Contents

ANALYSIS

Problem Outline

My project is inspired by Copter and Jetpack Joyride. These resemble platformers where the player aims to avoid obstacles by controlling their height via a binary input. The user's high score is determined by the number of obstacles pass. The obstacles are generated randomly and are of varying nature. Collision with an obstacle reduces the number of lives by 1. The game may be pausable and the meta-data saved in a txt. There may be a two-player version. If the user can control their horizontal velocity, then there will be a chasing missile that stops them from going to slowly. The chaser's speed would increase exponentially alongside the magnitude of the user's input effect (i.e. the acceleration increases). If the chaser hits an obstacle, their velocity becomes 0. The max speed of the user must be greater than the chaser always, this difference could decrease as the score increases, making the game progressively more difficult.

This will require a score counter box for each player; a velocity measure on the left; a game over/high-score screen; a pause menu that allows the user to save their progress; and a 'speed up' message every x points.

Hardware Requirements

Users require the following in their systems at a minimum since I plan to run the program on PyGame, so the user must be able to run Python 3.12 or later:

- A 32 bit or 64 bit processor (32 bit may struggle with processing power)

- 5 GB of secondary storage (sufficient for a 2D project on PyGame)

- 1GB of RAM (potentially more may be needed depending on the game/weather)

- Graphics API Dx10

- Monitor

- Keyboard

- Mouse

Software Requirements

- Python 3.12 or later to run the program
- Search engine enables troubleshooting
- Windows XP or later to run Python. I used the IDE 'Thonny'.

Computational Limitations:

Requirement	Justification	Why is this a limitation?	Potential Impacts
Abstraction and Decomposition	Simplifies program and creates focus on key details.	The extent to which I can decompose and abstract is limited. So complex characteristics like the physics of gravity will limit my computational development.	If I'm not careful, my game might be limited to simple features. I might miss precise detail like weather which would add to the experience.
Complex Program	This captures realistic aspects e.g. projectile physics. Object class hierachies will create a solid framework for future editing.An example of this is the use of object classes.	I mustn't make the program excessively time complex, else, the system may encounter latency issues because of the magnitude of computations/processes.	The game will not completely capture natural movement e.g. projectile motion; eagle flying.

Other Limitations

Requirement	Justification	Why is this a limitation?	Potential Impacts
Visual quality	Immersive game	I have other A Level subjects so I can't spend too much time learning and implementing higher quality graphics. Furthermore, this would require higher end user PCs. There is a trade-off between functionality and graphics.	Less immersive and realistic game. But the cartoonish style will appeal to younger kids.
Game Scale	Lots of content for user	The game could be infinite using auto generative algorithms. Distinct levels may not be feasible in 5 months nor will it be necessary for a good suer experience, so the game might finish within a few hours of successful gameplay.	Players cannot enjoy the game for more than about 10 hours before expunging all content.
Realistic Physics e.g. Projectile Motion	Makes the game relatable to users experience in the real world	I am only an A Level Physics student, my understanding of the practical implementation of formulae might be limited.	Motion isn't realistic and therefore is less immersive and predictable, perhaps frustrating users.
Loading paused instances	Easily let the user resume their 'runs'	The game is local, then saves must be transferred and uploaded manually between systems and independently by the user. It costs money to use a server which I don't have.	Users must realistically finish their game instance on one device, but can take multiple sittings.
Freedom of the Player	Enjoyable and non-repetitive game.	The game revolves around a binary input. This might appear	The user might get bored sooner than they would with a 3d runner.

		monotonous at first, but Flappy Bird proves this is an addictive concept as its difficulty increases e.g. closer and smaller obstacles. Movement in more dimensions requires a different game engine and more advanced physics which I am not equipped to implement.	

Hardware limitations

Requirements	**Justification**	**Why is this a limitation?**
Using a console for the game	This would enable more people to enjoy Elegant Eagles e.g. those without computers	I am creating the game using Python, so, there is no way to play on mobile unless I recreate the game in Lua or iOS.
Using a phone for the game	This would enable more people to enjoy Elegant Eagles e.g. those without computers	In addition to the reasons above, to release games for consoles I need to be a licensed developer which I am not.

Software limitations

Ideally, my game would be governed by precise laws of mechanics such as drag due to air resistance and preservation of momentum. However, as I explain in development iteration 1, the finite nature of delta time means that the graphs of displacement against time and its derivatives means that at some sufficient resolution, my engine does not conform to reality. Moreover, attempting to implement more precise physics via some Python engine such as mapping velocity to its corresponding polynomial, requires computational resources which would increase hardware requirements. In addition, such precision will not be appreciated by users and is therefore unnecessary, as discovered in the interview below.

Stakeholders and their roles

There is no dialogue in the game, this enables young children to play the game, then increasing the number of potential stakeholders. The goal age range is 3+.

The game will be played on a computer. Therefore, potential stakeholders cannot play on the go. We could increase the number of stakeholders by adapting the game to a touch screen via a digital keyboard.

I have selected a user to personify my target audience, James TC. He is a seventeen-year-old student in my computer science class who is an enthusiastic gamer. So, he will know what to expect from the type of game being created. Via the agile methodology, James will offer constructive criticism during biweekly meetings to improve the functionality and usability of my program. He will largely assess the game's difficulty. Furthermore, the head of physics Mr Newton, using his expertise in dynamics, will help me fact-check the equations of motions that dictate my game. I will ask for Mr Newton's opinion monthly on the formulae.

I also chose stakeholders from a range of ages and professions, for example: 9-year-old Oliver and 25-year-old David. I chose a range of stakeholders to represent the user base. Oliver for example is a young child, and David is a casual gamer. Oliver is an important stakeholder as young children have different opinions and skills to adults.

Based on stakeholder feedback, I can tailor the design phase to their likings. I need to ensure that I ask my stakeholders the right questions so that there is no bias in the responses. To do this, my questions should be open ended and not directed; the questions must also cover a variety of options.

For instance, asking for 'number of lives' as '1' or '2+' could suggest that '1' is an unpopular choice, whilst it might be the most popular when the questionnaire is generated with more discreteness.

INTERVIEW: 13/09/17 PLAN WITH JAMES TC

I hope to learn about what the player base will want from the game's style, difficulty, visuals and sounds. This is essential to figuring out how to approach the design stage:

For the list of questions see 'appendix 1'.

Interview Script

- Question
  James' Response
- What animation do you want for losing a life?
  It's not too important, but there should be a lives counter in the corner of the screen that is clear.
- What animation do you want for losing all lives.
  Mario esque jump and fall out of map.
- What animation do you want for the enemy
  Angry laugh repeating – almost like a pirate from Cuphead.
- Do you want to be able to win or lose?
  Its not too important – but a sense of progress is so a score board would be nice
- If yes how hard do you want it to be to win?
  I'd like it to be extremely difficult so I get more fun/time out of the game.

- What kind of theme do you want for the game? What genre, colour scheme.
  A scary bleak colour scheme that excites and scares me.
- Should the eagle make sounds? If yes what?
  Sounds will make the game less scary and kind of funny so it should be at the start and end only (start and loss)
- What sound do you want the jumps to make?
  Quiet wing flap sound – keeps the game scary
- Do you want a win/lose animation?
  No – I'd rather the game mechanics themselves improve
- How many obstacles per second?
  Increasing number, 0.3 to 0.5 per second at first
- Do you want lives? How many?
  3
- Do you want to be able to pause.
  Yes
- Do you want scoreboard while playing?
  Yes just mine though, not anyone else's
- How would you lose the game?
- Should the game get more difficult as you progress?
- Do you want a leaderboard?
  Yes please – adds to the main GUI
- Do you want levels? How many?
  Perhaps different weather environments rather than levels to add variety.
- Two player?
  Yes please!
- Would you want local or online two player?
  Online seems infeasible – the 'couch game' style is great.

I repeated these questions with some people in my school. James does computer science, and I got a range of backgrounds to represent the real userbase e.g. a variety of ages/professions.

Instead of conducting the interview with several other people, I made a multiple-choice form on Google Forms and collated the data in Pie Charts. This reduces the chance of bias and makes people more willing to help as it takes less time. Here are some examples of what they looked like:

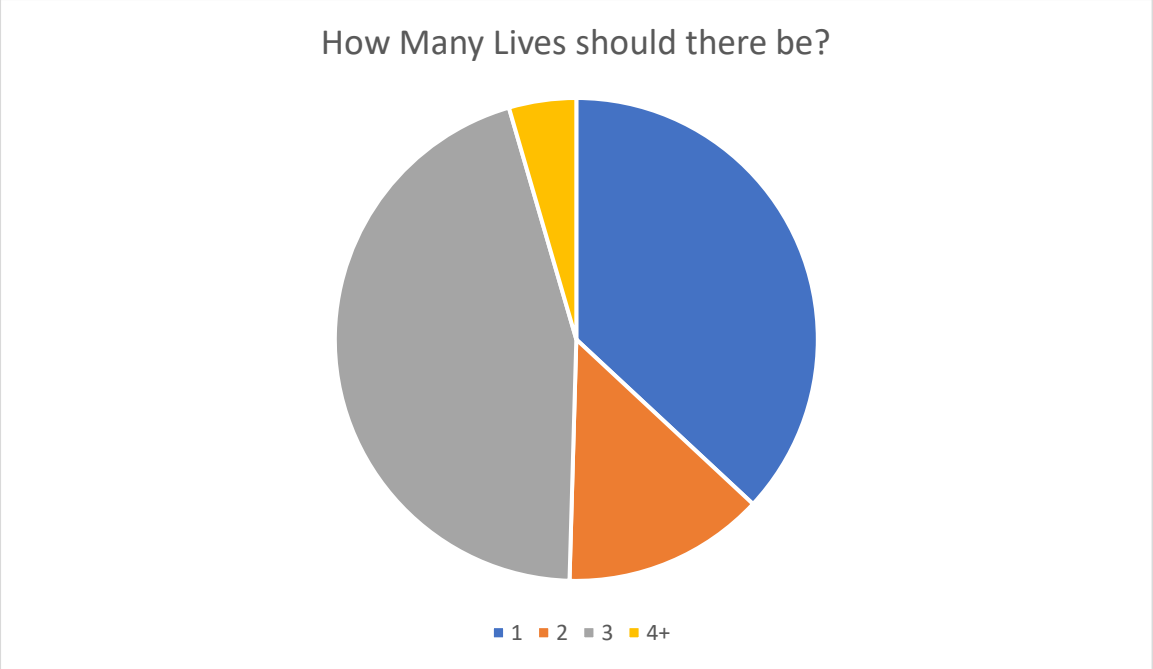


Fig. 1.1

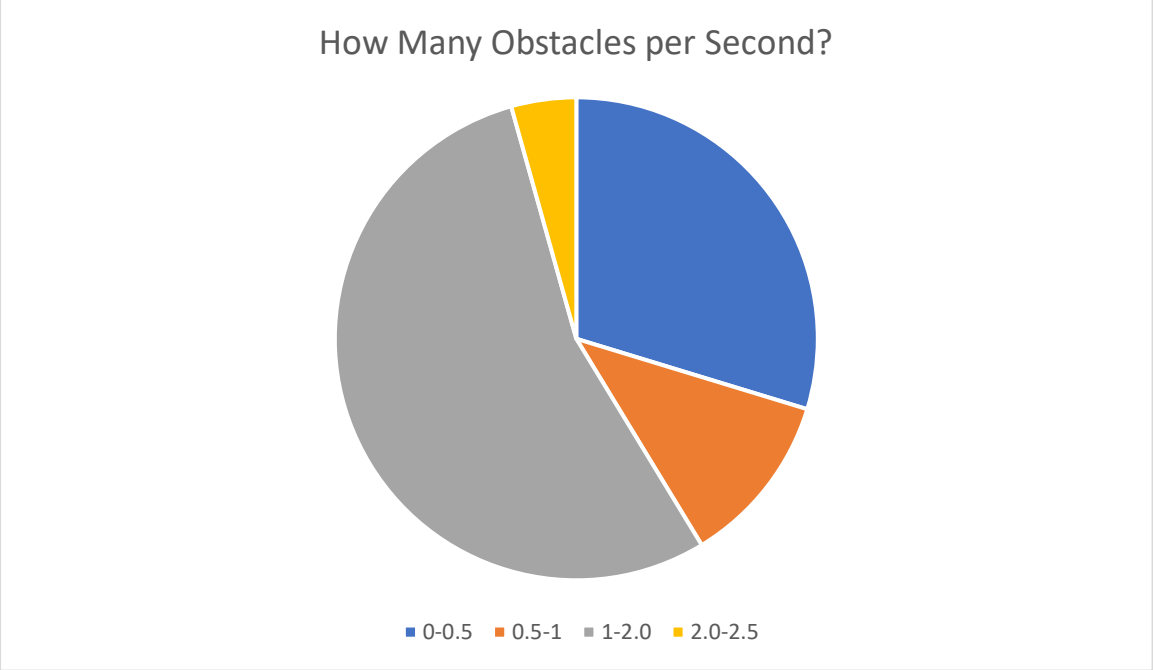


Fig. 1.2

Use of Computational Methods

This problem is best solved in a computer environment since there are many varying aspects. The use of a program, more specifically a game engine facilitates this.

Thinking Abstractly

Abstraction removes unnecessary info. This will keep the project feasible as well as clear to a user therefore easy to use. There are lots of objects such as players and obstacles that can be simplified.

- Change of basis to view a 3D world in 2D. In fact, the user only controls the vertical velocity, the horizontal velocity varies by a fixed function.
- Simplified cartoon style – starting with a 'point'
- Simplified soundscape – reduce the number of noises to make the user understand e.g. game over noise, jump noise, wind sound based on velocity.
- Obstacles passed counter in top right so that the user sees their progress.
- Simple GUI to change difficulty, colour scheme etc.

Thinking Ahead

This gives the program structure and therefore keeps the project progressing efficiently.

- I plan to use PyGame because it handles a range of input devices.
- The input methods will be by keyboard.
- The outputs will be visual and audible; the latter indicates game over/jump.
- I will uses score boxes to keep the user invested by letting them aim for their highscore. Each user's own high-score could be displayed as well as a session high-score for multiplayer competition.
- I will save files to a persistent store of data by saving an instance of a game as an object.
- I will use a database to import, check data in case of corruption, instance of object free marks.

Thinking Procedurally

Provided project structure, makes parts of code reusable, thus making the entire program consistent, bug tested, and easier to create. I can decompose my project into the following:

https://www.canva.com/design/DAGQdpCTRvs/Fku0NTEQhQJylHffw3qhRw/edit?utm_content=DAGQdpCTRvs&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

These processes comprise the backbone of the program's functionality

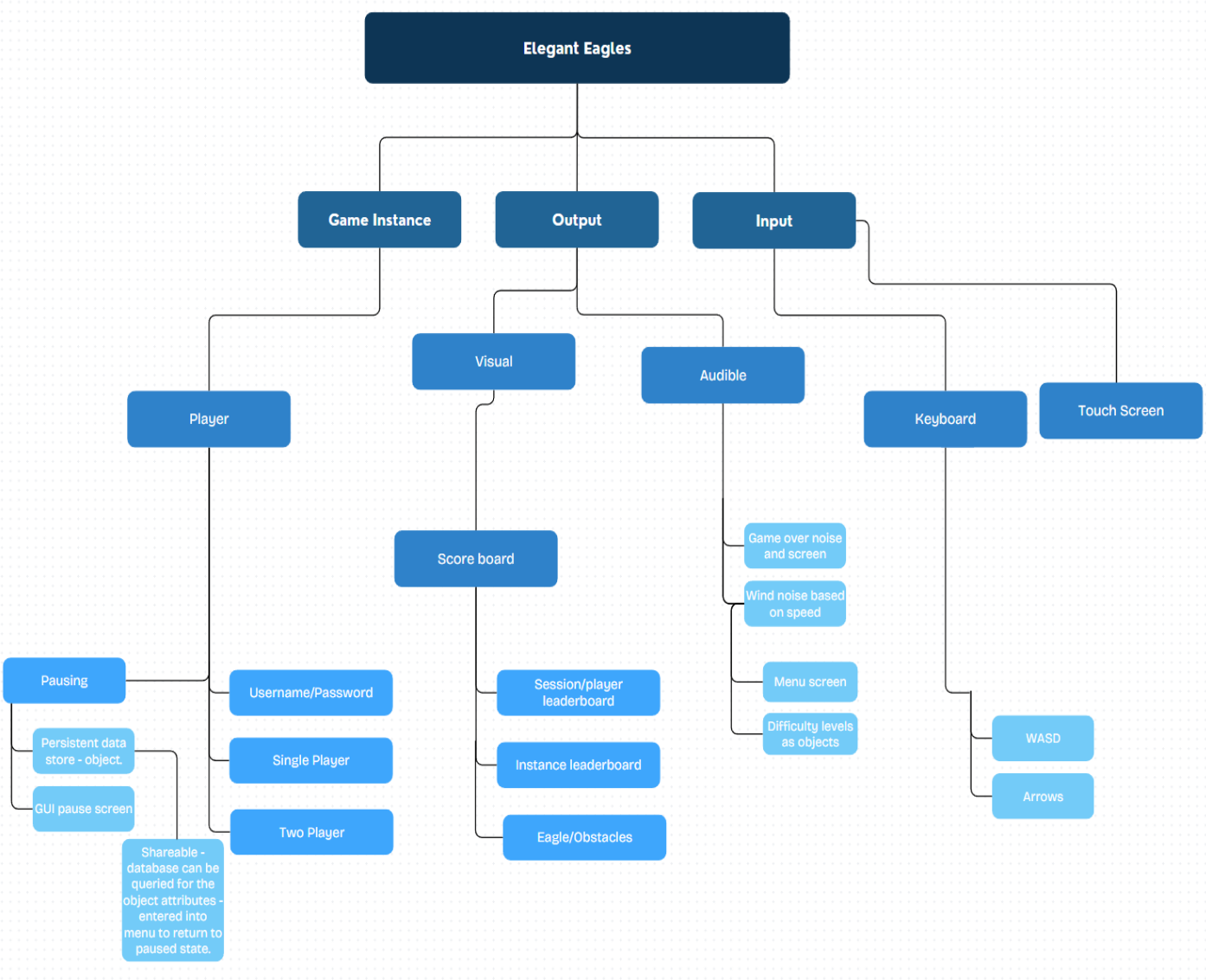


Fig. 1.3

Game instance

The player is identified by their username. Passwords could be hashed and checked from a database for verification. Pausing takes the user to a menu where they can save the data of the game instance (an object). This could be shareable and could be entered to the main menu to restart a paused instance.

Output

Outputs are essential. They enable reactionary input and the entire user experience. Visually, in an instance, the player can see their score, the session/personal leaderboard as well as the obstacles/eagle itself. Variable wind/jump/game over noises provide an immersive, customisable user experience.

Input

Similarly, the inputs are imperative, while simple. The space, W, and/or Up_arrow keys are the sole input in a game instance. The main menu is to be navigated by a mouse. Touch screen users can 'jump' by pressing an on screen 'button'.

Thinking Logically

The menu driven nature of the GUI requires decisions. Furthermore, the game itself requires constant iterative checking of conditions e.g. 'eagle_hit==True/False'.

I can create algorithms that represent the stages of gravity that control the movement of the players and some obstacles.

Namely:

$$F_G = G \cdot \frac{(M_1 M_2)}{d^2}$$

Fig. 1.4

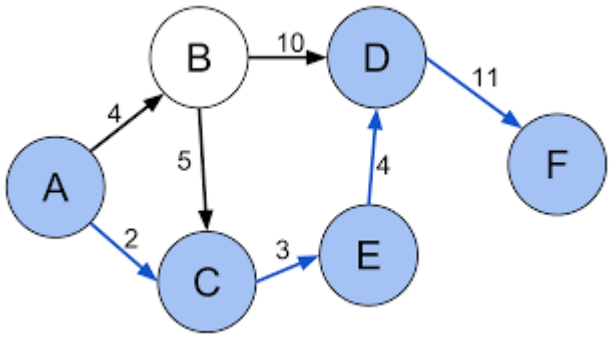


Fig. 1.5

The two following images are respectively the Euclidean and Manhattan distance formulae. The former is more appropriate given the continuous nature of the coordinate system that all objects in the game exist in.

$$h = \sqrt{(x_{start} - x_{destination})^2 + (y_{start} - y_{destination})^2}$$

Fig. 1.6

$$h = |x_{start} - x_{destination}| + |y_{start} - y_{destination}|$$

Fig. 1.7

Furthermore, I will likely implement the A* algorithm to control the movement of a predatory enemy eagle on some levels. Crucially, the algorithm must have some imperfections to allow the imperfect player to escape. Perhaps the enemy's pathfinding can approach perfection over time to end the game and let more skilled players have fun.

Thinking Concurrently

To optimise the program's running, processes must run simultaneously. For instance, checking 'object_count', and incrementing 'obstacles_passed' accordingly whilst checking 'user_input' or 'jump==True/False'.

I will create multiple object instances and have them running concurrently. For example, the multiple obstacles or eagles

Conclusion

The stated computational methods demonstrate that the problem must be solved by a computer program. They also simplify the problem, highlight the important aspects whilst improving the user experience.

If I am successful in designing a solution, stakeholders can play a game where they uncontrollably 'fly' horizontally at increasing velocities and aim to 'jump' over and under obstacles, hopefully entertaining them

Evidence of Looking at Other Systems/Ideas and How They Link to My Proposed Designs

After discussing with James TC on the 17.09 (agile iteration 1), I've concluded that stakeholders are after a unique game experience rather than high end SFX/VFX.

Jetpack Joyride

'Jetpack Joyride' is a 2D platformer where the user controls vertical velocity. The goal is to avoid obstacles which increase in difficulty as the horizontal speed increases. This is a good idea as it is simple to implement and makes Eager Eagles non monotonous as the game will be exciting.

Jetpack Joyride is simple so it's easy to learn. Jetpack joyride has 'character swapping' but it is up to chance, I intend to improve on this by giving the user flexibility.

Fig. 1.8

Flying obstacles add a thrilling element to the game. Users often encounter unique obstacles which they must improvise against e.g. direction/speed of missiles. This is a good idea because it adds variety to Eager Eagles.

Otherwise, the enemy could simply move x metres towards the player in a straight line per tick, if the velocity of the enemy is like the player and their displacement sufficiently small the enemy will rarely collide with obstacles. This is possible though and can be exploited by the player to escape the enemy as in Subway Surfers.

Fig. 1.9

The user's score is measured by the coin's they catch and the time they 'survive' for. Fascinatingly, Jetpack Joyride employs gravitational field equations in the character itself when attracting coins with the magnet. This idea of coins could be perfect for Eager Eagles because it adds an additional layer of difficulty to the first few obstacles as in 'Subway Surfers' - keeping the user always entertained.

Fig. 1.10

Via my description of an imperfect **A* algorithm**, the enemy eagle will path-find to the player with increasing speed and efficiency – this is like the missiles in Jetpack Joyride which spawn in at the players y coordinate.

Fig. 1.11

When modded, the multiplayer feature in Jetpack Joyride allows collaboration or sabotage between players, transforming an 'arcade game' to a 'couch game'.

Jetpack Joyrides retro style is fun. The vibrant colour scheme appeals to younger audience members, as my project will (3+).

The use of sidebar menus fills the screen and makes the logo on the right more immersive.

Temple Run

Fig. 1.12

Temple Run is a 2D runner (with 3D turns) where the player runs from an enemy while avoiding obstacles. Like Jetpack Joyride, Temple Run speeds up, ensuring the player eventually loses and adding inevitable difficulty to the game. The game employs obstacles that get rid or obstruct the path. As in Jetpack Joyride, there are 'power ups' that add variety to the gameplay.

Temple Run's retro style enables the enemy to be scary, immersing the player. The yellow colour scheme gives an 'abandoned' aura to the game, making the game more surprising and intense.

Fig. 1.13

Temple Run's game over screen includes an animation of the monster attacking you and segways into the 'start game' screen. This makes the gameplay cycle enjoyable as the user is encouraged to keep playing. I aim to have a 'lose' and 'win' animation.

The font from Temple Run is adventurous and the yellow letters makes it seem like its words from real life – decayed. This would contribute to the authenticity of Eager Eagle e.g. clouds in the logo. Perhaps this can be improved by letting the user customise the game font, but in an imromptu conversation with my stakeholder Oliver, I realised that too much customisation would make the game confusing and the important features obfuscated. This demonstrates an important use of **abstraction**.

Fig. 1.14

Temple Run's menu GUI lets users change 'skins' or buy powerups. The idea of powerups is a prevalent theme in similar games like Jetpack Joyride – they encourage repeat users and progress by collecting coins.

Fig. 1.15

Temple run uses flashing visuals to tell the user that a game phase is ending e.g. losing the 'invincibility' power up. This makes the user experience straight forward to understand. Whilst such graphics are not financially or temporally feasible for an A Level project as highlighted in my limitations, this gave me the idea to add an alert system for power ups when they end.

Flappy Bird

Fig. 1.16

Flappy Bird is a 2D platformer where the user moves up and down to avoid tubes (obstacles). Flappy Bird implements a cartoonish geometric graphic style which appeals to younger users. Flappy Bird is also the only game of the 3 that enables keyboard controls, this makes rapid inputs more feasible and thus a gameplay feature which enables the user to get better at an otherwise 'simple' game. This is very similar to Eager Eagles in the character's movement.

Fig. 1.17

The style and GUI is the perfect fit for Eager Eagles because the buttons are large and clear, making their use clear.

Fig. 1.18

The score system in Flappy Bird is simple and adds suspense to every consecutive obstacle in comparison to the large tallies built up in Jetpack Joyride.

Flappy Bird is special because of its functionality. The users entire screen is used for the binary input. This makes the game accessible to younger kids. On a keyboard, the user should be able to press any button to jump in Eager Eagle.

Key Features and their Justifications

My success criteria have been inspired by my research and reflect the requests/ideas of my stakeholders like James TC and Mr Newton.

FEATURE	**REASON**
Pause menu/Options panel	Lets the user take a break/change settings e.g. volume
Load GUI	To start the game
Simple geometric graphics	To enjoy the game/react to visual stimuli easily
Sound effects	To enjoy the game/react to audial stimuli easily
Keyboard controls	To move the character
GUI Menu	Check leaderboard (competetive games are more enjoyable), account info, level
Quit Game	Stop playing the game
Moving character model	To enjoy the game/react to visual stimuli easily
Flying obstacles	Makes the game fun and adds variety – more difficult than stationary obstacles

Game Over Screen	Satisfying end – more enjoyable game.
Saving pause info: player position, health, status	Play between devices on one game instance
Progress Map	Satisfaction and urge to continue playing – idea of progress.
Two characters	Play with friends.

Success Criteria

We will describe the success criteria using a table:

1. Criterion Description
2. Category; one of: Functionality, Robustness, or Usability
3. How we will measure whether we achieved the success criteria. Usability will be measured using quantised surveys illustrated via pie-charts. Success will be measured holistically using both Alpha and Beta tests.
4. Justifying the criterion category (for the categories 'what might be done' and 'what realistically cannot be done') and justifying the need for the criterion.

Success Criteria I will do

No.	**Criterion**	**Category**	**How to Measure Achievement**	**Justification of Criterion Categorisation**
1	Store high scores, pause states, and shared game data using a persistent data store	Functionality	Testing save/load functionality across multiple sessions	Ensures user progress and scores are stored reliably
2	Run without crashes or critical bugs that disrupt gameplay	Robustness	Extended stress-testing using edge cases and play-testing across different game states	Ensures stability and prevents crashes that disrupt gameplay
3	Allow player to control character's vertical movement smoothly and responsively	Functionality	Observation of test users playing the game	Ensures responsive controls, improving player experience
4	Support accessibility features, such as large text for visually impaired players	Usability	Observing GUI navigation of at least one individual with a relevant impairment e.g. visual impairment.	Makes the game inclusive for a wider audience
5	Be simple enough for players aged 3+ to learn within one minute of play	Usability	Playtesting with a variety of users, including young children. We will conduct **quantised surveys** and report data using pie-charts.	Ensures intuitive mechanics that require minimal instruction

6	Run on multiple Windows versions and hardware configurations	Functionality	Testing on different Windows versions and hardware configurations using beta tester computer systems.	Ensures broad compatibility for users on the intended platform
7	Execute game logic with precision, producing consistent and accurate outputs	Robustness	Reviewing output accuracy through repeated tests	Ensures the game mechanics are reliable and predictable
8	Feature an aesthetically pleasing, sleek design with smooth, rounded UI elements	Usability	User feedback on visual appeal. We will conduct **quantised surveys** and report data using pie-charts.	Creates a visually engaging and polished experience
9	Trigger game over when player collides with a mask object	Functionality	Testing collision detection in different scenarios	Ensures a clear losing condition that aligns with the game mechanics
10	Use object-oriented programming (OOP) for managing levels, pausing, and tracking player attributes like speed and health	Functionality	Code review and testing multiple level transitions	Ensures modular, scalable game design for varied gameplay
11	Include start screen options for resuming gameplay, viewing leaderboards, and accessing account information	Functionality	Verifying that all menu options function correctly	Ensures easy navigation and access to essential features
12	Provide enough variety to encourage 'replayability' and long-term engagement	Usability	Observation of user engagement over multiple sessions	Prevents the game from becoming repetitive and keeps players interested
13	Generate dynamic obstacles that the player must avoid	Functionality	Checking obstacle spawn logic and collision detection	Adds challenge and engagement to gameplay
14	Track and display the player's current score and remaining lives in real time	Functionality	Visual confirmation during gameplay	Keeps players informed of progress and remaining chances
15	Sort high score table in descending order	Functionality	Running tests with different score inputs to ensure correct sorting	Encourages competition and motivates players to improve their performance
16	Define a win condition that allows players to complete the game	Functionality	Playtesting completion conditions	Provides a satisfying endpoint and sense of achievement
17	Allow players to pause and save progress at any time	Functionality	Testing pause/save features across different game states	Ensures players can take breaks without losing progress

18	Start a new game from the main menu without errors	Functionality	Checking the functionality of the start button in different scenarios	Allows quick restarts for better user experience
19	Store user accounts and allow players to log in and continue progress	Functionality	Logging in and verifying saved data across multiple sessions	Ensures personalized experiences and long-term engagement
20	Use a distinct color palette that enhances immersion and differentiates elements	Usability	Gathering user feedback on visual themes	Enhances immersion and makes the game stand out stylistically
21	Display a clear start screen with necessary options for game navigation	Functionality	Verifying menu navigation and responsiveness	Ensures players can easily start the game and access features
22	Increase score by 1 point per successfully avoided obstacle	Functionality	Code review and in-game testing of score increments	Provides a clear and consistent scoring system that rewards progression
23	Display final score on game-over screen	Functionality	Checking if the game-over screen correctly displays the last score	Provides immediate feedback on performance and encourages 'replayability'
24	Show remaining lives in the top-right corner at all times	Functionality	Visual confirmation during gameplay	Keeps players aware of their remaining chances to continue playing
25	Pressing 'Esc' must bring up a pause menu, interrupting gameplay functions	Functionality	Testing key-mapping functionality in different game states	Provides quick access to essential settings without disrupting gameplay
26	Ensure graphical user interface (GUI) is intuitive and easy to use	Usability	Observing users interacting with the interface. At least 50% of users must be able to traverse to the relevant button within 10 clicks.	Ensures clear and intuitive control of game functions

Success Criteria I might do

No.	**Criterion**	**Category**	**How to Measure Achievement**	**Justification of Criterion Categorisation**
1	Generate a variety of dynamic obstacles that alter gameplay unpredictably	Functionality	Playtesting and reviewing obstacle variations	Adds variety and challenge but is complex to balance effectively
2	Play a distinct sound effect when the player loses a life	Usability	Audio cue testing in different scenarios	Provides immediate feedback but is a minor addition
3	Implement a lives system that decreases upon collision and resets upon game restart	Functionality	Visual confirmation during gameplay	Allows recovery from mistakes but changes game balance significantly
4	Display interactive controls that visually	Usability	UI testing with user interaction	Enhances accessibility but adds extra UI complexity

	indicate which buttons are being pressed			
5	Show and allow modification of horizontal velocity based on player actions	Functionality	Code review and in-game testing of velocity display	Provides better player awareness but is not essential
6	Implement multiple difficulty settings that adjust game parameters like speed and obstacle frequency	Functionality	Observation of player experience across difficulty modes	Allows accessibility for all skill levels but requires extensive tuning
7	Develop a replay system that records and allows players to review past gameplay	Functionality	Testing recorded game instances for playback accuracy	Useful for improvement but requires additional memory management
8	Provide visual feedback when the player takes damage, such as screen effects or character animation changes	Usability	User testing with damage indicators	Enhances clarity but is not critical for core mechanics
9	Allow players to log in and save their progress persistently	Functionality	User authentication testing	Enables progress saving but is unnecessary for local play
10	Display how drag affects the eagle's movement based on speed and direction	Usability	Physics tests and visual feedback	Adds realism but might not be noticeable to casual players
11	Simulate different weather conditions like rain affecting eagle movement	Functionality	In-game physics testing with different weather settings	Increases immersion but requires additional mechanics
12	Create enemies with pathfinding AI to track the player	Functionality	Testing enemy movement in various scenarios	Enhances challenge but requires complex AI programming
13	Implement enemies that only detect and chase the player within a specific line of sight	Functionality	Debugging AI logic and user playtesting	Adds stealth mechanics but is difficult to fine-tune e.g. despawn radius/timer
14	Develop an enemy AI using a Neural Network to improve tracking and behavior over time	Functionality	Training and testing AI behaviour in various scenarios	Highly advanced but requires significant computational resources
15	Allow the player to control or have weather generated randomly/based on score	Functionality	Observing weather changes across playthroughs	Increases variety but may feel unpredictable without clear indication
16	Introduce a two-player mode with simultaneous gameplay	Functionality	Testing local multiplayer functionality	Adds 'replayability' but significantly changes game structure

17	Implement a chasing enemy using the A* algorithm, considering heuristics and distance to player	Functionality	Reviewing AI decision-making process in different situations	Creates smart enemy behaviour but is computationally expensive
18	Enable gradual health regeneration over time	Functionality	Verifying health regeneration in various situations	Allows longer gameplay but may reduce challenge
19	Implement an enemy AI with randomized pathfinding that can persist across game instances or reset each session	Functionality	Running tests on enemy AI adaptability	Adds unpredictability but is difficult to balance and implement
20	Allow players to pick an eagle character, but recognize that it adds little to gameplay	Functionality	Observing user gameplay	Allows customization but does not significantly enhance gameplay
21	Enable players to customize UI element sizes, including text size	Usability	UI testing with different accessibility options	Improves accessibility but is a minor enhancement
22	Allow players to rebind keys for custom controls	Usability	Testing rebind feature with multiple inputs	Enhances control customization and is relatively straightforward to implement but is an insignificant improvement for usability and will require GUI extension.

Success Criteria I realistically cannot do

Number	**Criterion**	**Category**	**How to Measure Achievement**	**Justification of Criterion Categorisation**
1	Visual feedback on collision, such as flashing effects or screen shake, to enhance impact perception	Usability	Observe whether users find the lack of visual effects impacts their experience	Would take too much time to implement and does not significantly contribute to gameplay.
2	Display on-screen indicators of user input for better control awareness and accessibility	Usability	Observe if users struggle to understand their controls during play	Does not enhance core gameplay, as players can already see the effects of their input through movement.
3	Implement physics-based movement with accurate acceleration,	Functionality	Compare in-game physics to real-world expectations and assess deviation by surveying beta testers.	Too complex to implement accurately, and the stylized graphics make unrealistic physics acceptable.

	drag, and collision responses			
4	Display power-up icons with timers in the top right to inform players of active effects	Functionality	Testing whether alerts appear correctly when power-ups expire	Would be a useful feature but is non-essential; players can rely on visual cues or experience.
5	Animate wing flapping dynamically based on speed and user input for realism	Usability	Visual assessment of animation fluidity and user feedback	Would add polish but is unnecessary for core mechanics.
6	Allow users to design, save, and play custom levels with an intuitive level editor	Functionality	Testing a level editor's functionality and stability using a usability survey and by inspecting user gameplay.	Would require significant additional UI design and a complex editor, making it impractical.
7	Implement online multiplayer with real-time synchronization and latency compensation	Functionality	Latency and synchronization testing between players	Requires a server, which is expensive to maintain and beyond the project's scope.
8	Provide haptic feedback for key actions like collisions and power-ups (if supported by hardware)	Usability	Testing controller vibration	Intended for desktops, which lack haptic feedback, making this feature irrelevant.
9	Develop a pathfinding enemy that accounts for gravity, requiring jumping or flying to navigate	Functionality	Testing AI movement under gravity constraints	Pathfinding algorithms control position, not acceleration; implementing gravity would require counteracting it, making it redundant.
10	Introduce a day-night cycle with gradual lighting changes for	Usability	Observe player gameplay	Would improve immersion but does not impact gameplay directly, making it a low-priority feature.

	increased visual immersion			
11	Implement dynamic weather effects (rain, wind) that influence gameplay and movement	Usability	User feedback on visual appeal	Adds polish but does not contribute to the core game experience.
12	Add background animations like drifting clouds and flowing water for a more dynamic world	Functionality	Testing whether the terrain generates correctly without breaking collision logic	Would be interesting but is difficult to implement without affecting balance and difficulty.
13	Generate randomized terrain to ensure unique level layouts for each playthrough	Functionality	Testing whether the terrain generated is unique across multiple instances.	Functionality Testing whether

Design

Foreword:

I am going to develop my project in 3 agile phases. This creates sufficient time for detailed and iterative stakeholder feedback. It also lets me carefully break down the problem into smaller, structured, independent parts that can be developed individually.

To keep my project flexible, I will also add design elements throughout the development stage. For instance, in hindsight I can say, I did not consider the drawbacks of MySQL in the design stage but appropriately adjusted with thorough justification in phase 2.

Throughout there will be explanation outside the code screenshots and inside them in the form of comments e.g.

'#This generates the user'

Systems Diagram

Below is my systems diagram. I split it into two images for clarity

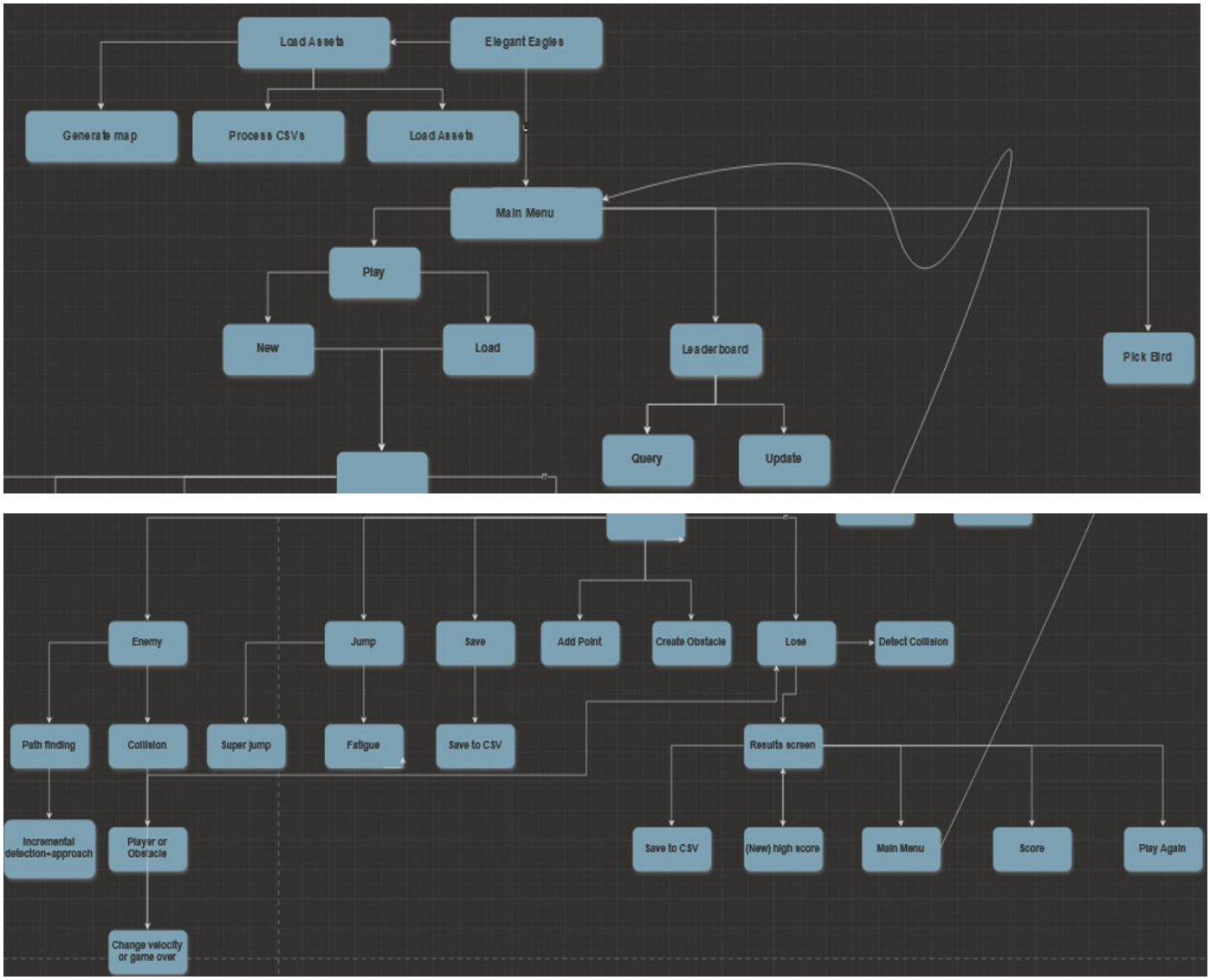


Fig. 2.1

The diagram outlines a possible framework for the program. The directional arrows indicate where users can make decisions. Except for the main menu—which is always available—once a decision is made, it cannot be reversed. This layout provides an overview of how users might interact with and navigate through the program. Although details might be refined as development progresses, the basic structure will remain constant. The diagram also illustrates how the application can be broken down into smaller, manageable segments.

Correspondingly, the development process will be decomposed into components of this size or less, before being integrated with the program and each other.
Naturally, new program components will appear in the iterative process as a leaf node of a parent node which exists in the program.

Top-Down Modular Design

I used top-down modular design to utilise decomposition. The problem is now more approachable and can be solved and tested individually and incrementally rather than all at once, making bug testing and tracing more difficult.

GUI and Usability Features

GUI Criteria:

1. Easy to use

   A. Intuitive navigation:

   The GUI will be menu-driven where events are triggered only by button presses to avoid confusion.

   The icons will be large and clearly labelled using industry standards, saving space. For instance, using a cartoon house for the home button and a cog for settings.

   B. Minimalistic Design:

   I will use few buttons and few submenus to avoid overwhelming the user. I will do that by grouping together related buttons e.g. all settings will be in one submenu.

   An example of a submenu is when the user presses ‘play’

   They should then be given the choice to ‘load’ an existing instance stored in the persistent data store or start a ‘new’ instance. ‘Load’ requires the user to drag and drop a file.

   C. Drag and Drop:

   When transferring files, users can drag and drop the files rather than needing to copy file paths which takes more time and care. I could even add a simple animation to show that the user is in the process of performing a drag and drop.

   D. Aid

   I will use tooltips that pop-up when the user hovers their cursor over a button. For example, when hovering over the ‘cog icon’, the user will see the text 'settings’.

   In addition, I will make use of **a colour-blind** toggle mode that will sharpen contrast in the violet side of the colour spectrum; by criteria 1A, this toggle will be easy to find. In addition, the use of icons (WIMP) will make the game accessible for the illiterate.

   Whilst I could create a tutorial using wireframes for new users to learn how to navigate the GUI, this shouldn't be necessary, otherwise the GUI would not be easy to use.

2. Aesthetically pleasing

   A. Colour Scheme:

   The GUI will use distinct, **vibrant** colours using industry standards to differentiate between buttons, menus, and interactive elements. For example, green for "Start" or "Confirm," red for "Exit" or "Close," and yellow for "Warning."

   The colours will be chosen to ensure accessibility, avoiding combinations that are difficult for colourblind users to distinguish. For example, I will use tools like colourblind simulators to test the palette.

   B. Icons

   I will use recognisable icons instead of text where possible, saving screen space and improving visual appeal. For example, a cartoon house for the "Home" button, a cog for "Settings," and a floppy disk for "Save".

   All icons will follow a consistent style (e.g., flat design or cartoonish) to maintain a cohesive

look throughout the GUI.
Icons will be large and clearly labelled to ensure they are easy to identify and interact with, even for users with limited technical experience.
The icons will be spaced evenly, and the icon sizes uniform while maximising icon size.
Additionally, I will maximise the number of lines of symmetry, as industry GUIs do.
C. Responsiveness:
For actions that take time e.g. loading games, progress bars will be used. This makes sure the user knows progress is occurring.
Moreover, there will be immediate user feedback on every button. For example, buttons will become dark when the cursor hovers over it, and even darker when pressed.

3. Consistent
   a. Consistent placement of buttons: For example, the "Back" or "Return" button will always be in the same location (e.g., top-left corner) across all submenus to ensure users can navigate naturally without relearning the layout.
   B. Standardised icon style:
   Buttons like the return/undo button will be the same icon, of the same size in all places.

An example of a submenu is when the user presses ‘play’

They should then be given the choice to ‘load’ an existing instance stored in the persistent data store or start a ‘new’ instance. ‘Load’ requires the user to drag and drop a file.

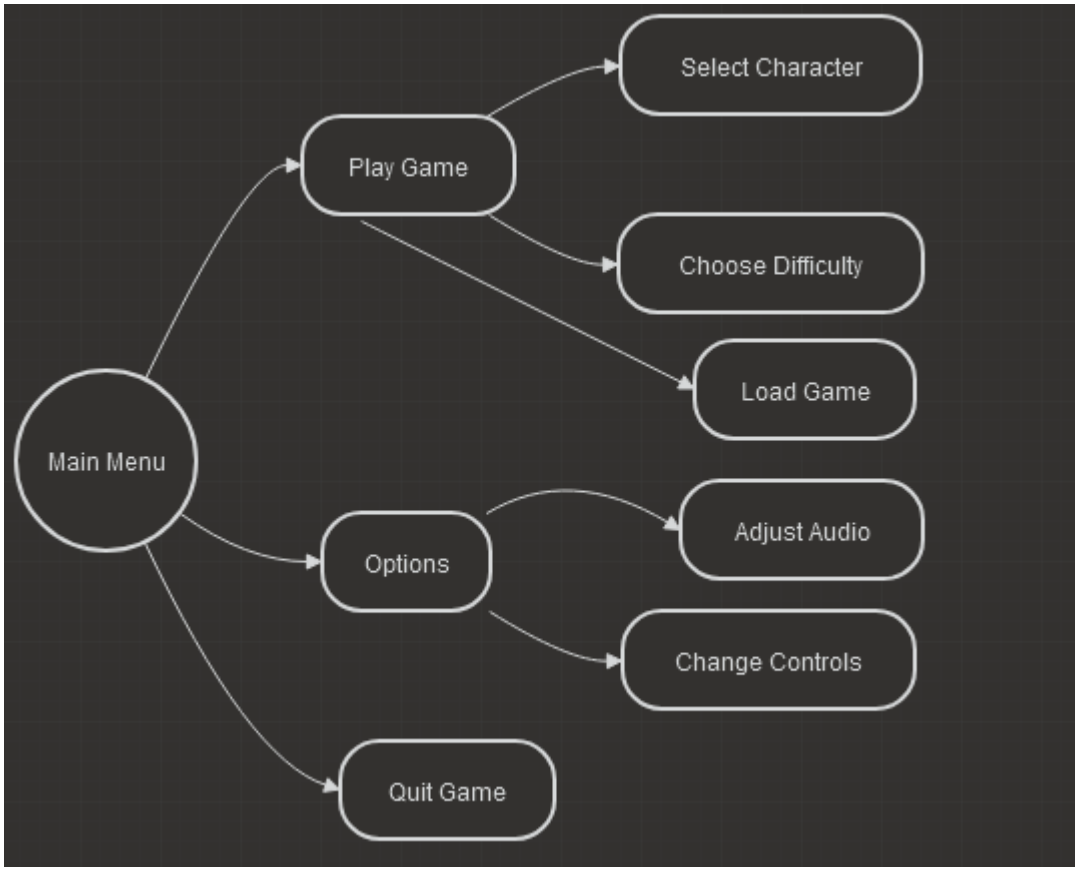


Fig. 2.2

EXAMPLE GUI:

While this GUI is heavily abstracted, *Elegant Eagles* as discussed with my stakeholder James doesn't benefit from a complex GUI, since the user makes all their choices in a 'run'.

Fig. 2.3

I developed this sketch using MS paint. Note that it can be directly transferred to the project development. Certainly preliminarily.

Player Design

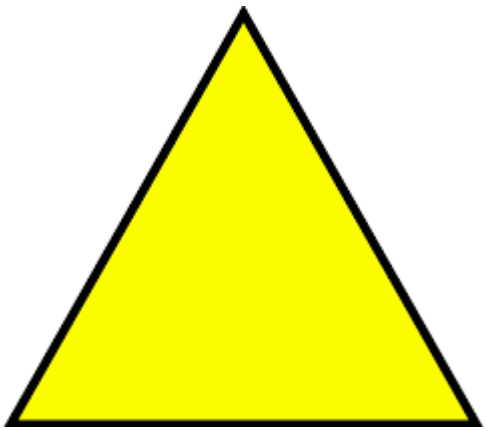


Fig. 2.4

This stands out because its a bright unnatural colour. This means that the user can follow their movement easily.

Enemy Design

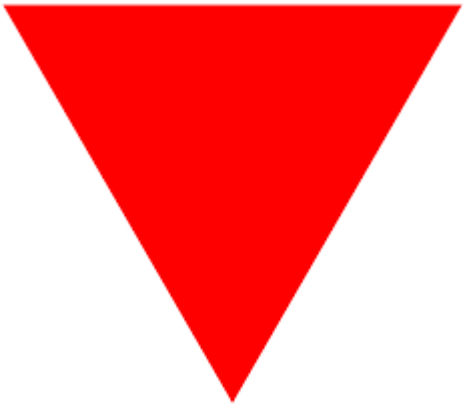


Fig. 2.5

Similarly, the sharp angular nature of polygons makes it stand out from the background. The use of red follows industry standard colour coding for 'danger'.

These are both very simple designs which lets me focus on the important features of the game itself.

Explanation of the Required Modules, Subroutines and Methods

Update:

Update is a module that, itself, will be created in development stage one before development in development stage three. Below, we will explore the intended progression of the module.

Development Iteration One

The movement of airborne particles is dictated by inertia and the Newtonian influence of gravity. The velocity and position of these particles are continuously updated within the game loop, ensuring smooth and realistic motion.

The update() method is responsible for modifying the xy-coordinates of airborne objects based on their velocity. Additionally, this method applies acceleration, altering velocity over time to simulate weight, drag and upthrust.

In accordance with Newton's Second Law, F=ma, the resultant force acting on the User Eagle is translated into acceleration. Instead of directly computing forces, the system considers accelerative effects to maintain computational efficiency.

The character's motion in *Elegant Eagles* is governed by multiple accelerative forces which together create a dynamic and challenging movement system. These include:

- An upwards acceleration triggered by the jump mechanic (iteration one)

- A constant downwards acceleration due to gravity (iteration one)

Development Iteration Three

- A horizontal acceleration caused by wind, which varies sinusoidally with time to simulate natural gusts (iteration **three**)
- **Air resistance**, which introduces velocity-dependent drag (iteration three)

All forces except air resistance can be implemented by selecting appropriate constants and applying them via method calls within the main game loop. These constants will be fine-tuned through stakeholder feedback and prototype testing. Air resistance requires further explanation to instruct implementation:

In real life, air resistance is a function of velocity, area and the coefficient of drag, via the formula:

$$F_{\text{drag}} = \frac{1}{2} C_d A \rho v^2$$

Where C is the coefficient, A area and v velocity.

However, since we operate in arbitrary units, a stakeholder-defined constant K replaces complex physical values. Therefore, the **lambda function** we will use to determine the acceleration will be:

$$D = K * (V_V)^2$$

Where K is the constant to be determined. Note that K is not universally constant since the User Eagle can move through a variety of fluids such as water and air.

Then as before, to create this acceleration, we simply call the appropriate method in the game loop.

User Interface:

The UI is fundamental to how users engage with the game. It impacts every facet of the user experience; ensuring that its design is visually appealing yet intuitive is crucial.

- The program aims to entertain. Therefore, the game should be consistent of vibrant colours and free shapes. One major source of my inspiration for this is 'Jetpack Joyride' which I explore in my Analysis.
- The UI must be simple to cater to the younger audience. As aforementioned, I aim for users of all ages, and gaming backgrounds to enjoy *Elegant Eagles; one mustn't speak English to enjoy an elegant game.*
- The colour scheme will consist soft, natural pastel and floral hues such as green and blue. Afterall, the game should be *natural* both in use and 'vibe' as my stakeholder Oliver put it. This should enhance the aesthetic. The seamless colour scheme transfer between the menu and game should encourage repeated and prolonged user enjoyment.

The UI will include the following elements. This list is flexible to change in my development diary:

Development Iteration One

- Welcome:
  The user receives a brief text-pictorial based tutorial about *Elegant Eagle's* features and challenges.
- Play:
  A central button used to start the experience.
- Settings:
  Where the user can change non-difficulty settings such as brightness/colour scheme. Colour scheme will be controlled with a colour picker as below.

Fig. 2.6

Development Iteration Three

- Difficulty:
  The user can vary the difficulty of the game using an integer input from 1 to 10.
  This input will require some simple validation. I considered sophisticated methods like using Regular Expressions, but given the narrow range of acceptable inputs, an inclusive rather than exclusive validation algorithm is most effective.
  The user input will take effect as varying the effective catch rate of pathfinding enemies; reducing enemy spawn rate; and shrinking/slowing obstacles.

<u>Splash Screen:</u>

An issue I've noticed in many Indy projects such as Temple Run, is that they take time to load. The solution to this, is to have a 'Splash Screen' which serves to assure the user, the program has not crashed and that progress is being made.

I have not yet decided what to use for the Splash Screen, perhaps just a loading sign.

Development Iteration Two

<u>Login</u>

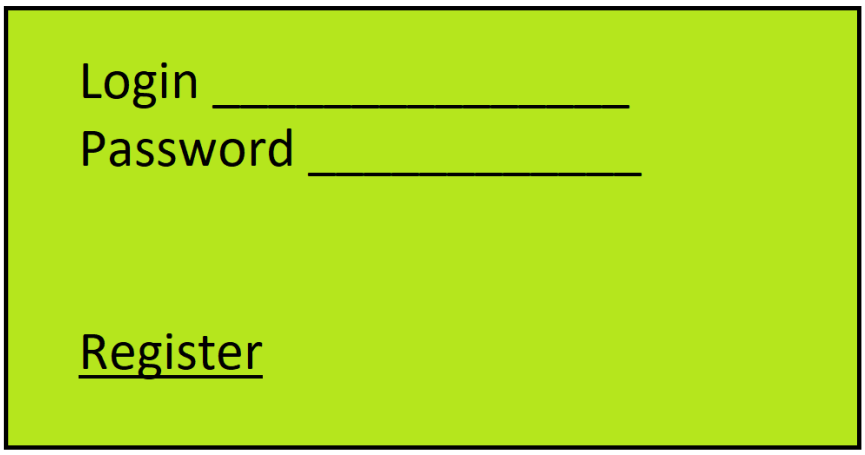


Fig. 2.7

The login system accepts a name and password which are validated using set structure **regular expressions:** preventing unnecessary database queries which will increase run time. If the input passes these local checks, then their attempt will be forwarded to the persistent data store for authentication using Try-Catch:

- If it is valid, the user's data will be imported to the relevant class objects e.g. brightness, colour scheme.
- If invalid, the user will receive details of the error using an Alert Box, they will then be prompted to enter a new input. I can also protect user data against brute force entry I can also timeout repeat failure. However, while trivial, the only hazard with such a weak point is the password leak itself e.g. if the user reuses the password.

Registration is handled in the same way, bar the authentication step. Instead, the entry is added to the persistent data store.

Saving:

I considered automatic saves to the users file, however there are two large issues with this:

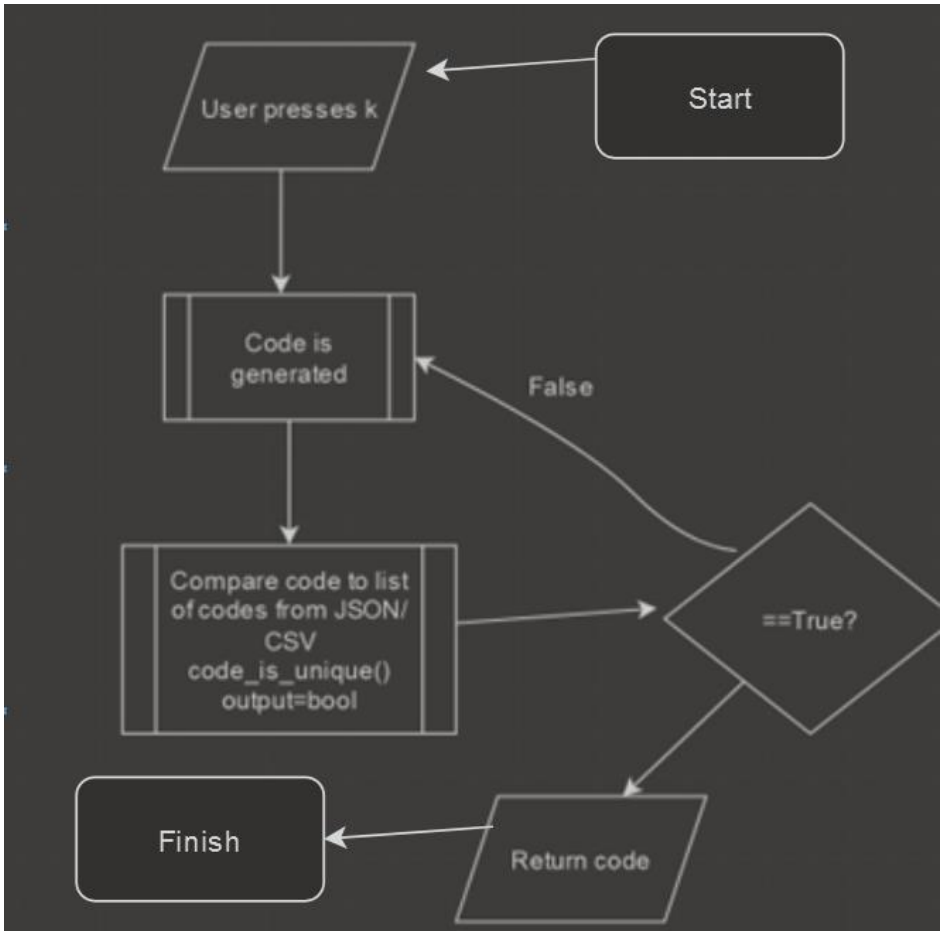
- The user may make a poor decision and want to boot a precious version.
- The user may want to go between devices without needing to arduously transfer files.

So the solution is that the user will generate a unique code at the click of the button. The following pseudocode explains the generation of this code and complimentary saving of data in a JSON.

- The user will press a button, perhaps 'k'. This is direct inspiration from the game 'Flappy Bird' from my analysis. This will trigger an algorithm to generate the unique save code.
- The responsible algorithm is nuanced. The code will be a 6-digit denary string. Crucially, the code must be unique else we risk overwriting a user's save. To start, the codes are generated pseudo-randomly. To guarantee uniqueness, we have two options. We can either perform a linear search every time to check if the generated code is new or not. This has drawbacks with respect to runtime, since we would need to search the same data-structure repeatedly. Naturally, I chose to make use of a **hash table** to optimise this search. This is because hash tables have query time complexit**y O(1) rather than O(n)** for linear searches. This does

however come at the cost of space complexity. Given that the codes are generated pseudo-randomly, there is no sophisticated hash table item:space required.

Below is the flowchart which summarises the decisions to be made in this algorithm.



Loading

The user's data will be imported from a JSON file. I decided to use JSON due to its readability since it makes debugging easier.

Processing JSON data:

The data must be validated before instantiating class objects. Some examples of the way they need to be validated include:

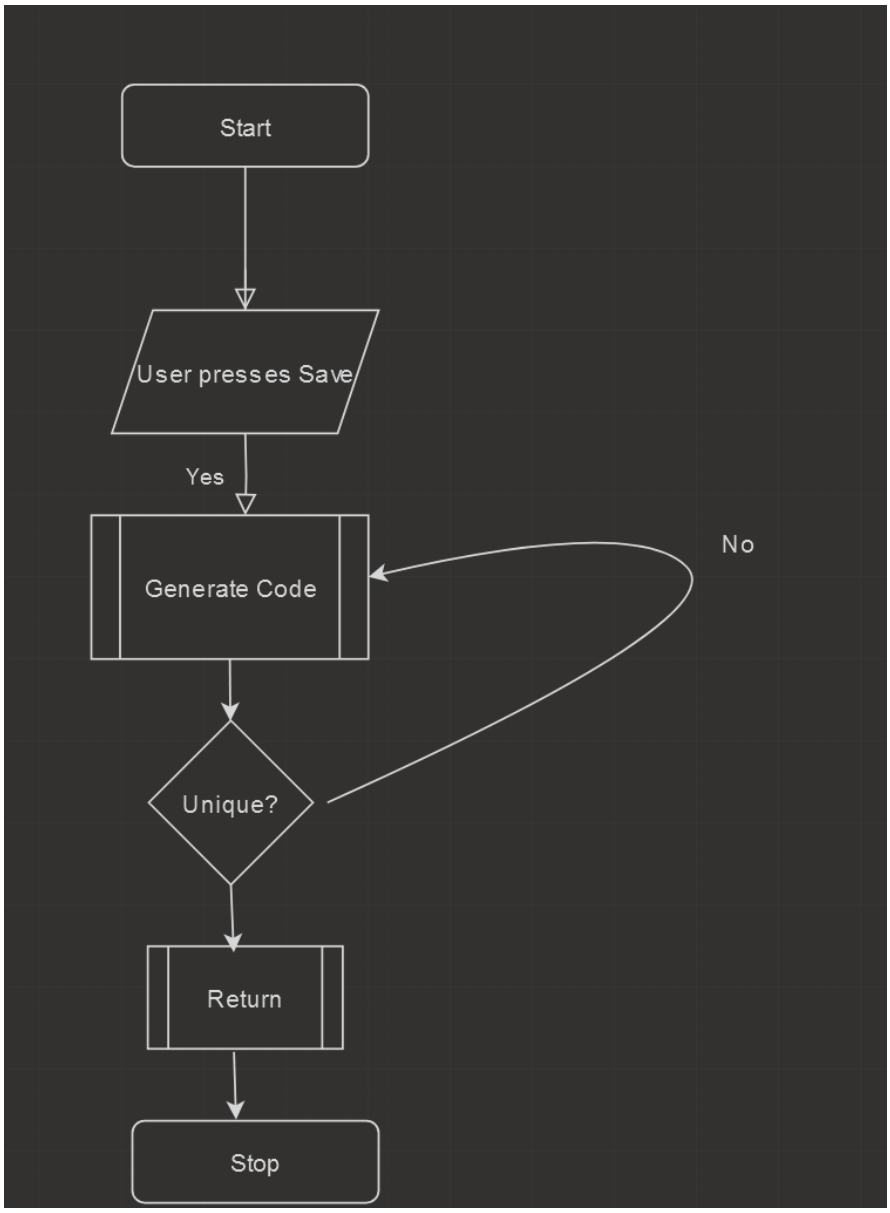
- Is there data
- Is the data type correct e.g. bool/str/int
- Is the data contradictory/possible.

Loading data:

Once the data is validated, we need to make class objects of this data.

- Class objects must be created seamlessly so that the user experience is continuous.
- The data from the JSON must be split into its relevant sections
- Then the split data is to be passed into object classes to create the game.

Below is the flowchart which describes the decisions to be made in this algorithm.



Development Iteration One

Creating the Ground

- The ground is a literal implementation of a **circular scrolling window**. Since the ground's appearance only serves to please aesthetically, the ground will cycle through the same 'animation' forever. In reality, we will be 'sliding' a jpeg.
- The update method from the Ground class needs to be called on every game iteration so that it appears like the ground is moving.

- The bijective movement between the ground and the obstacles will make it seem like the user is moving, whereas in reality the user only moves vertically. Rather than the converse, I chose this so that obstacles are generated on a need-only basis rather than pre-emptively. This is crucial in reducing time complexity in an up-scaled solution as it reduces the number of on-screen elements.
- The ground will also act as a pointer and an obstacle. Obstacles that pass the left boundary will be deleted from the obstacle object list and mask collision detected between the user-eagle and the ground will cost a life.

Here is how the ground might look:

Fig. 2.8

<u>Creating Obstacles</u>

- The implementation of obstacles can be effectively summarised by the UML diagram found overleaf.
- The basic obstacle that I intend to use in my first two development iterations is vertical pipes with gaps that the user must traverse. This was inspired by 'Flappy Bird' from my Analysis.
- When instantiating an obstacle object, we must pass attributes regarding the obstacles image-path and location. The Object-template parent class inherits from the Sprite Base template.

There are several child classes of the Object-Template parent class.

- Pipes:
  The pipes are the default obstacle. The user must pass through the gaps. Pipes are static and the pipe subclass requires the inverted and width attributes.
  Here is what the obstacle might look like:

Fig. 2.9

Development Iteration Two

- Orbs:
  The orb obstacle is the first instance of a dynamic obstacle. The orb motion is pseudorandom.
  Orbs will move vertically and will multiply its velocity by –1 upon mask collision e.g. with a pipe obstacle.
  The orb obstacle classes have velocity and radius attributes and inherit attributes from the Object-Template parent class.
- Flicker Orbs
  The Flicker-Orb class inherits the orb class. Its instantiation and behaviour is the same as the base Orb class, but it also has a Boolean *Visible* attribute which in effect, decides if a mask collision between the eagle and the orb results in the user losing a life. The Orb must 'flicker' in uniform periods so that the user can strategically plan whether they must evade it.
  Here is a potential graphic I designed for the orbs, inspired by *Pokémon's* 'Poke-Balls'.

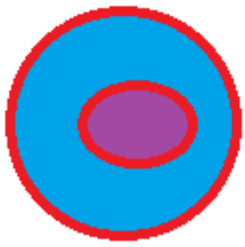


Fig. 2.10

Development Iteration Three

<u>Creating the user-eagle</u>

The user-eagle object class has several attributes; it inherits from the Sprite Base class.

The user can exist in multiple 'character states':

- *Gravity*
  The user experiences a state of semi-weightlessness. The specifics of *how* the gravity changes will be decided during the development process, utilising stakeholder testing and

feedback.
I considered using a ‘heavy ' eagle but after **stakeholder discussions** with James, I've realised that characters must inherently make the game **easier** for the user otherwise users simply will not use them.

- *Rotating*
  As previously described, the user will experience aerodynamic drag as a function of its area; area in turn varies with its flap-cycle-phase, that is, the fraction through its flap animation. This aims to emulate the flight of aerial animals. I modelled the flap-cycle-phase a function of velocity with respect to maximum/minimum velocities.
  When rotating, the eagle has no surface area therefore no air resistance acting on them. The state of rotation has been described as ‘bumper-rails’ by my Stakeholder James: it aims to make the user experience more straightforward, catering to the less experienced user.
  The user will also visually rotate. This is important as it clarifies the character state the user is in. The drag removal of drag was my stakeholder James’ idea; he suggested it because it will mesh nicely with our idea of an underwater map, which I will explain below.
- *Mini*
  The user is reduced in size while conserving its ratios, *mini* fractionally enlarges. This alters not only the user’s hitbox to aid evasiveness, but it fundamentally changes its movement. In accordance with the square-cube law, the effects of air resistance also reduce. *Mini* is therefore effectively the inverse of G*ravity.*
- *Time Control*
  The user reduces their horizontal velocity and acceleration due to gravity. *Time Control* aims to create more room for error from the user, since they can make more meditated inputs.
- *Reverse*
  Inspired by ‘Geometry Dash’ from my Analysis, the user switches the direction of their jumps and force due to gravity. *Reverse* is targeted at experienced users who want to correct misinputs; *Reverse* must reset the jump cooldown to make this possible.

Enemies

A* Pathfinding Enemy

I chose to make use of a pathfinding enemy so that the user had an ‘intelligence’ to play against. This adds a layer of difficulty to the more experienced user.

Detailed Heuristic Discussion:

An appropriate Heuristic function is quintessential to the algorithm since A* makes every single one of its decisions using the cumulative function *f* where:
f(x)=g(x)+h(x)
To choose a heuristic, we must consider:

1. The type of graph we will be traversing

2. The time complexity/resource requirements

3. The weighting and the nature of the function itself

- A perfect heuristic would guarantee perfect pathfinding. Perfect pathfinding would mean the user cannot escape the enemy at sufficient enemy velocities.
  Despite the dynamic nature of the graph structure being traversed, I decided to make the path-finding decisions iterative. This means that at low velocities, even perfect pathfinding would be avoidable by misleading the enemy.
  The heuristic should dynamic rather than a Pattern Database Heuristic, since the possible graphs to traverse are infinite, then the memory required for an effective PDH algorithm would be more than most commercial devices much less IDEs can handle.
- The boundedness of the Heuristic relative to the true distance controls the purpose of the algorithm.
  If we define the real cost of travelling between two nodes as *C,* then:
  If *C<<h(n)* then the algorithm is greedy, and the enemy becomes a 'glorified Line-of-Sight' enemy.
  If *C<h(n)* then the algorithm is imperfect.
  Else if *C>h(n)*, the algorithm is perfect but as *h(n)* decreases, the effectiveness of the algorithm is constant whilst the computational resources required increases with factorial complexity.

  I decided that a function *h(n)* that is the tightest upper bound on *C* is the best choice because the algorithm is imperfect but still very accurate.
  More precisely:
  $h(n) \leq C \ \forall n$
- Therefore, the velocity of the enemy mustn't necessarily be less than that of the user. This is a constant I will decide upon with stakeholder feedback after implementation. This is because it relies on the unknown of 'player skill' and game mechanics: in theory a user who can instantaneously control their vertical displacement could avoid the enemy, but this is not possible.
- Therefore, *h* is to compute the Euclidean distance between the node and the current position. This will certainly be an inclusive upper bound as:
  If there is an obstacle, this distance will be infinite and the algorithm greedy.
  If there is no obstacle, the enemy will traverse perfectly.
  In both cases *h<=C*. Euclidean distance is given by:

$$D_e = \left[(x_2 - x_1)^2 + (y_2 - y_1)^2 \right]^{\frac{1}{2}}$$

Fig. 2.11

Given that D_e is to be used comparatively, as an element of the cumulative function; I can make the optimisation of forgoing the exponentiation of the right-hand side, instead:

$$(D_e)^2 = [(x_2 - x_1)^2 + (y_2 - y_1)^2)$$

This last step is important since otherwise, the scalability of the program would be poor since the square root computation in Python has a large run-time cost.
A Euclidean heuristic is more suitable than Manhattan or Chebyshev heuristics because the game is free movement. In addition, I will use **Alpha-Beta pruning.** Since the enemy can only move in less than 180° and the pruning method is consistent, then the run-time saved is more than the pruning costs.

Pathfinding:
The A* algorithm consists of incremental distance towards the goal node. The user moves in discrete directions so that the algorithm is not just glorified line-of-sight. All obstacles are given a non-infinite score, so that the clever user can 'trick' the enemy into colliding with an obstacle that they themselves must avoid.
I considered incrementing acceleration rather than displacement so that motion felt smoother, however, since the direction of movement varies trigonometrically with the eagle which moves smoothly, I suspect distance will be sufficiently smooth. Moreover, using acceleration would make enemy movement less predictable and would make evasion trivial.
The pathfinding decision depends on the aforementioned cumulative function *f*. Accordingly, the algorithm moves to the node closest to the goal node. In classic A* we would ensure the node is unvisited, but given the dynamic graph, this is not a concern; backtracking is not possible.

The A* enemy will spawn incrementally; perhaps every 10 points and will despawn on mask collision. Mask collision includes collision with the floor, obstacles, other enemies and the user.

Reverse Deep-Q-Network; Predictive Dynamic Machine-Learning Enemy

To guarantee a challenge for the more experienced user, I will develop an enemy object class using machine learning. This enemy is unique to a 'run' and will never despawn. Otherwise, I would need to respawn it after each of its failures which would detract from the user experience of legitimacy. The enemy responsiveness aims to illusion user the user that the enemy is truly intelligent.

Python is very suitable for ML algorithms due to its powerful collection of data analysis libraries that I will use in my development.

The ML Enemy will approach a peak of McCoy's Evolutionary-Landscape, but its effective-Eagle-catch rate (EECR) making will eventually supersede that of its parent class, the A* enemy, since

- The Heuristic is imperfect
- It is not simply an iterative static structure, but a predictive dynamic one.

Where EECR is an acronym which I will use in my development iterations to refer to the threat an eagle poses to the user.

Fundamentally, this enemy will be driven only by negative feedback. The ML Enemy object class will inherit the A* object class. The main differences (polymorphism) will be that the ML enemy has no mask collision method. Instead, it operates via a feedback driven decision loop. The Enemy will always improve since it *must* move on every iteration. Otherwise, negative feedback ML models opt for intransigency. The enemy's negative feedback will consist of resets due to collisions with obstacles.

The ML enemy will therefore be an example of a reverse Deep-Q-Network.

To summarise the enemy, here is its neural network:

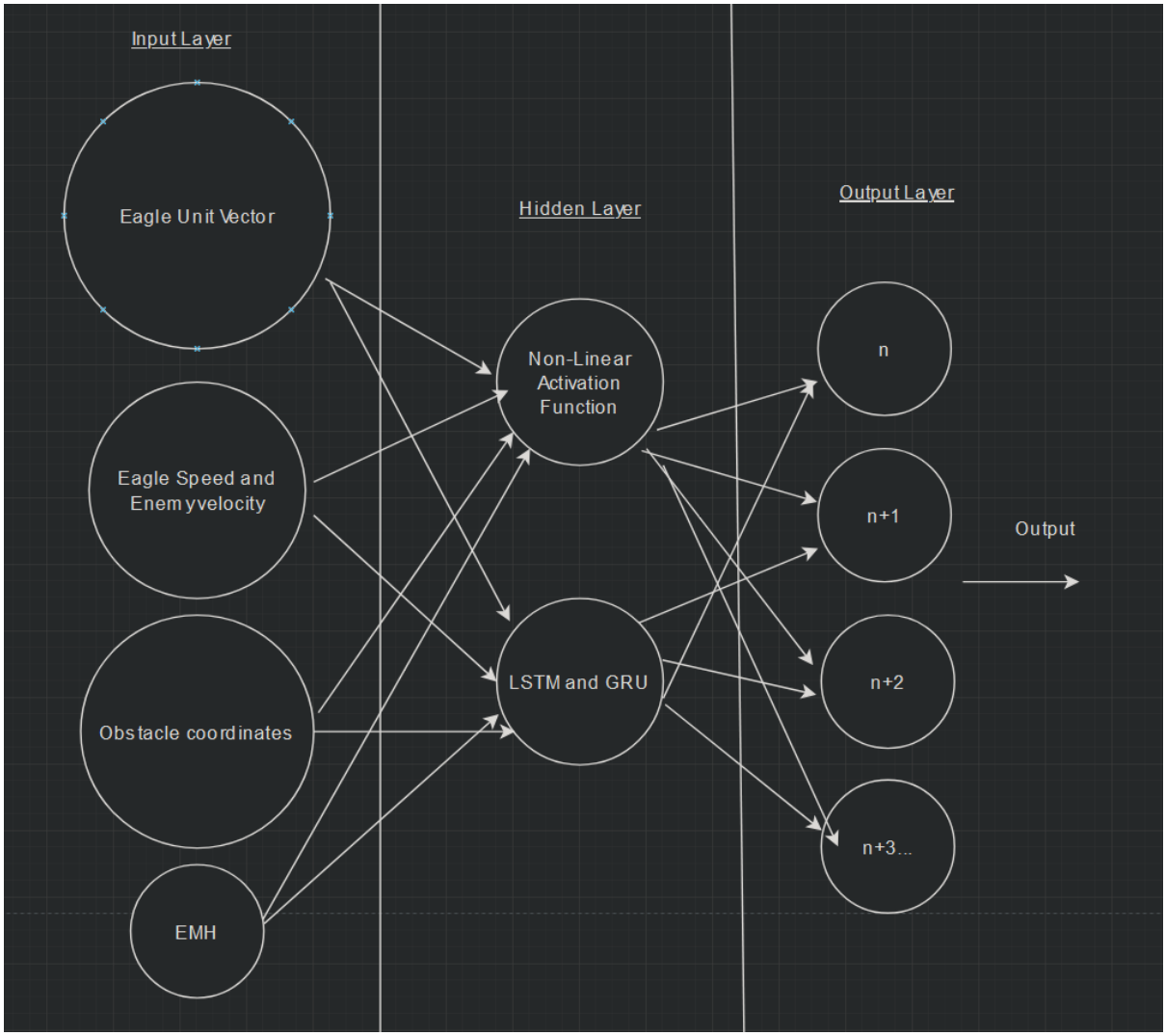


Fig. 2.13

The following describes the stages of the Neural Network:

Input

The ML enemy takes input parameters so that it can make predictions:

- The Enemy-Eagle unit vector.
  This is the straight-line path to the user, ignoring any obstructions. This serves as a directional cue.
- Eagle speed and ML-Enemy velocity.
  Eagle speed is a simplified parameter rather than velocity; we lose no granularity since the game is a 2D platformer centred around a 1D Eagle, therefore, when resolved correctly, the 2D eagle velocity has a 0 component.
- Obstacle coordinates
  This is crucial for navigation and collision avoidance. Given that the feedback directly promotes avoiding obstacles, some ML enemies may adapt a strategy of pacifism instead. Perhaps a simulative approach will be required to weed out such enemies so that the user experiences a quality enemy at all times.

- **Eagle movement history**
  Eagle movement history is a discrete parameter. It picks the option with the least percentage error to the real data of the mean eagle user data.
  EMH is therefore a compound parameter consisting of:
    - Mean speed (and therefore jump rate given that movement is 1D)
    - Mean vertical displacement
    - Median jump interval.

I chose these inputs is because they give a synoptic view of what varies between users who achieve a minimum score of 20. I intend to test this hypothesis in my development when I have a working prototype. I will test this by running trials with my stakeholders and discussing.

ML is delicate: it is plausible that the enemy develops TOO fast or slow for users to escape.

To fine-tune, I will vary the discreteness of EMH and perhaps the number of input parameters will vary.

Hidden Layer

The purpose of the Hidden Layer is to convert the many heterogenous parameters of different data types into a single pattern that the ML can be trained with.

- Non-linear activation
  Whilst the precise importance/weighting of the input layer is not necessarily determined, Non-linear activation performs a weighted summation of its inputs, followed by a non-linear activation likely **ReLU**.
- Long Short-Term Memory and Gated Recurrent Units
  Whilst Non-Linear Activation computes with static data. LSTM and GRU deal with temporally variable data. Data is considered over time and patterns are produced. This is essentially a multidimensional curve fitting exercise for Python.
  Therefore whilst Non-Linear Activation *decides,* LSTM and GRU *predicts* eagle behaviour.

Output Layer

The output layer filters the results of the Hidden Layer. It ensures that an appropriate balance of Non-Linear Activation and LSTM/GRU is implemented. This is decided upon using the negative feedback loop.

The output layer then makes a **decision**. Each neuron in the output layer corresponds to a potential action e.g. the anticlockwise angle about the x axis that the enemy will move in.

The network computes a Q-value, reflecting the EECR. The highest value EECR neuron is chosen.

Feedback throughout is collected as the time till collision. To determine time till collision, the motion of the ML Enemy is predicted and the coordinates are compared with the list of all of the coordinates of all obstacles. Feedback also reflects the decisions regarding weighting of the input layer.

Overall, the ML enemy uses a sophisticated learning algorithm to create a consistent yet dynamic challenge.

Line-of-Sight Enemy

The LOS enemy is a static enemy that behaves like the orb obstacle. That is, it is not affected by gravity of air resistance. The LOS enemy iteratively searches for a straight-line path connecting it to the user and moves a fixed distance along that line.

Movement:

- Define **P_e** as the position of the LOS enemy and **P_u** as the position of the user.
- Compute the vector **D** from **P_e** to **P_u**:
- $$D = (x_u - x_e)i + (y_u - y_e)j$$
  Fig. 2.14
- Normalize **D** and scale by a fixed velocity **V** to obtain the next position step P'_e:
  $$P'_e = P_e + \frac{VD}{|D|}$$
  Fig. 2.15
- V varies to balance the difficulty of the enemy.
- If an obstacle exists along the segment [P_e, P_u], the LOS enemy remains stationary.
- If there is no obstacle, the enemy moves uniformly along **D** until collision or user evasion.

Rather than have the LOS enemy *search* for the user, we will instead tell the LOS enemy where the user is and look for obstacles in the way by iterating through the list of obstacle objects. This produces the same semblance of a search to the user, whilst saving considerable processor run time.
If there is no obstacle, then the LOS enemy moves with uniform velocity.
If there is an obstacle the LOS enemy is stationary; this is the method by which the user will eventually escape the LOS enemy.
The LOS enemy should never collide with any non-user object, but it will despawn on Mask Collision.

The LOS enemy will spawn pseudo-randomly, scaling logarithmically the inputted difficulty.

If an enemy undergoes mask collision and it is not an ML enemy, then it will be de-spawned. This allows the user to evade most enemies.

Detecting obstacles:

- Using ray-cast along **D** to determine if any element of the obstacle object list intersects the segment **[P_e, P_u]**. Note that we exclude the enemies since they are dynamic. In fact, we only need to consider the static obstacles.
- If the ray-cast finds an object the enemy is stationary.
- Otherwise, the enemy travels along D.

This is very time efficient since ray casts operate like hash-tables; they are O(1).

Whilst the LOS enemy is more predictable and exploitable than the other enemies, it is important for scalability due to its low time complexity and will provide a surmountable challenge for the less experienced user.

Note that other enemies can still collide with the LOS enemy, de-spawning it permanently.

Pseudocode Examples

I chose to outline several algorithms behind my methods using pseudocode. This strategy is intended to streamline my future coding efforts and help others more easily grasp the underlying processes.

My program makes consistent use of object-oriented methods, therefore often individual algorithms are not self-contained. My solution to this is to refer to methods with clear placeholder names to ensure clarity. I will also choose suitable algorithms e.g. those that do not requrire comments where these names are sufficient as an explanation.

Load game

```
FUNCTION load_game(code) DECLARE current_score AS GLOBAL

TRY

IF is_valid_code(code) is FALSE THEN

    RAISE error "Invalid format; 6 digits required."



OPEN SAVE_FILE in read mode AS file

PARSE file content as JSON and ASSIGN to save_data



FOR EACH entry IN save_data DO

    IF entry["code"] EQUALS code THEN

        SET current_score TO entry["score"]
```

```
        PRINT "Success! Score: " + entry["score"]
        RETURN TRUE
    END IF
END FOR

PRINT "No matching save cold."
RETURN FALSE

CATCH FileNotFoundError:
PRINT "No file found"
RETURN FALSE

CATCH ValueError AS e:
PRINT "Error: " + e.message
RETURN FALSE

CATCH JSONDecodeError OR KeyError AS e:
PRINT "Corrupted save data: " + e.message
RETURN FALSE

END TRY
END FUNCTION
```

Update

```
Create function update(self):
#Update velocity
Self.speed += GRAVITY
        #Update displacement
Self.rect[1] += self.speed
End function
```

One limitation of this function is that it suggests that the graphs of the derivatives of displacemeent against time are polynomial. Whereas they are not truly polynomial because of variable drag. Moreover, all computer models are comprised uniquely of vertical and horizontal segments at some sufficient resolution, but this will not affect the user experience in any way.

A* Algorithm

```
create function MakePath(previousNodes, goal)
  path := [goal]
  while goal exists in previousNodes:
     goal := previousNodes[goal]
     path.insert(goal)
  return path

function AStarSearch(startNode, goalNode, heuristic)
  openNodes := {startNode}

=   previousNodes := an empty map
  cost := map initialized with Infinity
  cost[startNode] := 0

  heuristicCost := map initialized with Infinity
  heuristicCost[startNode] := heuristic(startNode)

  while openNodes is not empty:
     #Prioritises closest node
     current := node in openNodes with the smallest heuristicCost value
     if current = goalNode:
        return MakePath(previousNodes, current)
```

```
    openNodes.remove(current)
    for each adjacent_node of current:
        newcost := cost[current] + distance(current, adjacent_node)
        if newcost < cost[adjacent_node]:
            #Update to improved path
            previousNodes[adjacent_node] := current
            cost[adjacent_node] := newcost
            heuristicCost[adjacent_node] := newcost + heuristic(adjacent_node)

            if adjacent_node not in openNodes:
                openNodes.add(adjacent_node)
  return False #if no path is found
```

This is a spin on the typical A* pathfinding algorithm. A* aims to find the shortest path between a start and goal node on a weighted graph. It combines the strengths of Dijkstra's algorithm, which guarantees the shortest path, and Greedy Best-First Search, which uses heuristics to guide the search towards the goal. A* achieves this balance by evaluating nodes based on both the cost to reach the node from the start and an estimated cost to reach the goal from the node

The variation from the standard algorithm, is that this has been changed to deal with the dynamic nature of the graph. This means that the goal node is constantly changing. This will be handled by running the A* algorithm on a while loop, where the break condition is mask collision of the A* enemy.

Launch

This creates the 'jumpy' physics of *Elegant Eagles* (as my stakeholder James put it). Launch instantaneously multiplies velocity by –1. Note that we don't need to define velocity as vertical and horizontal because horizontal velocity is not in the user's hands. Moreover, it's a much simpler way of modelling the effect of a bird's wings flapping. In reality, there is a variable force exerted due to air resistance. I plan to introduce an air resistance function that does contact work against the Eagle's motion in development iteration 3. This will be a function of area, where area is a function of velocity.

Create function launch(self):

Self.speed=-SPEED

End function

Generate On-Screen-Element

This function takes coordinate, dimensional and some colour choice inputs and creates an on-screen element. This will be used as a template for all on screen elements. Perhaps in phase 3, this template can be made into a class e.g. if we want temporary on-screen elements e.g. showing that an enemy colliding with an obstacle rewards the player.

```
create function generate_OSE(screen, x_coordinate, y_coordinate, x_size, y_size, default_colour, active_colour, action=None) cursor = get mouse position press = get mouse button state

if x_coordinate < cursor[0] < x_coordinate + x_size and y_coordinate < cursor[1] < y_coordinate + y_size then
    draw rectangle on screen with active_colour at (x_coordinate, y_coordinate, x_size, y_size)

    if press[0] == 1 and action is not None then
        call action()
    end if
else
    draw rectangle on screen with default_colour at (x_coordinate, y_coordinate, x_size, y_size)
end if

text_surface = render text with font and color BLACK
text_rect = get rectangle of text_surface centered at (x_coordinate + x_size // 2, y_coordinate + y_size // 2)
draw text_surface on screen at text_rect

end function
```

UML Diagram of Non-Trivial Classes

Obstacle and Enemies:

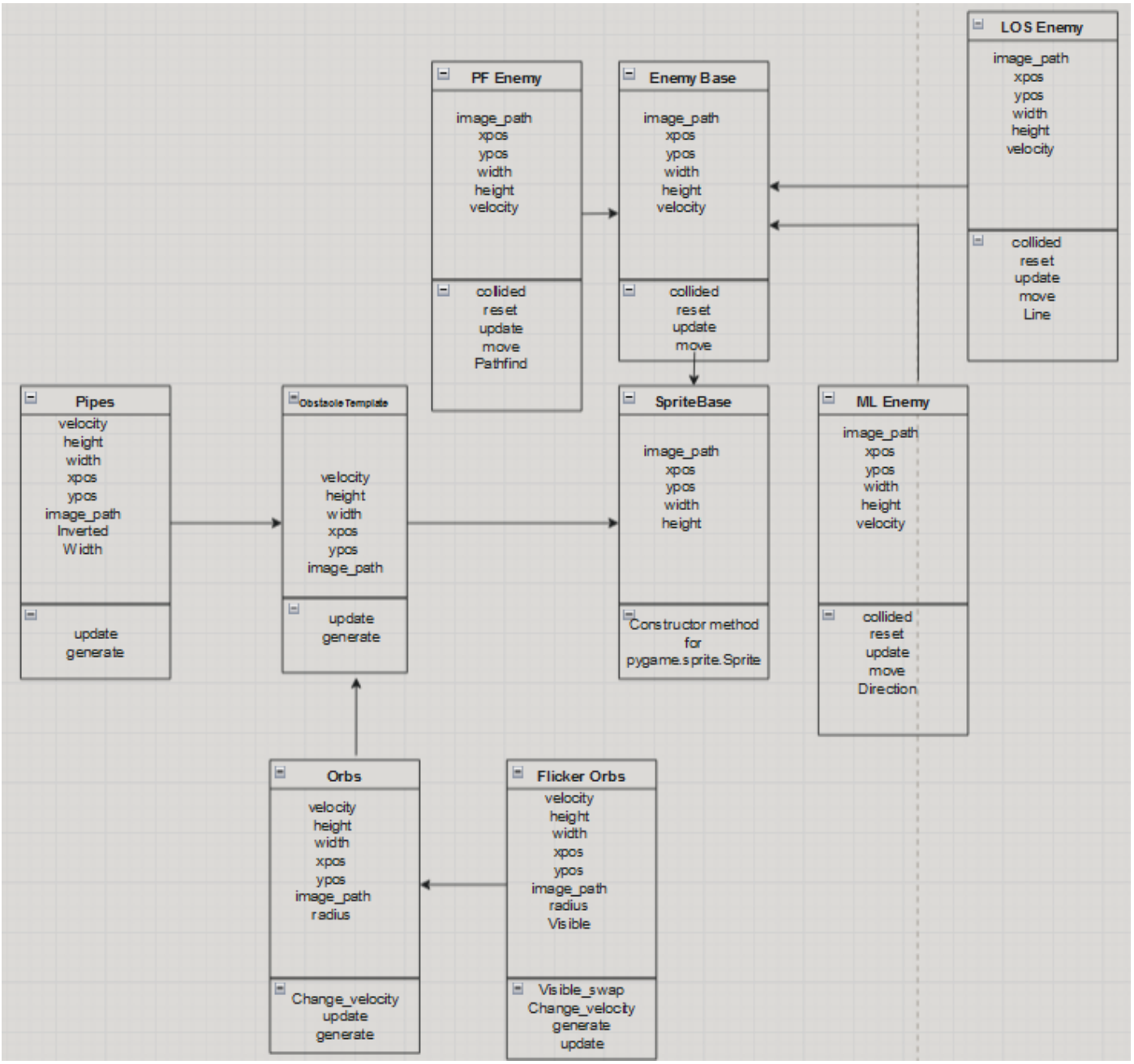


Fig. 2.16

User Eagles:

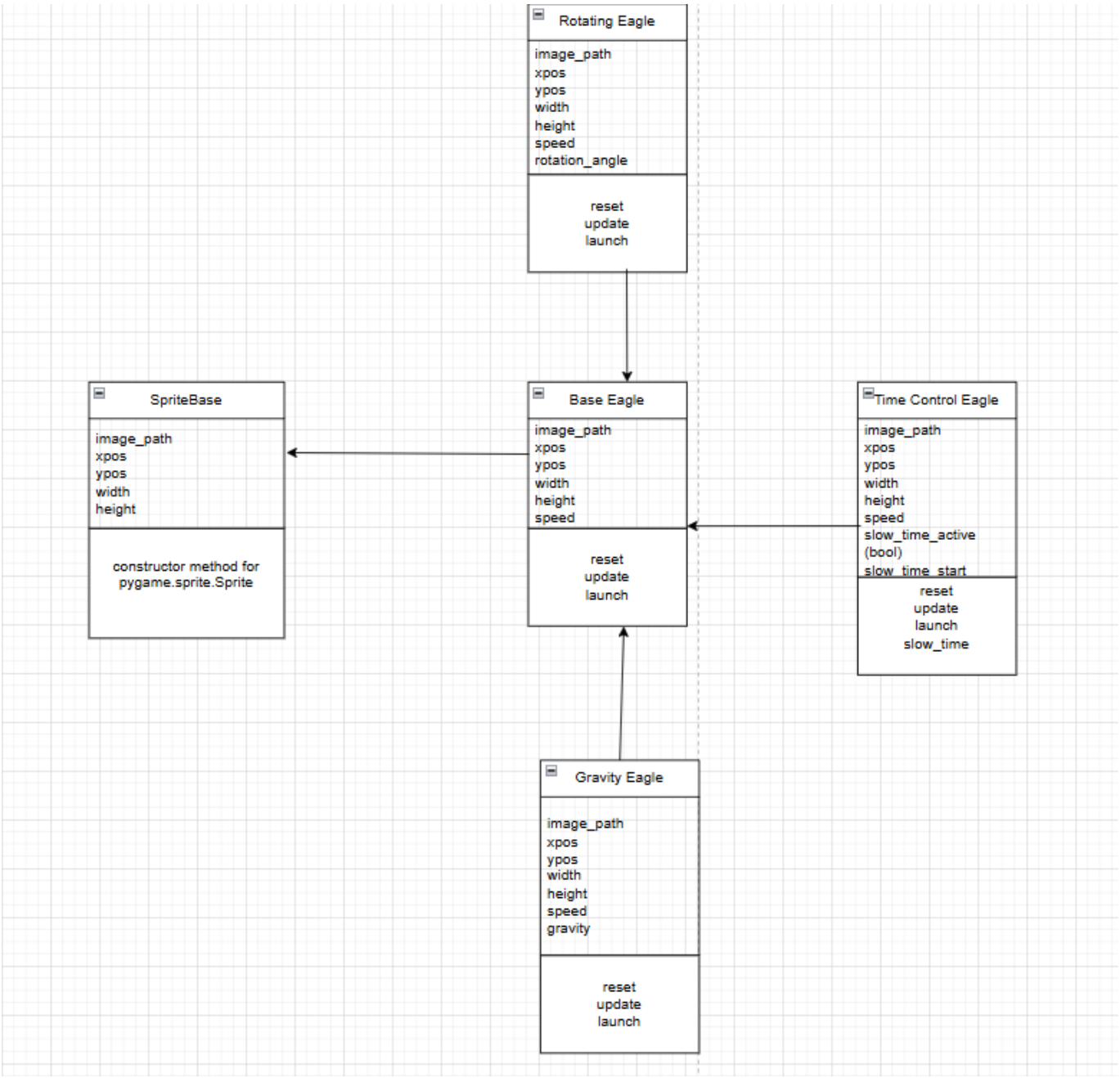


Fig. 2.17

Test Data to be Used During Development

The tables below show the test data split by iteration that will be used during development. Throughout the tables, an expected output of rejection is to be complimented by a relevant alert message.

Perhaps further test/design ideas will mean this list changes in my development. The updated version can be found throughout my development.

Iteration One – Core Mechanics

Test #	Description	Input Type	Expected Output / Behaviour	Reason for Inclusion
1	Press Space to jump	Valid	Player jumps vertically	Core platformer mechanic
2	Press Up Arrow or W to jump	Invalid	Input ignored	These are industry standard controls that users will plausibly try before trying the spacebar
3	Press @ to jump	Invalid	Input ignored	Random inputs should not have an effect
4	Player touches obstacle	Valid	Game over	Obstacle collision logic
5	Click 'Play' on GUI menu	Valid	Game starts successfully	Ensures GUI navigation launches gameplay correctly
6	Player moves above field of view	Boundary	Game over	Vertical boundary handling
7	Obstacle collides with ground or ceiling boundaries	Valid	Objects persist	Engine robustness; at this point in development obstacles should never de-spawn.
8	Click Quit button	Valid	System exits	Standard GUI exit behaviour
9	Click Options button	Valid	Returns alert: "Options will be added in future development"	Placeholders for future feature planning
10	Collision between user eagle and obstacle	Valid	User eagle de-spawns	Critical collision logic for game challenge
11	Collision between user eagle and ceiling or ground	Valid	User eagle de-spawns	Enforces game space boundaries

Iteration Two – GUI, Saving & Persistence

Test #	Description	Input Type	Expected Output / Behaviour	Reason for Inclusion
12	Save game using ‘k’	Valid	Game state saved	Provide save shortcut
13	Save game using ‘@’	Invalid	Input ignored	Symbol rejection

14	Save game using ‘Y’	Invalid	Input ignored	Handle incorrect inputs
15	Enter save code with [0–9]	Valid	Loads saved game	Accept valid save code
16	Enter save code with Space	Invalid	Input rejected	Save code must not include whitespace
17	Enter save code with letters only	Invalid	Input rejected	Only digits are allowed in save codes
18	Enter save code with mix of letters and numbers	Invalid	Input rejected	Save codes must only contain digits
19	Enter save code longer than 6 characters	Invalid	Input rejected	Save codes must be exactly 6 digits
20	Enter save code shorter than 6 characters	Invalid	Input rejected	Save codes must be exactly 6 digits
21	Enter correctly formatted code that does not exist	Boundary	Error message appears; no game loads	Ensure unused codes are handled gracefully
22	Enter valid existing save code	Valid	Game successfully loads	Confirms that saved state can be retrieved
23	Quit game using window X or Alt+F4	Valid	Game exits	Ensure standard and OS-level exit options work as expected
24	Score register prompt appears after clicking score	Valid	Pop-up shown	Score registration process
25	Score increments after passing obstacle	Valid	Score += 1	Gameplay progression logic
26	Enter Data into Leaderboard under unique name	Valid	Entry is saved	Tests basic leaderboard entry
27	Enter Data into Leaderboard under non unique name	Boundary	User is prompted to update or create duplicate	Ensures handling of duplicate names
28	Enter Unique but non alphabetical name	Invalid	Input rejected	Only alphanumeric names should be accepted
29	User enters an excessively long name	Boundary	Input rejected	Name length limit enforced
30	User overwrites leaderboard entry with larger score	Valid	Entry is updated	Only allows updates that improve score

31	User overwrites leaderboard entry with smaller score	Boundary	Entry remains unchanged	Prevents regression of user score
32	User declines to override score	Valid	New entry created with same name	Supports name reuse where desired
33	User enters score without name or name without score	Valid	Input rejected	Ensures complete entries
34	Scroll leaderboard	Valid	Leaderboard display updates	Supports usability of long scoreboards
35	User eagle collides with mask	Valid	Returns to main menu	Ensures core defeat condition is met

Iteration Three – Pathfinding Enemies, Physics & Advanced Logic

Test #	**Description**	**Input Type**	**Expected Output / Behaviour**	**Reason for Inclusion**
36	Switch character with keys [1–4]	Valid	Character changes	Character selection
37	Switch character with invalid keys: 0, 5	Boundary	Input ignored	Only 4 characters allowed
38	Repeated key press for character switch (same character)	Boundary	Handled gracefully; no change	Prevent unnecessary character state switching
39	ML enemy respawn on mask collision	Valid	Enemy respawns	Dynamic persistence
40	ML enemy spawns when score reaches 10	Valid	Enemy appears	Progressive difficulty
41	LOS enemy spawns pseudo-randomly	Valid	Enemy appears	Game unpredictability
42	Enter difficulty via text: -1, 11, 0.1, or non-integer string	Invalid	Input rejected	Ensure all improper difficulty values are safely rejected
43	Enter difficulty via scroll bar or text box: 0, 8, 10	Boundary	Accepted	Game starts at the selected difficulty
44	Use 'S' to slow time as Time Eagle	Valid	Game slows	Character ability
45	Time Eagle Slow Mode Duration	N/A	Valid	Ensure slow mode is time-limited
46	Time Eagle Manual Exit Prevention	Press 'S' again during slow mode	Invalid	Ensure user cannot prematurely exit slow mode
47	Time Eagle Cooldown	Attempt to activate slow mode before	Invalid	Ensure user waits 10 seconds before reuse

		cooldown ends		
48	Use 'T' or 'S' as non-Time Eagle	Invalid	No effect	Restrict slow mode to valid character
49	Enemy spawns every 10 points scored	Valid	New enemy appears at score milestones	Ensure enemy pacing increases with score
50	Difficulty meter controls A*, ML, and LOS enemy spawn frequency	Valid	A* is most common at medium difficulty, ML is most common at high difficulty, LOS is most common at low difficulty. Spawn chance is always non-zero. Constants will be calibrated through stakeholder input.	Balance enemy intensity with difficulty
51	Jump fatigue mechanic	Valid	Jump height/frequency reduces with repeated input	Ensure difficulty scales with user input behaviour
52	Air resistance mechanic	Valid	Player speed dampened over time	Simulates realistic drag effect
53	Randomised wind and rain	Valid	User eagle horizontal acceleration varies pseudo-randomly. Vertical acceleration varies sinusoidally.	Enhance realism and difficulty with weather effects
54	Difficulty meter GUI accepts and registers input	Valid	Input saved and difficulty modified	Verify correct GUI behaviour
55	AStar enemy pathfinding logic	Valid	Enemy takes shortest walkable path toward player	Verify correct use of A* algorithm
56	ML enemy pathfinding logic	Valid	Enemy path adapts to player pattern	Confirm AI learning and movement correctness
57	LOS enemy pathfinding logic	Valid	Enemy moves in straight line when in sight. Otherwise,	Confirm line-of-sight logic is functioning

			the enemy remains stationary.	
58	User eagle collides with A*, ML, or LOS enemy	Valid	User eagle de-spawns	Ensure collisions cause defeat regardless of enemy type
59	ML enemy collides with ground or ceiling	Valid	Enemy respawns in original location and retains learning	Ensure persistent behaviour across ML enemy lives
60	A* or LOS enemy collides with ground or ceiling	Valid	Enemy de-spawns	Remove enemies that exit bounds

Justification of Use of Libraries

1. Pygame:
   A. Provides tools for 2D mask collisions e.g. between obstacles and the user eagle. This is where images overlap.
   B. Handles user inputs e.g. spacebar to jump.
   C. Creates the Sprite parent class, and the surface used to render all on screen elements like the GUI
2. Random:
   A. Randomise obstacle behaviours, like the height of the pipes and the vertical distance between pairs of pipes.
3. JSON:
   A. Useful for saving and loading game data, such as high scores or user preferences and the list of all obstacles.
   B. JSON is lightweight and readable which makes it preferable to other methods of storing data persistently like CSV, since its easier to write and maintain.
4. Heapq:
   A. Heapq provides a lightweight and efficient implementation of a priority queue, which is essential for the A* Search Algorithm. This is because it allows the algorithm to quickly retrieve the node with the smallest heuristic cost (heuristicCost) from the openNodes set, ensuring optimal performance.
   B. HeapQ optimises the A* algorithm. Min-heap ensures that the node with the lowest estimated total cost (C+H) is always at the top of the queue. This minimizes the time complexity of retrieving and updating nodes, which is critical for real-time applications like *Elegant Eagles*.

During the Development stage, I may discover other necessary/more effective libraries, therefore, this list is not necessarily a complete list. This aims to demonstrate the need for libraries and the thought process behind choosing specific libraries for a given task.

Explanation and Justification of Data Structures and their Validation and Data Dictionary

I will need to validate lots of data inputted by the user.

I intend to create a save feature, so the user will need to input data. Perhaps the user will input a code, I will need to use **RegEx** to verify the length, characters, and legitimacy of the code. I will then need to instantiate objects using valid codes by taking data from a corresponding record of a **CSV** file.

Here is an example: my load algorithm flowchart.

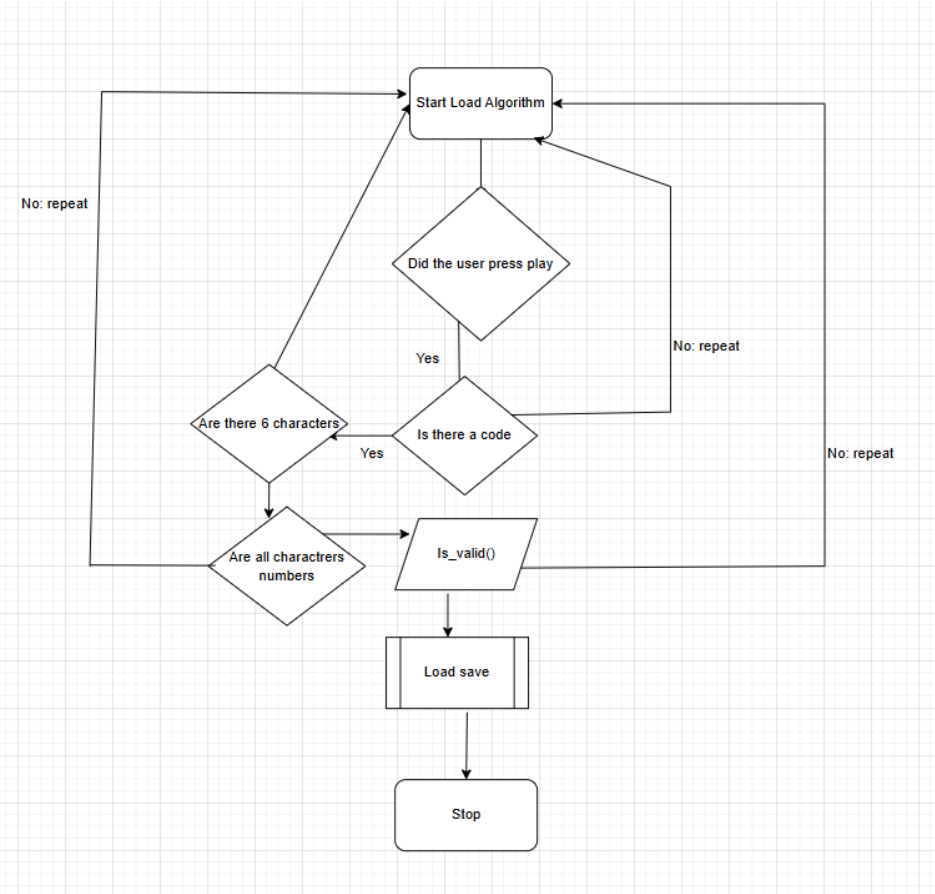


Fig. 2.18

The table below outlines the global variables, methods and objects that I plan to use in my project, along with the necessary validation required to handle this data. Each entry includes an explanation of the data's purpose in the game's functionality. Additional data items may be added during development as new requirements emerge.

This table will ensure that all critical validation is implemented, preventing unexpected error. This is exceptionally important given that the save-load feature will fabricate data that is supposedly copied

from a prior game instance. Validation will largely consist of Try Catch statements that return detailed error diagnostics to aid in debugging.

Name	Data Type	Explanation	Justification of Data Structure	Need for Validation	Justifying Validation
Leaderboard_Hash	**Hash Table**	Stores a mapping of player scores to player names.	**Allows fast lookups and sorting to display high scores efficiently.**	Ensure data integrity and avoid duplicate entries.	**Validate unique player ID enforce non-negative scores, and limit stored entries to a reasonable number. This ensures that leaderboard data remains accurate.**
Save_Structure_Hash	Hash Table	This allows objects to be instantiated from saved data efficiently.	Hash Tables are O(1) and therefore means that more queries can be made per second.	Ensure correct hashing to prevent collisions and corruption.	**Ensure correct hashing to prevent collisions and corruption. Verifying unique keys maintain**
Undo_Redo_LinkedList	**Circular Doubly Linked List**	Used for storing ordered game events to create the undo and redo feature.	**Stores the previous game state and the proceding game state**	1. Handle Edge Cases • **If the list is empty then no validation is needed.** • If the list has only one node (length = 1): ○ The node should point to itself **2. General Validation for Lists with Multiple Nodes (length > 1)** For each node in the list, check: • Each node must have a **non-null** next and prev pointer. • Maintain a **count** of how many times each node is referenced by another node's next and prev pointers.	Ensure head and tail pointers are correctly assigned, check for cyclic references, and validate proper node connections. This prevents data corruption.

				3. Ensuring a Single Circuit Covers the Entire List • The **last node to reach a reference count of 2** must be the **first node**. • The **first node to reach a reference count of 3** must also be the **first node**. • If these conditions are violated, the list is either not circular or has multiple disjoint cycles. **4. Detecting Cycles Efficiently (Floyd's Cycle Detection Algorithm)** • Use two pointers: o **Slow Pointer:** Moves **one** step at a time. o **Fast Pointer:** Moves **two** steps at a time. • If the list is **correctly circular**, the fast pointer will eventually catch up to the slow pointer **at the first node**. • If the fast pointer ever becomes null, the list is **not circular**. • To confirm the entire structure is correct: o Ensure the slow pointer	

				traverses each unique node at least once before returning success. ○ Predict when the fast pointer will pass the slow pointer to avoid infinite loops.	
Player_Eagle	Game Object	The object template that contains all data about the Player Eagle and its methods.	Multiple classes will need to reference the Player Object. For example, the enemy class will need to path find towards the player. Therefore, this data is most effectively stored in a class. It is also a parent class to similar child classes. Inheriting classes improves readability maintainability and stability since it reduces the amount of	• Direction constraint: Velocity.x>0 • Speed limit to prevent unrealistic movement: 0<abs(velocity • Ensure User Eagle is in suitable region: minX<coord.x<maxX minY<coord.y<maxY • Ensure dimensions are feasible; that is, less than the initial dimensions. • Ensure User Eagle does not spawn on an obstacle and therefore lose instantly: Compare all coordinate pairs in the obstacle_list (below in this	Ensure coordinates remain within bounds, validate velocity constraints, check against obstacle list to prevent immediate collision, and ensure all attributes are initialized. Ensuring that proper attributes are used prevents gameplay inconsistencies.

			code written and the code is logically connected throughout.	table) to those of the PlayerEagle, ensuring no co-incidence. Data Type Validation: Coordinates are floats Velocity is a vector ImagePath is an image in the list of images (below in this table) Float angle of rotation. Positice float dimensions. • Null cases; ensure all attributes are given: Ensure that both coordinates are given Ensure that velocity is given etc.	
ScoreKeeper	Variable	It is an integer counter that increments based on the number of obstacles passed.	It lets the program keep track of the user score. It also contains the text that the ScoreBoard (below) displays.	• Valid Data Type: Integer • Valid range: 999>Score>0	Validate integer type, prevent negative scores, and implement an upper limit to avoid overflow errors.
ScoreBoard	Game Object	It is the object that controls the physical score board on the user's screen.	It allows specific control of the text. For example, controlling a high-score message.	• High-score validation as above. • Message.length< 100; ensures legibility.	Ensure score updates in real-time, restrict message length for readability, and prevent special characters that may cause UI issues. This prevents scoring anomalies.
Gravity	Variable	It is the variable that holds the	Fundamental value that controls how	• Reasonable range to avoid erratic behaviour:	Validate gravity within set boundaries

		magnitude of the downwards acceleration the user experiences. It varies with difficulty the user chooses.	the User-Eagle will move. By making gravity variable rather than a constant, difficulty is variable, this creates a customisable challenge for all experiences.	0<gravity<x Note that x is a constant to be determined based on Stakeholder feedback from using a prototype since, real gravitational constants are based on metres per second squared. • Data Type: Float	(determined by testing and stakeholder feedback), ensure it's a float type, and avoid zero gravity for playability. This ensures UI readability. Keeping gravity in range prevents erratic movement.
ObstacleList	Object List	This stores a mutable dynamic structure of all obstacle objects.	This improves readability by storing all objects together. The iterability lets us keep track of and delete unneeded objects. Moreover, this is crucial for scalability and maintainability since additional objects can be appended.	Validation is complete from individual object validation.	Ensure all obstacles have valid attributes (position, size, type), prevent duplicates, and handle dynamic updates correctly. This prevents game-breaking errors.
EnemyList	List	This stores a mutable dynamic structure of all enemy objects.	As above	Validation inherited from individual enemy objects.	Ensure unique enemy instances, verify attributes exist, and prevent out-of-bounds movement.
Lives	Int	This stores the number of lives left.	This is an essential counter to keep track of	• Data Type: Integer • Appropriate range:	Enforce non-negative values, cap at max_lives, and ensure proper decrement

			the number of lives left.	0<=Lives<=Max_lives	behaviour. This ensures that users can lose the game.
HighScore	Int	Stores The highest recorded score.	As above	Ensure integrity and validity.	Validate integer type, enforce positive values, and prevent overflow or tampering. This ensures fair competition.
JumpForce	Float	As above	As above	• Data Type: Integer • Appropriate range: To be determined by stakeholder review during development using prototype.	Verify non-zero positive values, cap at a reasonable maximum, and prevent extreme jumps. This ensures that motion is realistic.
EnemyLOS	Game Object	It is the object that creates a working LOS enemy on the user's screen.	The class allows the LOS enemy template to be set up in few lines of code. It contains unique attributes like the search method for the user eagle. **Using a list of objects also lets the program efficiently move between user eagle characters.** Inheriting classes improves readability, maintainability and stability	• All validation from the User EagleGame object. • Possible x coordinate: 0<EnemyLOSCoord.x<UserEagleCoord.x This ensures that the LOS enemy has not somehow passed the user. This would be impossible since the LOS enemy can only travel rightwards since it is always on the left of the user eagle which can only travel rightwards. EnemyLOSCoord.x>0 means being on screen. If this is false, the object is to be deleted since once off screen,	Ensure LOS enemy only moves in valid directions, verify pathfinding logic, and restrict the number of active LOS enemies. This ensures AI behaves correctly.

			since it reduces the amount of code written and the code is logically connected throughout.	the LOS Enemy is deleted. • Unit vector(LOS Enemy, User Eagle): As explained, this is the direction that the LOS enemy will travel This vector must be non 0 and have a positive x component. If Unit vector is computed from the x coordinates, then this is implied by the previous validation step. • **Number of LOS Enemies is less than the limit.** This number can be queried via a linear search of the enemy object list.	
EnemyML	As above			As above and in addition • Ensure that the ML enemy is beatable: The age attribute of the ML enemy is below a time to be determined with the stakeholders by experimenting with a prototype. • Ensure that the ML enemy will not learn too fast: **R value of Non-Linear Function should not exceed a maximum. The lower the R value, the less**	Limit learning rate, cap age attribute, and prevent overfitting. This maintains balanced difficulty. This maintains balanced difficulty.

				accurate the function.	
EnemyAStar	Game Object	As above		• Correct heuristic: Euclidean Otherwise, the AStar Enemy might have a very high or low EECR	**Enforce Euclidean heuristic; check path validity e.g. on screen; and prevent looping or infinite searches.** This ensures smooth enemy movement.
maxSpeed	float	This is the max possible user eagle speed. This a result of applying the .launch() method on cooldown	Without max speed, the user eagle may start with high speed which causes instant loss.	• Data Type Float • Acceptable range maxSpeed>0	Validate speed is positive and within a set range. This prevents uncontrollable movement.
LOSEnemy User Vector	Vector	This is the unit vector that maps the LOS Enemy to the User.	This is the path that the LOSEnemy will travel on.	• Vector x component must be positive	Enforce a positive x-component and non-zero magnitude. This ensures that the enemy moves correctly.
AStar TargetNode	Vector	This is the unit vector of the user eagle.	A goal is required for the A* algorithm to traverse.	• Coordinates must be in playing region; i.e. minX<coord.x<maxX minY<coord.y<maxY	Validate coordinates are within minX, maxX, minY, maxY. This ensures AI navigates correctly.
GCost	Float	This is the approximated distance between two nodes.	This is a required component for the A* algorithm to traverse.	• GCost>0	GCost must be greater than zero and finite. Preventing negative values ensures valid pathfinding.
HCost	Float	This is the Heuristic approximated distance between the current node and the goal node.	This is a required component for the A* algorithm to traverse.	As above	Validate positive values and heuristic function. This ensures that the path taken is of the correct efficacy to control the EECR.
CCost	Float	This is the distance between the start node	This is a required component for the A*	As above	Ensure positive and finite values. This ensures correct path cost calculations.

		and the end node.	algorithm to traverse.		
LosEnemy Detection Radius	Float	This is the maximum distance from the LOS Enemy that the user eagle can be 'seen'inside of. Outside of this region (a circle), the LOS enemy is inactive.	This is a required attribute of the LOS enemy object	As above	Validate positive, finite radius, and prevent excessive range. This maintains balanced detection.
RaycastHit	Bool	This is a boolean variable that controls if there is an obstacle in the way of the LOS Enemy and the User Eagle .	This is a necessary condition to check if the LOS enemy is allowed to move or not.	• Data Type: Bool	Validate Boolean type and ensure proper collision detection logic. This is essential for making the enemy surmountable.
Collided	Bool	This is a boolean variable that all non ML Enemy class objects hold as an attribute.	This is a necessary conidition to control on screen element object de-spawning. WIthout it, the user couldn't evade obstacles and enemies.	• Data Type: Bool	Validate Boolean type and proper collision checks. This makes sure that the enemy despawns correctly.
JumpKey	Charact er	The key that the user must press to jump	Useful for program clarity	• Data Type: Character	Restrict to single characters, disallow special characters, and check for key binding conflicts. This ensures proper input handling.
SaveKey	Charact er		As above	• Data Type: Character	Validate against reserved keys and

					ensure correct assignment so that the save button works.
QuitKey	Character		As above	• Data Type: Character	Restrict to allowed keys and prevent accidental assignment of important controls so that the quit button works.
Character Change Key	Character		As above	• Data Type: Character	Validate against duplicate key bindings.
PauseScreen	GameObject	This is the object that controls the pause screen.	This makes controlling the on screen objects easier, since all of the data about the pause-screen is contained in one object.		Validate toggle functionality and prevent unintentional activation.
SplashScreen	GameObject	As Above			Ensure correct timing and transitions.
ImageList	List	The is the list of imagepaths and their identifiers that the program will use.	This makes using images easier, since instead of having file paths throughout, we refer to english-like image names. This makes the program more readable and scalable		Check for missing files, valid image formats, and correct identifiers.
CreateLogin	String	User inputs data and it is saved to this string to create a login.	This is necessary data to store to create a login.	• **RegEx:, specifically:** ^a-zA-Z0-9{2,29}$ • 0 spaces • Alphabetical characters only	**Enforce RegEx constraints, prevent spaces, restrict length,** and disallow special characters. This ensures valid usernames are chosen**.**

				• 2<Login.length()=<30	
Login Database	2D Array	Database queried by pairs of logins and passwords. If the pair is in the database, then data such as highscore is returned, otherwise an error is returned.	This is a required validation step to test login details. Without a persistent store of data, valid logins would not load any user specific data.	• **First Normal Form** • Blacklist reserve words so that SQL can be used e.g. 'null'; prevents **SQL injection**	**Enforce first normal form, blacklist SQL keywords, and prevent injection attacks.** This ensures security.
Difficulty	Integer	User inputs difficulty and it is saved to this integer to create a difficulty.	This is necessary data to store to vary the game's difficulty.	• **RegEx, specifically:** ^\d+(\.\d+)?$ Ensures float Within acceptable range e.g. 0<difficulty<10 Note that whereas elsewhere the constants were to be determined with a prototype and stakeholder feedback; here, it is the relative impact of the difficulty that is to be determined during development by stakeholder feedback.	Validate integer type, enforce positive values, and confirm within an acceptable range based on playtesting. This ensures that the extent of challenge is desirable.
Create Password	String	User inputs data and it is saved to this string to create a password.	This is necessary data to store to create a password.	• **RegEx, specifically:** ^(?=*.[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%?&])[A-Za-*	**Enforce RegEx constraints;** require a mix of character types; and blacklist common passwords from public surveys. This prevents weak passwords. Therefore improving security.

				*z\d@$!%*?&]{8,}$ • Sufficient complexity e.g. 1 number, 1 capital letter, 8 character minimum. • No common passwords like 'password'. The bank of 'common passwords' can be found here.	

Development Plan

I will work in development phases akin to those of the Agile Model.

Each cycle consists of Four Stages:

1. Objectives: Concise modular breakdown of intended implementation for the given phase. This depends on the modules from the design stage and any previous stakeholder feedback.
2. Implementation: Executing the objectives in the program
3. Testing: Ensuring the implementation works as intended including edge/erroneous/normal cases.
4. Evaluation: Verifying if the implementation meets the objectives and finding improvements based on detailed stakeholder feedback. All improvements will be carried out in the subsequent development stage. In the evaluation stage, my stakeholders and I will determine relevant arbitrary mathematical constants through the following process:

    A. Try the coefficient
    B. Through democratic feedback and discussion, we voted to either, increase, settle, or decrease the value.
    C. If the output of step B Is not 'settle', go to step A.

It is likely that even within the development phases, micro cycles will exist i.e. repeating steps two to four. Therefore, these cycles will be very short. This enables flexibility and a quick turnover from theoretical to tangible code and said code's testing. This flexibility will be instrumental in changing criteria e.g. for overly ambitious aspects or more effective methods.

Whilst impromptu chats with Stakeholders will occur throughout development, I've organised sit-down-sessions to test, hands-on, the game, so that improvements on the tangible product can be made where impossible theoretically. Notably, this development plan itself was subject to rigorous

scrutiny from my stakeholders James and Oliver e.g. specifying the nature of my semi-terminal testing.

DEVELOPMENT

Agile Iteration One

Using the interview discussion and recent discussion with my stakeholders James and Oliver, we have decided that the following is a comprehensive list of objectives.

Objectives

1. Create a simple GUI

2. Create Terrain
    (i) Tangible Circular Scrolling Window Floor

(ii) Background

3. Create base object classes

4. Create the User Eagle

4. Enable the User Eagle to jump

5. Set up simple vertical slit obstacles

6. Handle Mask Collision with the User Eagle

GUI

A GUI is necessary because it lets the user customise their experience and start the game.

I designed a preliminary GUI to learn how to create buttons in PyGame. Pygame is a library who's function I analyse in my Design stage.

```
def main_menu_new():
    pygame.init()


    pygame.display.set_caption('Quick Start')

    window_surface = pygame.display.set_mode((800, 600))


    background = pygame.Surface((800, 600))

    background.fill(pygame.Color('#000000'))


    manager = pygame_gui.UIManager((800, 600))


    hello_button = pygame_gui.elements.UIButton(relative_rect=pygame.Rect((350, 275), (100, 50)),
                                                text='Play Now!',
                                                manager=manager)


    clock = pygame.time.Clock()

    is_running = True
```

Fig. 3.1

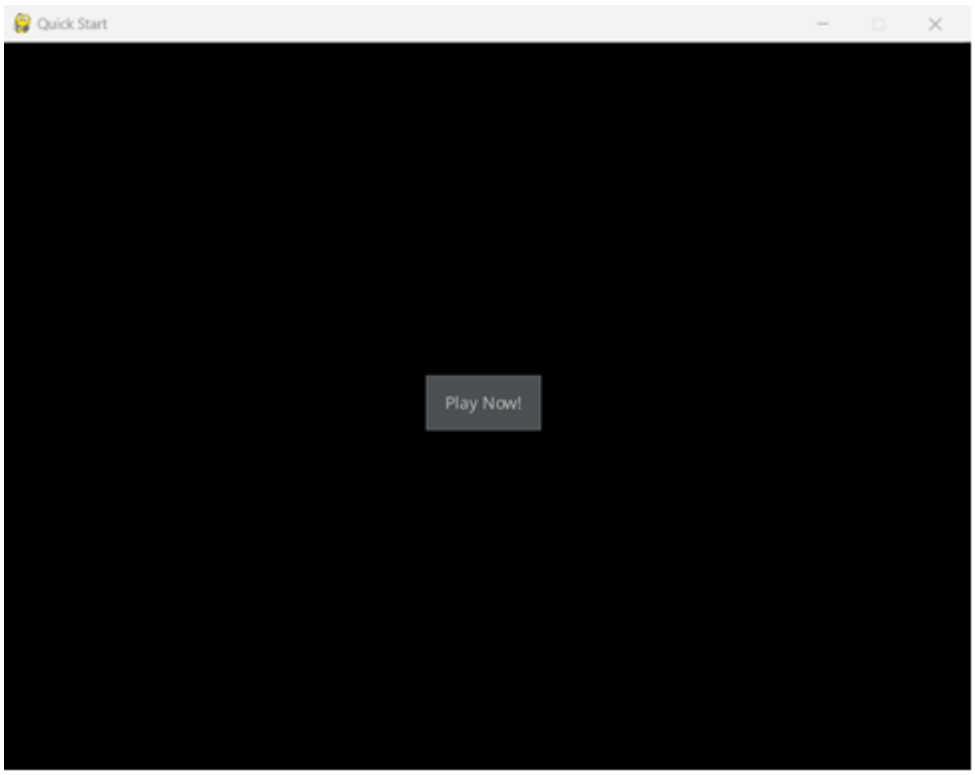


Fig 3.2

I then added a more visually appealing graphic in accordance with my design criteria. I considered reusing the GUI from my design because I really like the rounded buttons. Otherwise, I would need to import several images, which would make the image list long. This would be an issue because that would make finding the image I want more difficult, increasing the chance of error, contradicting the list's aforementioned purpose.

However, I concluded that the use of a jpg made the GUI experience static. Instead, I could detect the user's mouse location to highlight buttons when they were being hovered over. If I were to create a dynamic GUI using images, I would also need to use multiple images.

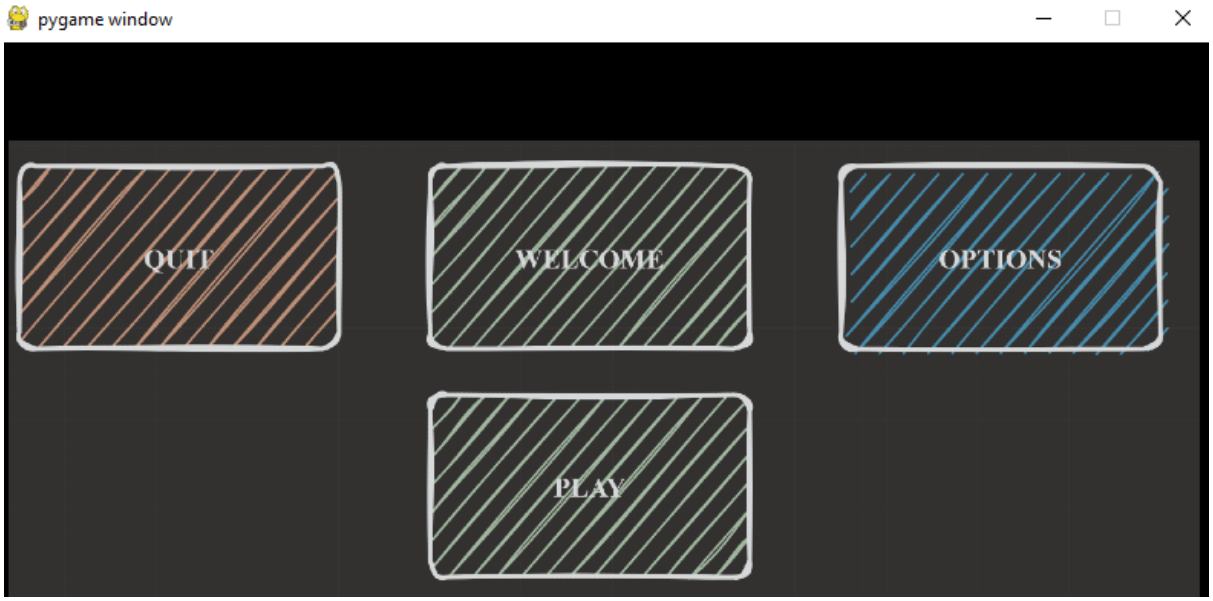


Fig. 3.3

```
<untitled> *   <untitled> *
1  import pygame
```

Fig. 3.4

I decided to write this program in its own page and in a class so that I could test components modularly. This also makes reusing components easier, since I can now find components using labels/page titles.

```
class ButtonInterface:
    def __init__(self, button_actions):
        pygame.init()
        self.screen = pygame.display.set_mode((800, 600))
        pygame.display.set_caption('Interactive Buttons')

        self.buttons = [
            pygame.Rect(100, 150, 200, 80),  # Button 1
            pygame.Rect(100, 250, 200, 80),  # Button 2
            pygame.Rect(100, 350, 200, 80),  # Button 3
            pygame.Rect(100, 450, 200, 80)   # Button 4
        ]

        self.button_labels = {1: "Quit", 2: "Welcome", 3: "Options", 4: "Play"}

        #Store the functions to run on button press
        self.button_actions = button_actions

        self.clock = pygame.time.Clock()
        self.running = True

```

Fig. 3.5

I set up the screen on which the buttons will be printed. Then I set default values. Crucially, I made use of Pygame's clock feature to allow the GUI to 'dynamic', since, we can now track how much time has passed since an event.

```
    def render_button(self, rect, label, font, base_color, hover_color, surface, mouse_position):
        #Change button color if hovered
        if rect.collidepoint(mouse_position):
            pygame.draw.rect(surface, hover_color, rect)
        else:
            pygame.draw.rect(surface, base_color, rect)

        text_surface = font.render(label, True, (0, 0, 0))  #Black
        text_rect = text_surface.get_rect(center=rect.center)
        surface.blit(text_surface, text_rect)
```

Fig. 3.6

This method controls 'creating' the button. The method 'blits' the button onto the screen.

This is the 'dynamic' feature of the GUI. When the user hovers their cursor over a button, it should change colours. This tells the user where their cursor is and makes the game feel more alive; reactive and energetic.

```
    def handle_button_click(self, button_id):
        print(f"Button {button_id} was clicked!")
        # Update the button label to indicate it was clicked
        self.button_labels[button_id] = "Clicked!"

        # Run the associated function for the button
        if button_id in self.button_actions:
            self.button_actions[button_id]()
```

Fig. 3.7

Next, this method handles the mechanism of the button. It calls the function passed in as an attribute to the class.

```
    def start(self):
        font = pygame.font.Font(None, 36)  ext
        button_base_color = (100, 200, 100)
        button_hover_color = (50, 150, 50)

        while self.running:
            time_elapsed = self.clock.tick(60) / 1000.0
            mouse_position = pygame.mouse.get_pos()  # Track mouse position

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.running = False

                if event.type == pygame.MOUSEBUTTONDOWN:
                    #Check if a button was clicked
                    for index, button_rect in enumerate(self.buttons, start=1):
                        if button_rect.collidepoint(event.pos):
                            self.handle_button_click(index)

            self.screen.fill((240, 240, 240))

            #Draws the buttonss
            for index, button_rect in enumerate(self.buttons, start=1):
                self.render_button(
                    button_rect,
                    self.button_labels[index],
                    font,
                    button_base_color,
                    button_hover_color,
                    self.screen,
                    mouse_position
                )

            pygame.display.flip()
```

Fig. 3.8

This method brings it all together. It tracks the time passed and the cursor location, to make the button colour change on-hover.

It gives the button colours and fonts that can always be changed based on stakeholder feedback. This method deals with all buttons simultaneously.

At this point I considered writing the class for each button, such that we would end up with a list of objects. However, given that the number of buttons/position of buttons is constant, I decided against this. Therefore, this would make the program confusing unnecessarily.

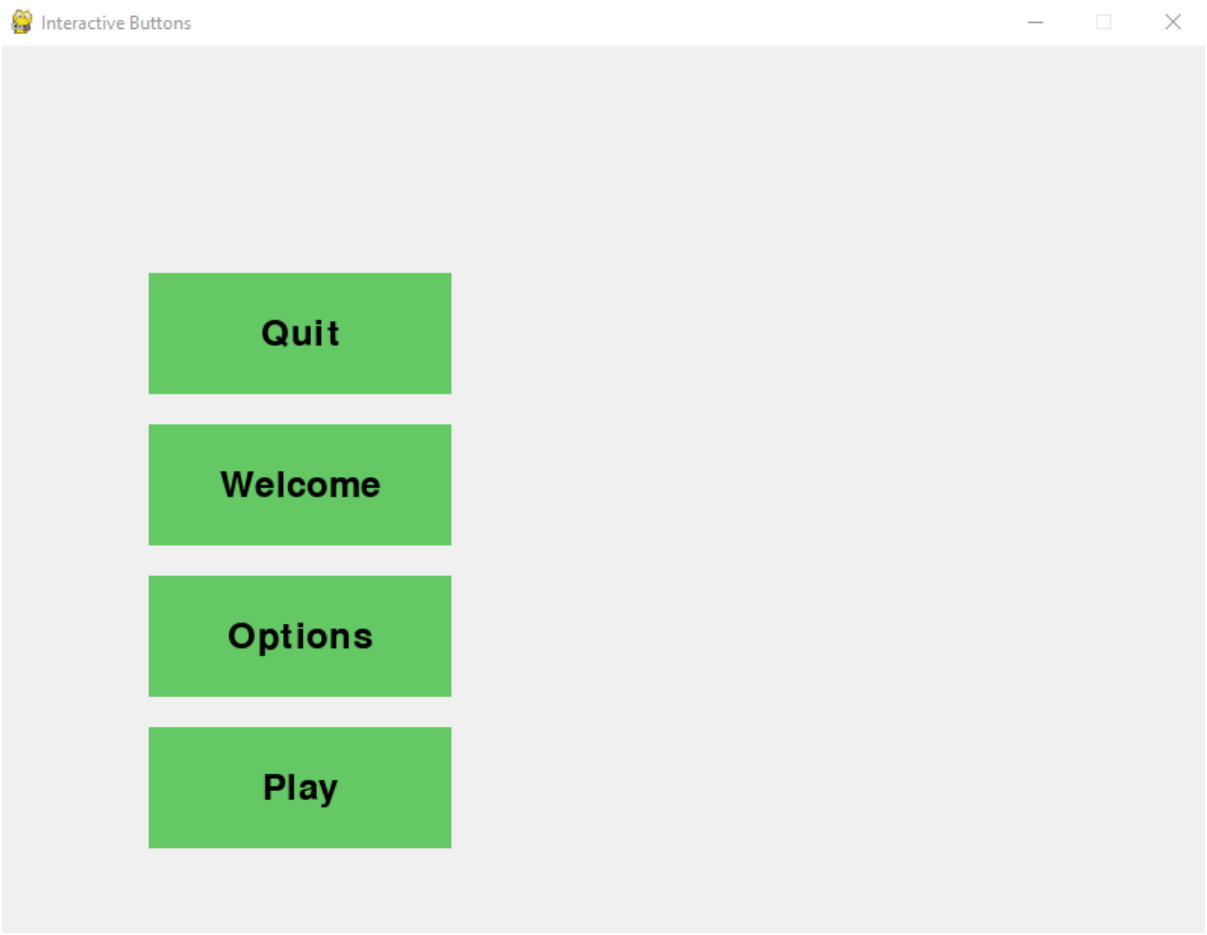


Fig. 3.9

This is the result. I will reuse this object class to make sub-menus in future development iterations. This is consistent with my plan from the design stage.

PROBLEM:

At first the dimensions of the window were constant, such that the windows were too large on some devices, such that they couldn't see the whole game.

```
pygame.init()
screen = pygame.display.set_mode((SCREEN_WIDHT, SCREEN_HEIGHT))
pygame.display.set_caption('Elegant Eagles')
```

Fig. 3.10

SOLUTION:

So, I made Elegant Eagles compatible with more devices by varying the window's dimensions with the user's device.

```
87  infoObject = pygame.display.Info()
88  SCREEN_width=infoObject.current_w/2
89  SCREEN_height=infoObject.current_h/2
```

Fig. 3.11

This gets the user's 'display.info' broken down into x, y dimensions. Then line 112 creates a window of ¼ the area of the user's display.

PROBLEM:

```
SCREEN_HEIGHT
```

Fig. 3.12

Inconsistent identifiers

SOLUTION:

Use my IDE's search and replace feature

PROBLEM:

I ran into an unexpected error when trying to install the settings library on my IDE, Thonny. My solution was to use several other libraries instead.

```
C:\Users\user\OneDrive - Bromsgrove School\Documents\Python>pip install python-settings
WARNING: Ignoring invalid distribution -ip (c:\users\user\appdata\local\programs\thonny\lib\site-packages)
Requirement already satisfied: python-settings in c:\users\user\appdata\local\programs\thonny\lib\site-packages (0.2.2)
WARNING: Error parsing dependencies of send2trash: Expected matching RIGHT_PARENTHESIS for LEFT_PARENTHESIS, after versi
on specifier
    sys-platform (=="darwin") ; extra == 'objc'
                 ~^
WARNING: Ignoring invalid distribution -ip (c:\users\user\appdata\local\programs\thonny\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\users\user\appdata\local\programs\thonny\lib\site-packages)
```

Fig. 3.13

<untitled> *

```
1  import pygame,sys,time,random,thorpy
2  try:
3      from sprites.abc import BG,Ground,Plane,Obstacle
4  except ImportError:
5      from sprites import BG,Ground,Plane,Obstacle
6  #https://stackoverflow.com/questions/69381312/importerror-cannot-import-name-from-collections-using-python-3-10
7
8
```

Fig 3.14

PROBLEM: Additionally, when using my personal device, I ran into issues importing from the spritesmodule (perhaps due to using MacOS).

```
import pygame,sys,time,random,thorpy,sprites
try:
    from sprites import Ground,Plane,Obstacle7
except ImportError:
    print('error')

#    from sprites.abc import Ground,Plane,Obstacle
#except ImportError:


#https://stackoverflow.com/questions/69381312/importerror-cannot-import-name-from-collections-using-python-3-10



class main:
    def __init__(self):
        #constructor method
        pygame.init()
        self.display_surface=pygame.display.set_mode((Display_x, DIsplay_y))
        self.clock=pygame.time.Clock()

    def play(self):
```

```
ell ×

> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<string>", line 8
    class main:
    ^^^^^
```

Fig 3.15

SOLUTION: I employed try/except to make *Elegant Eagles* compatible on more devices.

PROBLEM:

Thonny couldn't find the images I was using.

```
>>> %Run 'experiment code 20.11'

  pygame-ce 2.5.2 (SDL 2.30.8, Python 3.10.6)
  Traceback (most recent call last):
    File "C:\Users\user\OneDrive - Bromsgrove School\Documents\experiment code 20.11", line 195, in <module>
      bird = Bird()
    File "C:\Users\user\OneDrive - Bromsgrove School\Documents\experiment code 20.11", line 107, in __init__
      self.images = [pygame.image.load('skypic.').convert_alpha(),
  FileNotFoundError: No file 'skypic.' found in working directory 'C:\Users\user\OneDrive - Bromsgrove School\Documents'.
```

Fig. 3.16

SOLUTION:

This was solved by saving them to the same folder.

PROBLEM:

Loading many images caused a noticeable lag time in the program's compilation time.

SOLUTION:

I will import images with .covertalpha because it is more inefficient in terms of processor run time.

Login

Creating Terrain

Next, I need to create a 'map' for the user to traverse. The map is simple, but will be randomly generated subject to parameters like difficulty. Generating rather than creating the map saves development time and lets me focus on the game's functionality. Moreover, it improves the user experience since they have unlimited content.

PROBLEM:

```
BACKGROUND = pygame.image.load('Sky.jpg')
BACKGROUND = pygame.transform.scale(BACKGROUND, (SCREEN_width, SCREEN_height))
```

Fig. 3.17

Elegant eagles makes use of lots of images. At first I had to import each time. This caused syntax errors due to typos and made the program less readable.

```
FileNotFoundError: No file 'assets/sprites/bluebird-downflap.png' found in working directory 'C:\Users\user\OneDrive - Bromsgrove School\Documents'.
```

Fig. 3.18

SOLUTION:

I made a list of images that I could call whenever I wanted to use them.

```
self.images = [pygame.image.load('sky.jpg').convert_alpha(),
               pygame.image.load('green block.png').convert_alpha(),
               pygame.image.load('user_image.png').convert_alpha(),pygame.image.load('eagle.jpg').convert_alpha()]
self.image=self.images[2]
```

Fig. 3.19

PROBLEM:

```
20
21 images = [pygame.image.load('sky.jpg').convert_alpha(),
22                         pygame.image.load('green block.png').convert_alpha(),
23                         pygame.image.load('user_image.png').convert_alpha()]
24
25 GROUND_width = 2 * SCREEN_width
26 GROUND_HEIGHT = 100
27
28 PIPE_width = 80
29 PIPE_HEIGHT = 500
30 PIPE GAP = 150
```

```
Shell
>>> %Run -c $EDITOR_CONTENT
  pygame 2.6.1 (SDL 2.28.4, Python 3.10.6)
  Hello from the pygame community. https://www.pygame.org/contribute.html
  error
  Traceback (most recent call last):
    File "<string>", line 21, in <module>
  pygame.error: No video mode has been set
>>>
```

Fig. 3.20

SOLUTION:

Setting video mode first

Inheritance; Polymorphism; Scrolling Window

CIRCULAR SCROLLING WINDOW BACKGROUND

Instead of generating the background, I decided on making the background circular to save memory. This makes sense because the background only serves to give the user bearings of the 'safe-zone', that is, where they should stay.

```
class Ground(pygame.sprite.Sprite):

    def __init__(self, xpos):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load('grass.jpg').convert_alpha()
        self.image = pygame.transform.scale(self.image, (GROUND_width, GROUND_HEIGHT))

        self.mask = pygame.mask.from_surface(self.image)

        self.rect = self.image.get_rect()
        self.rect[0] = xpos
        self.rect[1] = SCREEN_height - GROUND_HEIGHT

    def update(self):
        self.rect[0] -= GAME_SPEED
```

Fig. 3.21

Throughout my project, classes will inherit the sprite class from pygame because it enables on-screen entities to have attributes, for example, coordinates, colours, dimensions.

```
34 # Base class for all sprites
35 class SpriteBase(pygame.sprite.Sprite):
36     def __init__(self, image_path, xpos, ypos, width, height):
37         super().__init__()
38         self.image = pygame.image.load(image_path).convert_alpha()
39         self.image = pygame.transform.scale(self.image, (width, height))
40         self.mask = pygame.mask.from_surface(self.image)
41         self.rect = self.image.get_rect()
42         self.rect[0] = xpos
43         self.rect[1] = ypos
```

Fig. 3.22

The major change required for the Ground class, is the 'update()' method which makes the 'scrolling background' scroll.

PROBLEM:

I got the following error when attempting to load the background.

```
Pygame.error: surface is not initialised
176     screen.blit(BACKGROUND, (0, 0))
177     #screen.blit(images[2], (120, 150))
```

Fig 3.23

SOLUTION:

The solution itself was a simple line of code:

```
screen=pygame.display.set_mode((infoObject.current_w/2, infoObject.current_h/2))
pygame.display.set_caption('Elegant Eagles')
```

To figure this out, I read through Stack-Exchange posts and official Pygame documentation regarding initialising surface. I reference these in the appendix.

The set of all displays is a subset of the set of all surfaces, but not all displays are surfaces. So, I was creating a display but not a surface to .blit() images onto.

All displays are surfaces, but not all surfaces are displays. So, when I made

Mathematical modelling:

Jumping and Obstacle Generation

So that I don't waste computational resources, I linked the random generation of obstacle dimensions to the user's jumping, since, the user must jump more than once per obstacle they overcome.

```
while play==True:

    clock.tick(15)

    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
        if event.type == KEYDOWN:
            if event.key == K_SPACE or event.key == K_UP:
                bird.launch()
                play = False
                SPEED = SPEED*1.2
                GAME_SPEED = GAME_SPEED*1
                PIPE_width = PIPE_width=random.randint(20,PIPE_width*2)
                PIPE_HEIGHT = 700
                PIPE_GAP = 300*0.95
```

Fig. 3.24

This shows that the difficulty decreases exponentially.

An impromptu chat with my stakeholder James made me realise there is a major flaw to this. After a certain amount of time, the pipes will be too small for the user to pass through.

In development iteration 3 I will implement a ceiling to the difficulty such that it's still possible to win the game.

```
def launch(self):
    self.speed = -SPEED
```

Fig. 3.24.1

I intend on developing this launch algorithm to more accurately mimic real physics in future development iterations.

The use of .launch() creates a unique style of movement to many games. 'It makes the game feel exciting, volatile and dangerous' as my stakeholder James put it.

Additionally, I made the game increase in difficulty exponentially with the number of jumps by multiplying the obstacle parameters by some constant scale factor every iteration. For example, the gap decreasing by a constant scale factor 0.95.

After reviewing with my stakeholder James, using the detailed methodology laid out in Design: Development Plan, we decided that 0.95 was too difficult since, sketching

y=width0·0.95n(jumps)y=width0·0.95njumps

Fig. 3.24.2

Will quickly become smaller than the size of *user_image* no matter the size.

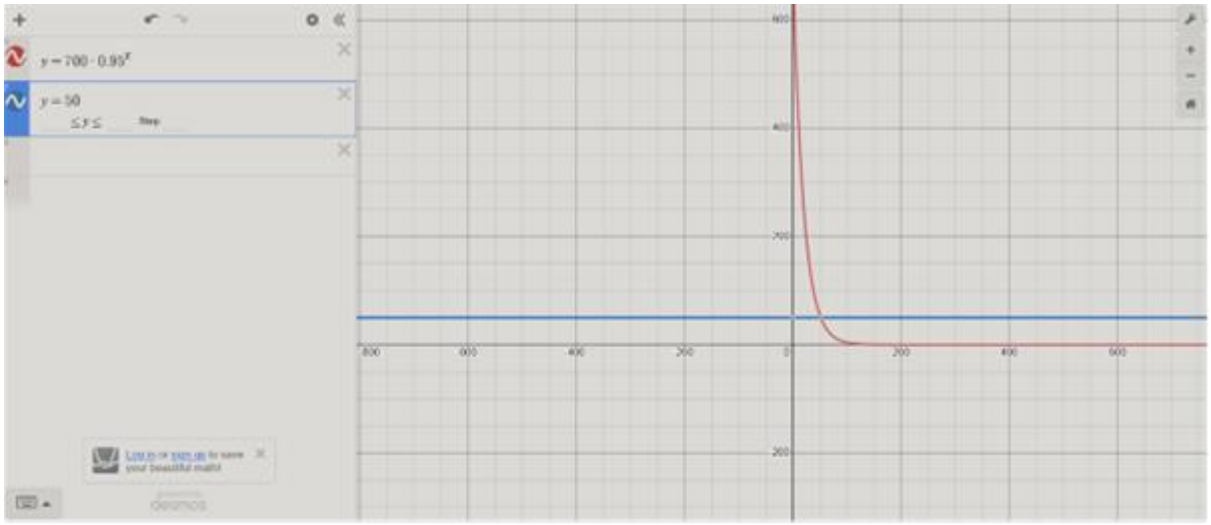


Fig. 3.24.3

PROBLEM: To change the animated state of the user, I was using a separate clock at first, this is inefficient.

SOLUTION: Instead, I utilised modular arithmetic as explained in the comments below. This function is inside the outer class, so it increments consistently with the rest of the program – preventing what my stakeholder James described as 'lag' in some informal testing.

```
def update_position(self):
    self.position=(self.position+1)%3 #Using the periodicity of modular arithmetic w.r.t. addition
```

Fig. 3.25

PROBLEM: After about 20 frames, the program crashed.

```
168     while play==True:
169         print('c')
170         clock.tick(15)
171         #Controling jumps
172         for event in pygame.event.get():
173             if event.type == QUIT:
174                 pygame.quit()
175             if event.type == KEYDOWN:
176                 if event.key == K_SPACE or event.key == K_UP:
177                     bird.launch()
178                     play = False
179                     SPEED = SPEED*1.2
180                     GAME_SPEED = GAME_SPEED*1
181                     PIPE_width = PIPE_width=random.randint(20,PIPE_width*2)
182                     PIPE_HEIGHT = 700
183                     PIPE_GAP = 300*0.95
184 
185 
186         screen.blit(BACKGROUND, (0, 0))
187 
188         if is_off_screen(ground_group.sprites()[0]):
189             ground_group.remove(ground_group.sprites()[0])
190 
191             new_ground = Ground(GROUND_width - 20)
192             ground_group.add(new_ground)
193 
194         #bird.play()
195         ground_group.update()
196 
197         bird_group.draw(screen)
198         ground_group.draw(screen)
199 
200         pygame.display.update()
```

Fig. 3.26

SOLUTION:

Since ‘stepping’ wasn’t a feature of my IDE, I used ‘print (‘c’)’ to see how many times the while loop ran.

I noticed that ‘c’ was only returned to terminal once. Therefore there is a logic error: I need to remove one indent from the ‘while’ statement.

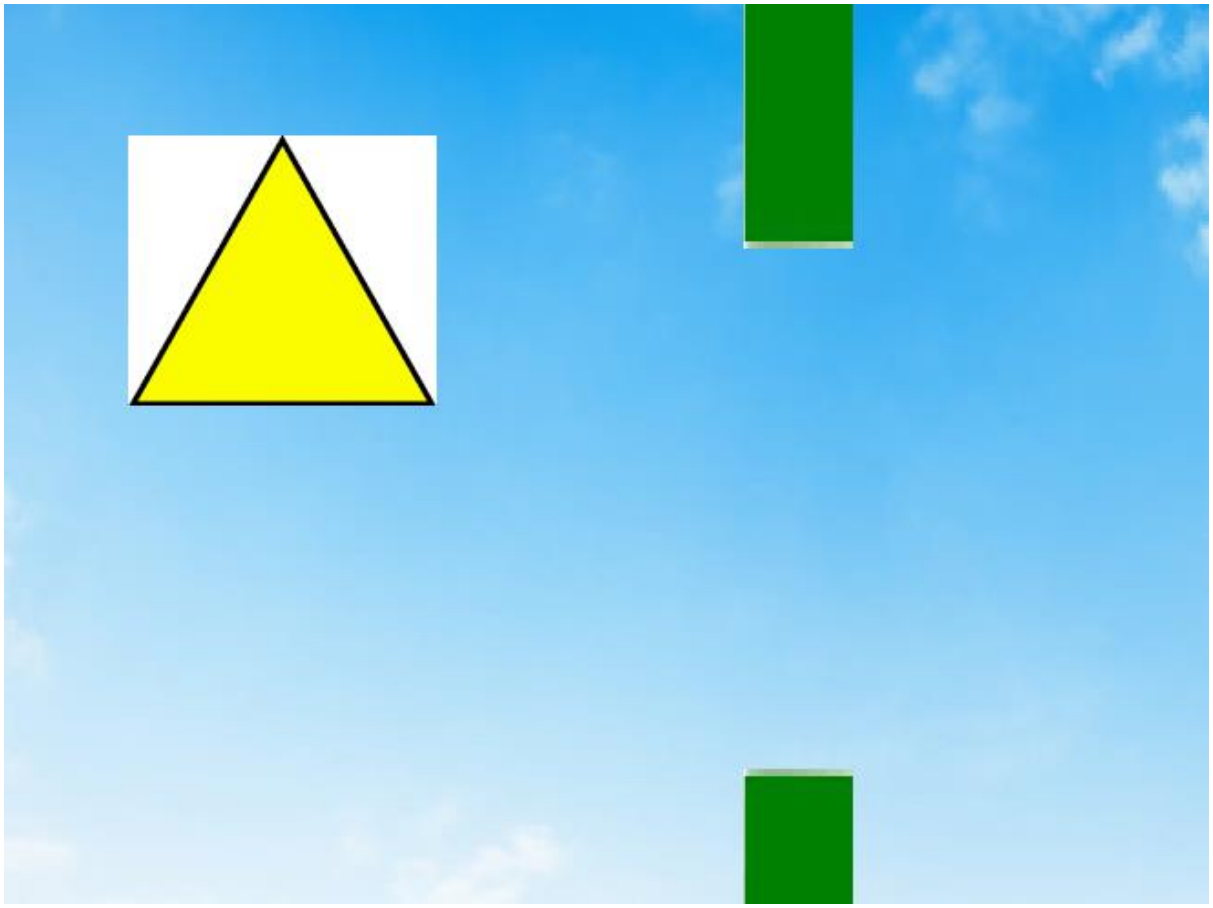


Fig. 3.27.1

```
307         for event in pygame.event.get():
308             if event.type == QUIT:
309                 pygame.quit()
310                 exit()
311             if event.type == KEYDOWN:
312                 if event.key == K_SPACE or event.key == K_UP:
313                     eagle.launch()
```

Fig. 3.27.2

As explained.

PROBLEM:

Generating obstacles with random nature and dimensions without a pattern,

SOLUTION:

Use the random library, a simple work around to the massive difficulty of random number generation. I considered the idea of true randomness by instead considering some insignificant decimal place in some meteorological phenomena mod4 – by inspection this distribution while random is not uniform.

```
def generate_obstacle(self):
    Obstacle_type=random.randint(1,4) #preliminarily - if there are 4 possible obstacles this generats them
    #Obstacle_x_coordinate=Player_x_coordinate+random.randint() #""
```

Fig. 3.28

Mask Collision

Next, I needed to handle the event of collision. This is how the user will lose the game; ending their 'run'. Pygame, facilitates mask collision with built in methods.

```
if (pygame.sprite.groupcollide(bird_group, ground_group, False, False, pygame.sprite.collide_mask) or
        pygame.sprite.groupcollide(bird_group, pipe_group, False, False, pygame.sprite.collide_mask)):

    time.sleep(1)
    break
```

Fig. 3.29

The if statement above checks for collision with either the floor or with pipes.

There is a slight pause at the end so that the user knows that they've lost. Perhaps a 'game over screen' could be added towards the end of the development process.

PROBLEM:

The game didn't close automatically upon failure and return the user to a main menu. While this doesn't hinder functionality it certainly slows the user experience and removes legitimacy.

SOLUTION:

```
#for testing
play=False
while play==False:
    quit_game()
```

Fig. 3.30

Test Table for Iteration One

Test #	Game function	Input	Type of Input	Justify Test	Outcome	Review
1	Jump	Space bar	Valid	Lets the user jump	Character successfully jumps when space is pressed	Success
2	Invalid Jump Keys	Up arrow, 'X', or '@'	Invalid	We ensure that users do not accidentally jump e.g. due to finger slip. In addition,	No jump occurs; input is ignored	Success

				users cannot circumvent the difficulty of rapidly jumping using the space bar.		
3	Play	Click	Valid	Lets the user play	Game starts successfully	Success
4	User touches obstacle	N/A	Valid	Obstacle collision logic	Game transitions to menu screen	Success
5	User touches floor	N/A	Valid	Fall detection	Game transitions to menu screen	Success
6	User moves above field of view	N/A	Boundary	Vertical boundary (ceiling) handling	Game over triggered	Success
7	Obstacle touches ground or ceiling	N/A	Valid	Engine robustness; at this point in development obstacles should never de-spawn	Obstacles persist	Success
8	Quit Program	Alt F4	Valid	Lets user close the program to stop playing	Game closes immediately	Success
9	Quit Program	Escape	Invalid	Invalid input makes sure the user cannot accidentally close the program.	Game remains open; no unintended exit occurs	Success
10	Quit Program	Click quit button	Valid	Standard GUI exit behaviour	Game exits and closes correctly	Success
11	Options Menu	Click options button	Valid	Placeholders for future feature planning	Returns text: "Options will be created in future development"	Success
12	Collision with obstacle (user eagle)	N/A	Valid	Critical collision logic for game challenge	User eagle de-spawns	Success
13	User eagle collision with ceiling or ground	N/A	Valid	Enforces game space boundaries	User eagle de-spawns	Success
14	GUI Play Button	Click 'Play' on GUI menu	Valid	Lets user start game via interface	Game starts successfully	Success

15	Quit Program	X button on window or Alt+F4	Valid	Ensure multiple standard ways to exit the game work as intended	Game closes immediately	Success

PROBLEM:

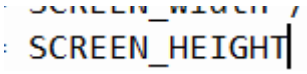


Fig. 3.31

Inconsistent identifiers

SOLUTION:

Use my IDE's 'search and replace' feature to ensure all variables were consistent.

Stakeholder Review One – Were the Objectives Achieved?

After letting my stakeholder James play around with the jump mechanic, we concluded that the cooldown mechanic was too harsh. This was simply solved by reducing the Fatigue_ratio variable.

We've decided that the GUI contains abnormal shapes, so I created symmetry by changing the size of the buttons. Otherwise, my stakeholders James and Oliver agree that the GUI is effective since its function is intuitive. James remarked that the navigation path to playing is effective because it is on the main menu.

My stakeholders Oliver and James agree that the background looks unique and effectively creates the illusion of movement. They agree that using a literal **circular slighting window** was an 'elegant and concise' way to create a background, given that the background is for visual effect.

Given James' coding expertise, he was the correct stakeholder to consult regarding my parent object classes and their efficacy. We agreed that they set up a good foundation for future child classes to inherit from it. This is because the methods and attributes are universally useful. This is effectively demonstrated by the UML Class Diagrams throughout my Design Stage. In phase 2, James is hoping that I make effective use of polymorphism to alter the movement of my multiple player eagle characters.

James recommended that I add a variable jump feature such that holding the space bar lets the user perform a super jump.

One significant issue that would've handicapped later agile iterations, is the structure of my code. My code at its preliminary stage lacked organisation in classes and methods. This will be imperative to creating the enemy and other obstacles.

One crucial practical issue we noticed was that the game was literally impossible to play for more than 47.3 seconds without losing. By considering the best-case scenario of instant generation of obstacles and using a dynamic programming-esque table as well as simple kinematic formulae such as

$$s = ut + \frac{1}{2}at^2$$

Fig. 3.32

we concluded that the least jumps would result from obstacles at the same height. We then computed a loose upper bound of

timemaxtimemax

by comparing the dimensions of the pipe and *user_image.*

James also complains that the user is sent into the game with no warning. He suggest that, as in the style of 'Jetpack Joyride' a game I took much inspiration from (see Analysis), the user should start without obstacles to get their bearings.

Going into stage 2, James wants to see more personality and unpredictability in *Elegant Eagles.* He also wanted to see details 'fleshed out' such as advancing the GUI.

Agile Development Iteration Two

Objectives

1. Create a score system.
    (i) Increment the Score based on relative obstacle-user position.
    (ii) Display this score with the GUI
        (a) Create a scoreboard

2. Create a leaderboard

    (i) Create the board

    (ii) Take inputs

    (iii) Validate name input

    (iv) Validate score input

    (v) Update GUI to display score

3. General optimisations and improvements to streamline future agile stages

4. Create a save/load mechanism
    (i) Save relevant data to a persistent data store

(ii) Load relevant data from the persistent data store.

(a) Make use of a flexible **validation** algorithm that **instantiates class objects** using **data from the persistent store. Examples of data to be saved and loaded in this iteration are score and the characteristics of obstacles.**

Handling Inherited Class Objects in a Data Structure

Creating The Score Board

Incrementing the score board

I need to keep count of the number of passed obstacles. One way of doing this is adding all instances of the class Obstacles to a list.

Then we have the choice of controlling the score by:

```
186     len(obstacles_list)-len(current_obstacles_list)
```

Fig. 3.33

However, this unnecessarily stores on screen elements that will never be needed again. This will be an issue when creating the save/load feature as obstacles will have lots of data that will need to be stored.

Instead, I tracked precisely when an obstacle passes the player.

PROBLEM

```
227         if is_off_screen(obstacle_set.sprites()[0]):
228             obstacle_set.remove(obstacle_set.sprites()[0])
229             obstacle_set.remove(obstacle_set.sprites()[0])
230             score=score+1 #increments score when an obstacle is passed
231             print(score)
```

Fig. 3.34

At first, I tried to simply increment the score whenever an obstacle is deleted, however, if someone loses, their score could be less than it really is because they've passed a pipe recently.

SOLUTION:

```
116 def is_off_screen(sprite):
117     return sprite.rect[0] < -(sprite.rect[2])
118
119 def is_past_eagle(sprite,SCREEN_width):
120     return sprite.rect[0]+SCREEN_width/6<-(sprite.rect[2])
```

Fig. 3.35

So, I solved this by adding a function that checks if the right edge of the obstacle sprite is past the left edge of the player sprite. This was made possible by the last index of rect[] being the width of the sprite.

Inheritance; Polymorphism

Creating the score board

Whilst I could simply initialise score as ‘score=1’, this doesn’t let me add different types of obstalces in the future.

```
def __init__(self,speed=SPEED,current_image=0,rect=0,score):
    pygame.sprite.Sprite.__init__(self)

    self.images = [pygame.image.load('sky.jpg').convert_alpha(),
                   pygame.image.load('green block.png').convert_alpha(),
                   pygame.image.load('user_image.png').convert_alpha(),pygame.image.load('eagle.jpg').convert_alpha()]
    self.image=self.images[2]

    self.speed = SPEED

    self.current_image = 0
    self.mask = pygame.mask.from_surface(self.image)
    self.rect = self.image.get_rect()
    self.rect[0] = SCREEN_width / 6
    self.rect[1] = SCREEN_height / 2
    self.score=score
```

Fig. 3.36

So, I tried to add score as an object to the class ‘Eagle’, however, I had an error with default arguments. I instead opted to make score a private attribute that would need to be accessed and changed using get and set methods.

```
def __init__(self,speed=SPEED,current_image=0,rect=0):
    pygame.sprite.Sprite.__init__(self)

    self.images = [pygame.image.load('sky.jpg').convert_alpha(),
                   pygame.image.load('green block.png').convert_alpha(),
                   pygame.image.load('user_image.png').convert_alpha(),pygame.image.load('eagle.jpg').convert_alpha()]
    self.image=self.images[2]

    self.speed = SPEED

    self.current_image = 0
    self.mask = pygame.mask.from_surface(self.image)
    self.rect = self.image.get_rect()
    self.rect[0] = SCREEN_width / 6
    self.rect[1] = SCREEN_height / 2
    self.__score=0
```

Fig. 3.37

At first, I ensured that the score incremented properly.

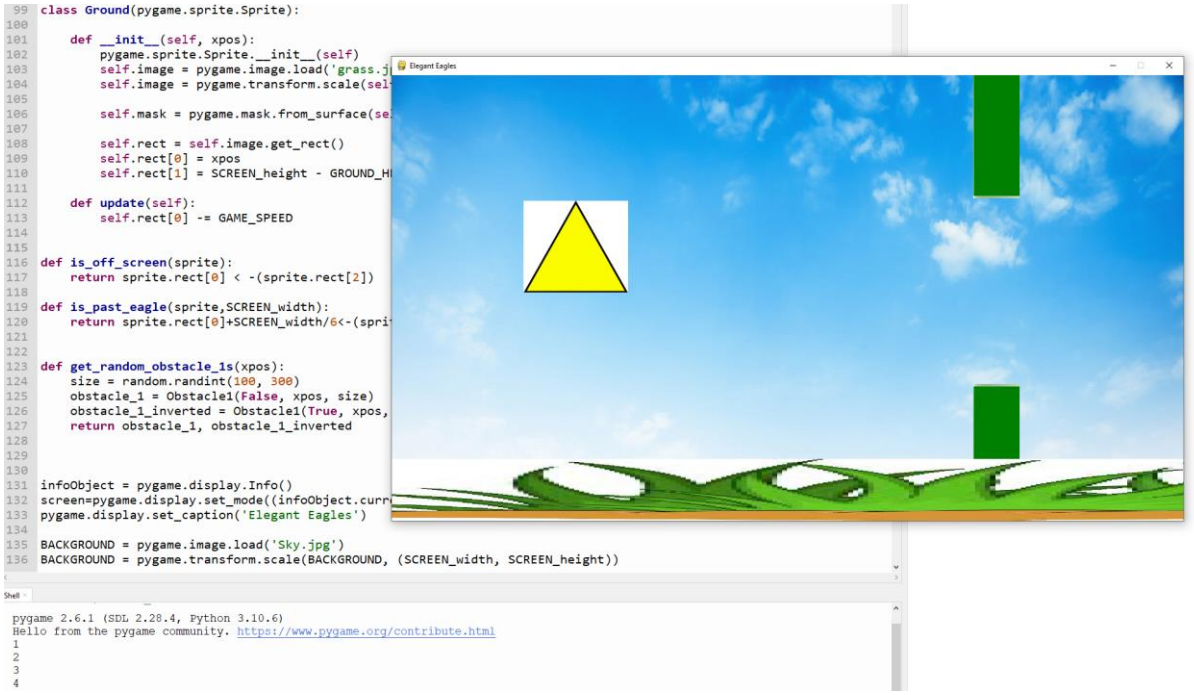


Fig 3.38

GUI Update

Next, I added a permanent GUI element to the screen using:

```
254                generate_OSE(screen_width*5/5.2,200,200 ,50,50,BLACK, (10,10,10),action=print(score))
```

Fig 3.39

I chose to develop my on-screen element (OSE) function from the design phase, by adding an 'active' and 'inactive' colour so that the player knows where their cursor is on the screen. This was a result of my stakeholder, James, complaining that he didn't know where his cursor was on the screen. This is a superior alternative to enlarging the cursor, as that is an unimportant on-screen element to a platformer.

```
def generate_OSE(screen, x_coordinate, y_coordinate, x_size, y_size, default_colour, active_colour, action=None):
    #get the current position of the mouse cursor
    cursor=pygame.mouse.get_pos()

    #check if the mouse button is pressed
    press=pygame.mouse.get_pressed()

    #check if the cursor is within the boundaries of the button
    if x_coordinate<cursor[0]<x_coordinate+x_size and y_coordinate<cursor[1]<y_coordinate+y_size:
        #change button color to active when hovered over
        pygame.draw.rect(screen, active_colour, (x_coordinate, y_coordinate, x_size, y_size))

        #check if the left mouse button is clicked and an action is assigned
        if press[0]==1 and action:
            action() #execute the assigned function
    else:
        #draw the button with the default color when not hovered over
        pygame.draw.rect(screen, default_colour, (x_coordinate, y_coordinate, x_size, y_size))

    #render the text onto the button
    text_surface=font.render(text, True, BLACK)

    #center the text within the button
    text_rect=text_surface.get_rect(center=(x+width//2, y+height//2))

    #draw the text onto the screen
    screen.blit(text_surface, text_rect)
```

Fig 3.40

```
pygame 2.6.1 (SDL 2.28.4, Python 3.10.6)
Hello from the pygame community. https://www.pygame.org/contribute.html
1
Traceback (most recent call last):
  File "C:\Users\agraw\OneDrive\Documents\Python projects\09.12 code", line 279, in <module>
    obstacles()
  File "C:\Users\agraw\OneDrive\Documents\Python projects\09.12 code", line 253, in obstacles
    generate_OSE(screen_width*5/5.2,200,200 ,50,50,BLACK, (10,10,10),action=print(score))
  File "C:\Users\agraw\OneDrive\Documents\Python projects\09.12 code", line 172, in generate_OSE
    pygame.draw.rect(screen, default_colour, (x_coordinate, y_coordinate, x_size, y_size))
TypeError: argument 1 must be pygame.surface.Surface, not float

>>>
```

Fig. 3.41

PROBLEM:

I was passing the wrong parameters into generate_OSE, instead of a dimensional paramater, element 1 of the parameter subarray requires a plane to create the scoreboard on.

SOLUTION:

```
253             generate_OSE(screen,200,200 ,50,50,BLACK, (10,10,10),action=print(score))
```

Fig. 3.42

Here is the result:

```
143         if is_off_screen(obstacle_set.sprites()[0]):
144             obstacle_set.remove(obstacle_set.sprites()[0])
145             obstacle_set.remove(obstacle_set.sprites()[0])
146             score += 1
147             generate_OSE(screen, 10, 10, 150, 50, (200, 200, 200),
148
149             obstacle_1s = get_random_obstacle_1s(_screen_width * 2)
150             obstacle_set.add(obstacle_1s[0])
151             obstacle_set.add(obstacle_1s[1])
```

Fig. 3.43

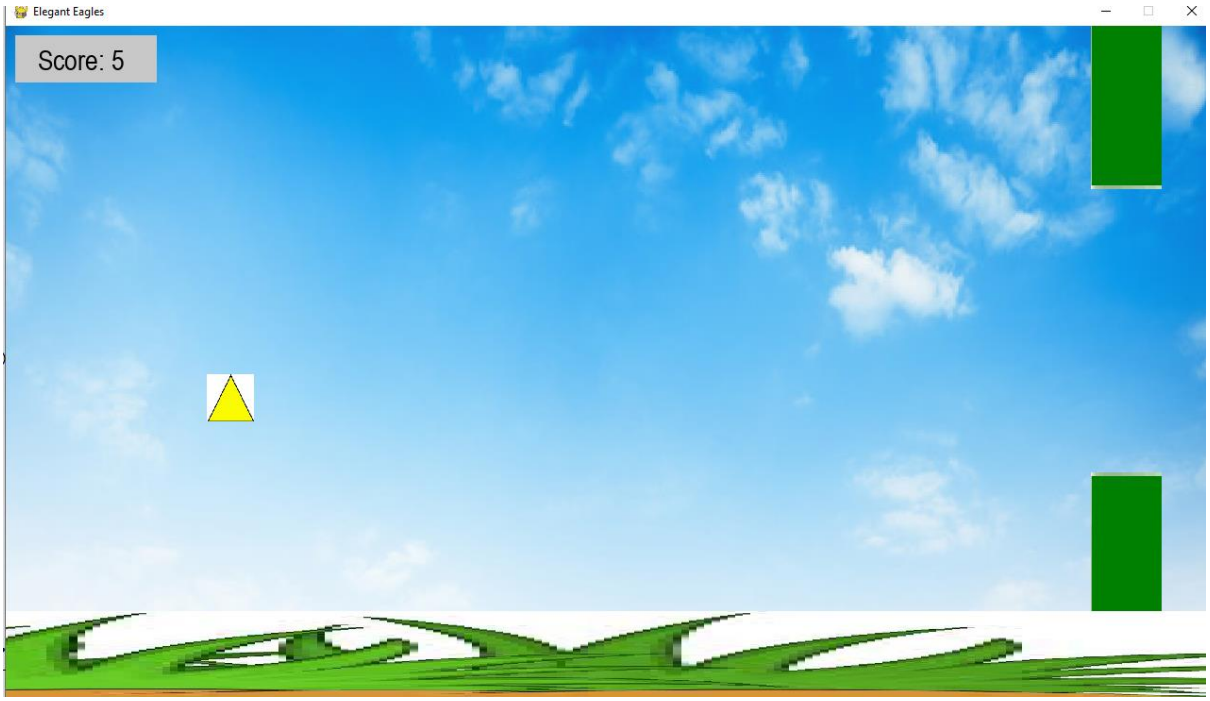


Fig. 3.44

Creating a Leaderboard

Before implementing a leaderboard into the game itself, I decided to

1. Create the leaderboard
2. Take direct user inputs
3. Validate usernames
4. Validate scores
5. Render leaderboard

And only then will I take inputs directly from the existing game program. Given that I am working with a persistent data store, I foresee issues pulling every single piece of data from the JSON to generate the score board. Sorting this data will further require an efficient sorting algorithm. I intend to implement quicksort.

Create the leaderboard

```
43  def main():
44
45      lb = {}  #note choice of data structure
```

Fig 3.45

I chose to use a dictionary for the leaderboard since, every leaderboard entry comes with a name, and its corresponding value.

Inputting the name

```
47          name = input("Enter name (or type 'exit' to stop): ").strip()
48          if name.lower() == 'exit':
49              break
50
51          if not is_valid_name(name):
52              print(f"Invalid name. Use only letters, numbers, and underscores, and ensure it's at most {MAX_name_LENGTH} characters long.")
53              continue
```

Fig 3.46

Here I reference a validation algorithm that I will write afterwards

Edge Case Handling; String Formatting

Below, I show how the algorithm deals with the edge case of repeated names. When I implement this in the actual game, I will need to decide if I want the user to have this choice. Perhaps instead of obfuscating the user experience with unnecessary input, I can simply ignore the edge case by only calling the *add_score* function after finding a bigger score.

```
54
55          #if repeat name
56          if name in lb:
57              while True:
58                  overwrite = input("name already exists. Do you want to update the score? (yes/no): ").strip().lower()
59                  #names consist of characters from the alphabet
60                  if not overwrite.isalpha():
61                      print("Invalid response. Please use letters only.")
62                      continue
63                  elif len(overwrite) > 3:
64                      print("Invalid response. Response too long.")
65                      continue
66                  #Accept only n, y, no, yes
67                  elif overwrite not in ['yes', 'y', 'no', 'n']:
68                      print("Invalid response. Acceptable responses: n, y, no, yes.")
69                      continue
70                  break
71              if overwrite in ['no', 'n']:
72                  continue
```

Fig 3.47

Here I do the same thing with the score input.

```
74          score_input = input("Enter score: ").strip()
75          valid, result = is_valid_score(score_input)
76          if not valid:
77              print(f"Invalid input: {result}")
78              continue
```

Fig. 3.48

Validation; RegEx; Edge Case Handling

Validating user input

```python
def is_valid_name(name):
    if not name: #non empty
        return False
    if len(name) > MAX_name_LENGTH:
        return False
    return bool(re.match(r'^[A-Za-z0-9_]+$', name))

def is_valid_score(score_str):
    # Checks if the score is a digit and within the specified range.
    if not score_str.isdigit(): #covers null case implicitly since '' is str()
        return False, "score is a natural number."
    score = int(score_str)
    if score < MIN_SCORE:
        return False, "invalid score!"
    elif score > MAX_SCORE:
        return False, f"score too big! The max is {MAX_SCORE}."
    return True, score
```

Fig 3.49

- Name:
  First the algorithm checks for the null case, that is, ensures there is really a name entered. Next it makes sure the name is not too long. I arbitrarily set this global constant to 20, without the condition, the leaderboard could be filled by a single name.

This is what the logic looks like.

```
Enter username (or type 'exit' to stop): gifpdog
Enter score: 234
Enter username (or type 'exit' to stop): 8hn
Enter score: bofg
Invalid input: Score must be a positive integer.
Enter username (or type 'exit' to stop):
Invalid username. Use only letters, numbers, and underscores, and ensure it's at most 20 characters long.
Enter username (or type 'exit' to stop):  $ £4
Invalid username. Use only letters, numbers, and underscores, and ensure it's at most 20 characters long.
Enter username (or type 'exit' to stop): £$£
Invalid username. Use only letters, numbers, and underscores, and ensure it's at most 20 characters long.
Enter username (or type 'exit' to stop): fe
Enter score: 24Q£
Invalid input: Score must be a positive integer.
Enter username (or type 'exit' to stop):
```

Fig. 3.50

Adding a score

```python
def add_score(lb, name, score):
    lb.append(name, score)
```

Fig. 3.51

Tuple Unpacking; String Formatting; Lambda Functions

Printing the leaderboard

```
10  def print_lb(lb):
11      # Sort the lb by score in descending order
12      sorted_lb = sorted(lb.items(), key=lambda x: x[1], reverse=True)
13
14      print("\nlb:")
15      rank = 1
16      prev_score = None
17
18      for i, (user, score) in enumerate(sorted_lb, start=1):
19          if score != prev_score:
20              rank = i  # Update rank only if the score changes
21          print(f"{rank}. {user} - {score}")
22          prev_score = score
```

Fig. 3.52

I made use of tuple unpacking and lambda function because these are concise methods which therefore improve readability.

Optimising with Hash Table and Big O() and Big Ω() notation.

I used pythons built in methods e.g. sort to make sure the leaderboard is ordered correctly. However, given that the leaderboard is updated, not regenerated from scratch. Therefore, I should store a sorted list.

Python's 'sort' procedure is an implementation of Timsort.

Moreover, when we want to check if a new score is indeed a high score, rather than performing a linear or binary search on the list every time, we should instead use hash tables since their complexity is O(1) or Ω(1) and

$$O(1)<O(log(n))<O(n)$$

$$Ω(1)<= Ω(1)<=log(2)$$

Fig. 3.53

Where hash query is the left most side. The use of Ω() notation is more appropriate given that we are working on a near finished data structure - close to if not a best case.

Here are some examples of how this algorithm runs. More test cases can be seen in my Appendix Two: Video Evidence Folder.

```
Enter username (or type 'exit' to stop): gifpdog
Enter score: 234
Enter username (or type 'exit' to stop): 8hn
Enter score: bofg
Invalid input: Score must be a positive integer.
Enter username (or type 'exit' to stop):
Invalid username. Use only letters, numbers, and underscores, and ensure it's at most 20 characters long.
Enter username (or type 'exit' to stop):  $ £4
Invalid username. Use only letters, numbers, and underscores, and ensure it's at most 20 characters long.
Enter username (or type 'exit' to stop): £$£
Invalid username. Use only letters, numbers, and underscores, and ensure it's at most 20 characters long.
Enter username (or type 'exit' to stop): fe
Enter score: 24Q£
Invalid input: Score must be a positive integer.
Enter username (or type 'exit' to stop):
```

Fig 3.54

General improvements and Optimisations

Inheritance and polymorphism

PROBLEM:

I had several classes with similar attributes. Creating the definitions for the classes separately made my code less readable and threatened scalability issues regarding consistency of identifiers/attributes.

As a solution, I set up a class structure hierarchy. For example, since both ‘eagle’ and ‘obstacle 1’ have y coordinates:

```python
class Eagle(pygame.sprite.Sprite):

    def __init__(self,speed=SPEED,current_image=0,rect=0):
        pygame.sprite.Sprite.__init__(self)

        self.images = [pygame.image.load('sky.jpg').convert_alpha(),
                       pygame.image.load('green block.png').convert_alpha(),
                       pygame.image.load('user_image.png').convert_alpha(),pygame.image.load('eagle.jpg').convert_alpha()]
        self.image=self.images[2]

        self.speed = SPEED

        self.current_image = 0
        self.mask = pygame.mask.from_surface(self.image)
        self.rect = self.image.get_rect()
        self.rect[0] = SCREEN_width / 6
        self.rect[1] = SCREEN_height / 2


    def update(self):
        self.speed += GRAVITY

        # UPDATE HEIGHT
        self.rect[1] += self.speed

    def launch(self):
        self.speed = -SPEED



class Obstacle1(Eagle):
    def __init__(self, inverted, xpos, ysize):
        # Call Eagle's constructor
        super().__init__()

        # Overwrite the attributes specific to Obstacle1
        self.image = pygame.image.load('green block.png').convert_alpha()
        self.image = pygame.transform.scale(self.image, (OBSTACLE_1_width, OBSTACLE_1_HEIGHT))

        self.rect = self.image.get_rect()
        self.rect[0] = xpos

        if inverted:
            self.image = pygame.transform.flip(self.image, False, True)
            self.rect[1] = - (self.rect[3] - ysize)
        else:
            self.rect[1] = SCREEN_height - ysize

        self.mask = pygame.mask.from_surface(self.image)

    def update(self):
        self.rect[0] -= GAME_SPEED


```

Fig. 3.55

However, I then had to use polymorphism, specifically overriding to modify the ‘update’ method such that the obstacles moved at the games speed

This will be useful in creating more obstacle object templates that will have similar properties to Obstacle.

PROBLEM:

Elegant Eagles booted quite slowly on the device of my stakeholder, James.

```
4 from pygame.locals import *
```

Fig. 3.56

SOLUTION:

One solution would be to increase the hardware requirements but that means some people wouldn't be able to enjoy the game. Instead, I made a significant optimisation in my 'import’ statements.

```
3 from pygame.locals import import QUIT, KEYDOWN, K_SPACE, K_UP
```

Fig. 3.57

I specified the parts of pygame that I planned on using. This makes the program more readable too, letting stakeholders like James understand and help me improve my program more easily. This also improves the project's scalability by reducing run times. The only possible draw back is that, should other parts of the library be used, then they will need to be imported as well. I will discuss this last point with James.

Circular Scrolling Windows and User Input

PROBLEM:

As explained in my stakeholder review from phase 1, users were sent into the game with ‘no warning’.

SOLUTION:

The player model now begins at rest with a sliding window for the ground still passing. No obstacles are being generated. The game starts when the player presses spacebar.

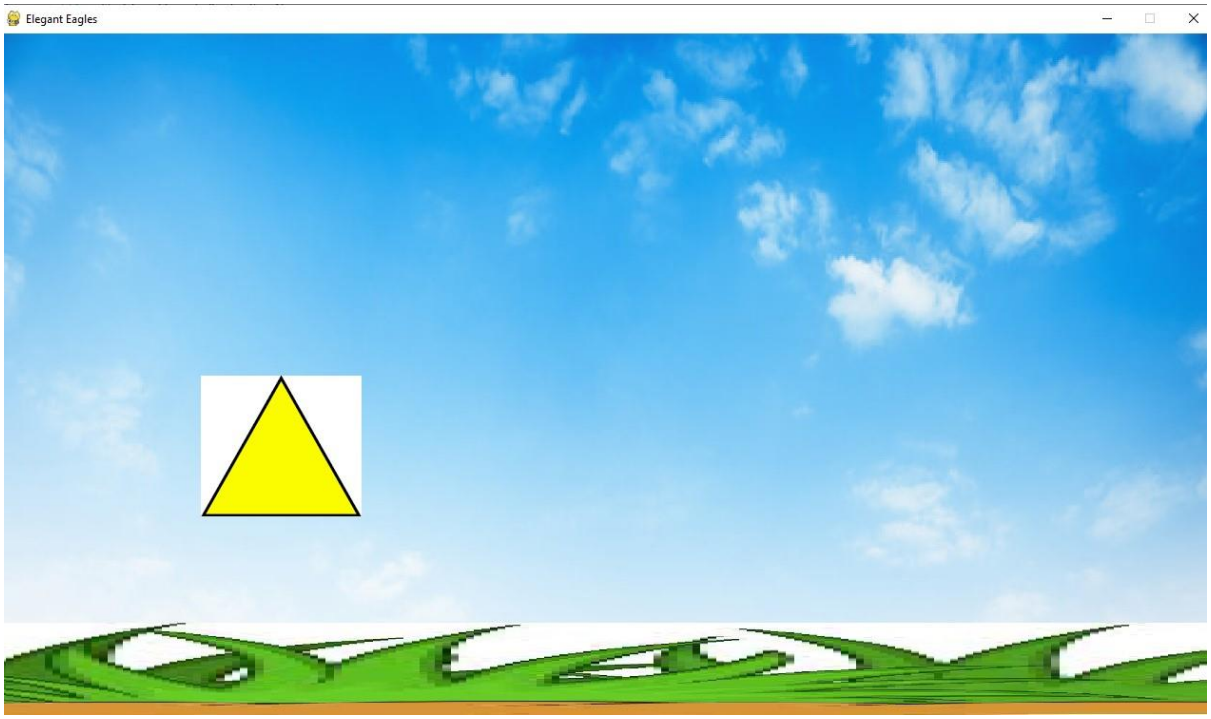


Fig. 3.59

Lamda Functions and Readability

PROBLEM:

The readability of my code was poor. There were lots of numbers in use when english would add clarity.

```
SPEED=SPEED*1.2
```

SOLUTION

I made use of lambda functions, such as one that simply changes its parameter by some scale factor.

```
149 def scale_factor(n):
150   return lambda a : a * n
151 change_speed = scale_factor(1.2)
```

Fig. 3.60

I also made use of 'F-strings' to improve clarity.

```
144             generate_OSE(screen, 10, 10, 150, 50, (200, 200, 200), (150, 150, 150), f"Score: {score}")
```

Fig. 3.61

PROBLEM:

I import my libraries in a try catch statement. This is unnecessary, as, it creates inconsistencies across devices.

SOLUTION:

Instead, I will use the backup library, that works on both devices that I am developing this project on: a school and personal device.

```
1 import pygame,random,time
```

Fig. 3.62

PROBLEM:

The robustness of my program needed improving.

SOLUTION:

I made class attributes private where possible. This improves the program robustness as users cannot directly interact with attributes and change them accidentally or maliciously.

e.g. below shows the speed attribute of the Eagle class.

```
self._speed = speed
```

Inheritance; Polymorphism; Encapsulation

PROBLEM:

The scalability of sprite usage in my program was poor.

SOLUTION:

I created a parent class from which all sprite using classes would inherit.

```
26 class SpriteBase(pygame.sprite.Sprite):
27     def __init__(self, image_path, xpos, ypos, width, height):
28         super().__init__()
29         self.image = pygame.image.load(image_path).convert_alpha()
30         self.image = pygame.transform.scale(self.image, (width, height))
31         self.mask = pygame.mask.from_surface(self.image)
32         self.rect = self.image.get_rect()
33         self.rect[0] = xpos
34         self.rect[1] = ypos
```

Fig. 3.63

This notably decreased the number of lines of code.

```
36 class Eagle(SpriteBase):
37
38     def __init__(self, speed=SPEED):
39         super().__init__('user_image.png', _screen_width / 6, _SCREEN_height / 2, 50, 50)
40         self.__speed = SPEED
41         self.__score = 0
42
43     def update(self):
44         self.__speed += GRAVITY
45         self.rect[1] += self.__speed
```

Fig. 3.64

Therefore, should my project be scaled up, the improved readability will be significant. This will also facilitate the testing and debugging of my program in the future.

PROBLEM:

I tried to move the code that controls the game in a running state into a subroutine to enable modular design and make the code more readable. The latter will be crucial when developing code in a few months time, after having forgotten what I've developed.

In doing so I ran into the logic error of the pop-up window cropping out everything except for the top right of the full window.

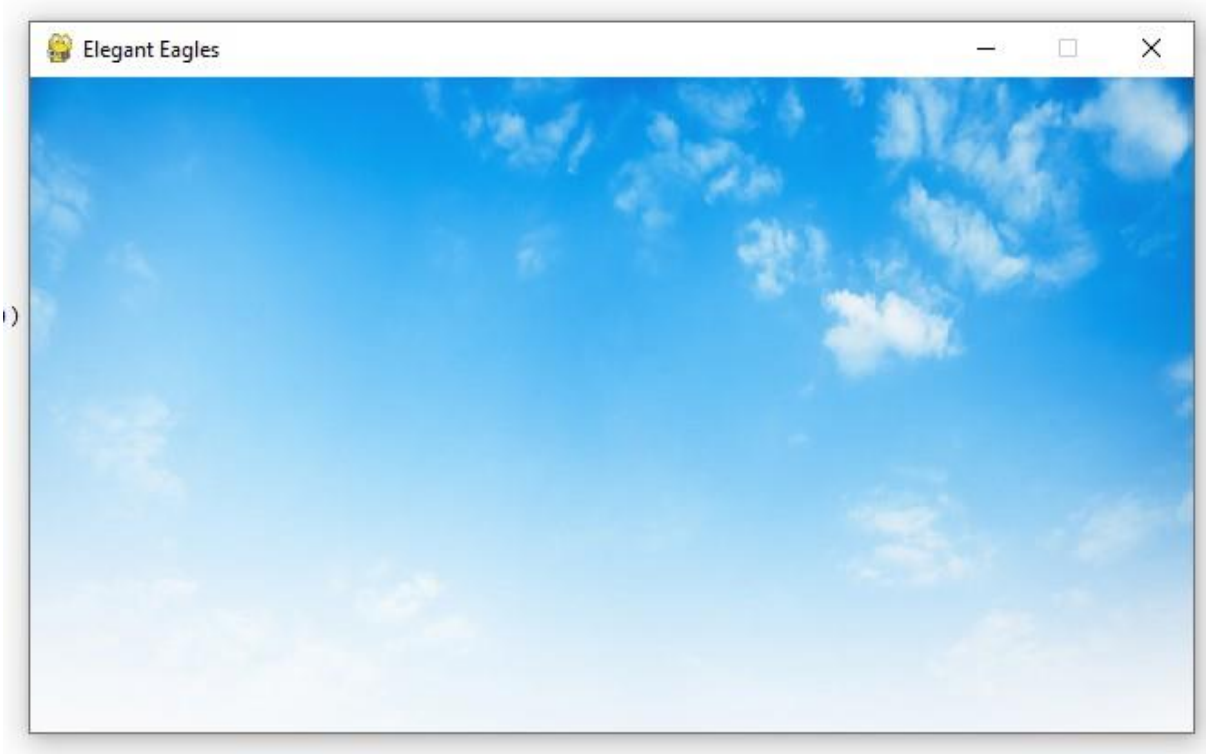


Fig. 3.65

SOLUTION:

Since my IDE doesn't have a built-in tracetable/stepping tool, I manually created one by adding simple ‘print(‘c’)’ and ‘print('d’)’ lines:

```
190     while play==True:
191         clock.tick(15)
192         #Controling jumps
193         print( 'd')
194         for event in pygame.event.get():
195             if event.type == QUIT:
196                 pygame.quit()
```

Fig. 3.66

```
>>> %Run '30.11 code'
pygame 2.6.1 (SDL 2.28.4, Python 3.10.6)
Hello from the pygame community. https://www.pygame.org/contribute.html
d
d
d
d
d
d
d
d
d
d
d
d
c
c
c
c
c
```

Fig. 3.67

```
222     #set up display
223     infoObject = pygame.display.Info()
224     SCREEN_width=infoObject.current_w/2
225     SCREEN_height=infoObject.current_h/2
226     screen=pygame.display.set_mode((infoObject.current_w/2, infoObject.current_h/2))
227     pygame.display.set_caption('Elegant Eagles')
228     BACKGROUND = pygame.image.load('Sky.jpg')
229     BACKGROUND = pygame.transform.scale(BACKGROUND, (SCREEN_width, SCREEN_height))
```

Fig. 3.68

So, I tried modifying the parameters ‘w/2’ and ‘h/2’ to see if this is what is controlling the window, but there was no change.

From Figure 3.66 I thought that the logic error could be found in figure 3.64.

```
256     while True:
257         print('c')
258
259         clock.tick(15)
260
261         for event in pygame.event.get():
262             if event.type == QUIT:
263                 pygame.quit()
264             if event.type == KEYDOWN:
265                 if event.key == K_SPACE or event.key == K_UP
266                     bird.launch()
267
268         screen.blit(BACKGROUND, (0, 0))
```

Fig. 3.69

So instead, I tried modifying the parameters here. Modifying line 268 only shifts the background, not the screen.

Fig. 3.70

Fig. 3.71

I was about to look further in the program, but then I realised that the error could've been created at the screen's creation.

It turns out, that I had initialised the screen earlier with some constants that I had forgotten to delete from an earlier iteration of the program.

```
20 screenw=100
21 screenh=10
```

Fig. 3.72

Then fixing the initialisation resulted in a working game again.

PROBLEM:

I noticed that my GUI was purely functional, whereas settings like volume had to be modified on the user's end. This took away from the user experience since the user has to open their settings to do this rather than everything being in one place.

SOLUTION:

To improve this, I created a parent class 'UIObjects' and then using polymorpism inherit the parent class and create a slider/switch/dropdown-menu, here is some pseudocode to explain:

```
class UiObject:
    def draw(self):
        pass

class Button(UiObject):
    ...

class Slider(UiObject):
    ...
```

Fig. 3.73

Improving the generation of obstacles

```
OBSTACLE_1_width = OBSTACLE_1_width=random.randint(20,OBSTACLE_1_width*2)
OBSTACLE_1_HEIGHT = 700
OBSTACLE_1_GAP = 300*0.95
```

Fig. 3.74

At first, a 'run's initial stages would be repetitive. The 'randomness' was generated recursively, such that, early obstacles would be boring. This is something my stakeholder James discovered when tinkering with the game.

So, we introduced an initial randomisation of the obstacle parameters.

```
120 def get_random_obstacle_1s(xpos):
121     size = random.randint(100, 300)
122     obstacle_1 = Obstacle1(False, xpos, size)
123     obstacle_1_inverted = Obstacle1(True, xpos, SCREEN_height - size - OBSTACLE_1_GAP)
124     return obstacle_1, obstacle_1_inverted
```

Fig. 3.75

Creation and Implementation of a Persistent Data Store

To create a leaderboard and eventually a pause/save system I needed to store data.

MySQL or JSON

Feature	MYSQL	JSON	Verdict
Ease of Use	Requires SQL queries	Simple to read and write	Whilst I can make SQL queries, there is more support for JSON online.
Scalability	Excellent for data integrity, particularly if Elegant Eagles is to be played concurrently by multiple people.	Suitable for local use.	My project won't see commercial use, then JSON is sufficient.

Performance	Slower for simple data sets	Faster	Elegant Eagles is not going to be used commercially, then the size of the data sets that it handles will always be small.
Portability	Tied to a database	Files can be shared	Manoeuvrability will be essential for stakeholders like James to test Elegant Eagles at home. Moreover, it lets Elegant Eagles work without a direct internet connection.
Backups	Automated backups are supported.	Backups must be done manually.	Only JSON carries the risk of someone forgetting to backup and losing all of their progress.

Fig. 3.76

So, I will use JSON given its ease of use.

Preliminary read and write functions to the JSON

I plan on instantiating class objects to store the attributes of a specific 'run'. Data will need to be read and validated from the JSON to do so.

I will also store the type of leaderboard users want to see in the JSON and this will be stored in a private attribute of the class, 'Eagle'.

Saving & Loading

Saving

Figure 3.77 demonstrates the sub-modules that I need to implement to create a save feature.

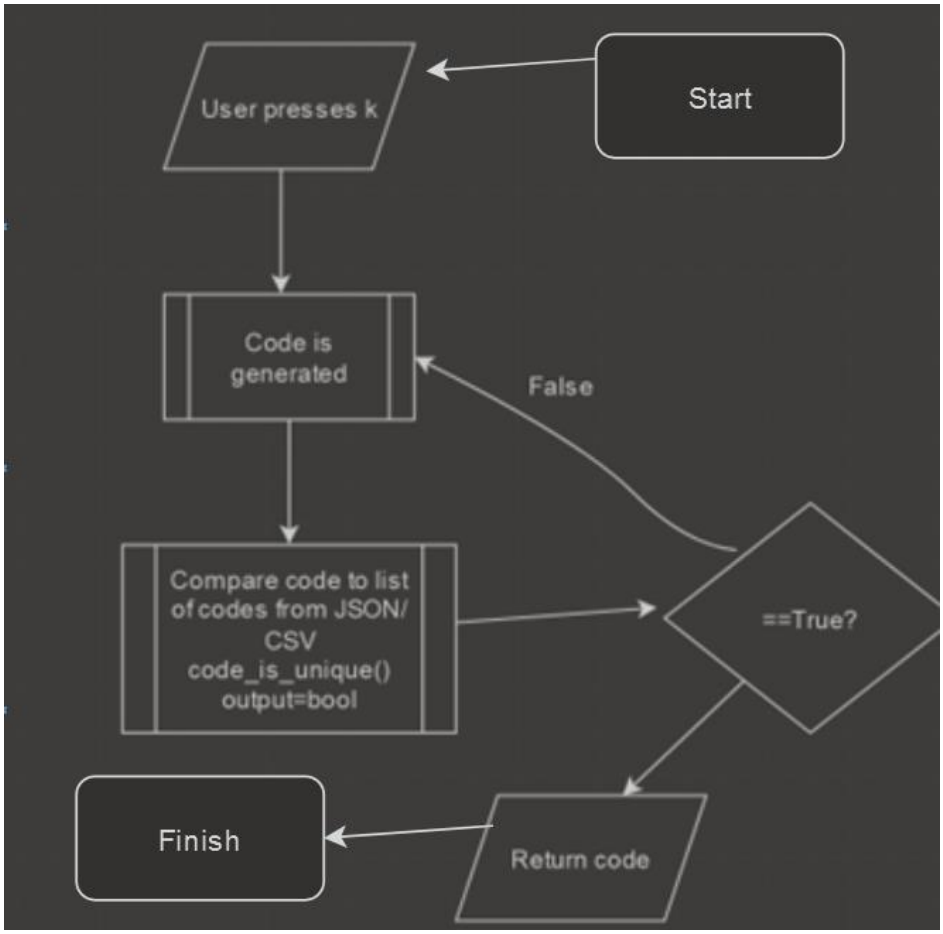


Fig. 3.77

The user must be able to pause and resume their progress across different sessions, even on a separate running instance. A login-based approach would be cumbersome for users and expose sensitive data. Instead, I implemented a save system using a randomly generated code, allowing players to retrieve their progress quickly.

However, a major challenge arises: ensuring that save codes are unique to prevent data loss or overwriting. If two players receive the same code, their game states may conflict, leading to unpredictable behaviour.

Use of Big O() and Ω() notation to achieve algorithmic optimality

PROBLEM

Whilst unlikely, it is possible for users to have the same code. This will inevitably lose users data should Elegant Eagles be used widely.

SOLUTION

To solve this, I explored two options:

- Sequential code generation: Ensures uniqueness but makes it easier for malicious users to guess another player's save code.
- Validated pseudo-random generation: Offers better security but requires an efficient method to confirm uniqueness.

I chose the latter, as it prevents unauthorized access through brute-force attempts. The validation process involves checking the generated code against existing ones in storage.

To optimise the generation, the codes should be sufficiently long. I decided that six digit codes are suitable since there is a one-million codes making the chance of coincidence on a simple game like this small. The game can be made more scalable in the future using non-numeric characters and longer codes; this is not suitable to implement yet as it will needlessly increase the memory required to play *Elegant Eagles.* Moreover, my **stakeholder** Oliver made me realise that the use of numbers makes entering the code easier for users.

I could use a binary search. This would require sorting the list.

On the one hand since we will query the list frequently, this could be worth it. However, we are also appending to the list frequently. I would've used an insertion sort, since we are sorting an almost sorted list, and binary search, giving an $O(n^2)$ solution, more appropriately given that the sorting is of a near best case. The solution is **Ω(log(2)+1).**

However, for the sake of **scalability** a hash table is the most suitable choice. This is because, at the expense of RAM, the O(1) query time ensures that save data can be accessed and validated almost instantly, regardless of the number of stored entries. This trade-off between memory usage and efficiency is justified, as fast lookups prevent performance bottlenecks when retrieving or verifying save codes. Additionally, hash tables eliminate the need for sorting and searching operations, making them ideal for handling large-scale game data with minimal processing overhead.

To ensure security, a lockout feature could be implemented to prevent brute force entry in development iteration three. E.g., after five unsuccessful queries, the user will be timed-out of using the load feature. This idea came to me after an impromptu chat with my stakeholder Oliver.

Use of Hash Tables

Data Structure

The data must be saved to the persistent data store. I decided to create a hash-map in the JSON because:

- Hash queries are O(1) rather than the time complexity of linear queries: O(n)
- Hash tables can deal with collisions quickly using linear probing or two dimensional sub-lists.

Coupled with the **scalability** with respect to time, hash tables are the clear choice.

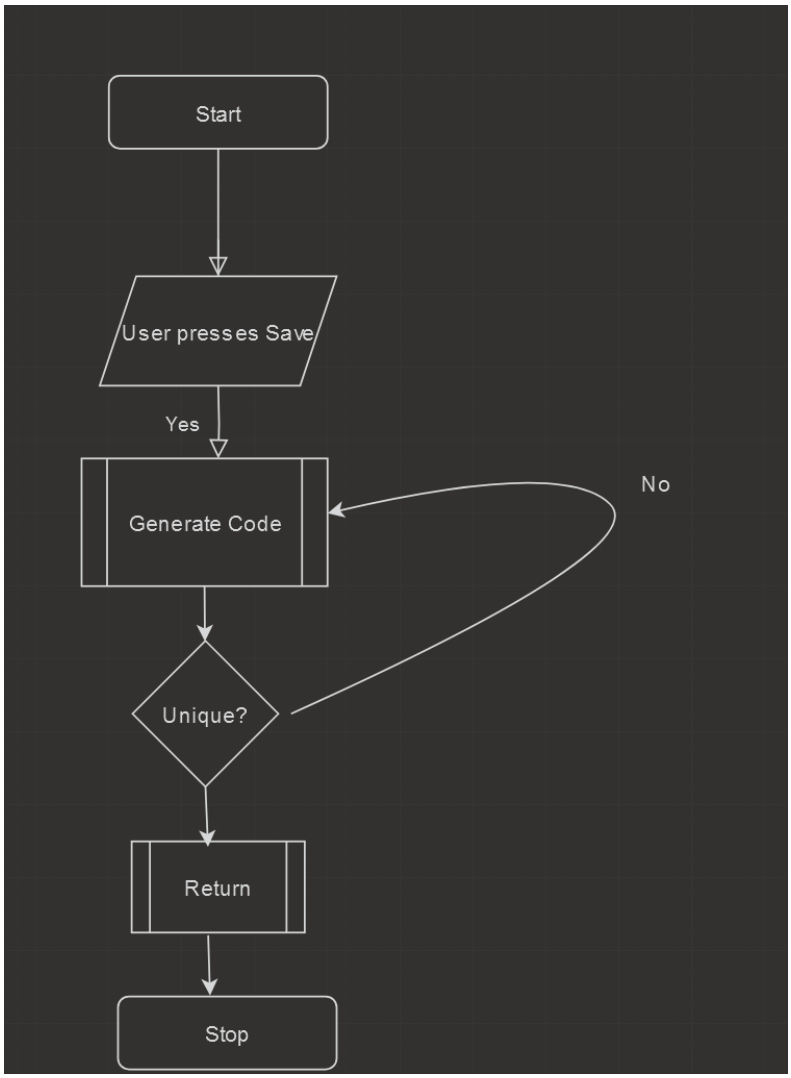
```
def save_game(score, eagle, obstacles):
    try:
        with open(SAVE_FILE, 'r') as file: #to minimise syntax errors
            save_data = json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        save_data = {} #validates existence of the file; if it doesn't exist because the game is a new run: file is created
```

Notably here, we **validate** the existence of the save file. If it is missing or corrupted (json.JSONDecodeError), the program does not crash but instead initialises a new save structure.

In addition to facilitating a hash table, the use of dictionaries makes the program more **maintainable** because to change a dictionary, a programmer must enter a key and field. Therefore, the rarity of dictionaries in the program means that it is unlikely to be unintentionally modified.

Save Algorithm

The flowchart below summarises the decisions that the save algorithm must make:



Validation with Try:Catch:

Generating the Code

```
#load existing save data
try:
    with open(SAVE_FILE, 'r') as file:
        save_data = json.load(file)
except (FileNotFoundError, json.JSONDecodeError):
    save_data = {}  #initialize as an empty dictionary if the file doesn't exist or is invalid

#generate a unique random code
while True:
    save_code = str(random.randint(100000, 999999))
    if save_code not in save_data:  #check if the code is unique
        break

#add the new save entry to the dictionary
save_data[save_code] = {"score": score}
```

```
Game Saved! Code: 990856

>>>
```

As described, we need to handle the pseudorandom generation of the code.

First, we need to initialise the file so that we can make queries to verify if the generated code is in fact unique. We use a try:catch: statement to make sure that the file exists. If the run has just started, then we simply create the list. Next, we need to generate.

I considered the limitation of pseudo randomness; that it is theoretically predictable. However, given the uniformity of pseudo randomness, its predictability cannot be maliciously exploited.

I chose ‘k’ to be the save button because that’s what some games from my Analysis did. The user is given a unique save code as shown above.

To make sure that the generated code was valid, I chose a lower bound that was six digits long too. Here, we sort the list of used save codes and then use Python’s built-in binary search to check if a generated code is non-unique in O(log(n)). As explained, insertion sort complements binary search since we are sorting a partially sorted list repeatedly. If the code is not valid, the while loops ensures that a valid code is eventually returned.

Saving the Data to the **Hash Table**

Creating Hash Buckets by Pulling all Attributes; Two Dimensional Dictionary

Here the system records the player's current state, including the eagle’s position, obstacles, and score, and assigns a unique code for retrieval.

```python
def save_game(code, eagle, obstacles, score):
    try:
        # Attempt to load existing save data
        with open(SAVE_FILE, 'r') as file:
            save_data = json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        # If file doesn't exist or is corrupted, start with an empty dictionary
        save_data = {}

    # Validate inputs before saving
    if not isinstance(code, str) or not code:
        print("Invalid save code! Must be a non-empty string.")
        return
    if not hasattr(eagle, 'x') or not hasattr(eagle, 'y'):
        print("Invalid eagle object! Missing coordinates.")
        return
    if not isinstance(score, (int, float)) or score < 0:
        print("Invalid score! Must be a non-negative number.")
        return

    # Store game data in dictionary
    save_data[code] = {
        "eagle": (eagle.x, eagle.y),  # Store eagle's position
        "obstacles": [(obs.x, obs.y, obs.width, obs.height) for obs in obstacles],  # Store all obstacles
        "score": score  # Store current score
    }

    # Write updated data back to file
    with open(SAVE_FILE, 'w') as file:
        json.dump(save_data, file)

    print(f"Game Saved! Code: {code}")
```

The three if statements **validate** the save algorithm. Without them, it would be difficult to detect logic errors.

Here, we save:

- Coordinates of every obstacle.
- UserEagle coordinates and velocity
- Current score

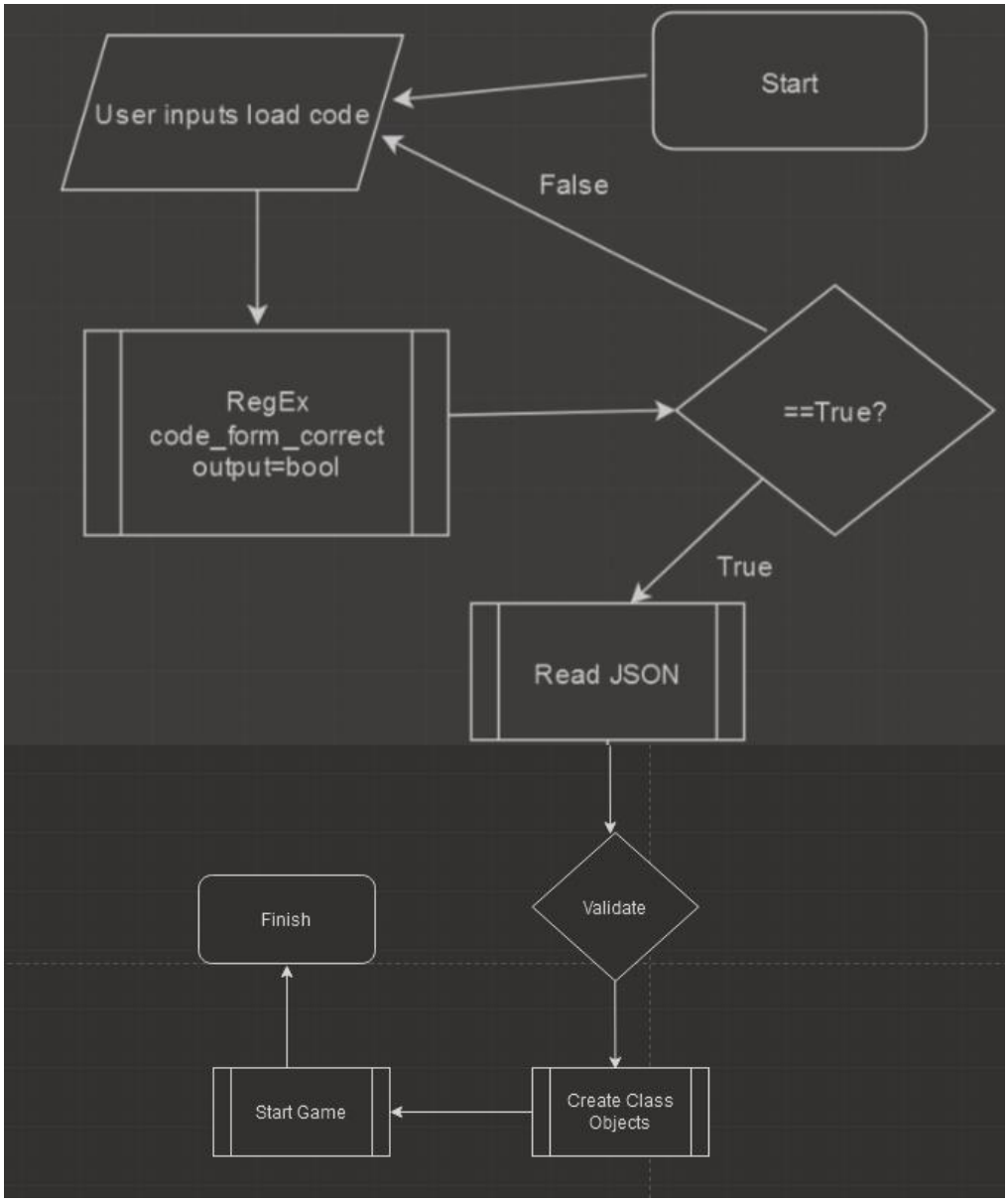
As the value for the dictionary entry, where the key is the save code. We previously validated the save code and added it to the hash table. These dump() lines will later be used to instantiate class objects when the user loads.

In the save feature, we utilise a **dictionary to create the hash table:**

- Dimension one refers to the type of object stored in the JSON file.
- Dimension two refers to the specific attribute of a given object stored in the JSON file.

Loading

The flowchart demonstrates the required sub-modules that I need to implement for the load feature.



The user must input the complimentary 1-1 save code to instantiate objects using attributes from the persistent data store.

Use of Lambda Function

First, we need to update the GUI.

```
290     while not play:
291         screen.blit(BACKGROUND, (0, 0))
292         generate_OSE(screen, _screen_width // 2 - 75, _SCREEN_height // 2 - 25, 150, 50, (200, 200, 200), (150, 150, 150), "Start", action=lambda: start_game())
293 
294         text_surface = font.render(f"Code: {input_code}", True, BLACK)
295         screen.blit(text_surface, (_screen_width // 2 - 75, _SCREEN_height // 2 + 50))
296 
297         pygame.display.update()
```

We achieve this using the ‘generate_OSE’ function that I created earlier in iteration two. The use of a lambda function ensures that the function is not called accidentally. The game loop runs continuously so that the system can take an input whenever the user acts.

Instantiating class objects using a persistent data store; String Formatting; RegEx

Loading from Persistent Data Store

Preliminary load_game function:

This function will be called whenever the user submits data in the box from before.

```
223 def load_game(code, eagle, obstacles):
224     try:
225         with open(SAVE_FILE, 'r') as file:
226             save_data = json.load(file)
227 
228         if code in save_data:
229             data = save_data[code]
230             eagle.reset(*data["eagle"])
231             obstacles.clear()
232             for obs in data["obstacles"]:
233                 obstacles.append(pygame.Rect(*obs))
234 
235             print(f"Game Loaded! Score: {data['score']}")
236             return data["score"]
237         else:
238             print("Invalid Code! No matching save found.")
239             return None
240     except (FileNotFoundError, json.JSONDecodeError):
241         print("No saved game found!")
242         return None
```

PROBLEMS

1. This currently acts as a pause feature not a save feature.
2. I saved a score before James had an unofficial play in class; the subroutine did not recognise my score.
3. The input is delicate e.g. letters can be used.

SOLUTION

1. There is a logic error. Line 236 should not use speech marks because '*"score"* 'is not the name of the entry, it is '*score'.*
2. Currently I am only checking for one parameter in 'save_data'.
3. Invalid formats need not be checked. We must validated the input. This is a concept I learned from Levitin's '*Algorithmic Puzzles'.*

Rigorous Validation

Here, we ensure that the code a user enters is of the correct format. I considered using RegEx but it is overkill here, since, by considering the chance of a random string of six characters being a valid save code, we realise that a valid code is rather particular. In contrast, RegEx is often used for broad criteria like email addresses.

```
217 def is_valid_code(code):
218     """
219     Validate that the code is exactly 6 digits long and contains only digits 0-9.
220     """
221     valid_digits = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}  # Set of valid digits
222 
223     # Check if the code has exactly 6 characters
224     if len(code) != 6:
225         return False
226 
227     # Check if every character in the code is a valid digit
228     for char in code:
229         if char not in valid_digits:
230             return False
231 
232     return True
```

Criteria validated against:

- Denary characters only
- Correct length

Validation with more rigorous Try:Catch:

```
235 def load_game(code):
236     global current_score
237 
238     try:
239         if not is_valid_code(code):
240             raise ValueError("Invalid code format. The code must be exactly 6 digits.")
241 
242         #Attempt to load the save data file
243         with open(SAVE_FILE, 'r') as file:
244             save_data = json.load(file)
245 
246         #Look for the code in the save data
247         for entry in save_data:
248             if entry["code"] == code:
249                 current_score = entry["score"]
250                 print(f"Game Loaded! Starting at Score: {entry["score"]}")
251                 return True
252 
253         #Code not found in save data
254         print("Invalid Code! No matching save found.")
255         return False
256 
257     except FileNotFoundError:
258         print("No saved game found! The save file does not exist.")
259         return False
260     except ValueError as e:
261         print(f"Error: {e}")
262         return False
263     except (json.JSONDecodeError, KeyError) as e:
264         print(f"Corrupted save data: {e}")
265         return False
```

Note the detailed try catch statement that aims to explain to the user the error, so that they may fix it.

This is the first major piece of information the user inputs. The user can do a lot wrong and they will need to be able to figure out what so that they can load their games.

```
Error: Invalid code format. The code must be exactly 6 digits.
Error: Invalid code format. The code must be exactly 6 digits.
Game Saved! Code: 697858
Invalid Code! No matching save found.
Game Loaded! Starting at Score: 1
Game Saved! Code: 531814
Game Loaded! Starting at Score: 3
Game Saved! Code: 377534
```

More working evidence can be found in Appendix Two: Video Evidence Folder

Test Table for Iteration Two

Test #	Game function	Input	Type of Input	Justify Test	Outcome	Review
1	Increment Score	Pass Pipe Obstacle	Valid	This is currently the only way to gain score.	The scoreboard increases its displayed value by 1.	Success
2	Invalid Score Increment	All other events e.g. pressing esc. or pressing space	Invalid	Since score is incremented by an iterative algorithm, it is crucial that the algorithm works correctly. This test ensures that the score is not only incremented when necessary, but that it does not increment unexpectedly.	Score is not incremented	Success
3	Load Game (valid code)	Entering save code 036742 (6 digit integer) where the save code exists in the JSON	Valid	This is how the user can load a game state.	The game is reloaded back to the desired game state. The save state is preserved in the JSON.	Success
4	Load Game (invalid code)	Entering save code ‘036742’ (6-digit integer), ‘A06461’ (non-integer), or ‘0461’ (non 6-digit integer) where the code does NOT exist in the JSON	Invalid	Invalid save codes should not create game states since they do not map to a JSON record.	Alert returned telling user that save code is incorrect. System accepts new save code to be input.	Success
5	Save Game using ‘k’	Press 'k'	Valid	Provide save shortcut	Save code displayed to user; feature success is determined by the load feature	Success
6	Invalid Save Game Key	Press unassigned key e.g. '&', space, or ‘@’	Invalid	Prevents unintended saves	No save created; input ignored	Success
7	Game Instance Ends on Collision	User eagle undergoes any mask collision	Valid	Lets the user save a ‘run’ to continue later	Game returns to menu screen.	Success
8	Load Game Menu Popup	Click ‘Load Game’ on the main menu	Valid	Let's user enter a save-load code.	Load Game Menu Pops up and the user can	Success

					immediately type in the load-game input bar.	
9	View Leaderboard	Click the Leaderboard button	Valid	Lets user view leaderboard entries.	Leaderboard pops up containing past scores from the JSON file.	Success
10	Invalid Leaderboard Access	Invalid input (e.g. keyboard press from menu)	Invalid	Invalid input makes sure the user doesn't accidentally get sent to the leaderboard.	No action occurs; input is ignored	Success
11	Enter Data into Leaderboard under unique name	Enter Name and Score (alphabetical and numerical)	Valid	Lets the user enter new high scores.	JSON file saves leaderboard entry.	Success
12	Enter Data into Leaderboard under non-unique name	Enter Name and Score (duplicate name)	Boundary	Lets the user enter new high scores under coincident names.	User is prompted to update the record under the same name or to create a new entry.	Success
13	Enter non-alphabetical name	Enter name using characters like £$%&(	Invalid	Ensures names only contain valid characters	The name is rejected, and the user is alerted that the name is non-alphabetical. The user is prompted for another input.	Success
14	Enter excessively long name	Name is more than 30 characters long	Boundary	Ensures user cannot enter names so long that the leaderboard becomes unreadable.	The name is rejected, and the user is alerted that the name is too long. The user is prompted for another input.	Success
15	Overwrite entry with larger score	Type 'Y' AND new entry is larger	Valid	Lets the user update their own entry	Leaderboard entry under same name is overwritten with new score.	Success
16	Overwrite entry with smaller score	Type 'Y' AND new entry is not larger	Boundary	Prevents the user from updating their own entry if it's worse.	Alert returned telling user that their new score is lower than the previous input. The leaderboard entry remains unchanged.	Success
17	Decline overwrite of score	Type 'N' after name conflict	Valid	Creates Duplicate	Creates a leaderboard entry with same name.	Success
18	Submit incomplete	Enter only name or only score	Valid	Makes sure that the user cannot enter an	Alert returned stating that an entry is missing. The	Success

	leaderboard entry			incomplete entry to the leaderboard.	user is prompted for another input.	
19	Scroll leaderboard	Arrow	Valid	Lets user look through different leaderboard entries	Leaderboard scrolls as expected.	Success

Stakeholder Review Two – Were the Objectives Achieved?

Using James' mathematical knowledge, constants for the fatigue algorithms have been decided through a series of 10 tests, where he provided feedback on proposed constants.

Regarding my alteration of the import lines, we've concluded that the inconvenience of importing separately is trivial and is worth the increase in performance.

Both Oliver and James commented that the motion of the user could be improved. The motion feels unrealistic. At first, we thought that the motion gave the game uniqueness, but I aim to preserve this uniqueness while making the motion more natural.

Oliver tried to get around the validating algorithms with non-alphanumeric characters; fortunately, as Levitin suggests in his Algorithmic Puzzles, validation should specify inclusion, not exclusion, so, the algorithm did not fail.

James said the load-save feature was extremely helpful and made progress a lot easier. The only concern he had was that adding features to this game would make the saving feature inflexible. However, I explained to him that the use of records facilitated the saving of additional data.

James wants the input decisions and save/load feature to be more rigorous. Currently direct input e.g. in the leaderboard is the weakest point in the system. Given that there was discussion about switching to mySQL for the persistent store of data for scalability reasons, this is pertinent to protect the system from SQL injection.

James queried my use of a unique save code rather than importing a file, but I explained that the use of files would not allow people to play the same 'run' on more than one device.

Both James and Oliver recommended that I implement a game-over menu for a more rounded user experience. At first I thought that restarting automatically would drive user engagement but after discussion, I realised that all professional games have this feature because it tells the user their score rather than just restarting automatically.

Overall, James wanted a final dash of flare to the game; with more accurate mimicry of physical phenomena. James wanted to see my implementation of Enemies from design because he complains that '*Elegant Eagles* becomes a bit repetetive after playing it for an hour'.

Agile Development Iteration Three

Objectives

1. Create a Game Over Screen

    (i) Show Score
    (ii) Accept inputs to restart/quit

2. Develop the **mathematics** and physical **modelling** using realistic formulae and laws:
    (i) Jumps
        (a) Jump fatigue

        (b) Super jumps

    (ii) **Air resistance**

    (iii) Using formulae for **Stokes Law and buoyant forces** to develop to develop realism.
        (b) Perhaps I can create a 'water' stage where the buoyant force and velocity changes dramatically. This could be an extension of one of the eagle characters below.

    (iv) **Randomised wind and rain with increasing magnitude**

        (a) Parent object class for weather, subclasses using polymorphism and inheritance for rain and wind, where rain varies with wind.

        (b) Fixing the save/load feature to reflect these changes

3. Add a **difficulty level** that effects various obstacles and enemies in different ways.
    (i) Suitable input methodology
        (a) Rigorous and suitable **validation** of this input
    (ii) Suitable effects

4. Employ inheritance and polymorphism to create real-time character swapping

5. Develop the leaderboard using the persistent data store:

    (i) **Rigorous validation**

    (ii) Instantiate leaderboard class object

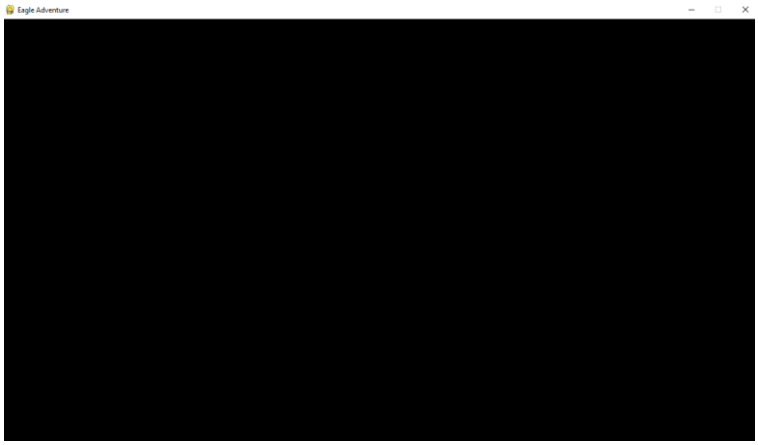
6. Create an enemy using a suitable **pathfinding** algorithm that chases the user:
    (i) Justify the use of **A*** over other algorithms
    (ii) Tailor intentional input lags and **heuristic** choice to control the enemy difficulty.
        (a) Utilise extensive stakeholder testing
    (iii) Implement A* on a **dynamic** graph using the **coordinates of obstacles and the user**.

Creating a Game Over Screen

We need to create a Game Over Screen because it will create a satisfying end for the user. Additionally, if we automatically start a new run, then the user will not be able to view their old score.

Creating game_over()

```
def game_over(score):
    #display game over screen with the final score
    screen.fill(BLACK)
```



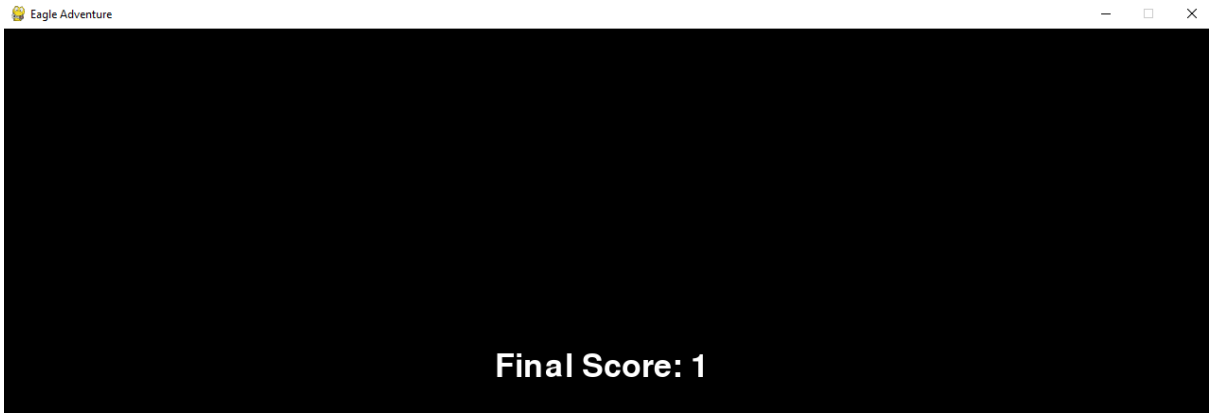
First, we create the black background for the window. This gives us a 'clean slate' without removing any OSE elements; we are essentially 'painting over' everything. We use the colour BLACK, one of my global constants; this makes the program more readable, because those inexperienced with PyGame might be unfamiliar with what parameter 'n' means in PyGame's methods such as .fill().

```
game_over_font = pygame.font.SysFont('freesansbold', 50)
game_over_text = game_over_font.render("Game Over", (240,240,240),True)
```

Next, we setup the text to be displayed. First, we create the font, freesansbold appears to be the industry standard. I observed this by comparing text from the Games from the Analysis stage with fonts provided by a WordPress software, 'Word'. Note that the size of text, 56, and the colours 240, 240, 240, were chosen using stakeholder feedback and the methodology laid out in Design: Development Plan.

```
score_text = game_over_font.render(f"Final Score: {score}", True, (240, 240, 240))
screen.blit(game_over_text, (SCREEN_WIDTH // 2 - 100, SCREEN_HEIGHT // 2 - 50))
screen.blit(score_text, (SCREEN_WIDTH // 2 - 120, SCREEN_HEIGHT // 2 + 20))
pygame.display.update()
```

Next, we return the score to the user. The second parameter 'True' just means that the render should be executed. Then, we utilise the .blit() method to produce the visible effect. As in iteration one, we use the global variables SCREEN_WIDTH and SCREEN_HEIGHT because it makes *Elegant Eagles* **scalable** to all devices; not just my own device, this will be pertinent to enhance user immersion in **Beta Tests and beyond.**



PROBLEM:

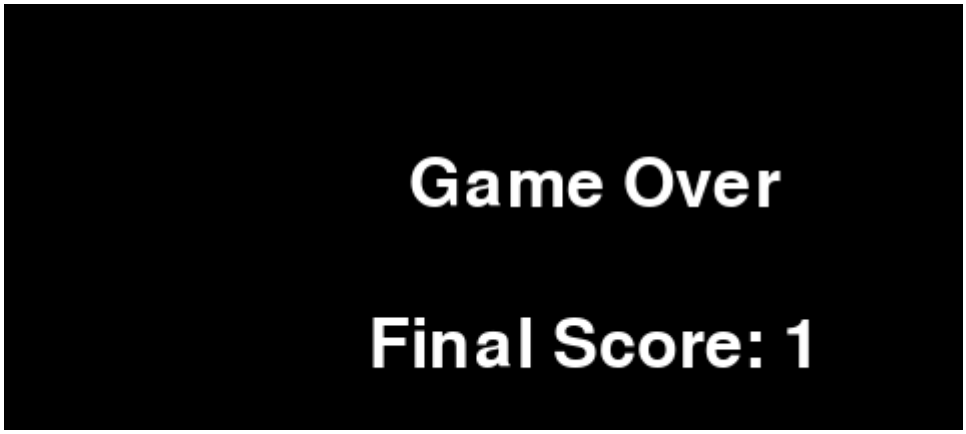
The 'Game Over' text is not appearing.

SOLUTION:

```
game_over_text = game_over_font.render("Game Over", (240,240,240),True)
```

By looking at the PyGame documentation (see References), I realised that I was passing the parameter in the wrong order. Then, the solution was a simple matter of swapping the second and third parameter.

```
game_over_text = game_over_font.render("Game Over", True, (240,240,240))
```



This fix produced the desired effect.

PROBLEM:

The user is not given a prompt on what to do next

SOLUTION:

We want the user to be able to start the game.

First, we will add instructions in the exact same way as we added the Game Over text.

```
instruction_text = game_over_font.render("Press SPACE to Restart or ESC to Quit", True, (240,240,240))
```

```
screen.blit(instruction_text, (SCREEN_WIDTH // 2 - 300, SCREEN_HEIGHT // 2 + 60))
```

Next, we need to handle the cases of the various results.

We use the built in method pygame.event.get() to detect inputs. Using the same documentation as before we know how to refer to each key.

```
#wait for user input to restart or quit
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            exit()
        if event.type == KEYDOWN:
            if event.key == K_SPACE:  #restart the game
                reset_game()
                game_loop()
                return
            if event.key == K_ESCAPE:  #quit the game
                pygame.quit()
                exit()
```

Conditions For Game Over

In the same if statement where we handle the case of User Eagle mask collision, we will call the game_over() subroutine which will produce the GUI.

```
if (pygame.sprite.groupcollide(eagle_group, ground_group, False, False, pygame.sprite.collide_mask) or
        pygame.sprite.groupcollide(eagle_group, obstacle_group, False, False, pygame.sprite.collide_mask)):
    game_over(current_score)
    return  # End the game loop
```

PROBLEM:

Escape isn't closing the game, I get this error

```
NameError: name 'K_ESCAPE' is not defined
```

SOLUTION:

```
from pygame.locals import QUIT, KEYDOWN, K_SPACE, K_UP, K_k, K_1, K_2, K_3,K_ESCAPE
```

I forgot to let PyGame detect the escape key as an event.

Returning Save Codes

We already have the following save algorithm:

```
def save_game(code, eagle, obstacles, score):
    try:
        # Attempt to load existing save data
        with open(SAVE_FILE, 'r') as file:
            save_data = json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        # If file doesn't exist or is corrupted, start with an empty dictionary
        save_data = {}

    # Validate inputs before saving
    if not isinstance(code, str) or not code:
        print("Invalid save code! Must be a non-empty string.")
        return
    if not hasattr(eagle, 'x') or not hasattr(eagle, 'y'):
        print("Invalid eagle object! Missing coordinates.")
        return
    if not isinstance(score, (int, float)) or score < 0:
        print("Invalid score! Must be a non-negative number.")
        return

    # Store game data in dictionary
    save_data[code] = {
        "eagle": (eagle.x, eagle.y),  # Store eagle's position
        "obstacles": [(obs.x, obs.y, obs.width, obs.height) for obs in obstacles],  # Store all obstacles
        "score": score  # Store current score
    }

    # Write updated data back to file
    with open(SAVE_FILE, 'w') as file:
        json.dump(save_data, file)

    print(f"Game Saved! Code: {code}")
```

Now, by using the exact same method, we can improve the method by which the user reads the save code. Previously we returned the result to the IDE, this is not user friendly. Instead, we create a simplified version of the previous GUI:

```
def print_code(code):
    #display final score on screen
    screen.fill(BLACK)
    score_font = pygame.font.SysFont('freesansbold', 50)
    score_text = score_font.render(f"final score: {score}", True, (255,255,255))
    screen.blit(score_text, (SCREEN_WIDTH // 2 - 120, SCREEN_HEIGHT // 2 - 20))
    pygame.display.update()
```

Then, we call the print_code() subroutine at the end of the save code, passing the code as a parameter.

```
# Store game data in dictionary
save_data[code] = {
    "eagle": (eagle.x, eagle.y),  # Store eagle's position
    "obstacles": [(obs.x, obs.y, obs.width, obs.height) for obs in obstacles],  # Store all obstacles
    "score": score  # Store current score
}

# Write updated data back to file
with open(SAVE_FILE, 'w') as file:
    json.dump(save_data, file, indent=4)

print_code(code)
```

This is the resulting window:

For demonstrations, see Appendix Entry Two: Video Evidence Folder.

Using Physics and Mathematics to Develop Modelling

Jump Fatigue

The input is binary, a space or an up arrow to jump, with a small cooldown to improve skill expression. 'Fatigue' will be a measure of the recent mean frequency of jumps and will be computed as follows:

$$\Delta Fatigue = k(\Delta t - t_{cooldown})^n$$
$$where\ \Delta t = t - t_{lastframe}$$

$$and\ at\ every\ iteration\ of\ the\ game:$$
$$fatigue = fatigue + \Delta$$

The purpose of 'jump fatigue' is to stop players from spamming the space button. This adds a layer of complexity to the game. The jump fatigue feature will be more severe at higher difficulties. I came up with jump by observing muscle fatigue in animals.

Note that this is a complete approximation of how the mechanic might work. 'n' is some arbitrary power such that its upper bound is derived from ignoring fatigue at the game's early stages and its lower bound to reward good timing with 'k' for the finer precision exponentiation strips.

These constants will be decided at my next stakeholder meeting with James through a micro-agile development lifecycle of repeated prototypes and improvements.

User Eagle Attributes

First, we must create several attributes in the User Eagle class.

```
self.vertical_speed = 0
self.gravity_force = PULL_FORCE
self.score = 0
self.is_playing = False
self.fatigue = 0 #Fatigue timer
self.last_jump_time = 0  #Track last jump time
self.k = 0.5 #Fatigue scaling factor determined through stakeholder feedback
self.n = 2#Exponential fatigue growth
self.cooldown = 0.5 #Base cooldown time
```

The purpose of these attributes will become obvious.

Validating Jump Request

To determine whether or not the user can jump, we could decrement the jump fatigue using a clock and then swap a Boolean variable allowing the jump.

However, this is an expensive operation with respect to time because we would need to call it every time the game loop passes. Instead, we will store the previous jump time as an attribute of the user eagle and check if sufficient time has passed using the .time() method.

```
def can_jump(self):
    """Checks if the player can jump based on fatigue."""
    return time.time() - self.last_jump_time >= self.fatigue
```

Next, we need to handle the validation and the jump thereafter.

```
if self.can_jump():
            self.is_playing = True
            self.vertical_speed = -FLIGHT_SPEED
            current_time = time.time()
            delta_t = current_time - self.last_jump_time
            self.last_jump_time = current_time
```

As in development phase 1, we must accelerate the user eagle. However, we must also update delta_t: the time since last jump.

Varying fatigue

$$\Delta\text{Fatigue} = k(\Delta t - t_{\text{cooldown}})^n$$

We must use this formula to increment fatigue.

```
if delta_t < 1.5:
    self.fatigue += self.k * (delta_t - self.cooldown) ** self.n
else:
    self.fatigue = max(0, self.fatigue - 0.1)
```

Here, n and k are attributes which will be set using the difficulty slider.

PROBLEM:

Fatigue appears to be acting erroneously. At uniform velocity, the cooldown time due to fatigue seems to vary unexpectedly. This variance was on a file scale; less than one half of a second.

SOLUTION:

Since the behaviour seemed random, I suspected inherent inconsistencies in processor run time due to background tasks, like checking for an internet connection or handling system interrupts. Such operations would cause these minute variations.

So, I suspected that the issue was exponentiating to with t. This is because t is calculated using a separate clock loop, and fatigue is implemented in the loop that handles the motion of the user eagle. As a result, there is not proportion between the time that has passed and the number of .update() calls.

Since we cannot merge the clock and update loops due to scope issues, we should increase fatigue every time the game loop (and therefore the update method) is called to create this consistency, by eliminating the reliance on the separate clock loop.

```
if delta_t < 1.5:
    self.fatigue *= 1+self.k * (delta_t - self.cooldown) * self.n
else:
    self.fatigue = max(1, self.fatigue * 0.9)
```

This conserves the function of jump fatigue, since we have created a geometric series rather than an exponential function - *since one cannot program a truly continuous function, the exponential version is but a glorified geometric series.*

Encapsulation

OOP Scaffolding for Future Development Iterations

My use of OOP has made the program behind *Elegant Eagles* more **modular**; reusable; readable; and **maintainable**.

However, by using encapsulation, we can make the program even more **maintainable** and **secure**. The latter is of critical importance if Elegant Eagles is to extend to an account system given the sensitive nature of login emails and passwords**.**

Specifically, by privatising attributes and restricting their access to get() and set() methods, we can reduce the probability of unintentional access and modification of said attributes.

Sprite Base Class

The Sprite Base Class is the global parent class in the class hierarchy. Therefore, privatising attributes here has the most significant effect, and is most likely to cause errors.

Constructor Method:

Before, this class looked like:

```
class SpriteBase(pygame.sprite.Sprite):
    def __init__(self, image_path, xpos, ypos, width, height):
        super().__init__()
        self.image = pygame.image.load(image_path).convert_alpha()
        self.image = pygame.transform.scale(self.image, (width, height))
        self.mask = pygame.mask.from_surface(self.image)
        self.rect = self.image.get_rect()
        self.rect[0] = xpos
        self.rect[1] = ypos
```

So, I privatised the attributes using the underscore syntax:

```
class SpriteBase(pygame.sprite.Sprite):
    def __init__(self, image_path, x_pos, y_pos, width, height):
        super().__init__()
        self._image = pygame.image.load(image_path).convert_alpha()
        self._image = pygame.transform.scale(self._image, (width, height))
        self._mask = pygame.mask.from_surface(self._image)
        self._rect = self._image.get_rect()
        self._rect.x = x_pos
        self._rect.y = y_pos
```

I then implemented the appropriate get() and set() methods. Given that the SpriteBase class is never instantiated, only inherited from, set() methods are not necessary.

```
def get_rect(self):
    return self._rect

def get_mask(self):
    return self._mask

def get_image(self):
    return self._image
```

An example of a set method can be found for my User Eagle class:

```
def set_score(self, value):
    self._score = value
```

PROBLEM:

A part of my program tried to access the coordinates of the Eagle object but failed.

```
  File "C:\Users\agraw\AppData\Local\Programs\Thonny\lib\site-packages\pygame\sprite.py", line 573, in draw
    surface.blits(
  File "C:\Users\agraw\AppData\Local\Programs\Thonny\lib\site-packages\pygame\sprite.py", line 574, in <genexpr>
    (spr.image, spr.rect, None, special_flags) for spr in sprites
AttributeError: 'Eagle' object has no attribute 'rect'
```

SOLUTION:

Unfortunately, my IDE returned line numbers of programs that don't exist.

Therefore, I had to deduce given the recent changes, what might have caused this issue.

Given that this is a new error, then I suspected that the solution would be found in the cascading use of get() methods.

First, I verified if this issue was unique to the *rect* attribute. Given that *rect* is the first attribute from the Eagle object that is called (for the .update() method), it is likely that this error is not unique to rect.

The get_rect() and get_image() methods are used to retrieve the rect and image attributes, respectively. However, Pygame's Sprite class expects these attributes to be directly accessible (self.rect and self.image) rather than through getter methods.

Then, I had to revert the constructor method:

```
class SpriteBase(pygame.sprite.Sprite):
    def __init__(self, image_path, x_pos, y_pos, width, height):
        super().__init__()
        self.image = pygame.image.load(image_path).convert_alpha()  # Directly expose 'image'
        self.image = pygame.transform.scale(self.image, (width, height))
        self.rect = self.image.get_rect()  # Directly expose 'rect'
        self.rect.x = x_pos
        self.rect.y = y_pos
        self.mask = pygame.mask.from_surface(self.image)  # Directly expose 'mask'
```

Note that these comments are not part of the final code (see Appendix Entry Two: Program Code). The comments serve to clarify the solution to this problem through annotation.

Comments

In addition, I made developments to my comments and use of line breaks to improve the readability for other programmers. This project was my first exposure to PyGame and I found many of the method names unintuitive. Therefore, this readability is especially important for other programmers who are less versed in PyGame.

For example, the Sprite Base Class from above:

```
class SpriteBase(pygame.sprite.Sprite):
    def __init__(self, image_path, x_pos, y_pos, width, height):
        super().__init__()  #initialize the parent sprite class

        #load and scale the image
        self._image = pygame.image.load(image_path).convert_alpha()
        self._image = pygame.transform.scale(self._image, (width, height))
        self.image = self._image  #pygame requires 'image' to be public for rendering

        #create a mask for better collision detection
        self._mask = pygame.mask.from_surface(self.image)

        #set up the sprite's hitbox
        self._rect = self.image.get_rect()
        self._rect.x = x_pos  #set X position
        self._rect.y = y_pos  #Set y position
```

This project has taught me that the use of line breaks can make intimidating blocks of code more approachable.

Stokes' Law on Aerodynamic Drag as a function of velocity; Inheritance; Polymorphism

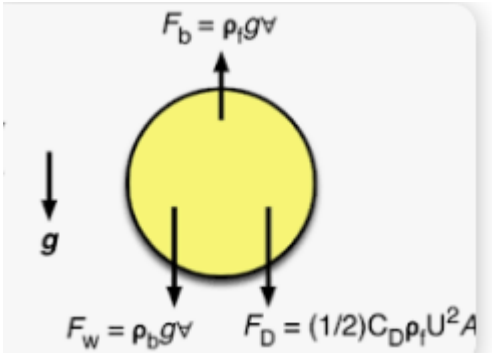
Air resistance

As per my **stakeholder** James' advice, I aim to improve the **physics** of my project. I aim to incorporate fluid resistance via Stokes' Law. This is inspired by the forces that act a real bird due to its adjacent surface area and velocity.

In addition, we will consider buoyant force as a function of volume and fluid density which varies with inverse proportion with vertical displacement.

$$F_{drag} = \frac{1}{2}\rho v^2 C_{drag} A$$

$F_{drag} = Drag\ force$
$\rho = Fluid\ density$
$v = Speed\ of\ object\ relative\ to\ the\ flui$
$C_{drag} = Drag\ coefficient$
$A = Cross\ sectional\ area\ of\ object$



In this diagram, the user eagle moves upwards. The subscript:

- b means upthrust
- w means weight
- d means drag

To summarise the physics:

Via Newtons 2nd Law

$$F_U = F_D = Drag + Weight = Buoyant\ \ Force$$

$$VP_b g + 6\,\pi r n v = P_f (gV)$$

I used the help of my stakeholder Mr Kettle to verify the accuracy of my formulae.

We decided that the buoyant force was inconsequential since its relative density is low.

Since we don't really exert forces, but we cause accelerations in Elegant Eagles, I decided that instead of using specific formula

Calculating the Area

We could use the **dot product** to determine the component of the velocity vector, **v** that is perpendicular to the face of the bird, **π**.

If we define the face of the bird as:

$$\mathbf{r} = \mathbf{a} + \lambda\mathbf{b} + \mu\mathbf{c}$$

Then we can use the cross product to determine the normal to the face:

$$\mathbf{n} = \mathbf{b} \times \mathbf{c}$$

And normalise to give

$$\hat{\mathbf{n}} = \frac{\mathbf{b} \times \mathbf{c}}{|\mathbf{b} \times \mathbf{c}|}$$

We would then consider a suitable triangle and use the dot product:

$$\mathbf{v} \cdot \hat{\mathbf{n}} = |\mathbf{v}| \cos\theta$$

However, given that we are working in 2D space and that we do not know how the horizontal velocity scales with respect to the vertical velocity, it makes more sense to consider only vertical drag and model the user eagle as a horizontal plane.

Updating the User Eagle Constructor Method

First, we need to add a drag coefficient to the user eagle's constructor method.

```
33    # Base class for the eagle
34    class Eagle(pygame.sprite.Sprite):
35        def __init__(self, speed=SPEED):
36            super().__init__()
37            self.image = pygame.image.load('user_image.png').convert_alpha()
38            self.image = pygame.transform.scale(self.image, (30, 30))
39            self.rect = self.image.get_rect()
40            self.rect.x = SCREEN_WIDTH // 6
41            self.rect.y = SCREEN_HEIGHT // 2
42            self.speed = 0
43            self.gravity = GRAVITY
44            self.drag_coeff = DRAG_COEFF
45            self.started = False
```

Note that, the drag_coefficient is really given by:

$$D_C = k_D * A$$

This simplifies the number of attributes that the user eagle has.

Next, we need to accelerate the user eagle.

We do this by adding the drag force to the update() method. This means that the force of drag will be calculated every iteration of the game loop and then applied to the user eagle. One possible addition to future development iterations is to make detailed models for all OSEs and implement drag for them.

```
def update(self, fluid="air"):
    # updates the eagle's position based on speed, gravity, and fluid dynamics
    if self._started:
        # Dictionary storing drag coefficients for different fluids
        fluid_drag_coefficients = {
            "air": 0.005,
            "water": 0.008,
            "honey": 0.01
        }

        # Choosing the fluid as the medium
        k = fluid_drag_coefficients[fluid]  # Drag coefficient

        # Compute drag force, opposing motion (F = kV^2)
        drag_force = k * self._speed**2 * self._speed/abs(self._speed)

        # Apply forces: gravity pulls down, drag resists motion
        self._speed += self._gravity + drag_force

        # Update position
        self.rect[1] += self. speed
```

Since we have added an attribute to the update() method, we need to cascade this change to all existing calls of the update() method:

The only call of the update() method is in the game_loop:

```
# Update groups
eagle_group.update("air")
ground_group.update()
obstacle_group.update()
enemies_group.update(eagle, obstacle_group)
```

In this case, passing a fluid is redundant, but when we specify the fluid we are moving in it will be necessary.

PROBLEM:

The object falls too fast and we get this error

```
  File "C:\Users\agraw\AppData\Local\Programs\Thonny\lib\site-packages\pygame\sprite.py", line 1689, in collide_mask
    return leftmask.overlap(rightmask, (xoffset, yoffset))
TypeError: offset must be two numbers
```

SOLUTION:

By running the program in test mode by temporarily importing the program to a device that uses the paid IDE, PyCharm.

```
Speed: -20, Drag Force: -80.0
Speed: -19.2, Drag Force: -73.728
Speed: -18.4, Drag Force: -67.584
```

I noticed that the magnitude of the Drag Force was far too severe. In addition, the drag force was always negative, no matter if the user jumped or not. Given that up is the positive direction, this indicates that:

- The plus/minus mathematics of the drag force isn't implemented correctly.
- The magnitude of the force is too large so the coefficient of drag needs to be reduced.

Typically, I would figure out the exact nature of constants through stakeholder feedback at the end of a development iteration. However, given the length of the iteration, I informally consulted my stakeholders James and Oliver. Using the methodology:

1. Try the coefficient
2. Through democratic feedback and discussion, we voted to either, increase, settle, or decrease the value.
3. If the output of 2. Is not 'settle', go to step 1.|

We determined more suitable coefficients:

```
fluid_drag_coefficients = {
    "air": 0.005,
    "water": 0.008,
    "honey": 0.01
}
```

To make sure that the plus/minus of the direction of the drag force works correctly, I had to analyse this line of code.

```
drag_force = k * self._speed**2 * self._speed/abs(self._speed)
```

It turns out that, since speed is a scalar quantity, the value of speed/abs(speed) is always one.

Therefore, instead, we replace the last term in the product with:

```
drag_force = k * self._speed**2 * (-1 if self._speed > 0 else 1)
```

It turns out that the error message went away when I fixed these two issues. Upon further investigation about the error, I realised that it was due to the exponential nature created by the rapid iteration: a downwards acceleration generated a large downward drag which in itself acts as a downwards acceleration, causing a downward drag and so on. A demonstration of this change can be found in Appendix Entry Two: Video Evidence Folder.

Polymorphism and Inheritance

Randomised Wind and Rain

To enhance gameplay unpredictability and realism, two environmental effects are introduced:

1. Wind – A dynamic force that influences the eagle's horizontal movement, making navigation more challenging. The wind force will change direction and magnitude over time, using trigonometric functions to create smooth oscillations rather than abrupt changes.
2. Rain – Affects air resistance by slightly modifying the drag coefficient, making movement more inconsistent.

Both features demonstrate how *Elegant Eagles* is a sophisticated and **realistic** program.

Class Attributes

First, we need to add private attributes to securely store the direction and magnitude of the force due to wind. In addition, we need an attribute to record the extent to which the drag coefficient will vary. As before, we privatise these attributes to improve the security of data and program maintainability.

```
class Eagle(SpriteBase):
    def __init__(self, speed=SPEED):
        super().__init__('user_image.png', SCREEN_WIDTH // 6, SCREEN_HEIGHT // 2, 30, 30)
        self._speed = 0  #current speed of the eagle
        self._gravity = GRAVITY
        self.__score = 0
        self._started = False  #flag to check if the game has started
        self._wind_force = 0  # wind force acting on the eagle
        self._wind_direction = 1  # 1 for right, -1 for left
        self._wind_magnitude = 0  # strength of the wind
        self._rain_effect = 0  #random variation in drag coefficient due to rain
```

The notable/non-intuitive additions are:

- The effectively boolean wind_direction attribute. In a later, method, we will compute the angle through which we exert the wind force, and this variable controls whether that angle is turned through clockwise vs. anticlockwise.
- The rain_effect attribute is a non-negative float. This float controls the severity of the variations in the coefficient of drag.

The relevance of these attributes to the inner workings is below.

Adding Methods

Wind:

```
def apply_wind(self):
    # Randomly generate wind magnitude and direction
    self._wind_magnitude = random.uniform(0.1, 0.5)  # Random wind strength
    self._wind_direction = random.choice([-1, 1])  # Random direction (left or right)

    # Smooth oscillation using trigonometry
    self._wind_force = self._wind_magnitude * self._wind_direction * math.sin(pygame.time.get_ticks()
```

Here, we utilise the random.uniform method to big a number in a continuous interval. The numbers 0.1 and 0.5 were chosen using stakeholder review via the methodology laid out in Design: Development Plan

Then, we utilise the get_ticks() method to make the continuous and smooth sinusoidal sin() graph vary **gradually**.

Rain:

```
def apply_rain(self):
    # Randomly vary the drag coefficient of air by a small amount
    self._rain_effect = random.uniform(-0.0001, 0.0001)  # Small random variation
```

We define the rain attribute similarly. The notable difference here is that rain_effect can be negative. This is because there is no binary direction attribute, instead, we use negative numbers to represent a *decreasing* drag coefficient (and positive numbers to represent increasing).

<u>Update Method</u>

Finally, we must actually exert the jerk on the User Eagle.

```
def update(self, fluid="air"):
    if self._started:
        # Drag coefficients for different fluids
        fluid_drag_coefficients = {
            "air": 0.005 + self._rain_effect,
            "water": 0.008,
            "honey": 0.01
        }

        # Select drag coefficient based on the medium
        k = fluid_drag_coefficients[fluid]

        # Compute drag force opposing motion
        drag_force = k * self._speed**2 * (-1 if self._speed > 0 else 1)

        # Apply forces: gravity pulls down, drag resists motion
        self._speed += self._gravity + drag_force

        # Apply wind force horizontally
        self.rect[0] += self._wind_force

        # Update vertical position
```

Here, note that we only add the rain effect to the 'air' fluid. This is because in real life water only acts in the air; this adds variety to the game; and most importantly, the relative effect from the previous image is negligible in denser fluids, therefore, in **future** development iterations, we could implement a scale factor rather than arithmetic change for the sake of **scalability**.

Finally, we must the wind and rain methods to the game_loop(), as described by the comments.

```
    if is_off_screen(obstacle_group.sprites()[0]):
        obstacle_group.remove(obstacle_group.sprites()[0])
        obstacle_group.remove(obstacle_group.sprites()[0])
        current_score += 1
        obs = get_random_obstacles(SCREEN_WIDTH * 2)
        obstacle_group.add(obs[0])
        obstacle_group.add(obs[1])

    # Generate enemies based on score
    generate_enemy(current_score, enemies_group)

    # Apply wind and rain effects
    eagle.apply_wind()
    eagle.apply_rain()

    # Update groups
    eagle_group.update("air")
    ground_group.update()
    obstacle_group.update()
    enemies_group.update(eagle, obstacle_group)
```

A demonstration of this change can be found in Appendix Entry Two: Video Evidence Folder.

Validation

Improving Program Validation

Throughout my program, there are instances of validation used to improve the maintainability, usability and security of *Elegant Eagles*. However, there are a few instances of validation that I did not think to implement in previous development. Here, we address these:

Refer to the section '*Testing to Inform Evaluation'* for test cases throughout.

Set Score

The Set Score method is used to instantiate a game object from the persistent data store. The score must be a natural number. Therefore, we add the second line in the below screenshot.

```
def set_score(self, value):
    if not isinstance(value, (int)) or value < 0:
        raise ValueError("Score must be a non-negative number.")
    self.__score = value
```

Update

The Update method is used to apply jerk to the User Eagle object.

```
def update(self, fluid="air"):
        if self._started:
            # Validate fluid type
            if fluid not in ["air", "water", "honey"]:
                raise ValueError("Invalid fluid type. Must be 'air', 'water', or 'honey'.")
```

Here, we validate that the fluid given is from the existing list of fluids. For the sake of **scalability**, I then created a dictionary of fluids changed the third line to:

```
if fluid not in fluid_list:
```

So that if additional fluids are to be added, only the fluid list needs to be updated, rather than all references to a fluid.

In addition, we validate the coefficient of drag:

```
for key, value in fluid_drag_coefficients.items():
    if not isinstance(value, (int, float)) or value < 0:
        raise ValueError(f"Drag coefficient for {key} must be a non-negative number.")
```

This is because the coefficient of drag must be positive since drag is a resistive force. There is a detailed alert returned for the sake of bug fixing.

Then we repeat the exact same things with the rest of the attributes in the Update() method. These exemplify the changes I made to other methods for example in the rest of the Eagle Class.

```
# Compute drag force, opposing motion (F = kV^2)
if not isinstance(self._speed, (int, float)):
    raise ValueError("Speed must be a valid number.")
drag_force = k * self._speed**2 * (-1 if self._speed > 0 else 1)

# Apply forces: gravity pulls down, drag resists motion
if not isinstance(self._gravity, (int, float)) or self._gravity < 0:
    raise ValueError("Gravity must be a non-negative number.")
self._speed += self._gravity + drag_force

# Apply wind force
if not isinstance(self._wind_force, (int, float)):
    raise ValueError("Wind force must be a valid number.")
self.rect[0] += self._wind_force

# Update position
if not isinstance(self._speed, (int, float)):
    raise ValueError("Speed must be a valid number.")
self.rect[1] += self._speed
```

Difficulty Meter

Next, we try to implement an intuitive way to set a difficulty level within a range of **0 to 10**. This meter allows users to input difficulty through either:

1. **A text entry field**: a faster method because entry is direct
2. **A slider**: an easier method because it is impossible to unsuccessfully validate

Flexible and Validated Input

Handling the Input

Validation with Try: Catch:

Validating User Input

```
def validate_input(user_input):
    try:
        difficulty = int(user_input)  #attempt to convert input to an integer
        if 0 <= difficulty <= 10:  #check if it's within the valid range
            return difficulty  #return the valid difficulty level
        else:
            return None  #return none if out of range
    except ValueError:
        return None  #return none if input is not a valid integer
```

This is the algorithm that we will use to validate direct entry from the test field. I chose the range zero to ten because, using stakeholder feedback and from the Design stage, when people want to 'rate' something they usually rate it out of 10. We will test the intuitiveness of this in the Evaluation section.

The ValueError case returns an error if a non-integer input is given.

To see test cases, see the *Final Beta Testing* section.

Creating the Slider

```
import tkinter as tk
from tkinter import ttk, messagebox
```

```
def update_difficulty_meter(difficulty):
    meter_bar.config(length=difficulty * 20)  #scale the progress bar length based on difficulty
    meter_label.config(text=f"difficulty level: {difficulty}/10")  #label to reflects difficulty
```

This algorithm prepares the slider for change.

Here, we use the library, tkinter, for its built-in slider input. Here, we let the length of the bar vary in proportion to the difficulty to create a striking visual effect. The constant 20 was determined through stakeholder testing via the methodology laid out in Design.

Note that the use of tkinter is particular suitable since, otherwise, the slider would appear discrete rather than continuous. Implementing a sliding animation requires animation skills that I do not have – even if I did, the use of a suitable library improves readability, decreases the likelihood of error and saves time.

```
def on_slider_change(value):
    difficulty = int(float(value))  #convert the slider value to an integer
    update_difficulty_meter(difficulty)  #update the difficulty meter
```

We then use this algorithm to execute the slide.

Creating the Input Text Box

```
def on_text_input():
    user_input = entry.get()  #get user input from the entry field
    difficulty = validate_input(user_input)  #validate the input
    if difficulty is not none:  #ensure the input is valid before proceeding
        update_difficulty_meter(difficulty)  #update the meter with valid difficulty level
    else:
        #show an error message if input is invalid
        messagebox.showerror("invalid input", "please enter an integer between 0 and 10.")
```

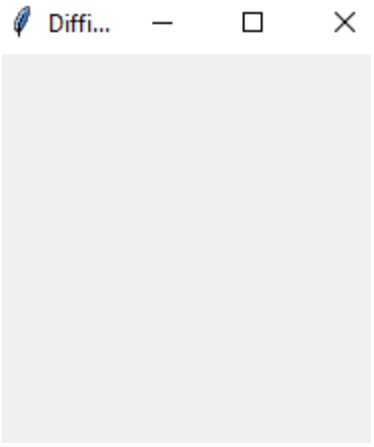
Here, we call the validation method from before to make sure that the user enters a valid input. The validation method returns True if and only if the input is valid.

Note that line four checks if the difficulty is ‘not None’ instead of just checking if it is true or false. This is because we can pass a difficulty of zero which would evaluate to false as a Boolean, but zero is a valid input. Other false inputs like “ “are not valid, hence, the need for validation at all.

Creating the GUI

```
# Create the main window
root = tk.Tk()
root.title("Difficulty Meter")
```

Using tkinter’s built in modules, we created a blank page.



Next, we need to accept an input.

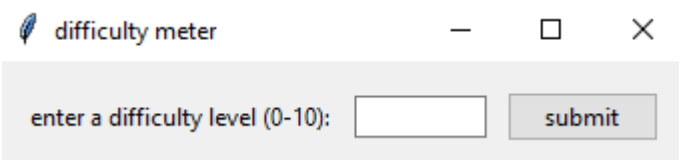
```
#create a text input field with a label
ttk.Label(input_frame, text="enter a difficulty level (0-10):").grid(row=0, column=0, padx=5, pady=5)
entry = ttk.Entry(input_frame, width=10)  #entry field for manual input
entry.grid(row=0, column=1, padx=5, pady=5)
ttk.Button(input_frame, text="submit", command=on_text_input).grid(row=0, column=2, padx=5, pady=5)
```

We use the pad arguments to add space around the widgets so that the program looks visually appealing and spaced out.

In addition, we use the row/column attributes to place the text within the window.

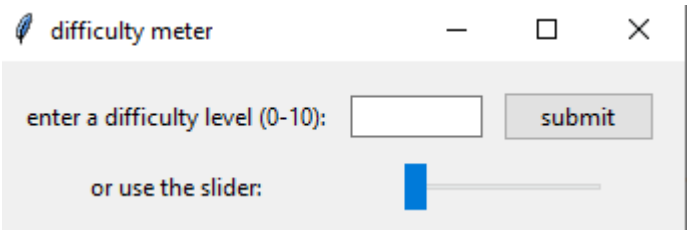
Then, we call on_text_input if the button is pressed, we do this using the ‘command’ argument.

However, this method lacked efficiency and was prone to errors, such as users entering invalid values or forgetting to click submit. In addition, the user had to manually press enter, which required an extra key stroke, a minor, but unnecessary amount of effort.



So, we added a slider for flexibility. This part of the program works the exact same as the screenshot above, but instead of a box, the input method is a slider.

```
#create a slider for adjusting difficulty
ttk.Label(input_frame, text="or use the slider:").grid(row=1, column=0, padx=5, pady=5)
slider = ttk.Scale(input_frame, from_=0, to=10, orient="horizontal", command=on_slider_change)
slider.grid(row=1, column=1, columnspan=2, padx=5, pady=5)
```



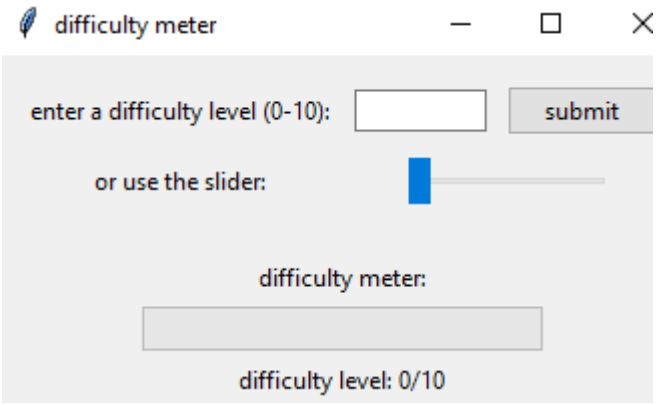
Next, we need to add a more visually appealing slider.

```
#create a frame for the difficulty meter visualization
meter_frame = ttk.Frame(root, padding="10")
meter_frame.pack()
ttk.Label(meter_frame, text="difficulty meter:").pack()  #label for the meter
meter_bar = ttk.Progressbar(meter_frame, orient="horizontal", length=200, mode="determinate")  #progress bar
meter_bar.pack(pady=5)
meter_label = ttk.Label(meter_frame, text="difficulty level: 0/10")  #label displaying difficulty level
meter_label.pack()
```

The first two lines let us create the frame. A frame is a technical term to refer to the window. Pack() is the fitting method used to 'pack' all of the information I want into the frame.

The 'orient' attribute refers to the direction of the length of the slider bar, we set it to horizontal because the window is horizontal, and it conforms with industry standards. The length 200 was defined using stakeholder feedback and the methodology outlined in the Design: Development Plan.

The 'mode' attribute makes sure that the bar only moves when the difficulty is changed either by the slider or by the direct input box.



Look at Appendix Entry Two: Video Evidence Folder, in the video called 'Video Evidence 2' for a demonstration of the difficulty feature in seconds zero through eight.

Instantiating a Structure of Inherited and Polymorphed Class Objects Based as a Factor of a Difficulty

Creating the effect

To create a tangible effect from the difficulty, we can do a few things.

Firstly, we can change the values of global variables using the methodology laid out in Design: Development Plan. The list of global variables I am referring to is:

- Gravity; a large gravity means that the movement fluctuates greatly
- Wind Force; a large wind force has a similar effect
- Obstacle Gap;
- Rain Effects

```
def set_difficulty(difficulty):
    global GRAVITY, OBSTACLE_GAP, ENEMY_SPEED, WIND_MAGNITUDE, RAIN_EFFECT

    #ensure difficulty is within the range 1 to 10
    difficulty = max(1, min(difficulty, 10))

    #scale parameters based on difficulty (1 = easiest, 10 = hardest)
    GRAVITY = 2.0 + (difficulty - 1) * 0.1  #gravity ranges from 2.0 to 3.0
    OBSTACLE_GAP = 400 - (difficulty - 1) * 20  #obstacle gap ranges from 400 to 220
    ENEMY_SPEED = 3 + (difficulty - 1) * 0.4  #enemy speed ranges from 3 to 6.6
    WIND_MAGNITUDE = 0.1 + (difficulty - 1) * 0.04  #wind magnitude ranges from 0.1 to 0.46
    RAIN_EFFECT = 0.00005 + (difficulty - 1) * 0.000015  #rain effect ranges from 0.00005 to 0.000185
```

- gravity ranges from 2.0 (easiest) to 3.0 (hardest).
- obstacle_gap ranges from 400 (easiest) to 220 (hardest).
- enemy_speed ranges from 3 (easiest) to 6.6 (hardest).
- wind_magnitude ranges from 0.1 (easiest) to 0.46 (hardest).
- rain_effect ranges from 0.00005 (easiest) to 0.000185 (hardest).

In addition, we can vary the way that obstacles are generated.

First, we need to modify the generate_obstacles function. We need to pass difficulty as an argument. This allows us to change the nature of obstacles

```
class Eagle(SpriteBase):
    def __init__(self, speed=SPEED, difficulty=5):
        super().__init__('user_image.png', SCREEN_WIDTH // 6, SCREEN_HEIGHT // 2, 30, 30)
        self._gravity = GRAVITY + (difficulty * 0.3)  #increase gravity
        self._speed = 0  #vertical speed of the eagle
        self._horizontal_speed = 0  #horizontal speed of the eagle
        self._difficulty = difficulty  # store difficulty

    def update(self, fluid="air"):
        if self._started:
            k = 0.005 + (self._difficulty * 0.0005)  # Increase drag slightly with difficulty
            drag_force = k * self._speed**2 * (-1 if self._speed > 0 else 1)
            self._speed += self._gravity + drag_force  # Apply adjusted gravity and drag
            self.rect[1] += self._speed
```

The Eagle class in this program is responsible for controlling the behavior of the player's character in the game. One of the key aspects influencing the eagle's movement is difficulty, which affects various physics-based attributes such as gravity and air resistance (drag).

The difficulty level is passed as a parameter during the initialization of the Eagle class and is stored as self._difficulty. The two primary effects difficulty has on the eagle's movement are:

- Increased Gravity:
  Higher difficulty values result in stronger gravitational pull, making the eagle fall faster when not flapping.
  This makes the game more challenging by requiring quicker reactions to maintain altitude.
- Increased Drag:
  k represents the drag coefficient, which resists motion.

An additional drag factor of (difficulty * 0.0005) is applied.
Higher difficulty increases the drag, making it harder to gain momentum when flapping.

Polymorphism; Inheritance

Real Time Character Swapping

Dynamic character swapping will improve the excitement factor of *Elegant Eagles*. This feature allows players to seamlessly switch between different character types, each with unique abilities and mechanics. To achieve this, an object-oriented programming (OOP) approach is used, ensuring modularity, scalability, and ease of future expansion.

The implementation will use inherited and polymorphed child classes of the Eagle Class(). The Character objects will be stored with the User Eagle Object in the Eagle Group. Each character type is built as a separate entity with distinct behaviours, making it easy to tweak or expand functionality iteratively.

The system will enhance gameplay variety while maintaining efficient game performance. Players will face different challenges depending on the character they control, leading to a more engaging experience. This approach also allows developers to incrementally improve or introduce new characters without overhauling existing mechanics.

There will be three characters:

- Rotating Eagle; by pressing '1' the user will rotate. This lets them evade obstacles with precise inputs.
- Heavy Eagle; by pressing '2', the user will gain fine control over the vertical User Eagle displacement.
- Time Control Eagle; by pressing '3', the user will slow down gameplay.

Throughout, we will utilise a **list of User Eagle's called the 'eagle_group'**. This structure is useful for scalability since we can store multiple User Eagles in this structure when we implement the multiplayer feature in future development iterations. Additionally, the use of a structure aids the readability of *Elegant Eagles* as a program since there are less lines of code and relevant data items are stored together.

<u>Rotating Eagle</u>

First, we need to create the class for the Rotating Eagle.

```
class RotatingEagle(Eagle):
```

Next, we need to adapt the constructor method, including the call of the parent class (Eagle) constructor.

```
def __init__(self):
    #initialize the rotatingeagle with a default rotation angle of 30 degrees
    super().__init__()
    self.__rotation_angle = 30  #private attribute to store the rotation angle
```

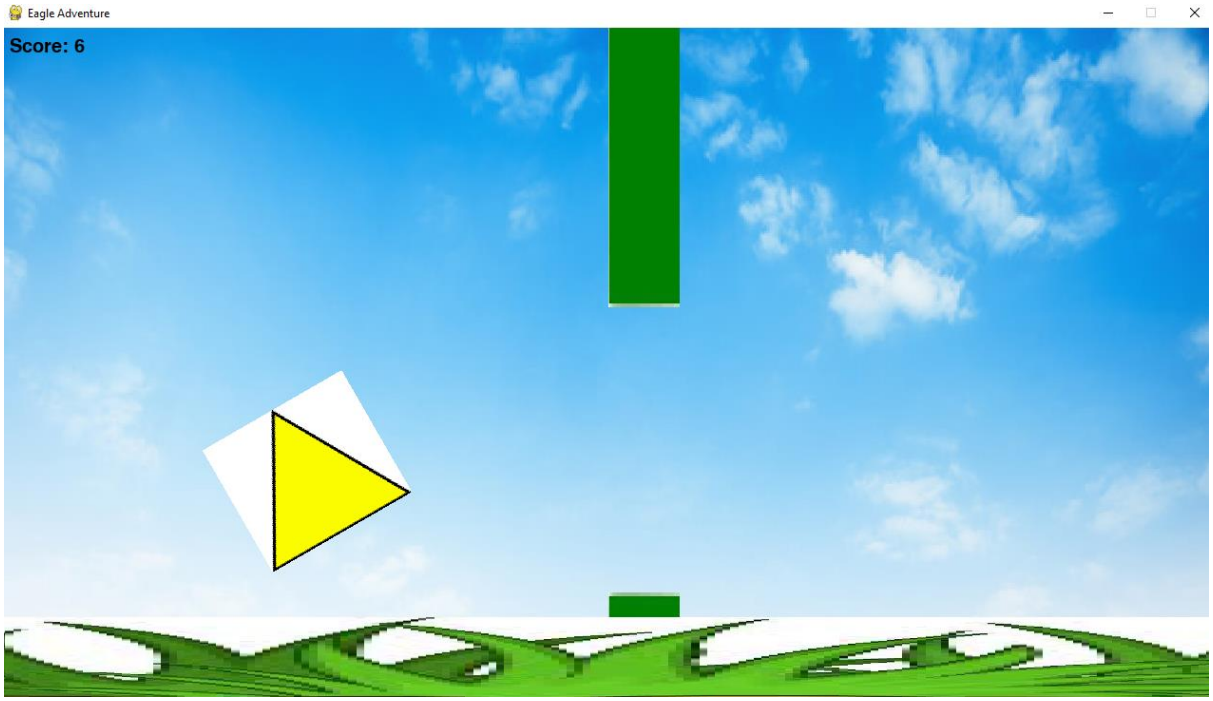
The angle thirty was chosen with Stakeholder feedback using the methodology laid out in Design: Development Plan. Here, I used a public attribute

Next, we need to create the actual effect:

```
def update(self):
    super().update() #call the base class update method
    #rotate the eagle image by the specified angle
    self.image = pygame.transform.rotate(pygame.image.load('user_image.png').convert_alpha(), self.rotation_angle)
```

Since the collision case is not done using collisions, but PyGame's mask collision, this visual rotation also handles the change in hitbox.

Changing the hitbox is necessary, otherwise, the user would experience disjointedness between what the visual and real hitbox is. In addition, the Rotating Eagle would not have any tangible effect.



Here is an example of what the rotation looks like. As stated, refer to Appendix Three for a demonstration.

Heavy Eagle

As above, first, we need to create the Heavy Eagle Class which inherits from the Eagle class.

```
class HeavyEagle(Eagle):
```

Next, we need to set up the constructor method. As above, we call Eagle()'s constructor method first to initialise the parent class attributes.

```
class HeavyEagle(Eagle):
    def __init__(self):
        super().__init__()  #initialize parent class attributes
        self.__gravity = GRAVITY * 1.2  #private attribute for increased gravity
```

Additionally, we must implement the necessary get() and set() methods:

```
def get_gravity(self):
    #getter method to access the private gravity attribute
    return self.__gravity

def set_gravity(self, value):
    #setter method to modify the gravity value with validation
    #ensures gravity is a positive number
    #raises:
    #valueerror: if the input is not a positive number
    if not isinstance(value, (int, float)) or value <= 0:  #check for valid gravity
        raise ValueError("gravity must be a positive number")
    self.__gravity = value  #update the attribute if valid
```

Here, we also validate the set method. We are ensuring that gravity is a positive float. Whilst one might think that checking *both* float and integer properties is overkill – since checking int takes less time than checking float in Python, we optimise for time to a minor extent. Set methods throughout are pertinent for the **instantiation of Character Objects using Save Data.**

Note that, the update method is unchanged:

```
def update(self):
    super().update() #call the base class update method
    self._speed += self._gravity #apply gravity; this being negative is needed to invert motion
```

This is because, we increase the rate of change of acceleration with respect to time that the User Eagle experiences in the constructor method.

Time Control Eagle

First, we create the class which inherits from the Eagle class.

```
class TimeControlEagle(Eagle):
```

Next, we create the relevant attributes in the constructor method.

```
def __init__(self):
    #initialize parent class attributes and time control properties
    super().__init__()
    self.__slow_time_active = False  #private attribute to check if slow time is active
    self.__slow_time_start = 0  #private attribute to store the timestamp
```

Then, we set up the appropriate get() and set() methods:

```
def is_slow_time_active(self):
    #getter method to check if slow time is active
    return self.__slow_time_active

def set_slow_time_active(self, value):
    #setter method to modify the slow time active flag with validation
    #ensures value is a boolean (true/false)
    #raises:
    #typeerror: if the input is not a boolean
    if not isinstance(value, bool):  #check if value is boolean
        raise TypeError("slow time active must be a boolean (true or false)")
    self.__slow_time_active = value  #update the attribute if valid

def get_slow_time_start(self):
    #getter method to access the slow time start timestamp
    return self.__slow_time_start

def set_slow_time_start(self, value):
    #setter method to modify the slow time start timestamp with validation
    #ensures the value is a positive number (timestamp)
    #raises:
    #valueerror: if the input is not a valid timestamp
    if not isinstance(value, (int, float)) or value < 0:  #check for valid timestamp
        raise ValueError("slow time start must be a positive number")
    self.__slow_time_start = value  #update the attribute if valid
```

The comments aid in the description of the validation performed by the set methods.

- Slow_time_active must be Boolean
- Slow_time_start must be a positive float.

Accepting User Inputs

Now, we modify the game_loop() so that:

- If the user performs a key stroke
- We check if the keystroke is relevant to the character swapping
- We remove the outdated Eagle Object from the Eagle_Group Object and add the new one.

```
if event.type == KEYDOWN:
    if event.key == K_SPACE or event.key == K_UP:
        eagle.launch()
    if event.key == K_k:
        save_game(current_score)
    if event.key == K_1:
        eagle = RotatingEagle()
        eagle_group.empty()
        eagle_group.add(eagle)
    if event.key == K_2:
        eagle = HeavyEagle()
        eagle_group.empty()
        eagle_group.add(eagle)
    if event.key == K_3:
        eagle = TimeControlEagle()
        eagle_group.empty()
        eagle_group.add(eagle)
    if event.key == K_s and isinstance(eagle, TimeControlEagle):
        eagle.slow_time()
```

Creating the effect

Now, we aim to create the actual effect as a result of the user input '3' to change to the Time Control Eagle. When the user pressed '3', they will trigger the slow-mode.

```
def update(self):
    #check if slow time is active and if duration exceeds 2000ms (2 seconds), deactivate it
    if self.slow_time_active and (pygame.time.get_ticks() - self.slow_time_start > 2000):
        self.slow_time_active = False

    #if slow time is active, skip gravity update to simulate slow motion effect
    if self.slow_time_active:
        return

    super().update()  #call the base class update method

    #apply rotation transformation only if the angle is nonzero
    if self.rotation_angle != 0:
        self.image = pygame.transform.rotate(self.original_image, self.rotation_angle)
```

First, we make sure that the slow time is active and then whether or not an excessive number of seconds have passed.

Using feedback from my stakeholder Oliver and the methodology from Design: Development Plan, we determined that the appropriate number of seconds to limit the Time Control Eagle to is two seconds. This is because, we do not want users to abuse the system and stay in the Time Control Eagle for an excessive amount of time.

Next, we must determine the scale factor by which 'time' will slow when the user enters the slow mode.

```
SLOW_MOTION_FACTOR = 0.5
```

In reality, we are simply reducing acceleration, the velocity of the ground, the velocity of the obstacles and the velocity of the User Eagle by the same scale factor.

Next, we must, make the ground travel slower.

```
class Ground(SpriteBase):
    def __init__(self, xpos):
        super().__init__('grass.jpg', xpos, SCREEN_HEIGHT - GROUND_HEIGHT, GROUND_WIDTH, GROUND_HEIGHT)

    def update(self):
        # Move the ground to the left, scaled by the slow motion factor
        slow_motion_factor = eagle.get_slow_motion_factor() if isinstance(eagle, TimeControlEagle) else 1
        self.rect[0] -= GAME_SPEED * slow_motion_factor
```

We do this using the last two lines. In the penultimate line, we:

- Check if the user Eagle is a Time Control Eagle
- If it is, then we make the slow_motion_factor less than one so that the game slows.

We do the exact same thing with the Obstacle Class:

```
class Obstacle1(SpriteBase):
    def __init__(self, inverted, xpos, ysize):
        image_path = 'green block.png'
        ypos = 0 if inverted else SCREEN_HEIGHT - ysize
        super().__init__(image_path, xpos, ypos, OBSTACLE_WIDTH, OBSTACLE_HEIGHT)
        if inverted:
            self.image = pygame.transform.flip(self.image, False, True)
            self.rect[1] = -(self.rect[3] - ysize)

    def update(self):
        # Move the obstacle to the left, scaled by the slow motion factor
        slow_motion_factor = eagle.get_slow_motion_factor() if isinstance(eagle, TimeControlEagle) else 1
        self.rect[0] -= GAME_SPEED * slow_motion_factor
```

Finally, we call the .slow_time() method if the eagle object in the eagle group structure is a TimeControl Eagle.

```
if event.key == K_s and isinstance(eagle, TimeControlEagle):
    eagle.slow_time()
```

For a demonstration of the Time_Control _Eagle, see the relevant videos (as described by their video titles) in Appendix Entry Two: Video Evidence Folder.

Inheriting Specific Attribute States

PROBLEM:

When we swap between eagles, the position and velocity reset to the original values.

SOLUTION:

```
def transfer_state(self, other_eagle):
    """
    Transfer the state (position, velocity, etc.) from another eagle to this one.
    """
    self.rect[0] = other_eagle.rect[0]  #transfer x position
    self.rect[1] = other_eagle.rect[1]  #transfer y position
    self.set_speed(other_eagle.get_speed())  #transfer velocity using setter
    self.set_started(other_eagle.get_started())  #transfer started flag using setter
```

We will call this method when we are updating the eagle group structure.

This lets us instantiate the ‘other_eagle’ before deleting the original eagle (‘self’).

We do this by passing relevant attributes for seamless transfer such as vertical velocity and coordinates.

First, we need to pass this ‘other_eagle’ as an attribute to the constructor. For example:

```
class RotatingEagle(Eagle):
    def __init__(self, old_eagle=None):
        super().__init__()  #initialize parent class attributes
        self._rotation_angle = 30  #private rotation angle
        if old_eagle:
            self.transfer_state(old_eagle)  #transfer state from the old eagle
```

In the last line, we call the transfer_state() method to update the attributes from ‘old_eagle’.

Note that the line ‘if old_eagle’ only calls the transfer_state() method given that the Rotating Eagle has an eagle to inherit specific state attributes from; it is possible in future development iterations that we will let the user start the game in a non-default user character.

Finally, we must modify game_loop() so that the old_eagle is passed into the instantiation of the new object. For example:

```
if event.key == K_3:
    # Swap to TimeControlEagle and transfer state
    new_eagle = TimeControlEagle(old_eagle=eagle)
    eagle_group.empty()
    eagle_group.add(new_eagle)
    eagle = new_eagle
```

This is where we will update the eagle_group structure after having passed all necessary attributes to the new User Eagle object.

PROBLEM:

Since once of the objectives in this iteration is to add enemies, we need to make sure that the changes, particularly the Time_Control_Eagle's slow mode impacts the entire game.

SOLUTION:

To do this, we simply change the GAME SPEED global variable whenever we enter or exit a slow mode:

```
def slow_time(self):
    # Activate slow time ability
    if not self._slow_time_active:
        self._slow_time_active = True
        self._slow_time_start = pygame.time.get_ticks()  # Record the start time of slow motion
        global GAME_SPEED
        GAME_SPEED = GAME_SPEED * SLOW_MOTION_FACTOR  # Reduce game speed
```

```
def deactivate_slow_time(self):
    #deactivate slow time ability
    if self._slow_time_active:
        self.set_slow_time_active(False)  #use setter method
    global GAME_SPEED
    GAME_SPEED = GAME_SPEED / SLOW_MOTION_FACTOR  #restore game speed
```

We simply, multiply the game speed by the slow motion factor when entering slow mode, and when exiting, we divide by the same slow motion factor.

Deductive Debugging

PROBLEM:

If a user exists Time_Control_Eagle after a sufficiently small number of milliseconds, then the slow mode is permanent.

SOLUTION:

Given that this bug hinges on a time condition (number of milliseconds), we look to our only use of Pygame's .clock() method and an inequality. We do this, because, if the exit condition is not met, then the slow-mode will be permanent.

If the user exits the eagle character within two seconds, then the GAME_SPEED global variable does not change.

```
def deactivate_slow_time(self):
    #deactivate slow time ability
    if self._slow_time_active:
        self.set_slow_time_active(False)  #use setter method
    global GAME_SPEED
    GAME_SPEED = GAME_SPEED / SLOW_MOTION_FACTOR  #restore game speed
```

So, we reset the GAME_SPEED variable whenever slow_time_active is false, This is because the attribute slow_time_active persists across all eagle states using the previous fix to seamless transfer.

PROBLEM:

The problem persists, the only exception is that the GAME_SPEED resets when the user re-enters the Time_Control_Eagle character.

SOLUTION:

Since the reset methods aren't called in other characters, we can deduce that the bug is caused by the slow_time_active parameter is not being passed to the other Eagles.

```
def transfer_state(self, other_eagle):
    """
    Transfer the state (position, velocity, etc.) from another eagle to this one.
    """
    self.rect[0] = other_eagle.rect[0]  #transfer x position
    self.rect[1] = other_eagle.rect[1]  #transfer y position
    self.set_speed(other_eagle.get_speed())  #transfer velocity using setter
    self.set_started(other_eagle.get_started())  #transfer started flag using setter
    self.slow_time_active=other_eagle.slow_time_active
```

Then, the fix is a simple matter of adding the last line in the image above, so that the slow_time_active attribute is passed between eagle characters.

Enemies

Now, we aim to implement an enemy that aims to collide with the User Eagle, making the user lose. As described in the design, we will implement the A* Pathfinding Enemy, 'AStar Enemy':

The AStar Enemy uses the A* pathfinding algorithm to navigate the game world in pursuit of the User Eagle. This algorithm relies on both actual movement cost and a heuristic estimate to determine the shortest path toward its target. However, in order to ensure fair gameplay and allow the user a chance to escape, the heuristic has been intentionally designed to be imperfect. For example, the heuristic may slightly underestimate obstacle density or over-prioritise straight-line distance, introducing exploitable patterns that skilled players can take advantage of.

The enemy operates within a 2D xy-plane representation of the game space, where static obstacles—such as pipes and ground boundaries—are encoded as high-cost or impassable regions in the coordinate grid. This forces the AStar Enemy to intelligently curve around these regions rather than passing through them. The grid resolution, obstacle cost function, and update frequency of the pathfinding logic are all adjustable parameters that can be tuned to create the desired level of challenge and realism.

Ultimately, this implementation aims to provide a dynamic and challenging pursuit experience that rewards strategic movement and map awareness, while maintaining the game's principle of learnability and fairness.

For every ten points that the score increments, an enemy spawn will trigger.

All enemies will de-spawn t seconds after being within r units of the user eagle, this is the evasion mechanic. The constants t and r will be determined below.

Iterative pathfinding algorithm; Predictive-tracking enemy with trigger mechanism

A* Enemy

Inheritance

First, I created a parent PathfindingEnemy class to allow for future expansion with different enemy types. This ensures modularity, reusability, and maintainability using inheritance and polymorphism.

```
class PathfindingEnemy(SpriteBase):
```

Here, we inherit the parent class of the entire class hierarchy: Sprite Base. This use of **inheritance** gives the AStar Enemy certain suitable attributes such as an image file path and x/y coordinates.

Validation; Encapsulation

Next, we set up an appropriate constructor method:

```
class PathfindingEnemy(SpriteBase):
    def __init__(self, xpos, ypos, speed, difficulty):
        try:
            super().__init__('enemy.png', xpos, ypos, 40, 40)
            self._speed = speed  #speed of the enemy
            self._target = None  #target position for pathfinding
            self._pathfinding_interval = 15  #update path every 15 frames
            self._frame_counter = 0  #frame counter for pathfinding updates
            self._difficulty = difficulty
        except Exception as e:
            print(f"Error loading enemy image: {e}")
```

We pass the relevant attributes required to instantiate an enemy object:

- Coordinates
- Speed

- Game difficulty

Next, we need to call the parent constructor to initialise the enemy image file-path, coordinate location and size. Then, we must set up the object attributes:

The speed attribute controls the vertical velocity since the enemy will have a fixed horizontal velocity. Using the vector between the AStar Enemy and the goal coordinate, we will use trigonometry to determine a suitable vertical to horizontal velocity ratio to make the enemy travel in the intended direction.

The notable attribute that we require is the coordinates of the target. The target attribute is a list because there is an x coordinate and a y coordinate that must be passed in that order.

We then use the pathfinding_interval and frame_counter attributes to control the rate at which the pathfinding algorithm is run. Frame_counter measures the number of frames that have passed since the last pathfinding update. It will be reset to zero whenever the pathfinding direction is updated. Pathfinding_interval measures the number of frames between consecutive updates.

Since we do not want the enemy to be perfect, this rate will not be very high. Since the system is running at 30 frames per second, 15 frames is a lag time of 0.5 seconds which is substantial enough to evade the enemy.

In fact, we can vary the number of frames as a way to control how hard avoiding the enemies will be. Otherwise, we'd need to change the heuristic, which could increase the algorithm's complexity.

Validation

PROBLEM:

Right now, we are not validating against issues loading the enemy image. For instance, there could be an issue with the file-path being changed e.g. if we change the name of our drive.

SOLUTION:

We utilise try catch with a general exception. We do a general case because the specific error is unimportant; the result is all the same: a relevant system alert. This will be especially pertinent in the Beta Tests when it is likely there will be issues sharing all relevant images to testers.

```
class PathfindingEnemy(SpriteBase):
    def __init__(self, xpos, ypos, speed, difficulty):
        try:
            super().__init__('enemy.png', xpos, ypos, 40, 40)
            self._speed = speed  #speed of the enemy
            self._target = None  #target position for pathfinding
            self._pathfinding_interval = 15  #update path every 15 frames
            self._frame_counter = 0  #frame counter for pathfinding updates
            self._difficulty = difficulty  #difficulty level of the enemy
        except Exception:
            print(f"Error loading enemy image: {Exception}")
```

Rigorous Validation

Creating the Get() and Set() methods

We will utilise lots of validation so that *Elegant Eagles* is robust and maintainable.

- Speed must be a float:

```
def get_speed(self):
    return self._speed

def set_speed(self, speed):
    if isinstance(speed, (int, float)) and speed > 0:  #validate speed is a positive number
        self._speed = speed
    else:
        raise ValueError("speed must be a positive number.")
```

Case Specific Optimisations

PROBLEM:
Firstly there is redundant validation.
Secondly, the error message is inaccurate
SOLUTION:
The set of all integers is a subset of the floats (rationals), therefore, we need only check the floats. **Earlier** in stage three, we validated against both integer AND float nature because validating against integer nature takes a smaller number of clock cycles to succeed. However, since speed is almost never an integer, an integer check would be a waste of space, and time and detracts from readability.
To fix the alert message, we note that speed can be negative in this case. Strictly, we should label speed as 'velocity' but calling the variable speed improves readability for programmers on the team who may be unfamiliar with

```
def set_speed(self, speed):
    if isinstance(speed, (float)) and speed > 0:  #validate speed is a positive number
        self._speed = speed
    else:
        raise ValueError("speed must be a float number.")
```

- Target must be a tuple (a coordinate) with two values:

```
#get and set methods for target
def get_target(self):
    return self._target

def set_target(self, target):
    if isinstance(target, tuple) and len(target) == 2:  #validate target is a tuple of length 2
        self._target = target
    else:
        raise ValueError("target must be a tuple of length 2 (x, y).")
```

We ensure dimensional consistency with the second clause of the if statement since there are two numbers corresponding to the x and y coordinates. The use of a tuple is standard industry practice (source: References: PyGame documentation exemplars). This is because tuples are immutable, therefore, protecting the coordinates of the target.

PROBLEM:
The validation is incomplete, we could pass ('hi', seven) as target successfully.
SOLUTION:
We must validate each of the tuple indices:

```
def set_target(self, target):

    if isinstance(target, tuple) and len(target) == 2:  #validate target is a tuple of lengt
        x, y = target
        if isinstance(x, float) and isinstance(y, float):  #validate x and y are numbers
            self._target = (float(x), float(y))
        else:
            raise ValueError("Target coordinates must be floats or integers.")
    else:
        raise ValueError("Target must be a tuple of length 2 (x, y).")
```

Here, we make sure that each of the indexes are floats, before returning the relevant ValueError alerts. Overall, we:
-Ensure that the target is a tuple
-Check that the tuple has exactly two values (x, y coordinates).
-Validating for float x, y coordinates.
-Raise case specific alerts

- So that the pathfinding algorithm is performed at a steady rate, we need to ensure that pathfinding_interval is a natural number:

```
#get and set methods for pathfinding interval
def get_pathfinding_interval(self):
    return self._pathfinding_interval

def set_pathfinding_interval(self, interval):
    if isinstance(interval, int) and interval > 0:  #validate interval is a positive integer
        self._pathfinding_interval = interval
    else:
        raise ValueError("pathfinding interval must be a positive integer.")
```

- So that the interval from above is correctly used, we must ensure that the updated frame_count is positive:

```
def set_frame_counter(self, counter):
    if isinstance(counter, int) and counter >= 0:  #validate counter is a non-negative integer
        self._frame_counter = counter
    else:
        raise ValueError("frame counter must be a positive integer.")
```

- Finally, we need to make sure that difficulty is an integer between one and ten inclusive. This is important, because, the tangible difficulty scales very precisely with the number. For instance, the speed of the game would decrease to less than zero if the difficulty input is negative. This would quite literally be a 'game-breaking bug', a contradiction of one of my fundamental **success criteria.**

```
#get and set methods for difficulty
def get_difficulty(self):
    return self._difficulty

def set_difficulty(self, difficulty):
    if isinstance(difficulty, int) and 1 <= difficulty <= 10:  #validate difficulty is between 1 and 10
        self._difficulty = difficulty
    else:
        raise ValueError("difficulty must be an integer between 1 and 10.")
```

Modified Manhattan Heuristic; Iterative Mathematics;

Development of Heuristic

When choosing a Heuristic, the typical consideration is with regards to run time. However, given that the application of the heuristic in *Elegant Eagles* is to make the enemy path find imperfectly, one might think that time complexity is irrelevant. However, we cannot be completely inconsiderate of time-to-execute because a slow execution time would inflate the lag time.

The choice from the Design stage is therefore Euclidean distance:

$$\left(H\left(x_{enemy}, y_{enemy}, x_{user\ eagle}, y_{user\ eagle}\right)\right)^2 = \left(y_{user\ eagle} - y_{enemy}\right)^2 + \left(x_{user\ eagle} - x_{enemy}\right)^2$$

Euclidean distance is very quick to execute and simple to implement. However, it will underestimate distances causing collision. However, use of the Euclidean Distance would make the AStar Enemy simply an LOS enemy that does not stop for obstacles. This defeats the point of the AStar Enemy; a challenging and unique enemy.

Another choice is, Manhattan distance:

$$H_M\left(x_{enemy}, y_{enemy}, x_{user\ eagle}, y_{user\ eagle}\right) = y_{user\ eagle} - y_{enemy} + x_{user\ eagle} - x_{enemy}$$

This would be more suitable because it is faster to execute and simpler to implement. However, like Euclidean Distance, the enemy would collide with obstacles since, on a fine scale, if a given path has a lesser Euclidean Distance, then it must have a lower Manhattan Distance. Otherwise, this algorithm is perfect.

Therefore, this motivates us to customise the Heuristic Function to our needs; we need to dissuade the enemy from colliding with obstacles.

Then we define H() as:

$$H_M\left(x_{enemy}, y_{enemy}, x_{user\ eagle}, y_{user\ eagle}\right) + \sum_{i=1}^{len(obstacle\_group)} H_M\left(x_{enemy}, y_{enemy}, x_{obstacle_i}, y_{obstacle}\right)$$

First, we must create the method:

```
def heuristic(self, enemy_pos, player_pos, obstacles):
    """
    Heuristic considering distance, obstacles, and momentum.
    """
```

Note the use of the parameters about:

- The position of the enemy
- The position of the player
- The list of all obstacle coordinates

This will allow the heuristic to create a ‘map’ of the relative position of the obstacles with respect to the enemy and the ideal path to the user (a straight line).

Then, for **readability**, we expand upon the use of variables:

```
ex, ey = enemy_pos  #extract enemy's position (e_x, e_y)
px, py = player_pos  #extract player's position (p_x, p_y)
```

We then initialise distance, using the case of zero obstacles.

```
#manhattan distance: |e_x - p_x| + |e_y - p_y|
distance = abs(ex - px) + abs(ey - py)
```

Next, we need to add the penalty to the result. We do this, by checking the proximity of every obstacle to the enemy and penalising closeness.

```
#obstacle penalty: add a penalty for obstacles in a straight line
obstacle_penalty = 0  #initialize penalty to 0
for obstacle in obstacles:  #iterate through all obstacles
    #calculate distance to the obstacle
    ox, oy = obstacle.rect.x, obstacle.rect.y  #extract obstacle's position (o_x, o_y)
    obstacle_distance = abs(ex - ox) + abs(ey - oy)  #manhattan distance to obstacle

    #add a penalty inversely proportional to the distance to the obstacle
    if obstacle_distance < 50:  #penalize if the obstacle is within 50 units
        obstacle_penalty += 1000 / (obstacle_distance + 1)  #add penalty
```

We only consider obstacles that are very close to the enemy, to optimise the performance of the algorithm and create scalability in *Elegant Eagles*.

Should the number of obstacles in future development iterations increase, then, the iterative calling of the heuristic would be expensive.

PROBLEM:

The optimisation will cause major heuristic inaccuracies.

SOLUTION:

The problem arises from our use of obstacle_distance when deciding if each obstacle is close enough to the enemy to warrant adding to the penalty of the given path.

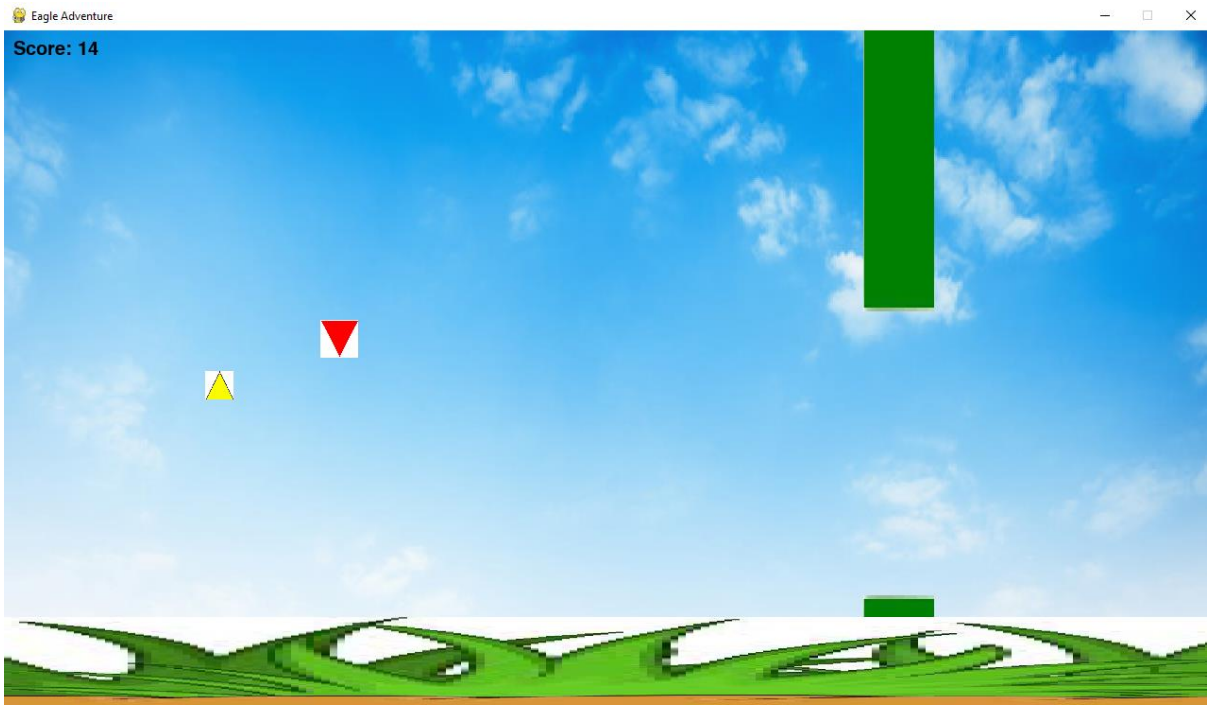
Instead of using Manhattan distance for this step, we will use Euclidean distance. This is because:

- Euclidean distance is a more relevant measure of proximity because the enemy will not necessarily move in solely northings and westings (almost certainly not so). Instead, the enemy will move in an apparently smooth curve.
- We used Manhattan distance in the first place because it created additional imperfections in the Enemy's path, making evasion possible.

```
#add a penalty inversely proportional to the distance to the obstacle
if obstacle_distance < 50^2:  #penalize if the obstacle is within 63 units
```

In addition, the constants 50 and 1000 are completely arbitrary and need to be developed. They will be developed in the Stakeholder Evaluation

This is the result:



Heap Queue Structure and Big O();

PROBLEM:

This algorithm is looking quite expensive so far.

Whilst it will almost certainly run given that one cannot eyeball the arbitrary computational order that is suddenly tangible on a human scale. We can improve program scalability, whilst decreasing lag-times by implementing careful **optimisations**. Typically, these would be inserted retrospectively but given the scale of an A* algorithm it is suitable to plan more thoroughly.

Throughout we will use Big O() notation, this is the suitable notation type because there is no trend in the set of neighbour nodes being considered.

SOLUTION:

At first, my data structure was going to be a dictionary of length one:

1. Store the first neighbour as the key, its heuristic cost as the value
2. Iterate through the list of neighbour nodes
3. Calculate the cost of each
4. If the new cost is less than the stored cost in the dictionary, update the node stored in the dictionary
5. Repeat until all nodes have been considered

Whilst, mathematically, this demonstrates the minimum complexity O(n) of one pass, **the pathfinding algorithm is called iteratively** throughout the user run. Therefore, it is imperative to **take a more holistic approach to what has shaped into a nuanced optimisation problem**.

Rather than taking the greedy approach, we will consider fundamentally what operation costs time. The vast majority of time taken in an A* algorithm comes from the heuristic computations on each

node. Crucially, one most note that each node will be considered at least two times. Therefore, we opt to store node costs.

One might therefore utilise a dictionary, but that raises some issues:

- What order do we store the nodes in this structure? If we simply append, then, queries (as opposed to computing the heuristic cost of a given node multiple times) will be O(n).
- How do we find the node with the lowest cost, do we need to perform an O(n) linear search?

To surmount these issues, I chose to use **heap queues**. Heap queues optimise the pathfinding algorithm in many ways:

1. Heap queues are priority queues, therefore, the lowest heuristic cost node is always stored in index zero. Then once all neighbours are considered, the query time to pick the optimal neighbour is O(1).
2. Heap queues maintain order on insertion. Insertion itself is only O(log(n)).
3. Since the queue is sorted, then the query time is O(log(n)), this means that, we calculate the heuristic cost of each node zero or one time(s). We know whether to query or compute using a node flag, as in standard A*.

We will explore the use of heap queue below.

Creating the Pathfinding Algorithm

```
def _a_star_pathfinding(self, target_pos, obstacles):
```

First, we must carefully consider the parameters that we are passing to the algorithm.

Typically, the target position is the list of user coordinates. We will use this parameter to control where the enemy will pathfind to. Next, as in typical A*, we create the graph structure by passing the list of obstacle coordinates. The coordinate attribute is of a consistent form: [x, y], whilst, 'obstacles' is a list of obstacle objects that must be iterated through and interrogated using relevant get methods.

Since, the target position itself is not relevant, it is the difference between the positions, we need to set up the position of the enemy as an attribute.

```
start = (self.rect.x, self.rect.y)
```

Here, we use the .rect() methods from the Sprite Base class that is inherited from far up the class hierarchy.

Next, we handle the case of the enemy being too far from the user. This prevents the enemy from pathfinding to the user when off screen. This is one of the aforementioned methods of escaping the enemy.

```
search_range = 200
```

```
#check if the target is within the search range
if abs(start[0] - goal[0]) > search_range or abs(start[1] - goal[1]) > search_range:
    return []  #skip pathfinding if the target is too far
```

Next we need to set up the relevant structures for the iterative pathfinding

Near Recursive Hash Map

```
open_set = []  #priority queue for exploring paths
heapq.heappush(open_set, (0, start))  #push start node with cost 0
came_from = {}  #dictionary to reconstruct path
g_score = {start: 0}  #actual movement cost from start
f_score = {start: self._heuristic(start, goal, obstacles)}  #estimated total cost
```

The open set refers to the set of all neighbour nodes that are yet to be visited. At the end of each pass open set will be empty before being updated to a non-empty state.

Next, we need to insert the start position into the open set. Note that Python's HeapQ library effectively casts a list into a priority queue, hence, we need not define open set as a heap. What we are doing in this line is adding the start position to the heap, and giving it priority 0. The priority is 0 because we simply need to make the structure non-empty such that we can enter the while loop, where we will search for neighbours and add them to the heap.

Next, we initialise the came_from dictionary to store the path that the enemy travels. We must store this path, because, whilst it appears like A* is varying continuously, in reality, there are complete A* traversals happening frequently, such that, the enemy does not travel until the entire path is decided.

The structure being a dictionary allows it to operate as **hash map that behaves recursively – every node points to a previous node, like a linked list. This allows the path to be reconstructed top down.**

Finally, as the comments describe, we initialise dictionaries for the estimated cost and real cost. We initialise the g list because, we know that the cost to get from the start to the start is zero. We also set up the framework for the f list to be added to very efficiently. All we need to do is iterate through the list of possible start values. Note that f, the estimated cost, is calculated as the sum of current distance travelled and the heuristic value.

Next, we need to set up the logic to return the node path to travel to the enemy eagle.

Termination via Backtracking

```
while open_set:
    _, current = heapq.heappop(open_set)  #get the node with the lowest cost

    if current == goal:
        #reconstruct path from goal to start
        path = []
        while current in came_from:
            path.append(current)
            current = came_from[current]
        path.reverse()
        return path
```

First, we are traversing from the most 'promising node'. This is the node with the highest priority. Via the nature of heap queues, this is index zero. Heappop() returns the value in this index.

Next, we are checking if the algorithm which we will describe below has reached its destination. We are checking if the aforementioned priority node is the final node. This is the termination condition.

Each ‘came_from[current]’ gives us the previous node in the best path. We reverse this list to get the node path in the correct order: from start to finish.

Next, we must populate the neighbour node list.

```
#generate 8-directional neighbour nodes
neighbours = [
    (current[0] + dx, current[1] + dy)
    for dx, dy in [(-self._speed, 0), (self._speed, 0), (0, -self._speed), (0, self._speed),
                   (-self._speed, -self._speed), (self._speed, -self._speed),
                   (-self._speed, self._speed), (self._speed, self._speed)]
]
```

Careful Scalability Justification

We do this by considering eight directions. Whilst we could consider more or less, I chose eight arbitrarily.

If we consider too many directions, the algorithm will be too accurate and therefore the enemy will be inescapable. In contrast, if we consider too few directions, the algorithm will be utterly unintelligent and trivial to evade. Nonetheless, eight is arbitrary. This is completely fine, since, we are controlling the difficulty of the enemy through other means such as, the frequency of path updates. It is crucial to vary frequency through one specific parameter, rather than many, such that stakeholder testing can operate in a transparent and measurable feedback loop.

Whilst there is a large number of words, we are simply brute forcing the possible coordinates in the eight cardinal compass directions (45 degree turns) with respect to the current enemy position.

The use of calculus language like ‘delta y’ here is useful because it sets us up for more sophisticated physics in future development iterations past A Level. For example, we can differentiate with respect to x to get acceleration, so that we can make the motion of the AStar enemy smoother e.g. by validating the angle between the intended acceleration vector and the current velocity vector using the dot product.

Next, we need to actually choose which node to visit next.

```
for neighbour in neighbours:
    #calculate cost to reach neighbour
    tentative_g_score = g_score[current] + self._heuristic(current, neighbour, obstacles)
```

First, we iterate through the list of neighbours. Then, we get a predicted cost that the enemy must travel if they were to travel through this neighbour. We get this cost, by summing the true cost to get to the current node and the heuristic cost from the next node.

```
for neighbour in neighbours:
    #calculate cost to reach neighbour
    tentative_g_score = g_score[current] + self._heuristic(current, neighbour, obstacles)

    #update path if it's the best so far
    if neighbour not in g_score or tentative_g_score < g_score[neighbour]:
        came_from[neighbour] = current
        g_score[neighbour] = tentative_g_score
        f_score[neighbour] = tentative_g_score + self._heuristic(neighbour, goal, obstacles)
        heapq.heappush(open_set, (f_score[neighbour], neighbour))
```

Next, we check whether or not we have visited this neighbour before. Then, we check if this new cost is better (lesser) than our existing best neighbour.

If we successfully find a 'better' neighbour (one with lesser g cost), then we need to:

- Add to the came_from dictionary so that we can reconstruct the path. This line of code means that the best neighbour so far is *current*.
- Update the f and g costs so that we can compare other neighbours to this one.
- Add the neighbour to the open set, using the f score as the priority. The lower the priority of a node, the better it is. The algorithm will then revisit the best node in the next pass as described.

Polymorphism and Try Catch Validation

Update method

Next, we need to execute the changes on the enemy object.

```
def update(self, player, obstacles):
```

Firstly, we need to pass the appropriate parameters. As before, the player object, and the obstacles object.

```
#increment frame counter
self._frame_counter += 1

#run a* only every few frames
if self._frame_counter >= self._pathfinding_interval:
    self._frame_counter = 0
    #set target to player location using accessor methods
    self._target = (player.get_x(), player.get_y())
    #recalculate path using a*
    self._path = self._a_star_pathfinding(self._target, obstacles)
```

This method is called every game loop. Hence, whilst not strictly a measure of frames, we increment the frame counter.

This if statement first checks if a sufficient number of frames has passed to make a new path for the enemy to travel. If this condition is satisfied, we need to reset the counter, and update the path node-object list attribute.

```
#if path exists, move toward the next point
if self._path:
    next_pos = self._path[0]
    dx = next_pos[0] - self.rect.x
    dy = next_pos[1] - self.rect.y
    distance = math.sqrt(dx ** 2 + dy ** 2)
```

Next, we consider the next node to visit in the node-object list called path. We then consider the distance that we must travel to get to that node. However, since we want the enemy to move at a constant velocity, we then need to normalise this vector.

```
if distance > 0:
    #normalize direction
    dx /= distance
    dy /= distance
    #move enemy in direction of target
    self.rect.x += dx * self._speed
    self.rect.y += dy * self._speed
```

We then need to implement the logic of moving on to the next node:

```
#if close to next point, remove it from path
if distance < self._speed:
    self._path.pop(0)
```

We 'move on' from a node before reaching it so that the movement appears smooth. This also means that the enemy pathfinds slightly less effectively, so, evasion is slightly easier. This is unimportant since, as discussed, we can control the difficulty of enemy evasion using the frame_counter threshold from above.

Finally, we implement another evasion mechanic: if the enemy goes off screen because the user evades it, then it will despawn.

```
    #despawn if enemy is off screen
    if self.rect.x < -self.rect.width or self.rect.x > SCREEN_WIDTH or self.rect.y < -self.rect.height or self.rect
        self.kill()
```

PROBLEM:

An impromptu discussion with my stakeholder James, made me realise a potential issue: we are calculating a complete path to travel, even though the enemy will decide upon a new path very shortly in the next iteration.

SOLUTION:

Upon further consideration, it is impossible to produce the same path without finishing the path. This is because, a greedy path may not be optimal. Therefore, the 'solution' I intended to implement would've made the enemy's pathfinding ineffective and indecisive because of the greediness.

Scalability Considerations

Now we need to make the procedure which **spawns** the enemy.

We are using the enemy group throughout.

```
enemies_group = pygame.sprite.Group()
```

As with the player characters and the obstacles, we use a group for iterability and containing all objects under one identifier. Whilst at the moment there is exactly one or zero enemies on the screen at all times, this improves program scalability. This is because, the Group() method from pygame's sprite library enables batch operations on multiple objects. This means that operations will act on every object in the group when a method is called on the group.

```
def generate_enemy(score, enemies_group):
    global last_enemy_spawn_score
    # Only add a new enemy when the score is a multiple of 10 and not on every frame
    if score % 10 == 0
        print("Spawning new enemy!")  # Debugging
        last_enemy_spawn_score = score  # Update the last score when an enemy was spawned
        # Spawn an enemy at a random vertical position
        enemy = PathfindingEnemy(SCREEN_WIDTH, random.randint(0, SCREEN_HEIGHT - 50), speed=5)
        enemies_group.add(enemy)
```

This function will be called during the game loop. Not the parameter enemies_group which is the list of all enemy objects. This allows for iterability e.g. if in the future, we want to be able to de-spawn all enemies for a powerup. Passing the enemy group lets us add to it as necessary.

The if condition ensures that score is a multiple of 10, only then will the enemy spawn. We then instantiate the enemy object using a little bit of pseudo-randomness to increase user replayability through variety. The constants chosen were determined through scrutinous stakeholder testing as described in Design: Development Plan.

```
generate_enemy(current_score, enemies_group)
```

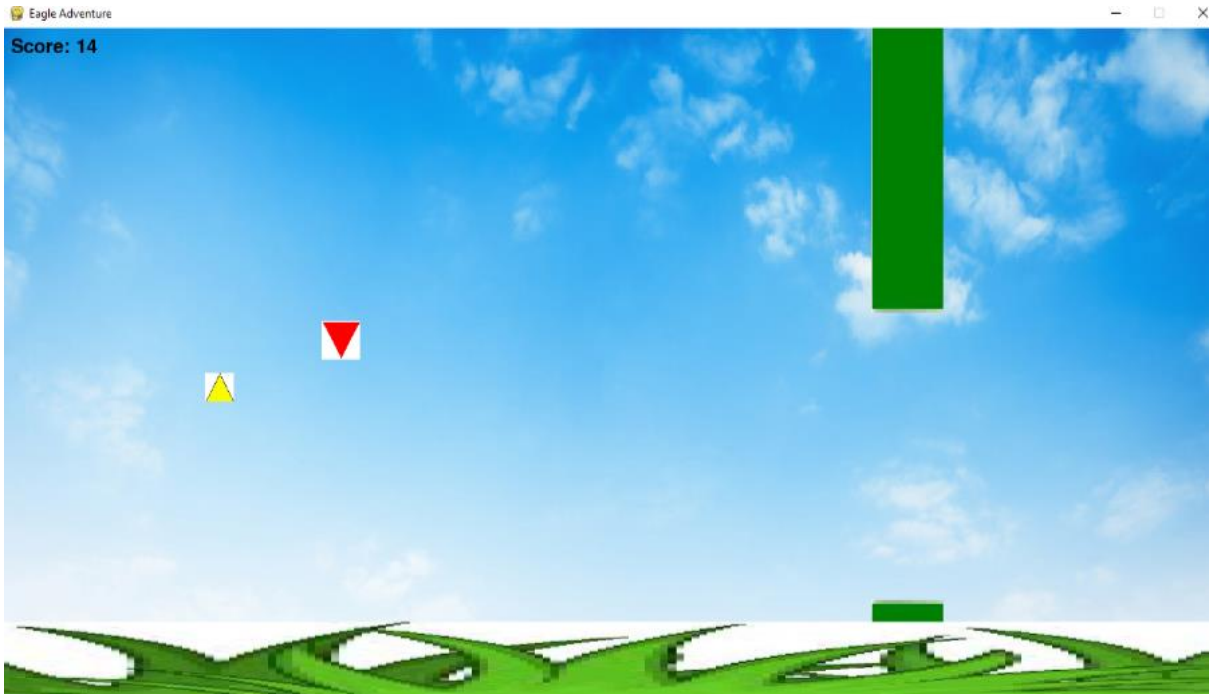
Finally, we need to update each of the enemies on every iteration of the game loop.

```
# Update groups
eagle_group.update("air")
ground_group.update()
obstacle_group.update()
enemies_group.update(eagle, obstacle_group)

# Draw groups
eagle_group.draw(screen)
obstacle_group.draw(screen)
ground_group.draw(screen)
enemies_group.draw(screen)
```

We do this, by first updating the attributes using the aforementioned .update() method which has been **polymorphed**. Then, we use .draw() to update what the user sees. As mentioned in development iteration one, .draw() deletes the previous instance automatically, so the movement of the enemy looks fluid.

This is what the result looks like. **Refer to Appendix Entry Two: Video Evidence Folder for a dynamic demonstration.**



Termination Logic Error; Discussion of Space/Time Requirements

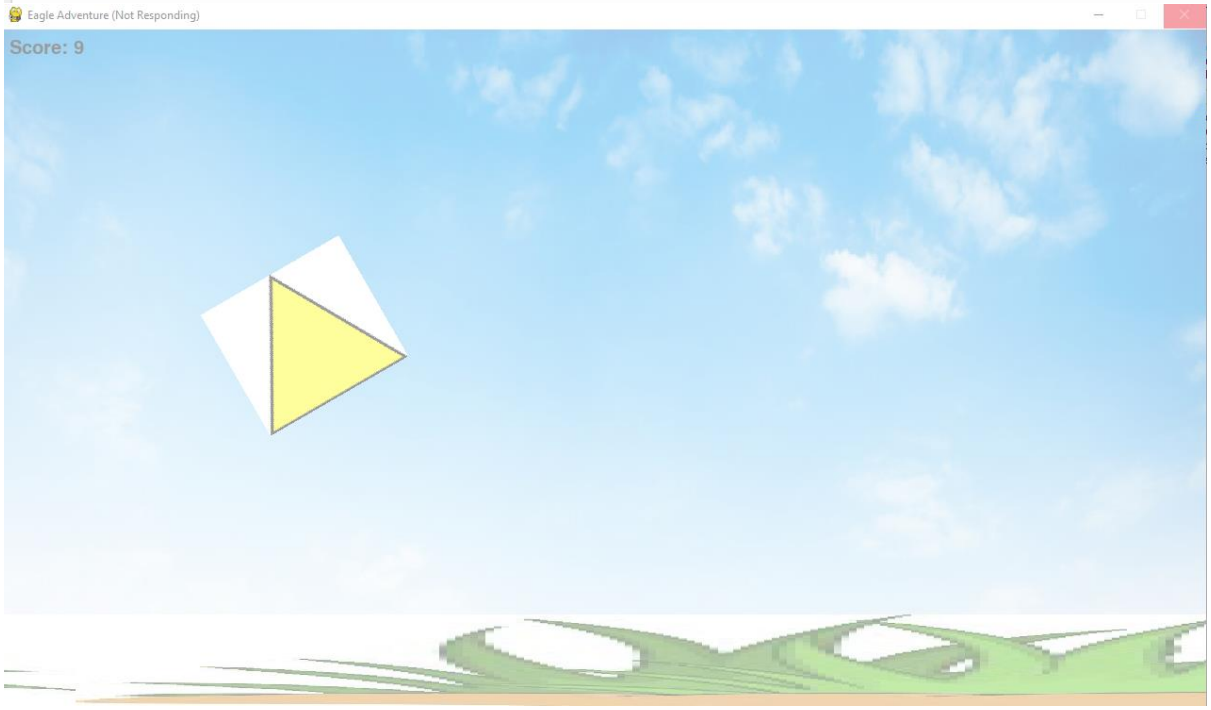
PROBLEM:

The game was spawning enemies at every frame whenever the score was a multiple of 10, leading to excessive resource consumption.

```
pygame 2.6.1 (SDL 2.28.4, Python 3.10.6)
Hello from the pygame community. https://www.pygame.org/contribute.html

Process ended with exit code 3489660927.

Python 3.10.6 (C:\Users\agraw\AppData\Local\Programs\Thonny\python.exe)
```



The scoreboard outputs this error once it gets to ten points.

Notably, there is no syntax error as there is no Python error returned, only an IDE exit code.

SOLUTION:

The exit code is essentially a resource issue because too much code is being run. We know this because the error is an IDE exit code. By googling the exit code in my IDE's documentation (see References), I realised that this exit code is returned when the program gets stuck on a particular line of code.

This implies that there is a loop that executes a very large number of times.

By inspecting the recently added code and using a tracing line:

```
print('I am called')
```

we discover that the game loop is trying to render too many user eagles. This is because, the game_loop() recognises every single frame, not just the first, frame, with a score of ten points, as a frame requiring an enemy spawn.

This increases the time to execute by a factor of the number of iterations, this is what caused the error.

Then, we replace:

```
if self.score % 10 == 0:
    self.spawn_enemy()
```

with:

```
if self.score % 10 == 0 and self.score != self.last_enemy_spawn_score:
    self.spawn_enemy()
    self.last_enemy_spawn_score = self.score  # Update last spawn event
```

Where we initialise last_enemy_spawn_score as -1.

I had an impromptu chat with my stakeholder James, to get his opinion on whether the on-screen-elements should be generated geometrically or not, since, this would dramatically reduce the space requirements. I argued that this would improve the scalability of *Elegant Eagles.*

We concluded that this was unnecessary, since the user image was simply a stand-in for an animation. Then any scalability improvements would be contradictorily temporary. We do not need to implement mask collision detection since this is built into PyGame and the Eagle Class. This is the benefit of using mask collision rather than comparing coordinates – it is incredible comprehensive and therefore scalable.

PROBLEM:
It generates at score =0 because the code is mod10=0, and 0 satisfies this, so add if clause

SOLUTION:

```
if self.score % 10 == 0 and self.score != self.last_enemy_spawn_score:
    self.spawn_enemy()
    self.last_enemy_spawn_score = self.score  # Update last spawn event
```

Using the modulo 10 logic, we will simply initialise the 'last_enemy_spawn_score' to be zero, such that, the condition does not evaluate to true at score = 0.

To see a demonstration of this feature, refer to Appendix Entry Two: Video Evidence Folder.

Improvements to the Save/Load Feature

PROBLEM:

Enemies are not being loaded in the Save; Load feature

SOLUTION:

We need to update he save/load feature.

Instantiating and Saving Specific Attributes States Using a Two Dimensional Dictionary as a Hash Table

Save Algorithm

First, we need to pass the correct data into the algorithm:

```
def save_game(code, eagle, obstacles, enemies, score):
```

The change here is that we now pass the list of enemies.

Next, we need to open up the persistent store of saves, if this does not exist, we must create it. We do the latter using the FileNotFoundError.

```
try:
    #attempt to load existing save data from the save file
    with open(SAVE_FILE, 'r') as file:
        save_data = json.load(file)
except (FileNotFoundError, json.JSONDecodeError):
    #if the file doesn't exist or is corrupted, initialize with an empty dictionary
    save_data = {}
```

Note that, if there is an error decoding the JSON file (a sign of corruption), then we also create a new file. This allows the algorithm to always proceed.

Next, we need to update the hash table so that it stores the enemy attributes:

- Vertical velocity
- X coordinate
- Y coordinate

```
#store game data in a dictionary under the provided code
save_data[code] = {
    "eagle": {"x": eagle.rect.x, "y": eagle.rect.y},  #store the eagle's position
    "obstacles": [{"x": obs.rect.x, "y": obs.rect.y, "width": obs.rect.width, "height": obs.rect.height} for obs in obstacles],
    #store all obstacles with their positions and dimensions
    "enemies": [{"x": enemy.rect.x, "y": enemy.rect.y, "speed": enemy._speed} for enemy in enemies],
    #store all enemies with their positions and speeds
    "score": score  #store the current score
}
```

In the save feature, as explained previously, we utilise a dictionary to create the hash table:

- Dimension one refers to the type of object stored in the JSON file.
- Dimension two refers to the specific attribute of a given object stored in the JSON file.

We do this to improve the rigour of the load function because the structure is explicit; this means that different OSEs are stored separately and clearly via the key of dictionary entries.

Another example of the rigour produced by a hash table is in handling the edge case of there being no enemy object. If instead of using a hash table I used a list structure that follows a pattern e.g. 'fourth item is the score', then there would be an issue if there is no enemy object to save, this would require additional validation that we can do without by using a hash table.

In addition, the use of a hash table facilitates scalability since if we want to save/load other pieces of data in the future, for example, another dimension corresponding to types of enemies, then, we don't need to make complicated cascading changes throughout the program because there is no dependence on the order in which data is saved to the JSON.

Load Algorithm

Next, we need to add to the load algorithm. We need to use the data saved to the persistent store to instantiate relevant objects.

```
#check if the "enemies" key exists in the save data
if "enemies" in save_data:
    #iterate over each enemy data in the save data
    for enemy_data in save_data["enemies"]:
        #create a new LOSEnemy object using the enemy data
        enemy = LOSEnemy(enemy_data["x"], enemy_data["y"], enemy_data["speed"])
        #add the enemy object to the enemies list
        enemies.append(enemy)
```

How this works is, we search the hash table for every entry where the key is 'enemies'. Then, we repeat in the second dimension of the dictionary, so that we instantiate the relevant enemy object with the correct attributes.

PROBLEM:

We have no way to choose which type of enemy to instantiate.

Decision Based Instantiation

SOLUTION:

First, we must save the enemy type into the JSON.

```
save_data[code] = {
    "eagle": {"x": eagle.rect.x, "y": eagle.rect.y},
    "obstacles": [{"x": obs.rect.x, "y": obs.rect.y, "width": obs.rect.width, "height": obs.rect.height} for obs in obstacles],
    "enemies": [
        {
            "x": enemy.rect.x,
            "y": enemy.rect.y,
            "speed": enemy._speed,
            "type": enemy._enemy_type.lower() if enemy._enemy_type.lower() in valid_types else 'astar'  # Validate enemy type
        }
        for enemy in enemies
    ],
    "score": score
}
```

I used line breaks to annotate the code to improve readability. Each line corresponds to a separate attribute being saved for each enemy. Also, for the sake of readability, I use aftermath list comprehension syntax so bring attention to the attributes.

Next, we must alter the load algorithm for enemies by considering cases based on the enemy_data[“type”] for entries in the dictionary where save_data is of the key: “enemies”.

```
#check if the "enemies" key exists in the save data
if "enemies" in save_data:
    #iterate over each enemy data in the save data
    for enemy_data in save_data["enemies"]:
        #determine the enemy type and instantiate the corresponding class
        if enemy_data["type"] == "LOS_Enemy":
            enemy = LOS_Enemy(enemy_data["x"], enemy_data["y"], enemy_data["speed"])
        elif enemy_data["type"] == "ML_Enemy":
            enemy = ML_Enemy(enemy_data["x"], enemy_data["y"], enemy_data["speed"])
        elif enemy_data["type"] == "AStar_Enemy":
            enemy = AStar_Enemy(enemy_data["x"], enemy_data["y"], enemy_data["speed"])
        else:
            raise ValueError(f"Unknown enemy type: {enemy_data['type']}")

        #add the enemy object to the enemies list
        enemies.append(enemy)
```

PROBLEM:
When solving this issue and altering the save algorithm, I noticed that there is a scalability issue. Only one obstacle is being saved.

Optimising for Scalability

SOLUTION:

I didn’t notice this in Development Iteration Two because currently, *Elegant Eagles* always has exactly one obstacle. However, in future iterations, we will add more obstacles.

Then, the solution is a simple matter of copying the syntax from what we did with the enemies:

```
#store all obstacles using a for loop
"obstacles": [
    {
        "x": obs.rect.x,
        "y": obs.rect.y,
        "width": obs.rect.width,
        "height": obs.rect.height
    }
    for obs in obstacles
],
```

The way we save obstacles is the exact same, but we added the line 'for obs in obstacles' to iterate through the list of obstacles.

PROBLEM:

There is insufficient validation in the save algorithm:

Rigorous Validation

SOLUTION:

Here, I will list off the way we are validating and then the method of validation; to see why we validate refer to the penultimate sub-chapter in Design:

- In Development Iteration Two, we validate the save file for existence/corruption and we validate for a non-negative integer score.
- Validating for the existence of save code and validating for a string save_code. This is a more time efficient method of validating against erroneous saves than by validating using the is_valid() method. If a saved code exists and is a string, we can reasonably expect that it will be valid, **but we validate nonetheless in Development Iteration Two**.

  ```
  #validate save code
  if not isinstance(code, str) or not code:
      print("invalid save code! must be a non-empty string.")
      return
  ```

  The isinstance() method is a Python method that lets us check if 'code' is an instance of a string (str). Next, we use the 'not' Boolean operator to validate against the null case.

- Validating for the existence of User Eagle coordinate coordinates; this is the only instance of validation when instantiating User Eagle coordinates and can easily lead to unexplained errors without it. This ensures that the User Eagle object is instantiated correctly.

  ```
  #validate eagle attributes
  if not hasattr(eagle, 'x') or not hasattr(eagle, 'y'):
      print("invalid eagle object! missing coordinates.")
      return
  ```

- Validating for the float nature of User Eagle coordinates.

```
#validate eagle coordinate types
if not isinstance(eagle.x, float) or not isinstance(eagle.y, float):
    print("invalid eagle object! coordinates must be float values.")
    return
```

- Finally, validating for the existence of a velocity attribute in the UserEagle

```
#validate eagle velocity attribute
if not hasattr(eagle, 'velocity'):
    print("invalid eagle object! missing velocity attribute.")
    return
```

Test Table for Iteration Three

The notable addition to this test table is the comments column.

Test #	Game Function	Input	Type of Input	Justify Test	Outcome	Review
1	Jump	Space bar	Valid	Ensures jumping remains functional	Character successfully jumps when space is pressed	Success
2	Invalid Jump Key	Up arrow	Invalid	Retest ensures fatigue development had no unintended repercussions	Nothing happens as intended.	Success
3	Max jump frequency is controlled by jump fatigue	Pressing spacebar at a very large frequency	Valid	Controls the user's jump fatigue.	The rate of jumping should decrease at a decreasing rate, but it did not. Resolved	Resolved
4	Jump Fatigue	Rapid Space Bar Presses	Valid	Ensures jump frequency decreases over time	Jumping slows down correctly	Success
5	Air Resistance	Movement in air	Valid	Ensures air resistance behaves correctly	Character's velocity decreases	Success
6	Wind Effect	Random wind change	Valid	Ensures that wind moves the user horizontally.	Horizontal displacement is altered by wind	Success
7	Rain Effect	Enter Rotation Mode	Valid	Tests rain interacting with wind	Upwards force varies randomly. The change is at a uniform frequency.	Success
8	Save Game	Click	Valid	Save works with new features	Game state is saved and reloadable	Success
9	Load Game	Click + Enter Save Code	Valid	Checks loading of saved state	Game loads as expected	Success
10	Load Invalid Save Code	Incorrect formats: non-	Invalid	Ensure incorrect codes do not load	Error message appears	Success

		integer, non-6-digit, mixed				
11	Difficulty Selection (Scroll Bar or Text Box)	Integer between 0 and 10	Boundary	Tests accepted range for difficulty	Game starts with chosen difficulty	Success
12	Invalid Difficulty Input	Below 0, above 10, non-integer	Invalid	Tests rejection of bad inputs	Error message shown	Success
13	Invalid Leaderboard Entry	Enter Symbols	Invalid	Only alphanumeric names allowed	Error shown; re-prompted	Success
14	Time Eagle Slow Mode Activation	Press 'T' as Time Eagle	Valid	Ensure correct key activates slow mode	Slow mode is activated	Success
15	Time Eagle Slow Mode Limit	N/A	Valid	Ensure slow mode is time-limited	Slow mode lasts only 3 seconds	Success
16	Manual Time Eagle Slow Mode De-Activation	Press ‘T’ in slow mode	Invalid	Ensures that, as intended, the user must remain in slow-mode for the full duration, to justify the cooldown.	No change	Success
17	Time Eagle Slow Mode Cooldown	Attempt to reactivate slow mode immediately after ending	Invalid	Ensure cooldown is enforced	User must wait 10 seconds before reusing slow mode	Success
18	A* Enemy Pathfinding	Enemy movement	Valid	Tests shortest walkable path	Enemy follows correct path	Success
19	A* Enemy Avoids Obstacles	Move around objects	Valid	Ensures obstacle avoidance	Enemy path curves around obstacle	Success
20	Character Switch: valid keys [1–4]	Key press 1–4	Valid	Switch between characters	Character changes	Success
21	Character Switch: invalid keys 0 or 5	Key press 0 or 5	Boundary	Ensure input range is enforced	No change	Success
22	Character Switch: repeat same key	Key press same character	Boundary	Should produce no change	No visual difference or state change	Success
23	A* Enemy collision with obstacle	Obstacle collision	Valid	Tests permanent de-spawn logic	Enemy de-spawns	Success
24	Enemy spawns every 10 points	Score milestones	Valid	Tests scoring-based spawning	Enemies spawn at correct intervals	Success
25	Holistic pathfinding match to reference (A*)	Observe path taken	Valid	Ensure that enemy path-finds in the intended style so that the user experience is tailored correctly. For instance, this prevents the enemy from being too easy to avoid or impossible to avoid.	Matches intended chase logic: • There is a noticeable lag time in the enemy’s reactions to new stimuli. • The enemy effectively avoids obstacles. • The enemy is always in motion. The enemy accelerates at a uniform frequency.	Success

Stakeholder Review Three – Were the Objectives Achieved?

James expressed his satisfaction with the final stage of *Elegant Eagles*, noting that the game had evolved into a polished and engaging experience. However, he offered a few minor suggestions to enhance user experience further. For one, he recommended that the 'heavy' character instead have negative gravity; reflecting a 'stage' from Geometry Dash, one of inspirational games I reference in the Analysis section. We decided that the idea needs work given that the characters function to make the game easier, with an initial learning curve.

Additionally, James pointed out that the original gravity constant used for the Heavy Eagle was too large, resulting in unbalanced gameplay. He also helped me determine the constants used to govern how jump fatigue varies. Both James and Oliver engaged in hands-on playtesting of various difficulty modes and the A* Enemy to identify more appropriate and balanced values. To see the updated constants in context, refer to Appendix Entry Three: Program Code. We used the methodology outlined in my Design: Development Plan to determine these constants.

Looking ahead, James proposed a future iteration of the project beyond the scope of A Level. He suggested that we implement the ML enemy and have it act as a user given its obstacle avoidance, this would lay the foundation to a two-player game that functions as a one player game.

Oliver added that the A* Enemy initially felt like a real player chasing him, which made the experience more intense and immersive. His feedback supports further development of the predator-prey dynamic as a core mechanic, potentially leading *Elegant Eagles* toward becoming a full-fledged competitive two-player game.

EVALUATION

Testing to Inform Evaluation (Beta Testing)

I had my stakeholders review all the alpha tests from the three development iterations and provide feedback on each feature as they progressed. Below are the results of these tests, along with noteworthy comments from the Beta Testers. The video recordings of these tests can be seen in Appendix Entry Two: Video Evidence Folder.

Pre-Analysis of Tester Demographic

Below, we analyse the sort of people that we distributed *Elegant Eagles* to for testing to inform evaluation. A spread of testers is imperative to both represent the real userbase and to evaluate the extent to which we achieve a given success criterion. **For further justification on testing methodologies, refer to Testing Subsection: *Usability Features…***

Demographic Subset	**Need for subset**	**Subset Size/People**
Children between the	*Elegant Eagles* must be accessible to. The need for this subset is mitigated due to my stakeholder, Oliver, being in this subset.	Three

ages of three and eight		
Older minors (under 18)	Many older minors are avid gamers. Therefore, they will likely be immediately competent, and we need to keep them entertained. In addition, this subset is a major subset and therefore the most important subset, as indicated by column three. The need for this subset is mitigated due to my stakeholders James and myself being in this subset	Eleven
Adults	*Elegant Eagles* should be accessible to mature individuals, including those with limited video game experience – such as some elderly persons.	Nine

Test Table with Post Development Testing

Evidence of Post Development Testing can be found in Appendix Entry Two: Video Evidence Folder and in 'Notable Tester Comments' below

Test #	**Game Function**	**Input**	**Type of Input**	**Justify Test**	**Outcome**	**Review**	**Notable Tester Comments**
1	Jump	Space bar	Valid	Lets the user jump	Character successfully jumps when space is pressed	Success	Jumping feels responsive and smooth. The animation makes it feel dynamic rather than floaty.
2	Invalid Jump Keys	Up arrow, 'X', or '@'	Invalid	Ensures users do not accidentally jump or circumvent difficulty of rapid jumps	No jump occurs; input is ignored	Success	
3	Play	Click	Valid	Lets the user play	Game starts successfully	Success	
4	User touches obstacle	N/A	Valid	Obstacle collision logic	Game transitions to menu screen	Success	Instant reset keeps the pace high. Feels fair.

5	User touches floor	N/A	Valid	Fall detection	Game transitions to menu screen	Success	
6	User moves above field of view	N/A	Boundary	Vertical boundary (ceiling) handling	Game over triggered	Success	
7	Obstacle touches ground or ceiling	N/A	Valid	Engine robustness; obstacles should never de-spawn	Obstacles persist	Success	
8	Quit Program	Alt F4	Valid	Lets user close the program to stop playing	Game closes immediately	Success	
9	Quit Program	Escape	Invalid	Ensures user cannot accidentally close the program	Game remains open; no unintended exit occurs	Success	
10	Quit Program	Click quit button	Valid	Standard GUI exit behaviour	Game exits and closes correctly	Success	
11	Options Menu	Click options button	Valid	Placeholders for future feature planning	Returns text: "Options will be created in future development"	Success	
12	Collision with obstacle (user eagle)	N/A	Valid	Critical collision logic for game challenge	User eagle de-spawns	Success	
13	User eagle collision with ceiling or ground	N/A	Valid	Enforces game space boundaries	User eagle de-spawns	Success	
14	GUI Play Button	Click 'Play' on GUI menu	Valid	Lets user start game via interface	Game starts successfully	Success	
15	Quit Program	X button on window or Alt+F4	Valid	Ensure multiple standard ways to exit the game work as intended	Game closes immediately	Success	
16	Increment Score	Pass Pipe Obstacle	Valid	This is currently the only way to gain score	The scoreboard increases its displayed value by 1	Success	Works well. Maybe display a confirmation message

							before loading.
17	Invalid Score Increment	All other events e.g. pressing esc. or pressing space	Invalid	Ensures score is only incremented when necessary	Score is not incremented	Success	Error message is clear and prevents confusion.
18	Load Game (valid code)	Entering save code 036742 (6 digit integer) where the save code exists in the JSON	Valid	Lets user load a game state	The game is reloaded back to the desired game state. The save state is preserved in the JSON	Success	Loading works well, but a small preview of the save state before loading could be useful.
19	Load Game (invalid code)	Entering save code ‘036742’ (6-digit integer), ‘A06461’ (non-integer), or ‘0461’ (non 6-digit integer) where the code does NOT exist in the JSON	Invalid	Ensures invalid save codes do not create game states	Alert returned telling user that save code is incorrect. System accepts new save code to be input	Success	
20	Save Game using ‘k’	Press 'k'	Valid	Provide save shortcut	Save code displayed to user; feature success is determined by the load feature	Success	
21	Invalid Save Game Key	Press unassigned key e.g. '&', space, or ‘@’	Invalid	Prevents unintended saves	No save created; input ignored	Success	
22	Game Instance Ends on Collision	User eagle undergoes any mask collision	Valid	Lets the user save a ‘run’ to continue later	Game returns to menu screen	Success	

23	Load Game Menu Popup	Click 'Load Game' on the main menu	Valid	Lets user enter a save-load code	Load Game Menu Pops up and the user can immediately type in the load-game input bar	Success	
24	View Leaderboard	Click the Leaderboard button	Valid	Lets user view leaderboard entries	Leaderboard pops up containing past scores from the JSON file	Success	Leaderboard loads fast, but a sorting option (e.g., alphabetical) would be useful.
25	Invalid Leaderboard Access	Invalid input (e.g. keyboard press from menu)	Invalid	Ensures user doesn't accidentally get sent to the leaderboard	No action occurs; input is ignored	Success	
26	Enter Data into Leaderboard under unique name	Enter Name and Score (alphabetical and numerical)	Valid	Lets the user enter new high scores	JSON file saves leaderboard entry	Success	The overwrite prompt is useful but could be clearer.
27	Enter Data into Leaderboard under non-unique name	Enter Name and Score (duplicate name)	Boundary	Lets the user enter new high scores under coincident names	User is prompted to update the record under the same name or to create a new entry	Success	
28	Enter non-alphabetical name	Enter name using characters like £$%&(	Invalid	Ensures names only contain valid characters	The name is rejected, and the user is alerted that the name is non-alphabetical. The user is prompted for another input	Success	
29	Enter excessively long name	Name is more than 30 characters long	Boundary	Ensures user cannot enter names so long that the leaderboard becomes unreadable	The name is rejected, and the user is alerted that the name is too long. The user is prompted for another input	Success	

30	Overwrite entry with larger score	Type ‘Y’ AND new entry is larger	Valid	Lets the user update their own entry	Leaderboard entry under same name is overwritten with new score	Success	Updating works great. Maybe highlight the changed entry.
31	Overwrite entry with smaller score	Type ‘Y’ AND new entry is not larger	Boundary	Prevents the user from updating their own entry if it's worse	Alert returned telling user that their new score is lower than the previous input. The leaderboard entry remains unchanged	Success	Good that it stops me from replacing my best score.
32	Decline overwrite of score	Type ‘N’ after name conflict	Valid	Creates Duplicate	Creates a leaderboard entry with same name	Success	Having an option is great. Prevents accidental overwrites.
33	Submit incomplete leaderboard entry	Enter only name or only score	Valid	Ensures user cannot enter an incomplete entry to the leaderboard	Alert returned stating that an entry is missing. The user is prompted for another input	Success	
34	Scroll leaderboard	Arrow	Valid	Lets user look through different leaderboard entries	Leaderboard scrolls as expected	Success	Scrolling is smooth. Maybe add a scrollbar for mouse users.
35	Max jump frequency is controlled by jump fatigue	Pressing spacebar at a very large frequency	Valid	Controls the user’s jump fatigue.	The rate of jumping should decrease at a decreasing rate, but it did not. Resolved	Resolved	Fatigue mechanic adds depth; a UI hint would clarify why jumps slow.
36	Invalid Jump Key	Up arrow	Invalid	Retest ensures fatigue development had no unintended repercussions	Nothing happens as intended.	Success	
37	Jump	Space bar	Valid	Ensures jumping is functional	Character successfully jumps when space is pressed	Success	

38	Jump Fatigue	Rapid Space Bar Presses	Valid	Ensures jump frequency decreases over time	Jumping slows down correctly	Success	Fatigue mechanic adds depth. Took a moment to realize why jumps slowed down—maybe a UI hint would help.
39	Air Resistance	Movement in air	Valid	Ensures air resistance behaves correctly	Character's velocity decreases	Success	Feels natural. Maybe display a slight drag animation to reinforce the effect.
40	Wind Effect	Random wind change	Valid	Ensures that wind moves the user horizontally.	Horizontal displacement is altered by wind	Success	Wind adds a nice challenge. A subtle visual cue would help players anticipate gusts.
41	Rain Effect	Enter Rotation Mode	Valid	Tests rain interacting with wind	Upwards force varies randomly. The change is at a uniform frequency.	Success	The rain feels great! Would be cool if it affected visibility slightly for added difficulty.
42	Save Game	Click	Valid	Save works with new features	Game state is saved and reloadable	Success	Works well, but a small preview of the save state before loading could be useful.
43	Load Game	Click + Enter Save Code	Valid	Checks loading of saved state	Game loads as expected	Success	Works fine. Maybe add a confirmation

							pop-up before loading.
44	Load Invalid Save Code	Incorrect formats: non-integer, non-6-digit, mixed	Invalid	Ensure incorrect codes do not load	Error message appears	Success	
45	Difficulty Selection (Scroll Bar or Text Box)	Integer between 0 and 10	Boundary	Tests accepted range for difficulty	Game starts with chosen difficulty	Success	Having options is great. Maybe show a brief explanation of how difficulty affects the game.
46	Invalid Difficulty Input	Below 0, above 10, non-integer	Invalid	Tests rejection of bad inputs	Error message shown	Success	
47	Invalid Leaderboard Entry	Enter Symbols	Invalid	Only alphanumeric names allowed	Error shown; re-prompted	Success	
48	Time Eagle Slow Mode Activation	Press 'T' as Time Eagle	Valid	Ensure correct key activates slow mode	Slow mode is activated	Success	Feels powerful but balanced. SFX is satisfying.
49	Time Eagle Slow Mode Limit	N/A	Valid	Ensure slow mode is time-limited	Slow mode lasts only 3 seconds	Success	Clear that it ends naturally—maybe flash a timer near end?
50	Manual Time Eagle Slow Mode De-Activation	Press ‘T’ in slow mode	Invalid	Ensures that, as intended, the user must remain in slow-mode for the full duration, to justify the cooldown.	No change	Success	Good that it's fixed duration, adds commitment to decision.

51	Time Eagle Slow Mode Cooldown	Attempt to reactivate slow mode immediately after ending	Invalid	Ensure cooldown is enforced	User must wait 10 seconds before reusing slow mode	Success	Adds tension—forces smart use of power-up.
52	A* Enemy Pathfinding	Enemy movement	Valid	Tests shortest walkable path	Enemy follows correct path	Success	Pathfinding works well. Sometimes feels a little too perfect—maybe add slight randomness?
53	A* Enemy Avoids Obstacles	Move around objects	Valid	Ensures obstacle avoidance	Enemy path curves around obstacle	Success	Impressive! Enemy movement feels smart but not unfair.
54	Character Switch: valid keys [1–4]	Key press 1–4	Valid	Switch between characters	Character changes	Success	
55	Character Switch: invalid keys 0 or 5	Key press 0 or 5	Boundary	Ensure input range is enforced	No change	Success	
56	Character Switch: repeat same key	Key press same character	Boundary	Should produce no change	No visual difference or state change	Success	
57	A* Enemy collision with obstacle	Obstacle collision	Valid	Tests permanent de-spawn logic	Enemy de-spawns	Success	This is very necessary. Otherwise, the game would be very difficult to win.
58	Enemy spawns every 10 points	Score milestones	Valid	Tests scoring-based spawning	Enemies spawn at correct intervals	Success	This is a healthy spawn rate. It allows a suitable pause for

							rest between the enemy and its successor.
59	Holistic pathfinding match to reference (A*)	Observe path taken	Valid	Ensure enemy path-finds in the intended style	Matches intended chase logic: noticeable lag time, obstacle avoidance, uniform acceleration	Success	

Examples of Remedial Action

I remedied syntax and logic errors in:

- **The Development Diary in the form**
  PROBLEM:…
  SOLUTION…
- Terminal Testing at the End of Each of the Three Iterations

However, there were four issues that slipped through these checks. The sheer volume and therefore variety of tests to inform were development were essential to identify them:

Issue	**Remedy**
Improper Validation of Difficulty Meter	My Test Cases were Incomplete, I failed to test the edge case of difficulty=0. This is an erroneous edge case because it is not possible for the game to have *zero* difficulty. This was a simple fix of changing a '>=' to a '>'
On occasion, the ML and AStar Enemies did not spawn when they were supposed to.	This was perplexing; the logic of 'spawn every ten points' is simple and a sufficient number of seconds passes between the score increments so the system successfully detected the current score state. **I eventually realised that since this happened sporadically, I should look to my randomised modules for the cause.** It turns out that the enemies would on occasion spawn on top of obstacles and therefore despawn instantly due to mask collision. This was confirmed by some testers reporting that the enemies 'blipped' onto the screen for a few frames before disappearing. This taught me that animations can be used to aid testing by showing *why* events like 'disappearing' enemies happens. To remedy this, I restricted the spawn coordinates of the enemy to be in a range of y coordinates where no obstacles could spawn.
Users would load a saved instance and lose	The rounding I used in my JSON file to save memory meant that if the user eagle was paused near to an obstacle, it might be loaded on top of said obstacle. This evaded my terminal testing because a pre-requisite for this issue is that the user is very close to the obstacle.

	To solve this, I simply increased the precision of saved values.
If two jump actions are performed just before mask collision, then the user would jump into the obstacle.	This detracted from the user experience since it seemed like the obstacles were not solid objects. It turns out that this was only possible at the start of the run. This indicated that the issue was in the jump-fatigue feature. I solved this by implementing a non-zero initial jump fatigue such that users could not jump twice rapidly even on the first jump.

Whilst these issues were minor and did not detract from *Elegant Eagles*'s function, fixing them has made *Elegant Eagles* a 'more rigorous and professional product' according to my stakeholder, James.

Evaluation of Solution

Cross-Referencing Test Evidence with Success Criteria

Here, I reference the criteria number [see above] with the relevant test evidence. Where possible, I referenced a video, see Appendix Entry Two: Video Evidence Folder.

The videos demonstrate many features that were not planned as success criteria.

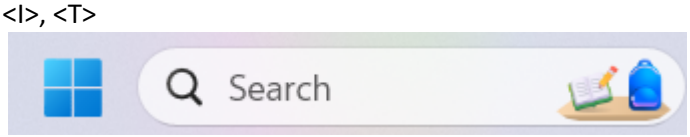
I will use the following tags to clarify what type of evidence I will show

Tag	**Descriptor**
<V>	Video Number: <V2> refers to Video Number Two For a List of Videos see Appendix Entry Two: Video Evidence Folder
<X>	Unmet Criteria; Solutions can be found in the Sub Chapter called, 'Success Criteria that were Not Completely Met and Criteria; How to Meet them through Future Development'
<T>	Text: Evidence will Use Text
<I>	Image: At least one image will be added net to every <I> tag

Evidence Table

Note that many of the unmet criteria <X> have been adapted and are not failures, but a result of changing project demands.

Criteria Number	**Evidence**
1	<V> Seven
2	<X>

3	<X>
4	<X>
5	<X>
6	<X>
7	<X>
8	<X>
9	<V> Three, Four
10	<X>
11	<V> Two
12	<V> Video Evidence Two
13	<V> Two
14	<X>
15	<X>
16	<V> Seven
17	<X>
18	<X>
19	<V> Two, Three, Four, Six, Eight
20	<X>
21	<V> One
22	<V> Two, Five
23	<V> Six, Eight
24	<V> One, Two, Three, Four, Five, Six, Eight
25	<V> One, Two, Three, Four, Five, Six, Eight
26	<V> One, Three, Four, Five, Six, Eight
27	<V> One, Two, Three, Four, Five, Six, Eight
28	<I>, <T> 
	My system runs on the OS Windows 11.
29	<V> One, Two, Three, Four, Five, Six, Eight

30	<V> One, Two, Three, Four, Five, Six, Eight
31	<V> Three, Four, Six, One, Five
32	<V> Five, Two, Seven
33	<V> Two
34	<V> One, Two, Three, Four, Five, Six, Eight
35	<V> One, Two, Three, Four, Five, Six, Eight
36	<V> Two
37	<V> Seven
38	<V> Two
39	<V> Two
40	<V> One, Three, Four, Five, Six, Eight
41	<V> Two
42	<V> One, Three, Four, Five, Six, Eight
43	<V> One
44	<V> Six, Eight
45	<V> One
46	<V> One
47	<V> One
48	<V> Six, Eight
49	<V> Seven

How Partially or Unmet Success Criteria can be met through **Future Development**

Here, we explore how my program compares to my success criteria from my Analysis. First, we will explore partially or unmet success criteria:

Success Criteria Number	**Success Criteria**	**To what extent was the criterion met?**	**How to meet the criterion through future development**	**Other Improvements**
1	Store high scores, user login credentials, and game state using a persistent data system	Partial	Use a central database so users can reuse save instances across devices.	By using a central database, users could reuse save instances across a range of devices.

2	Implement a life system to track player progress and allow multiple attempts per session	N/A	Give users extra lives through a temporary power-up (OSE) upon mask collision.	Instead of having lives, they currently have one life. The idea of multiple lives inspired the feature of having multiple Eagle Characters.
3	Display power-ups with icons and timers in the GUI to inform players of active effects	Fail	Use a PowerupDisplay class inheriting from Menu to manage icons and durations.	Each powerup would be stored as an object and deleted when its duration reaches zero.
4	Enable user accounts for saving and retrieving progress across multiple sessions	N/A	Implement a UserAccount class connected to a database to save/load progress.	This would allow players to save and load progress across sessions.
5	Use both sound effects and visual feedback to signal player damage	Fail	Add sound effects and visual cues through a PlaySound method in Spritebase.	Helps improve code readability and scalability.
6	Animate wing flapping dynamically based on player speed and input for realism	Fail	Create an AnimatedEntity superclass and override an animate() method in UserEagle.	AnimatedEntity would inherit from SpriteBase for reuse.
7	Develop a two-player mode with cooperative/competitive mechanics	Fail	Add a secondary UserEagle class for local multiplayer using different input keys.	Can later extend to online co-op with networking logic.
8	Create a machine-learning-based pathfinding enemy	Partial	Train and store behavior in a model per user in a persistent database.	Each account would have 1 ML Enemy Object.
10	Make the ML enemy more reactive and human-like	Partial	Add pseudo-random noise and reduce learning delay for more human-like behavior.	Improves realism and prevents robotic movement.
11	Allow eagle design customization	Partial	Update GUI to allow eagle selection before run start.	Add fatigue to prevent rapid character switching.
12	Fully customizable key-bind settings	Fail	Add key-binding options in the GUI and store user preferences.	Requires an account system to retain settings.
13	Accessibility options like colorblind palettes and TTS	Partial	Add dynamic accessibility settings that adapt to user needs.	Currently static and non-personalizable.
14	Multiple difficulty settings	Partial	Use sliders for enemy speed/pathfinding to let users customize difficulty.	Input values instantiate different class objects and behavior levels.
15	Custom level editor	Partial	Add drag-and-drop GUI and store obstacle layouts in persistent data.	Use saved attributes to instantiate objects during runs.
16	Interactive tutorial	Partial	Add a dedicated tutorial game with stepwise inputs and success alerts.	Link to tutorial video from GUI as fallback.
17	UI customization	Fail	Create a UserUI class with attributes like font size, stored per user.	Instantiated at login for personal setup.
18	Replay system	Partial	Use visual recording and large save files for replay viewing.	Users can download and review gameplay sessions.
19	Haptic feedback support	N/A	Add haptics when mobile compatibility is added through third-party support.	Requires mobile port of the game.

20	Dynamic music system	Fail	Base music changes on score, enemy count, and RMS of enemy distance.	Use weighted factors from stakeholder feedback.
21	Multiple game modes	Partial	Create GameMode class with children like TimeAttack or Survival, overriding modifyRules().	Show previews of rule changes before game starts.
22	Multiplayer functionality	Fail	Add local multiplayer and future online support with rollback networking.	Latency handled with interpolation.
23	Simulated weather effects	Partial	Apply weather effects to all entities and animate weather changes.	Update constants dynamically to improve scalability.

Completely met criteria

24. Ensure graphical user interface (GUI) is intuitive and easy to use
25. Generate dynamic obstacles that the player must avoid
26. Run without crashes or critical bugs that disrupt gameplay
27. Execute game logic with precision, producing consistent and accurate outputs
28. Allow player to control character's vertical movement smoothly and responsively
29. Be simple enough for players aged 3+ to learn within one minute of play
30. Run on multiple Windows versions and hardware configurations
31. Execute game logic with precision, producing consistent and accurate outputs
32. Feature an aesthetically pleasing, sleek design with smooth, rounded UI elements
33. Trigger game over when player collides with a mask object
34. Use object-oriented programming (OOP) for managing levels, pausing, and tracking player attributes like speed and health
35. Include start screen options for resuming gameplay, viewing leaderboards, and accessing account information
36. Provide enough variety to encourage 'replayability' and long-term engagement
37. Use a distinct colour palette that enhances immersion and differentiates elements
38. Sort high score table in descending order
39. Allow players to pause and save progress at any time
40. Start a new game from the main menu without errors
41. Display a clear start screen with necessary options for game navigation
42. Increase score by 1 point per avoided obstacle
43. Display final score on game-over screen
44. Show and allow modification of horizontal velocity based on player actions
45. Allow the player to control or have weather generated randomly/based on score
46. Implement a chasing enemy using the A* algorithm, considering heuristics and distance to player
47. Implement physics-based movement with accurate acceleration, drag, and collision responses
48. Display how drag affects the eagle's movement based on speed and direction
49. Simulate different weather conditions like rain affecting eagle movement

Evidence can be found below.

Usability Features and their Success through Testing with Evidence and Justification

Here we will discuss both **unmet and met usability features**

I distributed the program files to my stakeholders and asked them to share the game with their friends who play similar games. This approach allowed me to gather feedback to **test usability** from individuals who fit within my target user group but had no prior experience with my game.

Additionally, I took steps to minimize bias in stakeholder selection. Since my stakeholders were initially chosen from my personal network, there was a risk that their feedback might not fully represent the broader player base. By expanding the pool to include their friends, I increased the sample size and diversified the perspectives in my survey, resulting in more reliable and representative data about **usability**.

To **test usability** requested my stakeholders to observe and document their friends' experiences, noting their comments and feedback throughout gameplay. Additionally, they asked targeted questions at various points to gauge user perception of specific features. These questions were structured to assess intuitiveness and enjoyment, such as:

- How intuitive is X feature?
- How helpful is X feature?

Each question was tailored using stakeholder feedback to suit the relevant feature, for example, one cannot ask if the 'play button' is 'helpful' since its necessary – instead, my stakeholders asked, 'how easy was it to find the play button'. Question responses will be accompanied by an open request for general comments. This is crucial to prevent **bias** from the question set. This ensures that **usability testing** feedback is complete and therefore truly representative of the player base opinion. Below, I have outlined each feature along with the most notable and insightful feedback received.

Finally, responses were **quantized**, i.e., we had surveyed users give answers on a scale of 1-10. Such quantitative data ensured that the most pertinent issues were addressed first.

<u>Usability Testing Feedback Report</u>

Below is a diary of the feedback I received on every **usability feature**. The dialogue is representative of the responses feedback my stakeholders received for each question. The speech is comprised of quotes exclusively. Throughout, I will calculate the percentage of surveyed individuals who agree in at least one facet.

Diary Format:

- Question
- Responses

...

<u>Main Menu:</u>

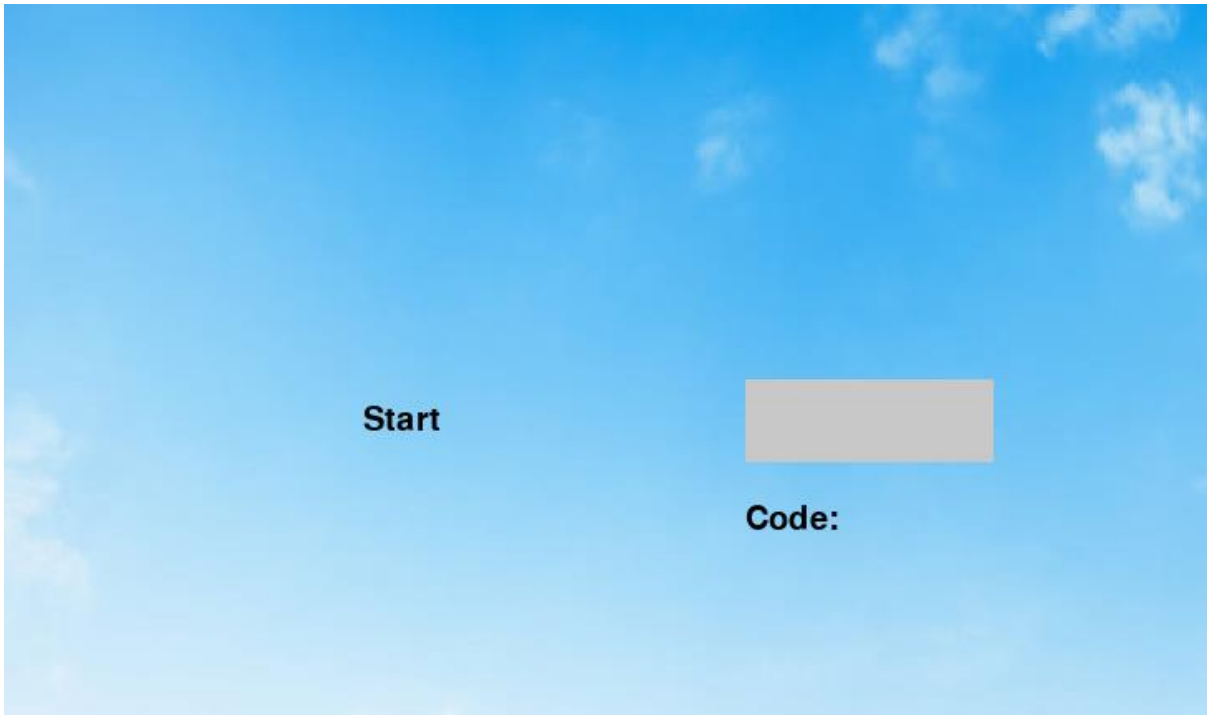
- **How intuitive is this feature:**
  '10/10, the background blends in with the menu sleekly and the buttons are self-explanatory, large and easy to find'.
- **Do you like the feature:**
  '9/10, I'm neutral on the menu but I don't see how you could improve it. I don't think I'd give any 'main menu' from any of the games I play a '10''.

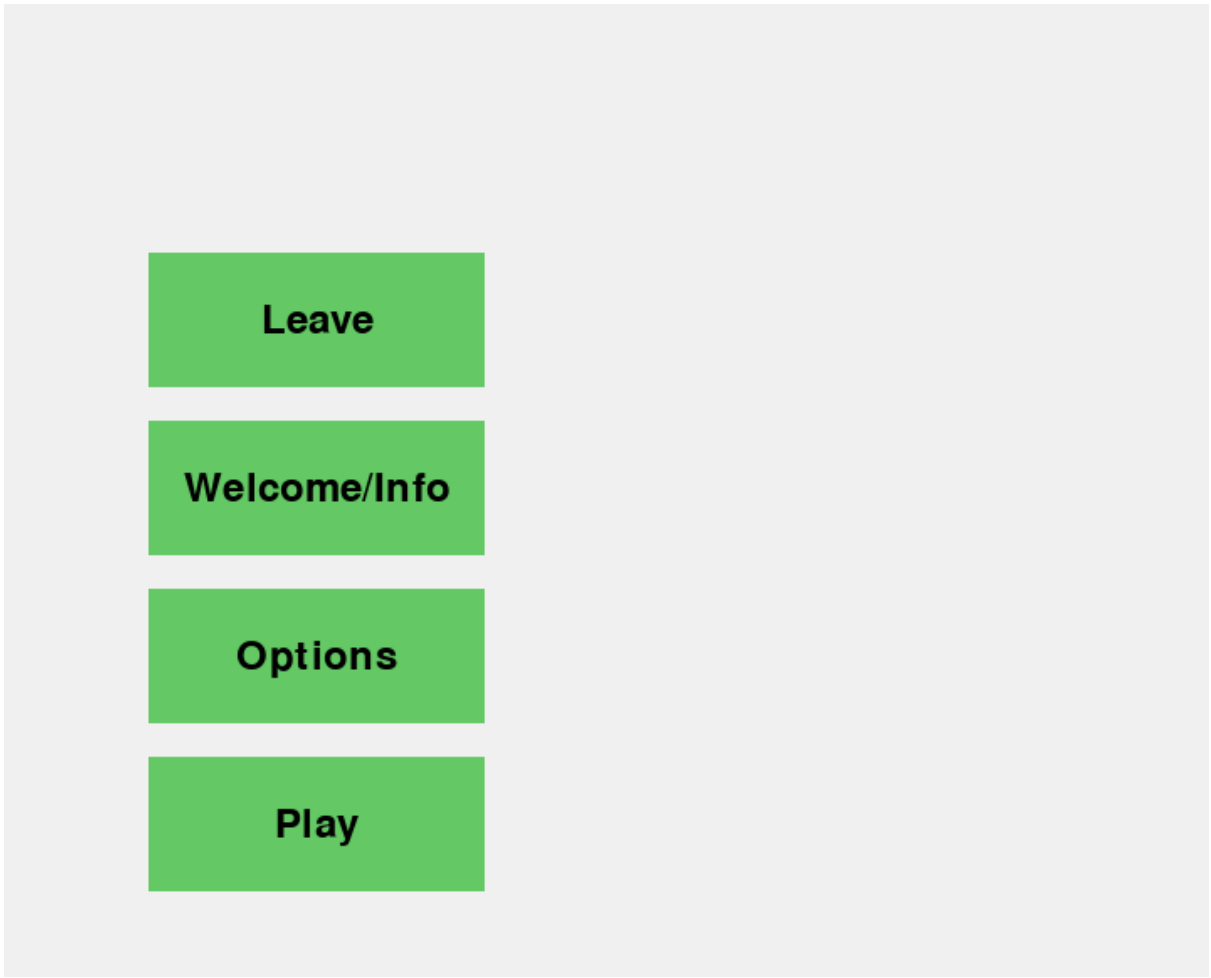
- **General Comments:**
  "I like how the buttons become dark when my cursor hovers over them, but I wish that they'd also enlarge to make it really clear where my cursor is, I often lose my cursor on the screen because the background is light'.

How to Improve

#	Issue Identified by Testers	My Proposed Solution with Stakeholder verification	Intended Method of Implementation
1	Users lose track of the cursor on the screen due to the light Background.	1. Enlarge buttons slightly when hovered over. 2. Add a subtle glow effect to highlight the cursor's position.	Modify the `Button` class to override the `on_hover()` method, increasing size dynamically. Apply a glow effect by modifying the `draw()` method in the UI renderer to add an animated border.

2	No option to customize the menu layout or settings from the main menu.	1. Add a 'Settings' button where users can adjust scale; keybinds etc.. 2. Allow users to change button size and contrast.	Instantiate a new `SettingsMenu` class that inherits from `MenuScreen`. Store preferences in a persistent JSON configuration file and update the UI dynamically using an observer pattern. To make use of the users keybinds, we must 1. Add input OSEs to the SettingsMenu class. 2. Update the save feature so that every account saves a **dictionary** of inputs. Where for every item, the key is the name of the key bind and the value the key bind itself. This means that we can query this dictionary when accepting inputs. Whilst I could use a hash map, the trade-off of space for time complexity is not worth it since, no matter the size of Elegant Eagles, the number of inputs will be small to make the game easy to learn, so a dictionary does not hinder scalability.
3	The menu performs well but could be optimized for faster load times.	1. Preload assets before displaying the menu. 2. Reduce unnecessary re-renders.	Implement asset preloading in the `MainMenu` class by loading textures asynchronously before menu display. I could also optimize rendering by checking state changes before re-drawing elements.


Start
Code:


Leave
Welcome/Info
Options
Play

Save/Load Feature:

- **How intuitive is this feature:**

3. ‘7/10, the layout is clean and straightforward, with clear labels. Add a ‘Settings’ button where users can adjust scale; keybinds etc..
4. Allow users to change button size and contrast.

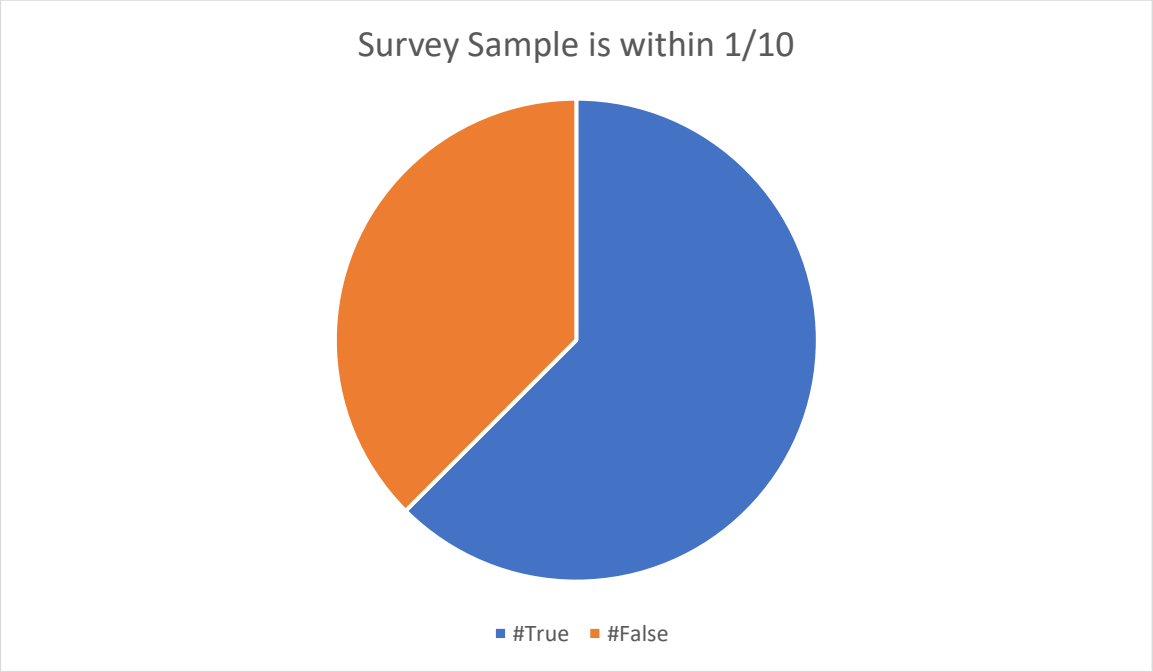
" Having both options on the same screen without needing a submenu is efficient and user-friendly. However, the use of save codes instead of a traditional save/load system feels a bit unintuitive since it’s not the norm in most games. A brief explanation or tutorial for first-time users could help bridge this gap.’

- **Do you like this feature:**

“Being able to save my progress in long games is usually a requirement for me to invest in it to begin to like it, so this feature allows me to enjoy the game. I love the simple, sleek look of this screen, and I like being able to continue games where I last left off. Even though save codes aren’t the norm, I really like this feature because it feels unique and adds a layer of personalization to the experience.”

- **How visually appealing is the feature?**

‘7/10, the design is minimalistic and functional, but it feels a bit plain compared to the main menu. Adding dynamic visuals like a faint background animation could make it more immersive and visually engaging.’



Here, we see lots of variation; this shows that the difficulty bar is a contentious feature. **This indicates that future development should focus on the following improvements. We should then**

conduct another survey to ensure changes match the wants of the Beta Testers.

How to Improve

#	Issue Identified by Testers	My Proposed Solution with Stakeholder verification	Intended Method of Implementation
1	It is not immediately obvious how to use the Save Feature	1. Video tutorial 2. Use of tooltips	Use of a hyperlink OSE via webbrowser.open() from Pygame. Add an attribute to the class of OSEs called 'hover action'; this will generate a local alert box with predetermined text that aims to explain features.
2	Users often lose track of the contents of their save codes	1.Thumbnails to show the current state of the save code 2. Return the attributes to the user e.g. showing user the score.	To implement thumbnails, I will essentially create a virtual machine that will load the instance as if the user were playing it, but shrink the display to a small size, make the game static and remove the possibility for input. To implement the latter, I will adapt the pause feature so that there is no pop-up menu. Returning the attributes will be a simple manner of using an f string to return attributes in a user-friendly format to not contradict issue number one. Given that users will be unable to interpret/visualise coordinate attributes, using stakeholder feedback in the would-be development stage 5, certain attributes would not be displayed.
3	Users wish they could label their saves in the application itself, rather than needing to write it down in a WordPress software.	1. When users press save, they can enter a name. 2. When users enter a valid code, before they	Entering a save name is a trivial matter of creating an additional OSE/Input Box. The only necessary validation will be length non-zero and sufficiently small. To return the name before entering, validation will need to be performed upon every keystroke. Returning errors after the user types the first character will be frustrating, so instead, any failed validation will not be returned; to make minor code

			press enter, the name is returned.	optimisations, the validation algorithm will only run in its completeness if the length of user input is of a certain size. Given len() is O(1), this will save runtime resources.

Overall, the consensus is that the feature is creative and effective, but difficult to use because it is new.

Movement

- **How intuitive is this feature?**

'9/10 - The specific key binds follow the industry standard, so, the movement feels natural. I felt like the movement was smooth and realistic; the movement was not janky yet it was still very dynamic.'

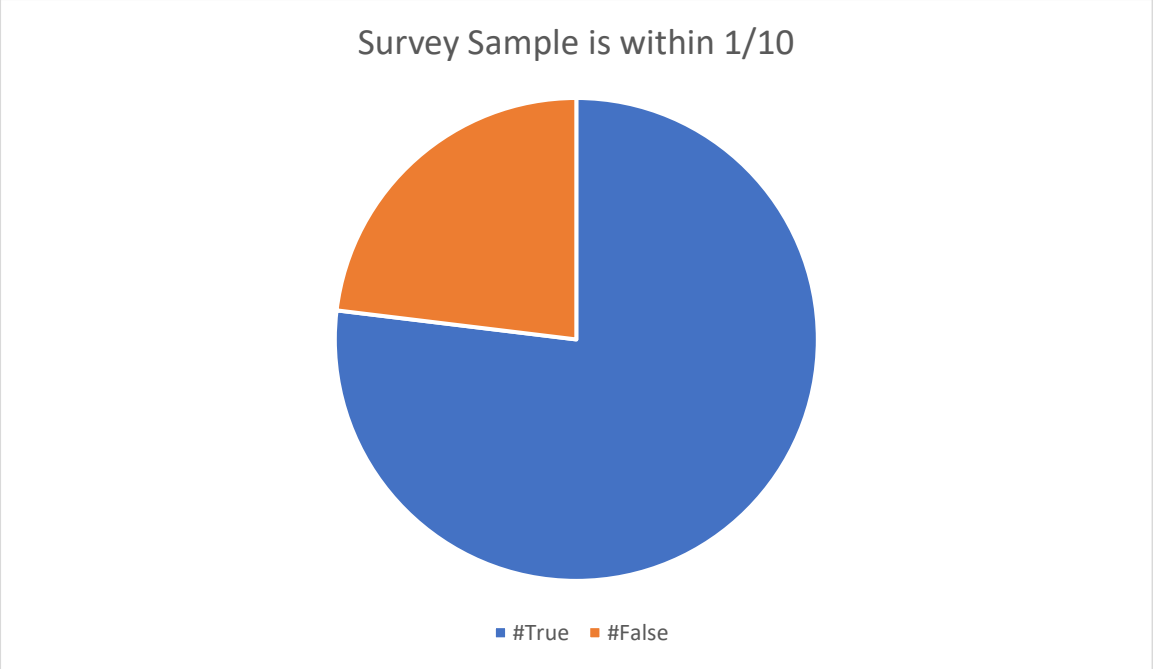
- **Do you like this feature:**

'9/10 I really like the granularity of weather physics; it adds a twist to a defined genre of platformers. The <jump fatigue> mechanic adds a sophisticated layer of difficulty; every input feels meaningful in contrast to games like the 'Dinosaur Game' where there is room for filler inputs.

- **How Visually Appealing is this Feature?**

7/10 The movement is smooth and fluid and responsive. However, immersion can be improved by using motion blur, and visual effects like jet streams due to drag. Small details like a soft wing-flap animation when gliding or a subtle background shift when reaching higher speeds could make movement feel even more dynamic.

For a visualisation, see Video Evidence 1 in Appendix Entry Two: Video Evidence Folder
https://bromsgroveschool-my.sharepoint.com/:f:/g/personal/20nagrawal_bromsgrove-school_co_uk/EvUf42AuviRJmUrBECYmFZsBrTCWB0NEzkFdjBqcin7yhg?e=xkjhgC



How to Improve

#	Issue Identified by Testers	My Proposed Solution with Stakeholder verification	Intended Method of Implementation
1	It is not immediately obvious how much jump fatigue the user experience has	1. Add an indicator for jump fatigue.	The maximum possible jump fatigue should be determined mathematically, i.e., the jump fatigue as a result of spamming the space bar. An OSE must then be created that updates iteratively and displays the proportion of max fatigue experienced. This can be made immersive by using a metre that resembles a thermometer.
2	The movement feels bland at times; the use of visual and sound effects could improve immersion	1. Implement a range of animation stages 2. Implement sound effects for jumping	For animation, I would update the image attribute of the UserEagle object. We would call a .animate() method which uses the .pause() method from Pygame to make the loop sufficiently slow. The specific time required for pausing would be determined using stakeholder testing. This would be performed in a similar way to how I calculated constants for jump fatigue in phase three. For sound effects, we call a 'play sound' function in the .launch() method. To create this function, we will use existing Pygame modules.

Difficulty Meter

- **How intuitive is this feature?**

8/10 - Whilst difficulty bars are a ‘familiar concept’ to seasoned gamers, the ‘continuous rather than discrete input was difficult to grasp’ for the newbie gamer - many tried writing ‘easy’ rather than a number.

A common proposed solution was ‘disallowing alphabetical input rather than validation’ let users type alphabetical characters in the first place; rather than rejecting the characters to save user effort. I could implement this feature using

‘Use of a slider or preset difficulty options (e.g., Easy, Normal, Hard) could make the process even more intuitive.’

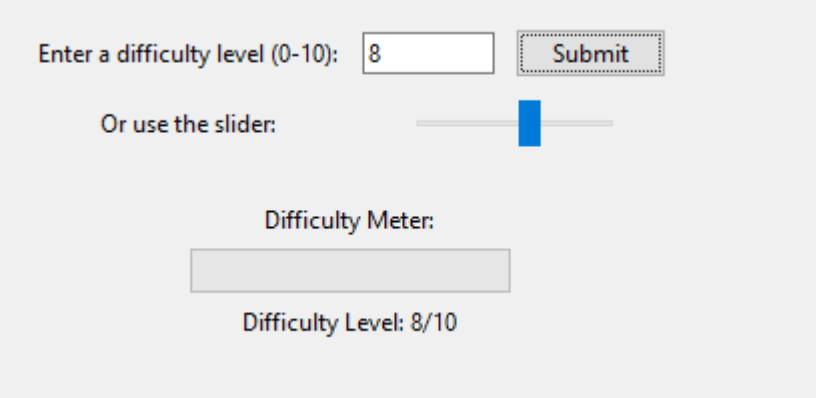
- **Do you like this feature:**

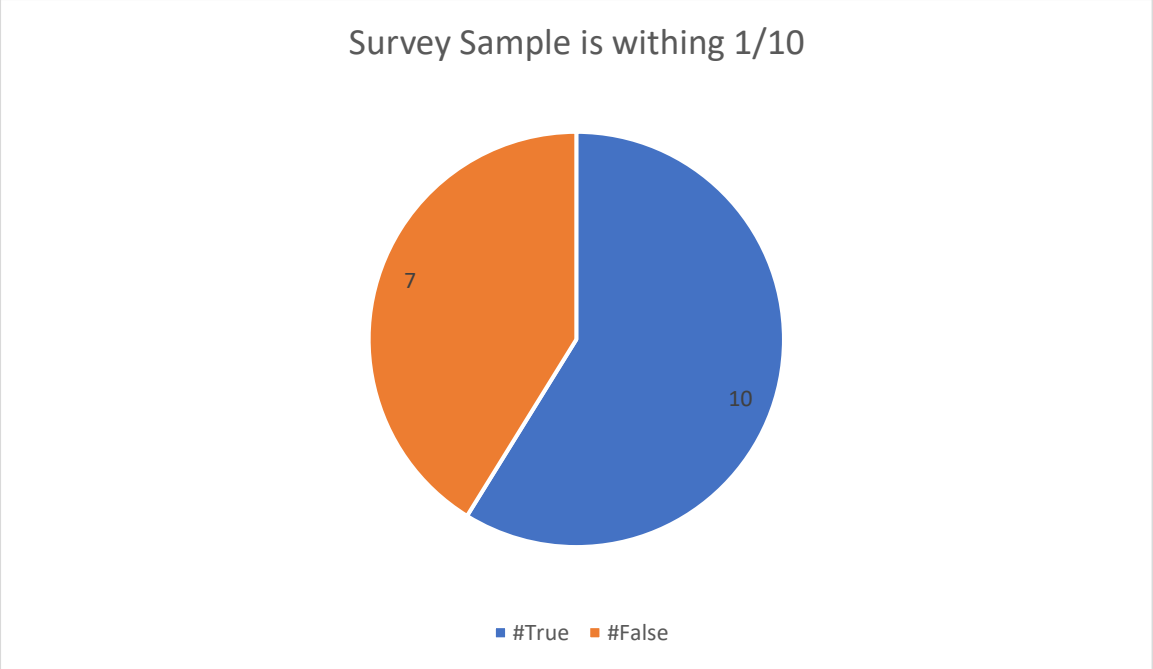
‘8/10 Once I understood how to use it, the flexibility allowed me to tailor the game to my skill level. However, the same flexibility made comparing scores with friends difficult, as, the leaderboard features a range of difficulties. The leaderboard could perhaps be separated by difficulty in future development’.

That said, the difficulty scaling is well executed - harder settings feel genuinely challenging rather than artificially punishing. Unlike static difficulty settings in other platformers, this system gives players control over their own experience.’

- **How Visually Appealing is this Feature?**

‘9/10 - The UI for the difficulty meter is clean and functional. The only room for improvement is a way to show the user the tangible difficulty that they are choosing. For example, an oscillating enemy in the GUI that moves as fast as it will in the real game.’





As with the save/load feature, we see lots of variation; this shows that the difficulty bar is a contentious feature. **This indicates that future development should focus on the following improvements. We should then conduct another survey to ensure changes match the wants of the Testers.**

How to Improve

#	Issue Identified by Testers	My Proposed Solution with Stakeholder verification	Intended Method of Implementation
1	Users cannot grasp how tangibly difficult a score of 'X' is.	Create a preview of the game with the updated difficulty	Visually, as I described in a previous table in the evaluation, I will essentially create a virtual machine that will load the instance as if the user were playing it, but shrink the display to a small size, make the game static and remove the possibility for input. To implement the latter, I will adapt the pause feature so that there is no pop-up menu.
2	User's find choosing the difficulty difficult	Create discrete game-modes like easy medium and hard, then enable users to enter a continuous difficulty value in an 'advanced mode'.	Create OSEs corresponding to: easy; medium; hard; advanced. The action to be called in the first three OSEs is difficulty_return() with a predetermined set of attributes. I will pass these attributes by using a for i in x loop where x is an array. The advanced OSE will prompt the user for the old method of input.

Enemies and Obstacles

- **How intuitive is this feature?**
  '10/10; it is immediately obvious that I must avoid the enemies and obstacles, since these are common mechanics in platformers like Jetpack Joyride [which I reference in my analysis].
  The behaviour of the enemy is extremely intuitive: the AStar enemy constantly pursues my location but cleverly avoids walls and barriers—its movement feels like a mix of precision and pressure, and this keeps the pacing tight and satisfying!'

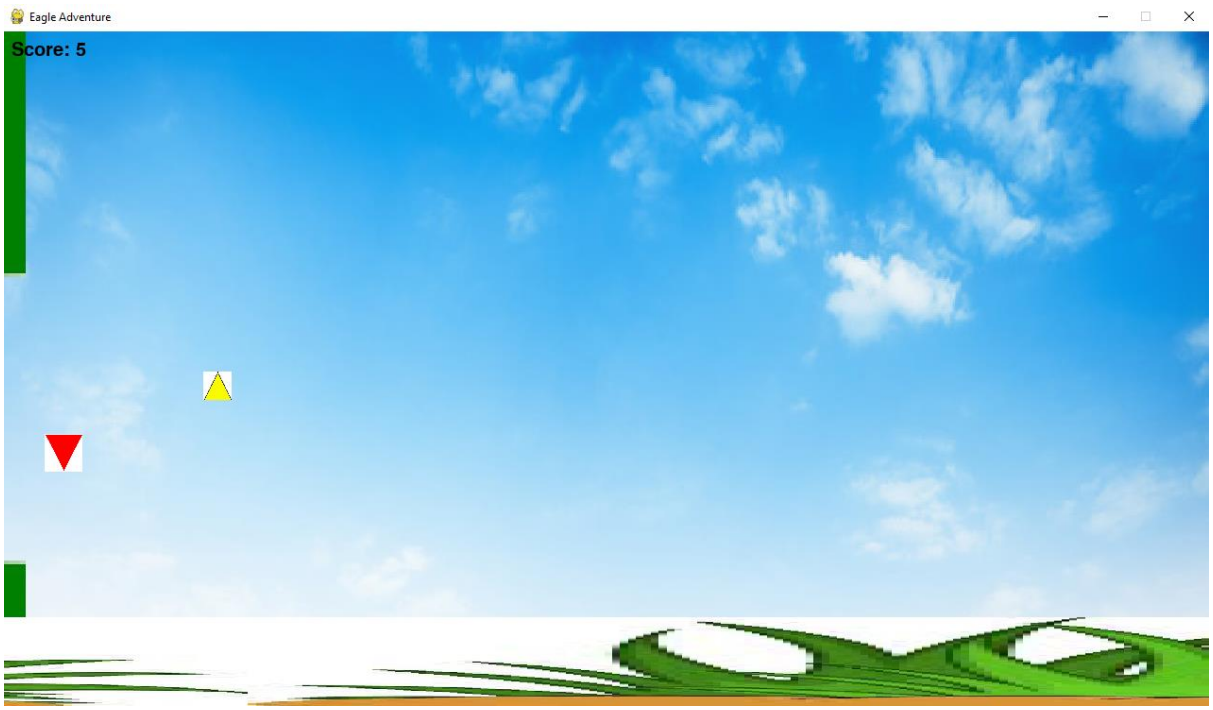
- **Do you like this feature:**

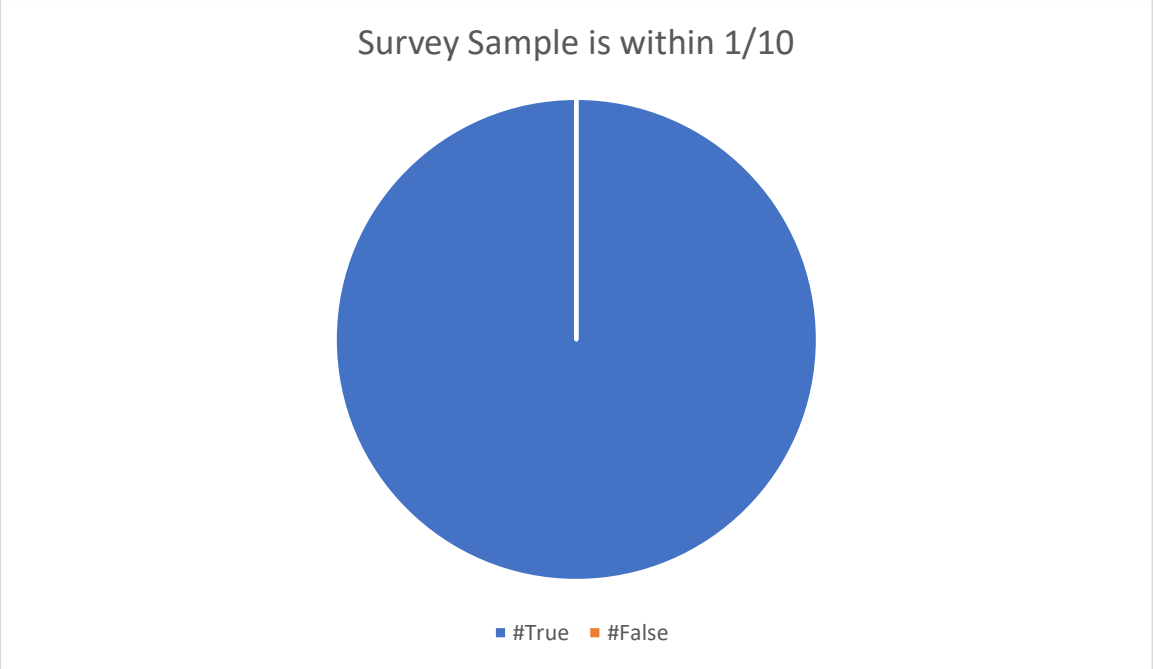
'8/10 Once I understood how to use it, the flexibility allowed me to tailor the game to my skill level. However, the same flexibility made comparing scores with friends difficult, as, the leaderboard features a range of difficulties. The leaderboard could perhaps be separated by difficulty in future development'.

That said, the difficulty scaling is well executed - harder settings feel genuinely challenging rather than artificially punishing. Unlike static difficulty settings in other platformers, this system gives players control over their own experience.'

- **How Visually Appealing is this Feature?**

'9/10 - The UI for the difficulty meter is clean and functional. The only room for improvement is a way to show the user the tangible difficulty that they are choosing. For example, an oscillating enemy in the GUI that moves as fast as it will in the real game.'





How to Improve

#	Issue Identified by Testers	My Proposed Solution with Stakeholder verification	Intended Method of Implementation
1	A* Enemy becomes predictable over time	1. Introduce pseudo-random variation in decision-making Add a small probability P that the enemy chooses a near-optimal path instead of the perfect one. This prevents the enemy from feeling overly robotic. P will be refined via stakeholder testing and applied by modifying node selection logic in the A* algorithm. 2. Add variable delay to path recalculation Introduce a randomized delay (e.g. 250–600ms) between pathfinding updates to simulate	1. Inject controlled randomness: Introduce a tunable probability $P \in [0.1, 0.2]$ that biases A* toward a near-optimal node (e.g. 2nd or 3rd best f(n)), simulating human-like imperfection. 2. Throttle path recalculation: Apply a randomized cooldown $\Delta t \in [250, 600]$ ms between A* invocations using pygame.time.get_ticks() to emulate delayed reactions. 3. Bias smoother routes: Modify the cost function to include a “turn penalty” — routes with fewer direction changes will be preferred over technically shorter ones. This creates more natural, flowing movement patterns.

		reaction time. This gives players brief windows to reposition and makes the enemy feel more human-like. Implemented using pygame.time.get_ticks() for timing control. 3. Occasionally prioritise speed over shortest path Modify the A* cost function to sometimes favour smoother, faster routes over minimal-distance ones. This creates more realistic movement and forces the player to adapt. Weighting will be calibrated through testing to ensure balance.	
2	Enemies Look too Similar	Introduce visual variety by changing colours, sizes, and animations. Vary movement patterns slightly	Change the image file path for each of the enemies. In the future I will implement animations for the enemies. These animations will vary between the enemies. We would call a .animate() method which uses the .pause() method from Pygame to make the loop sufficiently slow. The specific time required for pausing would be determined using stakeholder testing. This would be performed in a similar way to how I calculated constants for jump fatigue in phase three.
3	AStar enemy is too difficult to beat	Reduce search granularity for more human-like pathing. Introduce slight randomness or error to pathfinding. Limit enemy reaction speed or add a delay before changing paths. Allow some obstacles to be "confusing" for the enemy.	Add a slight probability of the enemy making a random choice; the easiest way to implement this is using a probability P (to be determined via stakeholder testing), that the algorithm will not be run. To make some obstacles ‘confusing' for the enemy, the weighting of the obstacle would be decreased so that the enemy is more likely to collide with it. To introduce a reaction time, the pathfinding enemy would run the exact same algorithm but using data from a previous decision iteration. This means that the game state from a previous iteration will be

			saved and the enemy will run the A* algorithm using the saved data. How 'previous' will be determined using stakeholder testing. To reduce the granularity of the coordinate plane, I just need to change the rounding algorithm of the ML enemy's coordinate system to be more precise. For example, instead of rounding to the nearest unit, I will round to the nearest 0.5 units.

Upon reflection, the choice of algorithms for this project may not have been the most suitable. While both the A* Enemy and ML Enemy achieve the intended user effect to some extent, their practical limitations hinder their effectiveness.

The ML Enemy suffers from slow learning, which prevents it from adapting in a meaningful way - the initial state carries out most of the decision-making. As a result, the enemy behaviour appears static over time, failing to demonstrate the intelligent progression that was initially intended.

The A* Algorithm, on the other hand, is most effective in well-defined graph structures. My implementation involved converting a coordinate plane into a graph representation, centred around hyper dense paths about obstacles. By reducing granularity, I introduced artificial imperfections rather than leveraging a more natural or efficient approach. Moreover, my gravitational model of warping space detracts greatly from readability because it is a verry abstract concept. A pseudorandom algorithm could have been a better alternative. This would have been less computationally expensive, less time taking and more readable.

Maintenance Issues and Limitations + Potential Improvements

Overall, I believe my program is complete, functional, and viable. While I have successfully met most of my criteria, there are a few aspects that I was not able to fully implement. However, I plan to address these in future updates.

Save Feature

One of the biggest long-term maintenance issues involves the save code system. Whilst the code itself is built for performance scalability, the concept of a save code is currently imperfect.

1. Each save is assigned a unique code, but as more players create saves, the available codes may eventually run out.

- Proposed Solution: I could Increase the length of the save codes to generate a larger pool of unique codes. However, this creates a major issue—older save codes would become invalid, potentially deleting past progress for players.
- Mitigation Strategy: Implement a system that allows both old and new save codes to exist simultaneously for a transitional period. Alternatively, convert old save codes to the new format automatically upon loading.

2. Save Security Risks

Since save codes are player-generated and relatively short, there is a risk that someone could guess another player's code and access their progress. This poses a security concern, especially for competitive or progression-based gameplay.

- Proposed Solution: I would Introduce a lockout feature, where multiple failed attempts to enter a save code temporarily disables access to that save. This would prevent brute-force entry.

  Players could be given an optional password or PIN tied to their save code for added security. This PIN could be unique to their account and utilise 2FA. This would be complex to implement because it requires validation of the user's internet connection and working on a server.

3. Compatibility Issues with Future Updates

As the game evolves, new features or mechanics might break compatibility with older save files. For example, if a future update adds a new type of enemy or world, old save files may not recognize these attributes properly. This would result in errors/objects would be instantiated with the wrong attributes.

- Proposed Solution: Implement a versioning system for save files. When a player loads an old save in a newer version of the game, the system should automatically convert outdated save data into a compatible format.
- **Alternatively, I could utilise the rigorous validation from developer phases two and three and simply allow values to be omitted using default values. Any new saved data would be at the end of the array of attributes and therefore would not displace any saved data.**

4. Bug Fixes & Performance Optimization

With continued updates, new bugs may emerge, and maintaining smooth performance across different devices could become challenging.

- Proposed Solution: Establish a bug reporting system where players can submit issues directly from the game menu.
- Moreover, it is possible that if I need to perform a hotfix without testing, I might create game breaking bugs e.g. overwriting save codes. As a solution, I would store **backups** of the persistent data store to further the **rigour** of *Elegant Eagles.*
- Performance Considerations: Introduce graphical settings that allow players to adjust game performance based on their device capabilities.

5. Server & Multiplayer Maintenance

If multiplayer is introduced in the future, server upkeep and network stability become major concerns. Maintaining online functionality requires constant monitoring, updates, and possibly a dedicated server infrastructure.

- Proposed Solution: Use cloud-based services or peer-to-peer networking to reduce server costs. Implement automatic server restarts and monitoring tools to detect downtime.

1. Expanding Content Without Breaking Balance

Adding new content like weapons, enemies, or worlds can be exciting, but it can also create balancing issues, making the game too easy or too difficult.

- Proposed Solution: Before releasing major updates, conduct playtesting sessions to ensure new content integrates smoothly without disrupting game balance.

Disuse of SI Units for Physical and Mathematical Formulae

As I entered phase three, I realised that, not only would elaborate formulae be required to model motion accurately, but that said formulae rely on SI units and well-established constants. My decision to work in arbitrary units in phase one, though time saving in the short term, could make future development iterations challenging.
The cascading use of constants eyeballed through 'stakeholder feedback' will create imprecision and therefore a poor model of reality.

- Proposed Solution: Convert existing formulae such as those controlling gravity and air resistance to SI units using a single constant to be determined through rigorous stakeholder feedback. Given that there is only one constant to be determined, more time can be spent on determining the precise value and the compound uncertainty will be zero.

While the game is in a stable and playable state, maintenance challenges such as save system security, future updates, performance optimisation, and multiplayer support will require ongoing attention. By implementing structured versioning, security measures, and playtesting, these challenges can be managed effectively, ensuring the game remains functional and engaging for years to come.

Future Updates & Enhancements

To improve the game over time, I would release updated versions that:

- Implement intended updates yet to be addressed
- Address new challenges that may arise after deployment.
- Implement additional features based on stakeholder feedback.

Some challenges that I anticipate but cannot pre-emptively solve are

- Air resistance is negligible; I cannot fix this yet because I run the risk of making the coefficient of air resistance too small and therefore making the feature pointless
- Change the jump fatigue constant; it is possible that users find jump fatigue too challenging of a mechanic to cope with; after all, as developer, I have bias as an **avid gamer who has played games with similar mechanics.**
- The difficulty bar doesn't create hard enough runs; there will certainly be users more skilled than me and my stakeholder team. Through practice they may find the game trivial.

Other necessary updates would be determined through user feedback. To enable user feedback, I could create a feedback-report option in *Elegant Eagles,* but this would require an internet connection. An easier method would be providing an email address in the GUI.

To make these updates easily accessible, I would add a "CHECK FOR SOFTWARE UPDATE" button in the main menu. This would lead to a new screen where users could see if a newer version is available and download it directly.

I would implement this using an X-community; sub-Reddit; or website. Here I would list the versions in the following order:

- V1.0.0 DOWNLOAD NOW
- V1.1.0 <Explanation of updates> DOWNLOAD NOW
  …

Expanding Game Content

One concern I recognise is that playing in a single closed world could become repetitive. To enhance engagement, I would:

- Add multiple, larger worlds featuring different settings and biomes to change the mood of the game.
- Introduce **dynamic, procedurally generated** worlds so each playthrough feels unique rather than having predefined maps.
- Introduce a "CHOOSE WORLD" button in the main menu, allowing players to select their preferred environment.
- Offer downloadable content (DLCs), such as new skins and character customizations, to keep the game fresh and encourage player investment.
- Introduce companion AI or pets, allowing players to customize and level up an ally that aids them in combat or exploration.

Addressing Current Limitations

Many of the limitations I identified during the analysis stage are still applicable. However, one major feature I would prioritize in future iterations is a multiplayer mode.

Implementing multiplayer would require:

- A networking system to allow real-time interactions between players.
- Ensuring server stability for smooth gameplay.
- Requiring users to have an internet connection to access multiplayer features. Therefore, I would need to validate internet connection when users press play.

While this would introduce additional software and hardware constraints, I believe it would significantly enhance the game's longevity and appeal.

In addition, I have noticed that, because I am in an academic exam year, I spend less than thirty hours per week on this project - in contrast, a full-time developer would spend upwards of fifty hours. Therefore, there are many features, though simple, that I chose not to implement given **time restraints**, for example, I wanted to create an orb obstacle that has no pathfinding. This would be solved by either collaborating with other developers or by simply spending more weeks developing *Elegant Eagles.*

Since the algorithm behind the AStar Enemy determines directional decisions rather than the exact vector of motion, reducing the frequency of these decisions causes the enemy to overcompensate, leading to noticeable positional errors. This introduces a potential future improvement: striking a better balance between pathfinding precision and computational efficiency.

Additionally, I have come to understand that the enemy's behaviour lacks realistic imperfection. This is due to the nature of the environment it traverses. Unlike a traditional sparse graph, the coordinate plane used in *Elegant Eagles* is highly connected—each node (coordinate) has many neighbouring nodes that can be travelled to. As a result, the A* algorithm performs exceptionally well, even when functioning greedily. The sheer number of edges effectively flattens the decision space, allowing the enemy to make very few mistakes.

In future iterations, introducing imperfections—such as **path noise, broad-node quantisation** or delayed decision-making — could help the enemy feel more natural and human-like, further enhancing gameplay challenge and immersion.

Dealing with gravity in enemy behaviour proved to be particularly challenging. Gravity does not affect displacement linearly, which means we cannot simply have the enemy “jump” every iteration to keep up with vertical changes. Jumping every frame would prevent the enemy from falling at all, making its movement feel unnatural and disconnected from the rest of the game's physics. On the other hand, increasing the interval between jumps would allow the enemy to descend—but at the cost of responsiveness, as it would struggle to reach the player in time or react to dynamic movement.

To resolve this, I had two options: either use gravity as a **deliberate design constraint**, making it a known limitation that players could exploit, or **bypass gravity altogether** for the enemy characters. I chose the latter. Enemies in *Elegant Eagles* are not affected by gravity. This allows them to move in clean, responsive lines across the map, enabling the A* algorithm to function with greater precision and making the enemy feel intelligent and aggressive.

While this introduces a discrepancy between the player and enemy physics—players must account for gravity while enemies do not—it was a conscious design choice. Given the scope of the project, simulating gravity-driven motion in the enemy while maintaining real-time pathfinding would have added significant complexity and likely reduced performance. In future updates, I may revisit this decision and experiment with a hybrid model, where enemies follow a gravity-like trajectory but still use A* for prediction.

Despite the challenges, I am confident that my game provides a strong foundation. With future updates, expanded content, and potential multiplayer functionality, I can further refine the experience and ensure long-term engagement for players.

Conclusion

This project, though exasperating at times, has been a very enjoyable educational experience. Without the fear of a looming deadline, it is a calming pastime – encouraging my projects of the future.

From the outset, I knew ambition is easier than implementation. Nonetheless, I underestimated many of *Elegant Eagles*'s features like dynamic obstacles. I have learnt that balancing ambition with practicality is necessary.

This project has honed my practical applications of, graph theory, physics, object-oriented programming, and optimisation decisions. Much of my experience in Computer Science till now has been following instructions. I found that the freedom of choice and the freedom therefore to fail, causes as much indecision as it does flexibility.

This project taught me that persistence and planning can make or break a directed product. Iterative development and stakeholder review; though formal in this project; are skills I will carry forward.

This project has taught me that we must test, refine and scrap ideas before implementation to ensure that the disjoint, technically sound modules can fuse into a cohesive product. I know how a deeper appreciation for the massive programs that we use daily.

This experience has reminded me that failure is an integral part of growth, and that every challenge, no matter how frustrating, is an opportunity to learn and improve.

Appendix

Entry One: Interview Questions

Questions:

- What animation do you want for losing a life?
- What animation do you want for losing all lives.
- What animation do you want for the enemy
- Do you want to be able to win or lose
- If yes how hard do you want it to be to win?
- What kind of theme do you want for the game? What genre, colour scheme.
- Should the eagle make sounds? If yes what?
- What sound do you want the jumps to make?
- Do you want a win/lose animation?

Game play

- How many obstacles per second?
- Do you want lives? How many?

- Do you want to be able to pause.
- Do you want scoreboard while playing?
- How would you lose the game?
- Should the game get more difficult as you progress?
- Do you want a leaderboard?
- Do you want levels? How many?
- Two player?
- Would you want local or online two player?

Entry Two: Video Evidence Folder

Instructions:

The contents of each video file are described in:

1. Evaluation cross referencing with success criteria
2. File names in the One Drive Folder

Videos

Video Evidence Folder

- READ FIRST
- Video Evidence 1; Advanced Physics.mp4
- Video Evidence 2; GUI.mp4
- Video Evidence 3; Characters; Stage 1 Enemy.mp4
- Video Evidence 4; Characters Two.mp4
- Video Evidence 5; Game Over Screen.mp4
- Video Evidence 6; AStar Enemy.mkv
- Video Evidence 7; Save and Load.mp4
- Video Evidence 8; AStar Enemy2.mp4
- Assets
- Miscellaneous Evidence

Some users who were not logged into one-drive have found that only using the first link (which contains the other files) works.

Entry Three: Code Listings

```
import pygame
import random
import time
import json
from pygame.locals import QUIT, KEYDOWN, K_SPACE, K_UP, K_k

#initialize pygame
pygame.init()

#global constants
SPEED = 20  #speed for eagle movement
GRAVITY = 2.5  #gravity affecting the eagle
GAME_SPEED = 30  #initial game speed
OBSTACLE_1_width = 75  #width of obstacles
OBSTACLE_1_HEIGHT = 700  #height of obstacles
OBSTACLE_1_GAP = 300  #gap between upper and lower obstacles
BLACK = (0,0,0)  #color constant for black

#get screen dimensions
infoObject = pygame.display.Info()
_screen_width = infoObject.current_w  #screen width
_SCREEN_height = infoObject.current_h  #screen height

#ground dimensions
_GROUND_width = 2 * _screen_width  #ground width
_GROUND_HEIGHT = 100  #ground height

#initialize pygame mixer for sound effects
pygame.mixer.init()
```

```python
import pygame

class ButtonInterface:
    def __init__(self, button_actions):
        pygame.init()
        self.screen = pygame.display.set_mode((800, 600))
        pygame.display.set_caption('Interactive Buttons')

        self.buttons = [
            pygame.Rect(100, 150, 200, 80),
            pygame.Rect(100, 250, 200, 80),
            pygame.Rect(100, 350, 200, 80),
            pygame.Rect(100, 450, 200, 80)
        ]

        self.button_labels = {1: "Leave", 2: "Welcome/Info", 3: "Options", 4: "Play"}

        #Store the functions to run on button press
        self.button_actions = button_actions

        self.clock = pygame.time.Clock()
        self.running = True

    def handle_button_click(self, button_id):
        print(f"Button {button_id} was clicked!")
        # Update the button label to indicate it was clicked
        self.button_labels[button_id] = "Clicked!"

        # Run the associated function for the button
        if button_id in self.button_actions:
            self.button_actions[button_id]()

    def render_button(self, rect, label, font, base_color, hover_color, surface, mouse_position):
        #Change button color if hovered
        if rect.collidepoint(mouse_position):
            pygame.draw.rect(surface, hover_color, rect)
        else:
            pygame.draw.rect(surface, base_color, rect)

        text_surface = font.render(label, True, (0, 0, 0))  #Black
        text_rect = text_surface.get_rect(center=rect.center)
        surface.blit(text_surface, text_rect)
```

```
pygame.mixer.init()

#base class for all sprites
class SpriteBase(pygame.sprite.Sprite):
    def __init__(self, image_path, xpos, ypos, width, height):
        super().__init__()

        #load and scale image
        self.image = pygame.image.load(image_path).convert_alpha()
        self.image = pygame.transform.scale(self.image, (width, height))

        #create a mask for collision detection
        self.mask = pygame.mask.from_surface(self.image)

        #set position
        self.rect = self.image.get_rect()
        self.rect[0] = xpos
        self.rect[1] = ypos
```

```
45
46  # Class for the player-controlled eagle
47  class Eagle(SpriteBase):
48
49      def __init__(self, speed=SPEED):
50          super().__init__('user_image.png', _screen_width / 6, _SCREEN_height / 2, 50, 50)
51          self.__speed = 0  # Vertical speed
52          self.__score = 0  # Player score
53          self.started = False  # Whether the game has started
54
55      def reset(self):
56          self.rect[1] = _SCREEN_height / 2  # Reset position
57          self.__speed = 0  # Reset speed
58          self.started = False  # Set game to not started
59
60      def update(self):
61          # Apply gravity if the game has started
62          if self.started:
63              self.__speed += GRAVITY
64              self.rect[1] += self.__speed
65
66      def launch(self):
67          # Launch the eagle upwards
68          self.started = True
69          self.__speed = -SPEED
70
71  # Class for obstacles
72  class Obstacle1(SpriteBase):
73      def __init__(self, inverted, xpos, ysize):
74          image_path = 'green block.png'
75          # Initialize obstacle position and size
76          super().__init__(image_path, xpos, 0 if inverted else _SCREEN_height - ysize, OBSTACLE_1_width, OBSTACLE_1_HEIGHT)
77          if inverted:
78              # Flip the obstacle if it is inverted
79              self.image = pygame.transform.flip(self.image, False, True)
80              self.rect[1] = - (self.rect[3] - ysize)
81
82      def update(self):
83          # Move the obstacle to the left
84          self.rect[0] -= GAME_SPEED
85
86  # Class for the ground
87  class Ground(SpriteBase):
```

```
            self.image = pygame.transform.flip(self.image, False, True)
            self.rect[1] = - (self.rect[3] - ysize)

    def update(self):
        # Move the obstacle to the left
        self.rect[0] -= GAME_SPEED

# Class for the ground
class Ground(SpriteBase):

    def __init__(self, xpos):
        super().__init__('grass.jpg', xpos, _SCREEN_height - _GROUND_HEIGHT, _GROUND_width, _GROUND_HEIGHT)

    def update(self):
        # Move the ground to the left
        self.rect[0] -= GAME_SPEED

# Check if a sprite has moved off-screen
def is_off_screen(sprite):
    return sprite.rect[0] < -(sprite.rect[2])

# Generate random obstacles
def get_random_obstacle_1s(xpos):
    size = random.randint(100, 300)  # Random size for obstacle
    obstacle_1 = Obstacle1(False, xpos, size)
    obstacle_1_inverted = Obstacle1(True, xpos, _SCREEN_height - size - OBSTACLE_1_GAP)
    return obstacle_1, obstacle_1_inverted

# Setup the game screen
infoObject = pygame.display.Info()
screen=pygame.display.set_mode((infoObject.current_w/2, infoObject.current_h/2))
pygame.display.set_caption('Elegant Eagles')

# Load background image
BACKGROUND = pygame.image.load('Sky.jpg')
BACKGROUND = pygame.transform.scale(BACKGROUND, (_screen_width, _SCREEN_height))

# Initialize player group
eagle_group = pygame.sprite.Group()
eagle = Eagle()
eagle_group.add(eagle)

# Initialize ground group
ground_group = pygame.sprite.Group()
```

```
# Initialize ground group
ground_group = pygame.sprite.Group()
for i in range(2):
    ground = Ground(_GROUND_width * i)
    ground_group.add(ground)

# Initialize obstacle group
obstacle_set = pygame.sprite.Group()
for i in range(2):
    obstacle_1s = get_random_obstacle_1s(_screen_width * i + 800)
    obstacle_set.add(obstacle_1s[0])
    obstacle_set.add(obstacle_1s[1])

# Setup clock for frame rate
clock = pygame.time.Clock()

# Game state variables
play = False
font = pygame.font.SysFont('freesansbold', 30)

# Save Game
current_score = 0
save_code = None

# Save file path
SAVE_FILE = 'save_game.json'

```

```
def reset_game():
    global play, current_score, obstacle_set, eagle
    play = False  # Return to the main menu
    current_score = 0  # Reset score
    eagle.reset()  # Reset eagle position and state

    # Clear and regenerate obstacles
    obstacle_set.empty()
    for i in range(2):
        obstacle_1s = get_random_obstacle_1s(_screen_width * i + 800)
        obstacle_set.add(obstacle_1s[0])
        obstacle_set.add(obstacle_1s[1])

SAVE_FILE = "save_data.json"
```

```
def save_game(code, eagle, obstacles, enemies, score):
    try:
        #attempt to load existing save data from the save file
        with open(SAVE_FILE, 'r') as file:
            save_data = json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        #if the file doesn't exist or is corrupted, initialize with an empty dictionary
        save_data = {}

    #validate inputs before saving
    if not isinstance(code, str) or not code:
        print("Invalid save code! Must be a non-empty string.")
        return
    if not hasattr(eagle, 'rect') or not hasattr(eagle.rect, 'x') or not hasattr(eagle.rect, 'y'):
        print("Invalid eagle object! Missing coordinates.")
        return
    if not isinstance(score, (int, float)) or score < 0:
        print("Invalid score! Must be a non-negative number.")
        return
```

```
save_data[code] = {
    "eagle": {"x": eagle.rect.x, "y": eagle.rect.y},

    #store all obstacles using a for loop
    "obstacles": [
        {
            "x": obs.rect.x,
            "y": obs.rect.y,
            "width": obs.rect.width,
            "height": obs.rect.height
        }
        for obs in obstacles
    ],

    #store all enemies using a for loop
    "enemies": [
        {
            "x": enemy.rect.x,
            "y": enemy.rect.y,
            "speed": enemy._speed,
            "type": enemy._enemy_type.lower() if enemy._enemy_type.lower() in valid_types else 'astar'  #validate enemy type
        }
        for enemy in enemies
    ],

    "score": score  #store the current score
}
```

```
    #write the updated save data back to the file
    with open(SAVE_FILE, 'w') as file:
        json.dump(save_data, file)

    #print a confirmation message indicating the game has been saved
    print(f"Game Saved! Code: {code}")
```

```
    # Load existing save data
    try:
        with open(SAVE_FILE, 'r') as file:
            save_data = json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        save_data = []

    # Generate a unique random code
    while True:
        save_code = str(random.randint(100000, 999999))
        if not binary_search(save_data, save_code):
            break

    # Append the new save entry
    save_data.append({"code": save_code, "score": score})

    # Sort the save data
    save_data = insertion_sort(save_data)

    # Write back to the JSON file
    with open(SAVE_FILE, 'w') as file:
        json.dump(save_data, file, indent=4)

    print(f"Game Saved! Code: {save_code}")
# Load the game state from a JSON file
```

```
    print(f"Game Saved! Code: {save_code}")
# Load the game state from a JSON file


def is_valid_code(code):
    """
    Validate that the code is exactly 6 digits long and contains only digits 0-9.
    """
    valid_digits = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}  # Set of valid digits

    # Check if the code has exactly 6 characters
    if len(code) != 6:
        return False

    # Check if every character in the code is a valid digit
    for char in code:
        if char not in valid_digits:
            return False

    return True


```

```
                return True

        #Code not found in save data
        print("Invalid Code! No matching save found.")
        return False

    except FileNotFoundError:
        print("No saved game found! The save file does not exist.")
        return False
    except ValueError as e:
        print(f"Error: {e}")
        return False
    except (json.JSONDecodeError, KeyError) as e:
        print(f"Corrupted save data: {e}")
        return False

# Generate and handle interactive buttons
def generate_OSE(screen, x_coordinate, y_coordinate, x_size, y_size, default_colour, active_colour, text, action=None):
    cursor = pygame.mouse.get_pos()
    press = pygame.mouse.get_pressed()
    if x_coordinate < cursor[0] < x_coordinate + x_size and y_coordinate < cursor[1] < y_coordinate + y_size:
        pygame.draw.rect(screen, active_colour, (x_coordinate, y_coordinate, x_size, y_size))
        if press[0] == 1 and action:
            action()
    else:
        pygame.draw.rect(screen, default_colour, (x_coordinate, y_coordinate, x_size, y_size))
    text_surface = font.render(text, True, BLACK)
    centre=(x_coordinate + x_size) // 2, (y_coordinate + y_size // 2)
    text_rect = text_surface.get_rect(center=centre)
    screen.blit(text_surface, text_rect)

# Main game loop with obstacles handling
```

```
def load_game():
    #load the game score and state from a file
    try:
        #open the save file in read mode
        with open('save_game.json', 'r') as save_file:
            #load the json data from the file
            save_data = json.load(save_file)

            #check if the loaded data is a dictionary
            if not isinstance(save_data, dict):
                raise ValueError("invalid save format.")

            #load the score
            if "score" not in save_data or not isinstance(save_data["score"], (int, float)):
                raise ValueError("missing or invalid score in save data.")
            score = save_data["score"]

            #load the user eagle
            if "eagle" not in save_data:
                raise ValueError("missing eagle data in save data.")
            eagle_data = save_data["eagle"]
            if "x" not in eagle_data or "y" not in eagle_data:
                raise ValueError("missing eagle coordinates in save data.")
            user_eagle = Eagle(eagle_data["x"], eagle_data["y"])  #instantiate the user eagle

            #load obstacles
            obstacles = []
            if "obstacles" in save_data:
                for obstacle_data in save_data["obstacles"]:
                    if "x" not in obstacle_data or "y" not in obstacle_data or "width" not in obstacle_data or "height" not in obstacle_data:
                        raise ValueError("invalid obstacle data in save data.")
                    obstacle = Obstacle(obstacle_data["x"], obstacle_data["y"], obstacle_data["width"], obstacle_data["height"])
                    obstacles.append(obstacle)

            #load enemies
            enemies = []
            if "enemies" in save_data:
                for enemy_data in save_data["enemies"]:
                    if "type" not in enemy_data or "x" not in enemy_data or "y" not in enemy_data or "speed" not in enemy_data:
                        raise ValueError("invalid enemy data in save data.")
                    if enemy_data["type"].lower() == "los":
                        enemy = LOS_Enemy(enemy_data["x"], enemy_data["y"], enemy_data["speed"])
                    elif enemy_data["type"].lower() == "ml":
                        enemy = ML_Enemy(enemy_data["x"], enemy_data["y"], enemy_data["speed"])
                    elif enemy_data["type"].lower() == "astar":
                        enemy = AStar_Enemy(enemy_data["x"], enemy_data["y"], enemy_data["speed"])
                    else:
                        raise ValueError(f"unknown enemy type: {enemy_data['type']}")
                    enemies.append(enemy)

            #return the loaded game state
            return user_eagle, obstacles, enemies, score
```

```
#handle exceptions that may occur during file loading or data processing
except (FileNotFoundError, ValueError, KeyError) as e:
    #print an error message if an exception occurs
    print(f"error loading game: {e}")
    #return default values (empty objects and score 0) in case of an error
    return Eagle(0, 0), [], [], 0
```

```
282
283 # Main game loop with obstacles handling
284 def obstacles():
285     global play, current_score
286
287     input_code = ""
288     code_active = False
289
290     while not play:
291         screen.blit(BACKGROUND, (0, 0))
292         generate_OSE(screen, _screen_width // 2 - 75, _SCREEN_height // 2 - 25, 150, 50, (200, 200, 200), (150, 150, 150), "Start", action=lambda: start_game())
293
294         text_surface = font.render(f"Code: {input_code}", True, BLACK)
295         screen.blit(text_surface, (_screen_width // 2 - 75, _SCREEN_height // 2 + 50))
296
297         pygame.display.update()
298         for event in pygame.event.get():
299             if event.type == QUIT:
300                 pygame.quit()
301                 exit()
302             if event.type == KEYDOWN:
303                 if event.key == pygame.K_RETURN:
304                     if load_game(input_code):
305                         start_game()
306                 elif event.key == pygame.K_BACKSPACE:
307                     input_code = input_code[:-1]
308                 elif event.key == pygame.K_k:
309                     save_game(current_score)
310                     return
311                 elif event.key <= 127:
312                     input_code += chr(event.key)
313
314     speed_increment = 0
315
```

```
313
314     speed_increment = 0
315
316     while play:
317         clock.tick(15)
318
319         for event in pygame.event.get():
320             if event.type == QUIT:
321                 pygame.quit()
322             if event.type == KEYDOWN:
323                 if event.key == K_SPACE or event.key == K_UP:
324                     eagle.launch()
325                 if event.key == pygame.K_k:
326                     save_game(current_score)
327                     play = False
328                     return
329
330         screen.blit(BACKGROUND, (0, 0))
331
332         if is_off_screen(ground_group.sprites()[0]):
333             ground_group.remove(ground_group.sprites()[0])
334             new_ground = Ground(_GROUND_width - 20)
335             ground_group.add(new_ground)
336
337         if is_off_screen(obstacle_set.sprites()[0]):
338             obstacle_set.remove(obstacle_set.sprites()[0])
339             obstacle_set.remove(obstacle_set.sprites()[0])
340             current_score += 1
341
342             obstacle_1s = get_random_obstacle_1s(_screen_width * 2)
343             obstacle_set.add(obstacle_1s[0])
344             obstacle_set.add(obstacle_1s[1])
345
346         speed_increment += 0.05
347         global GAME_SPEED
348         GAME_SPEED = 30 + int(speed_increment)
349
350         eagle_group.update()
351         ground_group.update()
352         obstacle_set.update()
353
354         eagle_group.draw(screen)
355         obstacle_set.draw(screen)
```

```
        eagle_group.update()
        ground_group.update()
        obstacle_set.update()

        eagle_group.draw(screen)
        obstacle_set.draw(screen)
        ground_group.draw(screen)

        # Permanent scoreboard
        score_display = font.render(f"Score: {current_score}", True, BLACK)
        screen.blit(score_display, (10, 10))

        pygame.display.update()

        if (pygame.sprite.groupcollide(eagle_group, ground_group, False, False, pygame.sprite.collide_mask) or
                pygame.sprite.groupcollide(eagle_group, obstacle_set, False, False, pygame.sprite.collide_mask)):
            time.sleep(1)  # Small delay before returning to the menu
            reset_game()  # Reset game without quitting
            return


def start_game():
    global play
    play = True

while True:
    obstacles()
```

```
    def start(self):
        font = pygame.font.Font(None, 36)  # Default font for button text
        button_base_color = (100, 200, 100)
        button_hover_color = (50, 150, 50)

        while self.running:
            time_elapsed = self.clock.tick(60) / 1000.0
            mouse_position = pygame.mouse.get_pos()  # Track mouse position

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.running = False

                if event.type == pygame.MOUSEBUTTONDOWN:
                    #Check if a button was clicked
                    for index, button_rect in enumerate(self.buttons, start=1):
                        if button_rect.collidepoint(event.pos):
                            self.handle_button_click(index)

            self.screen.fill((240, 240, 240))

            #Draws the buttonss
            for index, button_rect in enumerate(self.buttons, start=1):
                self.render_button(
                    button_rect,
                    self.button_labels[index],
                    font,
                    button_base_color,
                    button_hover_color,
                    self.screen,
                    mouse_position
                )

            pygame.display.flip()

# Define custom functions for each button
def button1_action():
    print("Running custom function for Button 1")

def button2_action():
    print("Running custom function for Button 2")
```

```
# Define custom functions for each button
def button1_action():
    print("Running custom function for Button 1")

def button2_action():
    print("Running custom function for Button 2")

def button3_action():
    print("Running custom function for Button 3")

def button4_action():
    print("Running custom function for Button 4")

if __name__ == "__main__":
    button_actions = {
        1: button1_action,
        2: button2_action,
        3: button3_action,
        4: button4_action,
    }

    app = ButtonInterface(button_actions)
    app.start()

import re

MAX_name_LENGTH = 20
MIN_SCORE = 0
MAX_SCORE = 197

def add_score(lb, name, score):
    lb.append(name, score)
```

```
111
112 def print_lb(lb):
113     # Sort the lb by score in descending order
114     sorted_lb = sorted(lb.items(), key=lambda x: x[1], reverse=True)
115
116     print("\nlb:")
117     rank = 1
118     prev_score = None
119
120     for i, (user, score) in enumerate(sorted_lb, start=1):
121         if score != prev_score:
122             rank = i  # Update rank only if the score changes
123         print(f"{rank}. {user} - {score}")
124         prev_score = score
125
126
127 def is_valid_name(name):
128     if not name: #non empty
129         return False
130     if len(name) > MAX_name_LENGTH:
131         return False
132     return bool(re.match(r'^[A-Za-z0-9_]+$', name))
133
134 def is_valid_score(score_str):
135     # Checks if the score is a digit and within the specified range.
136     if not score_str.isdigit(): #covers null case implicitly since '' is str()
137         return False, "score is a natural number."
138     score = int(score_str)
139     if score < MIN_SCORE:
140         return False, "invalid score!"
141     elif score > MAX_SCORE:
142         return False, f"score too big! The max is {MAX_SCORE}."
143     return True, score
144
```

```
144
145 def leaderboard_main():
146
147     lb = {}  #note choice of data structure
148     while True:
149         name = input("Enter name (or type 'exit' to stop): ").strip() #see references 1.2 for relevant post from stackexchange
150         if name.lower() == 'exit':
151             break
152
153         if not is_valid_name(name):
154             print(f"Invalid name. Use only letters, numbers, and underscores, and ensure it's at most {MAX_name_LENGTH} characters long.")
155             continue
156
157         #if repeat name
158         if name in lb:
159             while True:
160                 overwrite = input("name already exists. Do you want to update the score? (yes/no): ").strip().lower()
161                 #names consist of characters from the alphabet
162                 if not overwrite.isalpha():
163                     print("Invalid response. Please use letters only.")
164                     continue
165                 elif len(overwrite) > 3:
166                     print("Invalid response. Response too long.")
167                     continue
168                 #Accept only n, y, no, yes
169                 elif overwrite not in ['yes', 'y', 'no', 'n']:
170                     print("Invalid response. Acceptable responses: n, y, no, yes.")
171                     continue
172                 break
173             if overwrite in ['no', 'n']:
174                 continue
175
176         score_input = input("Enter score: ").strip()
177         valid, result = is_valid_score(score_input)
178         if not valid:
179             print(f"Invalid input: {result}")
180             continue
181
182         score = result
183         add_score(lb, name, score)
184     print_lb(lb)
185
```

```
43         self.rect[1] = ypos
44
45 # Base class for the eagle
46 class Eagle(SpriteBase):
47     def __init__(self, speed=SPEED):
48         super().__init__('user_image.png', SCREEN_WIDTH // 6, SCREEN_HEIGHT // 2, 30, 30)
49         self._speed = 0
50         self._gravity = GRAVITY
51         self.__score = 0  # Player score
52         self.started = False
53
54     def reset(self):
55         self.rect[1] = SCREEN_HEIGHT // 2
56         self._speed = 0
57         self.started = False
58
59     def update(self):
60         if self.started:
61             self._speed += self._gravity
62             self.rect[1] += self._speed
63
64     def launch(self):
65         self.started = True
66         self._speed = -SPEED
67
68 # Specialized Eagle classes
69 class RotatingEagle(Eagle):
70     def __init__(self):
71         super().__init__()
72         self.rotation_angle = 30
73
74     def update(self):
75         super().update()
76         self.image = pygame.transform.rotate(pygame.image.load('user_image.png').convert_alpha(), self.rotation_angle)
77
78 class HeavyEagle(Eagle):
79     def __init__(self):
80         super().__init__()
81         self._gravity = GRAVITY * 1.2
82         self.rotation_angle = 0
83
```

```
        self._speed = -SPEED

# Specialized Eagle classes
class RotatingEagle(Eagle):
    def __init__(self):
        super().__init__()
        self.rotation_angle = 30

    def update(self):
        super().update()
        self.image = pygame.transform.rotate(pygame.image.load('user_image.png').convert_alpha(), self.rotation_angle)

class HeavyEagle(Eagle):
    def __init__(self):
        super().__init__()
        self._gravity = GRAVITY * 1.2
        self.rotation_angle = 0

    def update(self):
        super().update()
        self._speed += self._gravity #this beiung negative is needed to invert motion
        self.image = pygame.transform.rotate(pygame.image.load('user_image.png').convert_alpha(), self.rotation_angle)

class TimeControlEagle(Eagle):
    def __init__(self):
        super().__init__()
        self.slow_time_active = False
        self.slow_time_start = 0
        self.rotation_angle = -30

    def slow_time(self):
        self.slow_time_active = True
        self.slow_time_start = pygame.time.get_ticks()

    def update(self):
        if self.slow_time_active:
            if pygame.time.get_ticks() - self.slow_time_start > 2000:
                self.slow_time_active = False
            else:
                return  # Skip gravity update to simulate slow time
        super().update()
        self.image = pygame.transform.rotate(pygame.image.load('user_image.png').convert_alpha(), self.rotation_angle)

```

```python
def save_game(code, eagle, obstacles, score):
    try:
        # Attempt to load existing save data
        with open(SAVE_FILE, 'r') as file:
            save_data = json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        # If file doesn't exist or is corrupted, start with an empty dictionary
        save_data = {}

    # Validate inputs before saving
    if not isinstance(code, str) or not code:
        print("Invalid save code! Must be a non-empty string.")
        return
    if not hasattr(eagle, 'x') or not hasattr(eagle, 'y'):
        print("Invalid eagle object! Missing coordinates.")
        return
    if not isinstance(score, (int, float)) or score < 0:
        print("Invalid score! Must be a non-negative number.")
        return

    # Store game data in dictionary
    save_data[code] = {
        "eagle": (eagle.x, eagle.y),  # Store eagle's position
        "obstacles": [(obs.x, obs.y, obs.width, obs.height) for obs in obstacles],  # Store all obstacles
        "score": score  # Store current score
    }

    # Write updated data back to file
    with open(SAVE_FILE, 'w') as file:
        json.dump(save_data, file)

    print(f"Game Saved! Code: {code}")
```

```python

# Class for obstacles
class Obstacle1(SpriteBase):
    def __init__(self, inverted, xpos, ysize):
        image_path = 'green block.png'
        ypos = 0 if inverted else SCREEN_HEIGHT - ysize
        super().__init__(image_path, xpos, ypos, OBSTACLE_WIDTH, OBSTACLE_HEIGHT)
        if inverted:
            self.image = pygame.transform.flip(self.image, False, True)
            self.rect[1] = -(self.rect[3] - ysize)

    def update(self):
        self.rect[0] -= GAME_SPEED


class PathfindingEnemy(SpriteBase):
    def __init__(self, xpos, ypos, speed):
        try:
            super().__init__('enemy.png', xpos, ypos, 40, 40)
            self.speed = speed
            self.target = None
            self.path = []
            self.pathfinding_interval = 15  # Update path every 15 frames
            self.frame_counter = 0
        except Exception:
            print(f"Error loading enemy image: {Exception}")

    def heuristic(self, enemy_pos, player_pos, obstacles):
        """
        Heuristic considering distance, obstacles, and momentum.
        """
        ex, ey = enemy_pos
        px, py = player_pos

        # Basic Euclidean distance
        distance = math.sqrt((ex - px) ** 2 + (ey - py) ** 2)
```

```python
class PathfindingEnemy(SpriteBase):
    def __init__(self, xpos, ypos, speed, difficulty):
        try:
            super().__init__('enemy.png', xpos, ypos, 40, 40)
            self._speed = speed  #speed of the enemy
            self._target = None  #target position for pathfinding
            self._pathfinding_interval = 15  #update path every 15 frames
            self._frame_counter = 0  #frame counter for pathfinding updates
            self._difficulty = difficulty  #difficulty level of the enemy
        except Exception:
            print(f"Error loading enemy image: {Exception}")

    #get and set methods for speed
    def get_speed(self):
        return self._speed

    def set_speed(self, speed):
        if isinstance(speed, (int, float)) and speed > 0:  #validate speed is a positive number
            self._speed = speed
        else:
            raise ValueError("speed must be a positive number.")

    #get and set methods for target
    def get_target(self):
        return self._target

    def set_target(self, target):
        if isinstance(target, tuple) and len(target) == 2:  #validate target is a tuple of length 2
            self._target = target
        else:
            raise ValueError("target must be a tuple of length 2 (x, y).")

    #get and set methods for pathfinding interval
    def get_pathfinding_interval(self):
        return self._pathfinding_interval

    def set_pathfinding_interval(self, interval):
        if isinstance(interval, int) and interval > 0:  #validate interval is a positive integer
            self._pathfinding_interval = interval
        else:
            raise ValueError("pathfinding interval must be a positive integer.")

    #get and set methods for frame counter
    def get_frame_counter(self):
        return self._frame_counter
```

```
def set_frame_counter(self, counter):
    if isinstance(counter, int) and counter >= 0:  #validate counter is a non-negative integer
        self._frame_counter = counter
    else:
        raise ValueError("frame counter must be a non-negative integer.")

#get and set methods for difficulty
def get_difficulty(self):
    return self._difficulty

def set_difficulty(self, difficulty):
    if isinstance(difficulty, int) and 1 <= difficulty <= 10:  #validate difficulty is between 1 and 10
        self._difficulty = difficulty
    else:
        raise ValueError("difficulty must be an integer between 1 and 10.")

def heuristic(self, enemy_pos, player_pos, obstacles):
    """
    heuristic considering distance, obstacles, and momentum
    """
    ex, ey = enemy_pos  #extract enemy's position (e_x, e_y)
    px, py = player_pos  #extract player's position (p_x, p_y)

    #manhattan distance: |e_x - p_x| + |e_y - p_y|
    distance = abs(ex - px) + abs(ey - py)

    #obstacle penalty: add a penalty for obstacles in a straight line
    obstacle_penalty = 0  #initialize penalty to 0
    for obstacle in obstacles:  #iterate through all obstacles
        #calculate distance to the obstacle
        ox, oy = obstacle.rect.x, obstacle.rect.y  #extract obstacle's position (o_x, o_y)
        obstacle_distance = abs(ex - ox) + abs(ey - oy)  #manhattan distance to obstacle

        #add a penalty inversely proportional to the distance to the obstacle
        if (ex-ox)^2+(ey-oy)^2 < 63^2:  #penalize if the obstacle is within 63 units
            obstacle_penalty += 450 / (obstacle_distance + 1)  #add penalty

    return distance + obstacle_penalty  #return total heuristic value
```

```python
def a_star_pathfinding(self, target_pos, obstacles):
    """
    a* pathfinding algorithm to move toward the target position
    """
    start = (self.rect.x, self.rect.y)
    goal = target_pos
    open_set = []
    heapq.heappush(open_set, (0, start))  #priority queue storing nodes with the lowest cost first
    came_from = {}  #dictionary to store the best path
    g_score = {start: 0}  #cost from start node to each node
    f_score = {start: self.heuristic(start, goal, obstacles)}  #estimated total cost from start to goal

    while open_set:
        _, current = heapq.heappop(open_set)  #retrieve node with lowest estimated cost

        if current == goal:
            #reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.reverse()
            return path

        #neighbor nodes (8-directional movement)
        neighbors = [
            (current[0] + dx, current[1] + dy)
            for dx, dy in [(-self._speed, 0), (self._speed, 0), (0, -self._speed), (0, self._speed),
                           (-self._speed, -self._speed), (self._speed, -self._speed),
                           (-self._speed, self._speed), (self._speed, self._speed)]
        ]

        for neighbor in neighbors:
            tentative_g_score = g_score[current] + self.heuristic(current, neighbor, obstacles)

            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + self.heuristic(neighbor, goal, obstacles)
                heapq.heappush(open_set, (f_score[neighbor], neighbor))  #store the neighbor with updated cost

    return []  #no path found
```

```python
    def update(self, player, obstacles):
        try:
            #simple logic to move left toward the player for now
            self.rect.x -= self._speed
            if self.rect.x < 0:  #remove enemies off-screen
                self.kill()
        except Exception as e:
            print(f"Error in PathfindingEnemy update: {e}")



last_enemy_spawn_score = -10  # Keeps track of the score when the last enemy was spawned
```

```python
def generate_enemy(score, enemies_group):
    global last_enemy_spawn_score
    # Only add a new enemy when the score is a multiple of 10 and not on every frame
    if score % 10 == 0 and score > last_enemy_spawn_score:
        last_enemy_spawn_score = score  # Update the last score when an enemy was spawned
        # Spawn an enemy at a random vertical position
        enemy = PathfindingEnemy(SCREEN_WIDTH, random.randint(0, SCREEN_HEIGHT - 50), speed=5)
        enemies_group.add(enemy)

enemies_group = pygame.sprite.Group()


def spawn_enemy():
    enemy = Enemy(SCREEN_WIDTH, random.randint(50, SCREEN_HEIGHT - 50))
    enemy.set_target(eagle)
    enemy_group.add(enemy)

# Class for the ground
```

```python
            # Neighbor nodes (8-directional movement)
            neighbors = [
                (current[0] + dx, current[1] + dy)
                for dx, dy in [(-self.speed, 0), (self.speed, 0), (0, -self.speed), (0, self.speed),
                               (-self.speed, -self.speed), (self.speed, -self.speed),
                               (-self.speed, self.speed), (self.speed, self.speed)]
            ]

            for neighbor in neighbors:
                tentative_g_score = g_score[current] + self.heuristic(current, neighbor, obstacles)

                if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = tentative_g_score + self.heuristic(neighbor, goal, obstacles)
                    heapq.heappush(open_set, (f_score[neighbor], neighbor))

        return []  # No path found

    def update(self, player, obstacles):
        try:
            # Simple logic to move left toward the player for now
            self.rect.x -= self.speed
            if self.rect.x < 0:  # Remove enemies off-screen
                self.kill()
        except Exception as e:
            print(f"Error in PathfindingEnemy update: {e}")

last_enemy_spawn_score = -10  # Keeps track of the score when the last enemy was spawned

def generate_enemy(score, enemies_group):
    global last_enemy_spawn_score
    # Only add a new enemy when the score is a multiple of 10 and not on every frame
    if score % 10 == 0 and score > last_enemy_spawn_score:
        last_enemy_spawn_score = score  # Update the last score when an enemy was spawned
        # Spawn an enemy at a random vertical position
        enemy = PathfindingEnemy(SCREEN_WIDTH, random.randint(0, SCREEN_HEIGHT - 50), speed=5)
        enemies_group.add(enemy)

enemies_group = pygame.sprite.Group()
```

```
enemies_group = pygame.sprite.Group()


def spawn_enemy():
    enemy = Enemy(SCREEN_WIDTH, random.randint(50, SCREEN_HEIGHT - 50))
    enemy.set_target(eagle)
    enemy_group.add(enemy)

# Class for the ground
class Ground(SpriteBase):
    def __init__(self, xpos):
        super().__init__('grass.jpg', xpos, SCREEN_HEIGHT - GROUND_HEIGHT, GROUND_WIDTH, GROUND_HEIGHT)

    def update(self):
        self.rect[0] -= GAME_SPEED

# Utility functions
def is_off_screen(sprite):
    return sprite.rect[0] < -sprite.rect[2]

def get_random_obstacles(xpos):
    size = random.randint(100, 300)
    obs = Obstacle1(False, xpos, size)
    obs_inverted = Obstacle1(True, xpos, SCREEN_HEIGHT - size - OBSTACLE_GAP)
    return obs, obs_inverted

def reset_game():
    global play, current_score, obstacle_group, eagle
    play = False
    current_score = 0
    eagle.reset()
    obstacle_group.empty()
    for i in range(2):
        obs = get_random_obstacles(SCREEN_WIDTH * i + 800)
        obstacle_group.add(obs[0])
        obstacle_group.add(obs[1])

def save_game(score):
    save_data = {
        "score": score,
        "timestamp": time.time()
    }
    try:
        with open('save_game.json', 'w') as save_file:
            json.dump(save_data, save_file)
    except Exception as e:
        print(f"Error saving game: {e}")

def load_game():
    try:
        with open('save_game.json', 'r') as save_file:
            save_data = json.load(save_file)
            if not isinstance(save_data, dict):
                raise ValueError("Invalid save format.")
            if "score" not in save_data or not isinstance(save_data["score"], int):
                raise ValueError("Missing or invalid score in save data.")
            return save_data["score"]
    except (FileNotFoundError, ValueError, KeyError) as e:
        print(f"Error loading game: {e}")
        return 0

# Initialize game objects
pygame.display.set_caption('Eagle Adventure')
screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
BACKGROUND = pygame.image.load('Sky.jpg')
BACKGROUND = pygame.transform.scale(BACKGROUND, (SCREEN_WIDTH, SCREEN_HEIGHT))

font = pygame.font.SysFont('freesansbold', 30)
clock = pygame.time.Clock()

# Player eagle group
eagle_group = pygame.sprite.Group()
eagle = Eagle()
eagle_group.add(eagle)
```

```
eagle_group = pygame.sprite.Group()
eagle = Eagle()
eagle_group.add(eagle)

ground_group = pygame.sprite.Group()
for i in range(2):
    ground = Ground(GROUND_WIDTH * i)
    ground_group.add(ground)

obstacle_group = pygame.sprite.Group()


current_score = 0
play = False

```

```python
import pygame
import re

#Button System to handle UI interactions
class BtnSystem:
    def __init__(self, btn_funcs):
        pygame.init()
        self.win = pygame.display.set_mode((800, 600)) #Setup game window
        pygame.display.set_caption('Btns!') #Window title

        #Creating button rectangles (position & size)
        self.btns = [
            pygame.Rect(100, 150, 200, 80),
            pygame.Rect(100, 250, 200, 80),
            pygame.Rect(100, 350, 200, 80),
            pygame.Rect(100, 450, 200, 80)
        ]

        #Button labels, assigned based on order
        self.btn_text = {1: "Quit", 2: "Info", 3: "Settings", 4: "Start"}
        self.btn_funcs = btn_funcs #Assigning functions to buttons

        self.fps = pygame.time.Clock() #Used to maintain framerate
        self.active = True #Game loop flag

    #Handles when a button is clicked
    def btn_clicked(self, btn_id):
        print(f"Btn {btn_id} clicked!") #Debug print
        self.btn_text[btn_id] = "Clicked!"
        if btn_id in self.btn_funcs:
            self.btn_funcs[btn_id]() #Execute assigned function

    #Handles button appearance & hover effect
    def draw_btn(self, rect, label, font, color, color_hover, win, mpos):
        if rect.collidepoint(mpos):
            pygame.draw.rect(win, color_hover, rect) #Change color if hovered
        else:
            pygame.draw.rect(win, color, rect) #Normal button color

        #Render text inside button
        txt = font.render(label, True, (0, 0, 0))
        txt_rect = txt.get_rect(center=rect.center)
        win.blit(txt, txt_rect) #Draw button label
        win.blit(txt, txt_rect) #Draw button label

    #Main loop for event handling and rendering
    def run(self):
        fnt = pygame.font.Font(None, 36) #Button font
        clr1 = (100, 200, 100) #Normal button color
        clr2 = (50, 150, 50) #Hover color

        while self.active:
            dt = self.fps.tick(60) / 1000.0 #Maintain framerate
            mpos = pygame.mouse.get_pos() #Track mouse

            for evt in pygame.event.get():
                if evt.type == pygame.QUIT:
                    self.active = False #Exit condition

                if evt.type == pygame.MOUSEBUTTONDOWN:
                    for i, btn in enumerate(self.btns, start=1):
                        if btn.collidepoint(evt.pos):
                            self.btn_clicked(i) #Check which button was clicked

            self.win.fill((240, 240, 240)) #Background color
            for i, btn in enumerate(self.btns, start=1):
                self.draw_btn(btn, self.btn_text[i], fnt, clr1, clr2, self.win, mpos)

            pygame.display.flip() #Update screen

#Functions for buttons (customizable)
def f1(): print("Function for Btn1")
def f2(): print("Function for Btn2")
def f3(): print("Function for Btn3")
def f4(): print("Function for Btn4")

#Main menu system
if __name__ == "__main__":
    btn_funcs = {1: f1, 2: f2, 3: f3, 4: f4} #Assign functions
    app = BtnSystem(btn_funcs)
    app.run()

#================= LEADERBOARD SYSTEM =================#
#Max username length & score range
MAX_NAME = 20
MIN_PTS = 0
```

```python
#Max username length & score range
MAX_NAME = 20
MIN_PTS = 0
MAX_PTS = 197

#Adds user & score to leaderboard
def add_score(lb, n, s):
    lb[n] = s

#Prints sorted leaderboard
def show_lb(lb):
    sorted_lb = sorted(lb.items(), key=lambda x: x[1], reverse=True)
    print("\nLeaderboard:")
    r = 1 #Keeps track of ranking
    prev_s = None #Handles same score rankings

    for i, (user, s) in enumerate(sorted_lb, start=1):
        if s != prev_s:
            r = i #Only update rank if score changes
        print(f"{r}. {user} - {s}")
        prev_s = s


def valid_name(n): #Validates username input
    if not n: return False #Name can't be empty
    if len(n) > MAX_NAME: return False #Max length check
    return bool(re.match(r'^[A-Za-z0-9_]+$', n)) #Only letters, numbers, underscore allowed

#Validates score input
def valid_score(s):
    if not s.isdigit(): return False, "must be a whole number"
    s = int(s) #Convert to integer
    if s < MIN_PTS: return False, "score too low!"
    if s > MAX_PTS: return False, f"score too high! Max={MAX_PTS}"
    return True, s

#Leaderboard input loop
def main():
    lb = {} #Dictionary to store scores
    while True:
        n = input("Enter name (type 'exit' to stop): ").strip()
        if n.lower() == 'exit': break #User quits input

```

```python

def is_valid_name(name):
    if not name: #non empty
        return False
    if len(name) > MAX_name_LENGTH:
        return False
    return bool(re.match(r'^[A-Za-z0-9_]+$', name))

def is_valid_score(score_str):
    # Checks if the score is a digit and within the specified range.
    if not score_str.isdigit(): #covers null case implicitly since '' is str()
        return False, "score is a natural number."
    score = int(score_str)
    if score < MIN_SCORE:
        return False, "invalid score!"
    elif score > MAX_SCORE:
        return False, f"score too big! The max is {MAX_SCORE}."
    return True, score

```

```python

def leaderboard_main():

    lb = {}  #note choice of data structure
    while True:
        name = input("Enter name (or type 'exit' to stop): ").strip() #see references 1.2 for relevant post from stackexchange
        if name.lower() == 'exit':
            break

        if not is_valid_name(name):
            print(f"Invalid name. Use only letters, numbers, and underscores, and ensure it's at most {MAX_name_LENGTH} characters long.")
            continue

        #if repeat name
        if name in lb:
            while True:
                overwrite = input("name already exists. Do you want to update the score? (yes/no): ").strip().lower()
                #names consist of characters from the alphabet
                if not overwrite.isalpha():
                    print("Invalid response. Please use letters only.")
                    continue
                elif len(overwrite) > 3:
                    print("Invalid response. Response too long.")
                    continue
                #Accept only n, y, no, yes
                elif overwrite not in ['yes', 'y', 'no', 'n']:
                    print("Invalid response. Acceptable responses: n, y, no, yes.")
                    continue
                break
            if overwrite in ['no', 'n']:
                continue

        score_input = input("Enter score: ").strip()
        valid, result = is_valid_score(score_input)
        if not valid:
            print(f"Invalid input: {result}")
            continue

        score = result
        add_score(lb, name, score)
    print_lb(lb)
```

```
# Main game loop
def game_loop():
    global play, current_score, eagle
    while True:
        screen.blit(BACKGROUND, (0, 0))

        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                exit()
            if event.type == KEYDOWN:
                if event.key == K_SPACE or event.key == K_UP:
                    eagle.launch()
                if event.key == K_k:
                    save_game(current_score)
                if event.key == K_1:
                    eagle = RotatingEagle()
                    eagle_group.empty()
                    eagle_group.add(eagle)
                if event.key == K_2:
                    eagle = HeavyEagle()
                    eagle_group.empty()
                    eagle_group.add(eagle)
                if event.key == K_3:
                    eagle = TimeControlEagle()
                    eagle_group.empty()
                    eagle_group.add(eagle)
                if event.key == K_s and isinstance(eagle, TimeControlEagle):
                    eagle.slow_time()

        if is_off_screen(ground_group.sprites()[0]):
            ground_group.remove(ground_group.sprites()[0])
            ground_group.add(Ground(GROUND_WIDTH - 20))

        if is_off_screen(obstacle_group.sprites()[0]):
            obstacle_group.remove(obstacle_group.sprites()[0])
            obstacle_group.remove(obstacle_group.sprites()[0])
            current_score += 1
            obs = get_random_obstacles(SCREEN_WIDTH * 2)
            obstacle_group.add(obs[0])
            obstacle_group.add(obs[1])

        # Generate enemies based on score
        generate_enemy(current_score, enemies_group)

        # Update groups
        eagle_group.update()
        ground_group.update()
        obstacle_group.update()
        enemies_group.update(eagle, obstacle_group)

        # Draw groups
        eagle_group.draw(screen)
        obstacle_group.draw(screen)
        ground_group.draw(screen)
        enemies_group.draw(screen)

        score_display = font.render(f"Score: {current_score}", True, BLACK)
        screen.blit(score_display, (10, 10))

        pygame.display.update()
        clock.tick(30)

        if (pygame.sprite.groupcollide(eagle_group, ground_group, False, False, pygame.sprite.collide_mask) or
                pygame.sprite.groupcollide(eagle_group, obstacle_group, False, False, pygame.sprite.collide_mask)):
            time.sleep(1)
            reset_game()

reset_game()
game_loop()
```

References

Below, I have listed the titles of the videos containing my test evidence along with the numbers of the Success Criteria which it also provides evidence for. I have done this in order to cross reference my test evidence with my success criteria and justify the success I have stated for each criterion.

- **General Tutorials**:
- Flappy Bird Game in Python (YouTube)
- A* Pathfinding Algorithm Explained (YouTube)
- A* Algorithm Implementation Tutorial (YouTube)
- Stanford CS on the Heuristic
- Saving in Pygame

- **Documentation:**
- Pygame GUI Quick Start Guide
- OCR UML Documentation (PDF)
- Heap in Python (Board Infinity)
- OpenAI
- **Helpful Stack Overflow Discussions**:
- ImportError: Cannot Import Name from Collections (Python 3.10)

- **Specific Tutorials**
- Flappy Eagle Game in Python (AskPython)
- Flappy Bird Clone in Python (ThePythonCode)
- Pygame Flappy Bird Project
- Common Password Research for Security