ANGRY BIRDS

Computer Science
Non-Exam Assessment

Oreoluwa Ajibade (LINC_WLLk1-13)

Table of Contents

ANALYSIS

Outline

My project will be a recreation of my childhood favourite (and very popular) game called "Angry Birds". Angry Birds is a classic 2000s game involving flinging projectiles (Birds) at various structures containing targets (pigs) to hurt them sufficiently. The player is restricted from advancing to the next level unless all the targets have been eliminated. Though eliminating the game is the primary target, the player is also further meant to cause as much destruction as possible to gain as many points as possible. The elimination of targets and the amount of destruction caused in the process will add up to form the final score.

In the game, the entities included will be: projectiles (birds) to launch at the target; the target (pig) to be the aim of the projectile; projectile launcher (slingshot) to launch the birds, breakable and unbreakable obstacles (blocks) to make the game difficult. Whereas, objects visible to the player only will be the score indicator to show the current score and the stored highscore; bird (projectile) count to show how many birds the user has left, and a menu button to pause and play the game.

Stakeholders

The original Angry Birds game is a mobile game, however, I have decided to make a PC adaptation of the game (If time permits, I can recreate the game in mobile for more convenience). Therefore, this means this game is suited for the casual PC gamer, a person that takes the time out of his day to play PC games. Furthermore, due to the casual, puzzle solving nature of the game, the target demographic will be between mostly 10 - 22 years of age because this the optimal PC gamer age according to Statista.

This is why I have chosen Feyisayo Zollner as a representation of my target audience. He is a seventeen-year-old student in my computing class that is interested in video games. We have also both enjoyed playing Angry Birds together in the past and he will be delighted to see a PC version. Since we are in the same computing class, I will also have regular contact with him.

How this Problem can be Solved by Computational Methods

The game can ideally be solved by computers due to how easy it is to abstract unnecessary variables and implement computational algorithms to simulate a certain game scenario.

Thinking Abstractly

Abstraction is the process of removing unnecessary details for simplicity purposes. My game isn't a based fully in a realistic scenario, therefore, abstraction is required. The following are use cases of abstractions in my Angry Birds Game:

- Remove unnecessary variables like Air resistance.
- Add a bright, user friendly, cartoony environment to make the game visually appealling
- Restrict the game to 2 dimensions
- Add a bar at the top and bottom to show information like the number of birds remaining or number of points scored

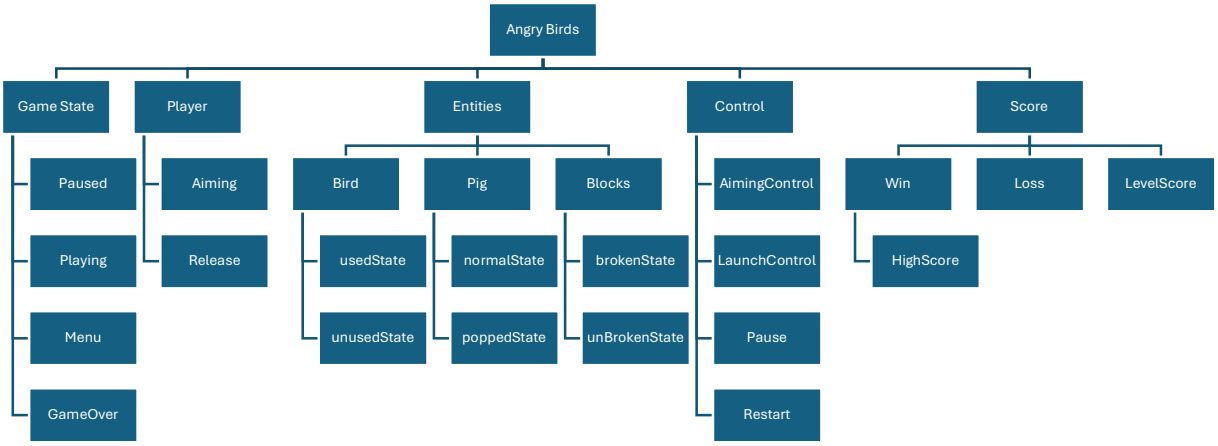
Thinking Ahead

Thinking ahead involves identifying the preconditions of a system like the inputs, outputs and reusable components. This is a way of looking ahead and knowing what you want. It allows you to plan a bit giving the solution of your problem a bit of structure.

- A points system will be utilized to give the user feedback on their performance in the game.
- Each game entity will be written as an object with different methods so it can easily be re-used in different components of the code
- Due to the nature of controlling a projectile launcher (slingshot), the user will be advised to make use of a mouse to play the game effectively
- I plan to use the Unity engine to simulate the physics because of its open source nature
- Due to my use of the Unity Engine, I plane to use the Unity IDE to develop my game

Thinking Procedurally

A game as complex as Angry Birds will be required to be broken into many parts to simplify the process of making it. I have decided to make each entity in the game an object with a bunch of methods and attributes.



I have broken down Angry Birds into five main areas to ensure smooth gameplay and user interaction. These areas focus on the essential elements, from how the game progresses to how players control the birds and score points:

1. **GAMESTATE:** This area defines the various phases of the game, such as:

- **Menu:** Where the game begins, allowing the player to start a new game or review instructions.
- **Playing:** The primary state where the action occurs as the player aims and releases birds.
- **Paused:** Halts the game, providing a break when necessary.
- **Over**: Indicates whether the player has won or lost after all pigs are defeated or attempts exhausted.

2. **PLAYER:** This section handles the actions controlled by the player, such as:

- **Aiming:** The player adjusts the bird's trajectory using the slingshot.
- **Release:** The player lets go of the slingshot, launching the bird toward the pigs.

3. **ENTITIES:** These are the in-game objects like birds, pigs, and blocks, which react to player actions:

- **Birds:** Can be in an unusedState (ready to be launched) or usedState (launched and interacting with blocks or pigs).

- **Pigs:** Can exist in a normalState (undamaged) or poppedState (eliminated after being hit).
- **Blocks:** Remain in an unBrokenState (intact) until hit, transitioning into a brokenState (destroyed).

4. **CONTROL:** This section manages how the player interacts with the game, including:

- **AimingControl:** Adjusts the slingshot's direction and power.
- **LaunchControl:** Releases the bird from the slingshot.
- **Pause:** Freezes gameplay.
- **Restart:** Resets the current level, allowing a fresh attempt.

5. **SCORE:** This is the system that tracks and rewards player performance:

- **Win:** Achieved by defeating all pigs and achieving the HighScore.
- **Loss:** Occurs if the player fails to clear all pigs.
- **LevelScore:** Determined by the number of blocks destroyed, pigs popped, and remaining birds.

This structure ensures the game flows smoothly, with each area working together to create a fun and challenging experience for players.

Thinking Logically

Thinking logically involves identifying points in which decisions will be made throughout the game. As the game is running, there will be a constant iteration searching for the following conditions:

- The "menu" state will determine if the game is paused or playing
- If a pig is hit, a decision if made to award the player more points than if a regular block
- The number of birds and number of pigs left will be continuously monitored to determine if the game should transition to a "game over" or "win" state. For instance, if there are no birds remaining but targets are still present, it results in a game over.

Thinking Concurrently

Being a game, a lot of processes will be required to run at the same time to give real time updates to the user. Processes such as:

- Audio effects like bird launches, pig squeals, and block destruction occur in sync with the game's visuals, ensuring that sound is triggered concurrently with corresponding events.

- The game constantly calculates the player's score based on the number of pigs popped, blocks destroyed, and remaining birds. This score calculation is happening in real-time, even while the player is still launching birds.
- As the game progresses, the game continuously updates the states of objects (e.g., birds, blocks, and pigs). Birds switch between unused and used states, pigs between normal and popped states, and blocks between unbroken and broken states, all happening independently and concurrently.

Conclusion

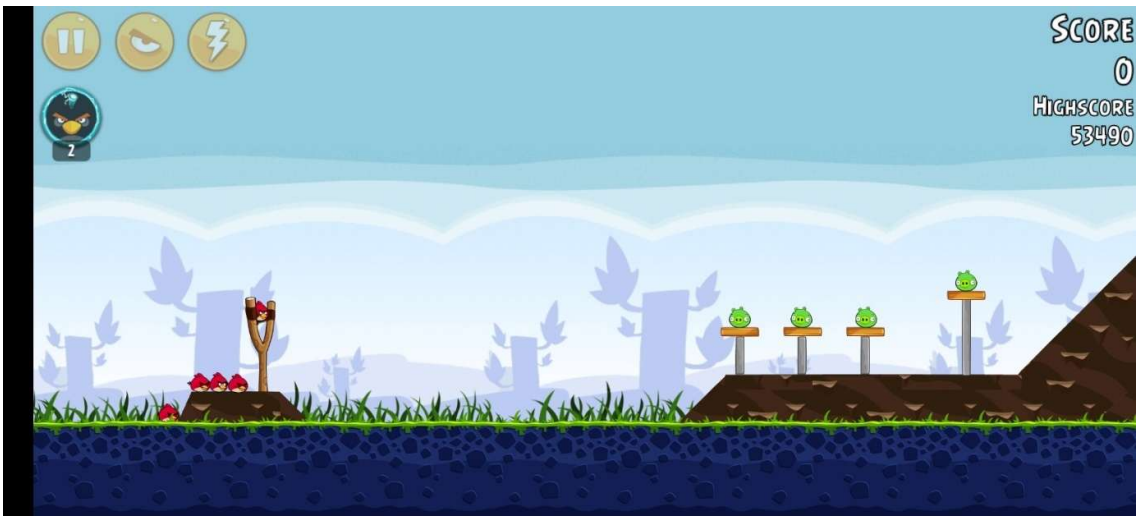
The earlier examples have shown that the characteristics of my problem can be addressed using computational methods, making it ideal for a computer program. These methods help structure the problem, making it easier to solve and thus well-suited for a computer-based solution.

Research

Game Comparisons

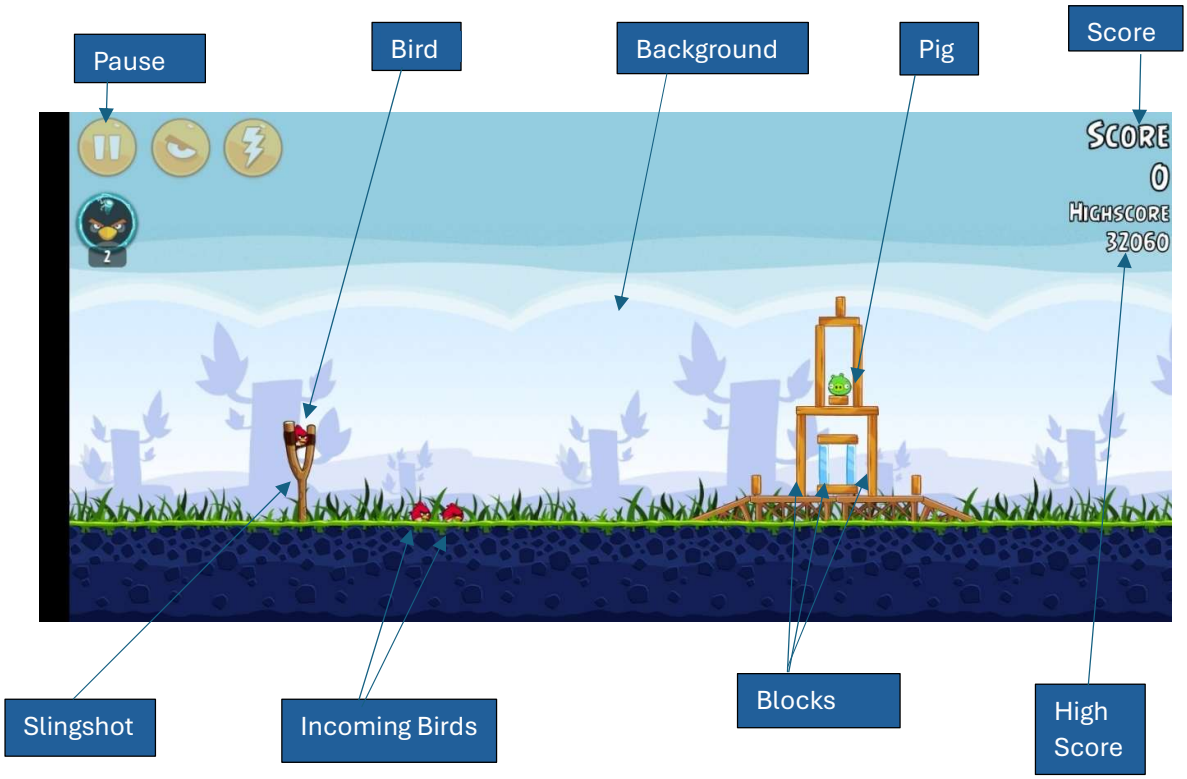
To properly research on how the game is meant to look like, I decided to examine a fairly similar game to what I want to create, Angry Birds 2 and Angry Birds Classic.

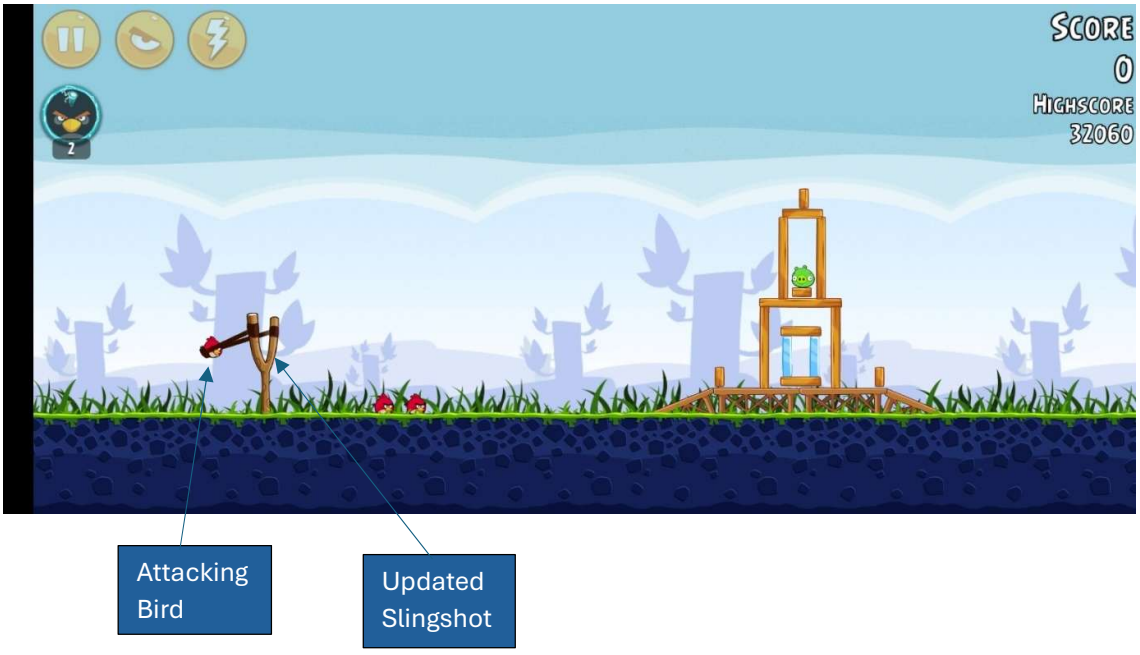
Angry Birds Classic is a popular puzzle game developed by Rovio Entertainment. In this game, players use a slingshot to launch various birds at green pigs stationed in or around different structures. The goal is to destroy all the pigs on the playing field. Each bird has unique abilities that can be used strategically to dismantle the pigs' defenses. This is how I aspire the game to look like. This is a screen shot of a classic level in the game:



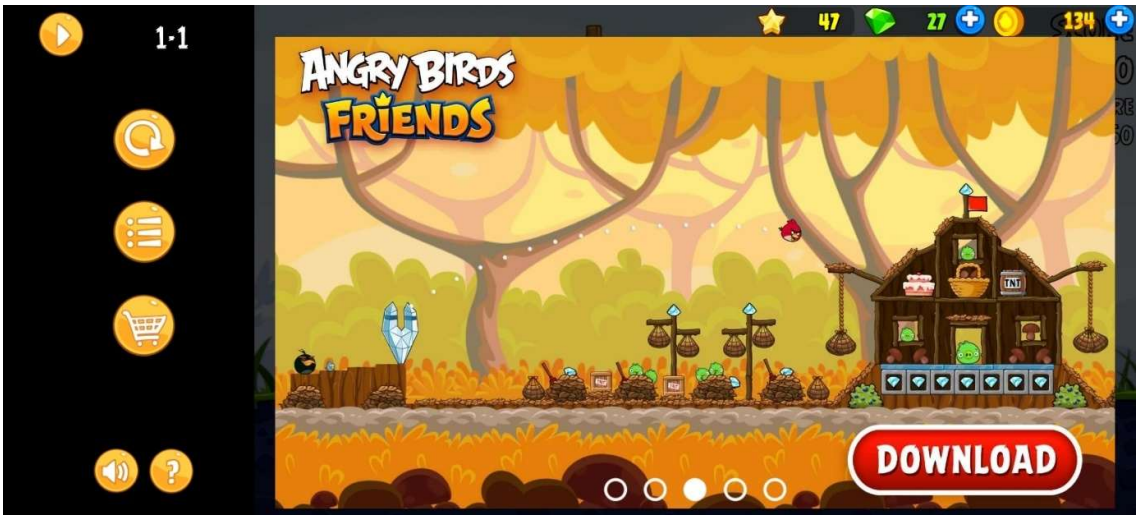
From the game, I have decided to implement the following features into my game:

Gameplay



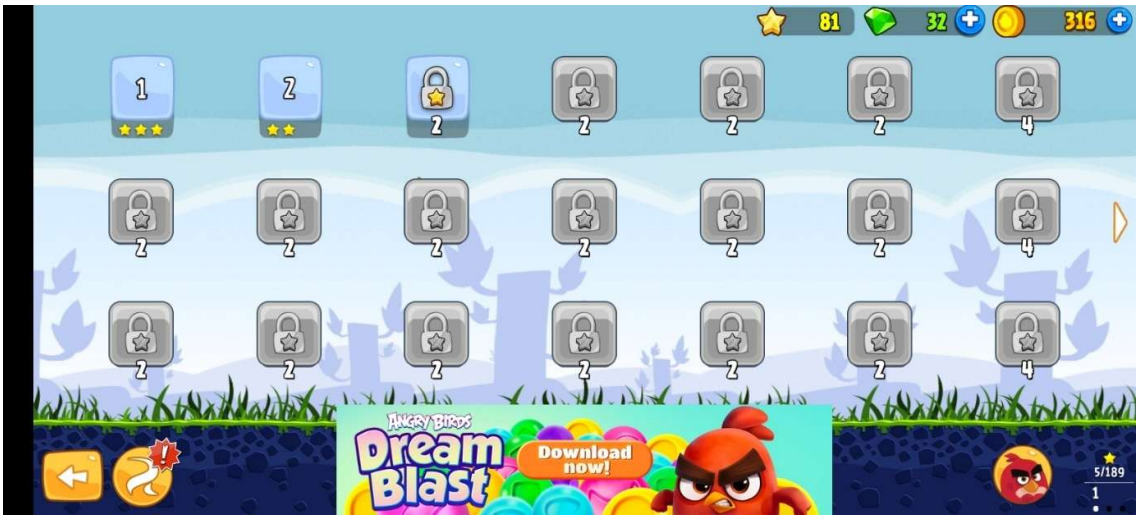


Pause State



Win State

Level Selection



Splash Screen

Next is Angry Birds 2, Which is a newer version of the Angry Birds Classic. They both have the same basic features except a radical change in design. The main reason I looked at this game for inspiration was because I thought some design changes like the card system worked better than the classic implementation incoming birds.

Gameplay

Card System

Level Selection

Splash Screen

Interview with Stakeholder

Plan

The main points I hope to get from this interview include:

- Basic Requirements for the game
- Design Changes to the game
- Changes in the physics of the game

Expanding on these main points, I have come up with questions under the topic of these main points:

Basic Requirements for the game

- How many different birds should be made for the bare requirements?
- Should these birds have unique abilities like in the classic game?
- What other features would you want the birds to have?
- Is a score panel a key element for the game, or do you think the game could work without it?
- How important is a high score panel to keep players engaged?
- Should this be a feature, and how would you like it displayed (global leaderboard, personal best)?
- What must absolutely be included for this to feel like a complete Angry Birds game?

Design Changes to the game

- Having played Angry Birds Classic before, what changes will you wish to have implemented design wise?

- Should the game be more realistic, or should it keep its original cartoonish style?
- How important is the look of the birds and pigs in enhancing the player's experience?
- Would you like the backgrounds and levels to be more interactive or dynamic in nature?
- What sort of animations or visual elements would you want to see?
- Should there be more exaggerated reactions when pigs are popped or blocks are destroyed?
- Should the game feature more detailed environments, weather effects, or lighting changes?

Changes in the physics of the game

- Should the game physics be more realistic, or do you prefer the exaggerated, cartoonish physics from the original?
- If so, what would you like to implement?
- Should bird flight paths be affected by environmental factors (wind, etc.) to add realism or new challenges?
- Should blocks break more realistically based on material type (e.g., wood vs. stone)?
- Should birds interact differently with various obstacles or environments (e.g., water, ice, etc.)?

Interview Script

Q: To start, how many different birds do you think should be made for the bare requirements of the game?

A: I think just adding "Red", the original bird with no special abilities, would be enough to start. We can always introduce more birds with abilities as the game progresses.

Q: Should these birds have unique abilities like in the classic game?

A: Yes, as the game progresses, introducing birds with unique abilities will definitely add variety and keep it interesting.

Q: What other features would you want the birds to have?

A: Keep them mostly the same as the original. People already like the way they are, and it worked well before, so no need to reinvent the wheel there.

Q: Is a score panel a key element for the game, or do you think the game could work without it?

A: It's nice to have, but not essential for the bare bones of the game. The main thing is that I can fling birds at pigs and eliminate them. The score is more like motivation to replay a level or compare with friends. If you are implementing the score panel though, I think that you should

make it so that you can compare scores on different levels with people all over the world or at least your friends.

Q: How important is a high score panel to keep players engaged?

A: It's not essential but would be awesome to have. I used to love comparing my high scores with friends, and it definitely made the game more competitive. But first, focus on the core gameplay, then add this.

Q: Should this be a feature, and how would you like it displayed? (Global leaderboard, personal best?)

A: Both would be great! It's always fun to see a personal best, but having a global leaderboard adds that extra competitive edge.

Q: What else must absolutely be included for this to feel like a complete Angry Birds game?

A: It just needs to be a cool puzzle game where I fling birds at pigs and deal with a few obstacles in between. That's what makes Angry Birds great for me.

Q: Having played Angry Birds Classic before, what changes would you wish to see design-wise?

A: I think the new "Angry Birds 2" is a bit too animated for my liking. I prefer the simplicity of the original. But I do like how in "Angry Birds 2" the birds are shown in a bar on the screen—it's nice to see all the birds lined up like that rather than how it is in the classic.

Q: Should the game be more realistic, or should it keep its original cartoonish style?

A: Keep it cartoonish. I don't want it looking like the Minecraft movie

Q: How important is the look of the birds and pigs in enhancing the player's experience?

A: Very important. I think they already look good, so no need to change their art style much. The original design worked really well.

Q: Would you like the backgrounds and levels to be more interactive or dynamic in nature?

A: Yeah, a static background could look boring after a while. Maybe a background with trees and some movement to make it more interesting.

Q: What sort of animations or visual elements would you want to see?

A: Keep the animations similar to the classic game. Nothing too fancy.

Q: Should there be more exaggerated reactions when pigs are popped or blocks are destroyed?

A: Nah, keep it simple. The reactions in the classic game were fun without being over the top.

Q: Should the game feature more detailed environments, weather effects, or lighting changes?

A: It would be nice, but it risks making the game look too much like Angry Birds 2. So, be careful with adding too many effects.

Q: Should the game physics be more realistic, or do you prefer the exaggerated, cartoonish physics from the original?

A: I like the cartoonish physics, but the only thing I didn't like was how the birds lost all their momentum when hitting a block. It gets kinda annoying. Maybe tweak that a bit.

Q: Should bird flight paths be affected by environmental factors like wind, or would that complicate things?

A: No, please don't add wind resistance. It would make things unnecessarily difficult.

Q: Should blocks break more realistically based on their material type, like wood versus stone?

A: Yeah, I think that would be cool. Blocks should definitely break differently depending on their material.

Q: Should birds interact differently with various obstacles or environments (e.g., water, ice, etc.)?

A: That would be harder to implement, but yeah, it would be good to see birds interact differently with different materials.

Review

After discussing the key elements, the stakeholder and I agreed that simplicity is essential for the game's core design, with the main objective being a puzzle-solving experience where the player uses birds to eliminate pigs.

For the basic requirements, the stakeholder expressed that adding just the original "Red" bird with no special abilities would suffice to start, ensuring the focus is on delivering the fundamental mechanics of the game. The idea is to keep the game simple at first, then progressively introduce more features, such as birds with unique abilities and additional challenges as the game advances.

Regarding gameplay features, the stakeholder felt that tracking progress, such as scoreboards and high score panels, isn't crucial for the game's initial version. The primary focus should be on allowing players to complete the core objective of eliminating pigs using birds. However, once the basic gameplay is polished, the inclusion of a score panel or leaderboard would be a great addition to enhance competitiveness and replayability.

In terms of visual design, the stakeholder preferred keeping the cartoonish style of the original Angry Birds, avoiding an overly animated or realistic look. The key takeaway here was that the birds, pigs, and overall aesthetic should stay true to the classic game, with minor tweaks like having backgrounds that are slightly more dynamic to prevent the environment from feeling static.

When discussing physics, it was agreed that the exaggerated, cartoonish physics from the original should remain, with the only requested change being to adjust how birds interact with blocks to prevent them from losing all momentum when hitting obstacles. This small tweak would improve gameplay while still keeping the game fun and familiar.

In conclusion, the stakeholder emphasized that the game should start simple, focusing on its core mechanics of solving puzzles by eliminating pigs. Additional features, such as more birds, complex scoring systems, and enhanced visuals, can be scaled up later, once the core gameplay has been fine-tuned.

Success Criteria

As per the feedback from the interview, I am going to divide my success criteria into parts; The Bare Essentials and the Extra Features.

Bare Essentials

Index	Feature	Descriptions
1	Slingshot	This is the main tool used by the player to launch birds at the structures. It adds a layer of strategy as players must aim and control the power of each shot.
1.1		Slingshot successfully launches birds
1.2		Slingshot animates when interacted with by the player
2	Bird	The bird acts as the projectile in the game, with each type having different abilities. This creates variety and adds depth to the gameplay.
2.1		Bird interacts with blocks by pushing them around depending on the force when in contact.
2.2		Bird interacts with pig by popping them when they are hit with a certain force
2.3		When bird gets launched by slingshot, bird obeys the laws of physics (ignoring unnecessary factors like Air resistance)
2.4		Number of Birds reduces when a target bird has been launched
3	Blocks	These form the structures that protect the pigs.
3.1		Basic Physics for Blocks; Physics governs how blocks fall which creates more dynamic and engaging gameplay as players have to predict how structures will collapse.

4	Pig	The main target in each level. Eliminating all pigs is necessary to complete the level. Their placement within structures makes strategic destruction essential.
4.1		Pig pops when hit with a certain force
5	Background	This sets the visual tone of the game and helps differentiate levels. While not crucial to gameplay, it adds to the game's aesthetic and experience.
6	Gameplay	The general gameplay of the game
6.1		The game determines if a player has won or lost based on the amount of pigs and birds left.
6.2		The core gameplay mechanic. Successfully aiming the bird at the pigs and causing destruction is the main objective of the game.
6.3		Ability to restart the game when it has been won or lost

Extra Features

Index	Feature	Descriptions
1	Score System	Allows players to track their best performances across different levels, adding a competitive element as players aim to beat their previous records or compete with friends.
2	High Score System	Allows players to track their best performances across different levels, adding a competitive element as players aim to beat their previous records or compete with friends.
3	Levels	The individual stages of the game, each with its own unique layout of pigs, structures, and challenges. Players progress by completing each level.
4	Sound Effects	Simple sound effects, such as the bird being launched or blocks breaking, enhance the game's feedback and immersion.
5	Splash Screen	A simple Splash Screen that plays when the game begins
6	Game Over Screen	A simple Game Over screen that activates if the player loses a game
7	Game Win Screen	A simple Game win screen that activates if the player wins a game

Limitations

The stakeholder mentioned that if we implement the score panel feature, that it should be available to compare with other players around the world. Though this would be a really good feature to implement, I noticed that implementing a feature like this will involve finding a place to store the user data online, and socket request which is beyond the scope of my project because there isn't enough time to implement this

Software and Hardware Requirements

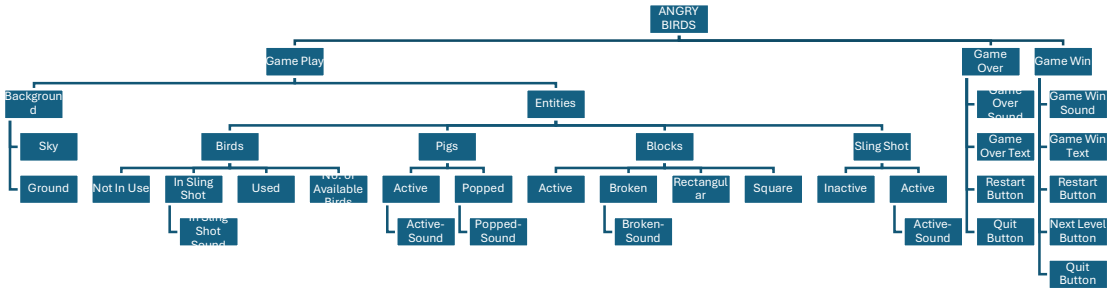
Requirement	Justification
Hardware	

Processor: 2GHZ+	Will need around 2GHZ in order to do the graphics and animations of the game as well as the tasks.
Memory: 1GB+ RAM	
Graphics: 12*MB+	You will require minimum graphics to run this game.
Hard Drive: No more than 250MB	
Mouse	Required to launch the bird
Monitor	Way of being able to view the game and play it.
Speaker	To hear Sound effects of the game
Software	
Operating system: WINDOWS 7/10/11 , macOS Mojave 10.14+, Linux Ubuntu 20.04	These are the operating systems that Unity will work with

DESIGN

Systems Diagram

As recommended by my stakeholder, I have chosen to keep the project simple for now before implementing more features. After my analysis, I did some further research into how the game can be broken down into more concise components, to reduce the problem of creating the game into more achievable tasks.



This hierarchy ensures clarity in structuring the game's components, states, and interactions, making it easier to develop and expand upon the Angry Birds game. By employing a top-down module design, I can effectively identify the necessary components before beginning the coding process. This approach allows me to decompose the problem into smaller, more manageable parts, enabling a focused examination of each segment. Utilizing the computational thinking skill of "thinking ahead," I can determine the required inputs and outputs. Additionally, "thinking logically" helps me pinpoint where decisions are needed and understand their impact on other parts of the solution. This structured methodology ensures that my working solution is both systematic and efficient, guiding me towards a well-organized and effective outcome.

The Angry Birds object serves as the main container for the entire game, encapsulating its three major states:

1. **Gameplay:** The interactive phase where the player engages in launching birds and targeting pigs.

2. **Game Over:** Triggered when all available birds are used but some pigs remain active.

3. **Game Win:** Triggered when all pigs are eliminated.

Game Play

This state represents the core interactive gameplay, comprising:

Background

1. **Sky:**

- Represents the top portion of the screen. The sky provides aesthetic appeal and acts as a visual backdrop.

2. **Ground:**

- Forms the base where structures, pigs, and the slingshot are placed.
- Ground physics ensure birds, blocks, or pigs that fall onto it behave realistically.

3. **Camera Movement:**

- The camera follows the currently active bird. This ensures the player maintains focus on the action and enhances immersion.

Entities

These are interactive elements within the game world:

1. **Birds:**

The primary projectile, with distinct states:

- **Not In Use:** Birds waiting in a queue for their turn to be launched. A subtle idle animation and sound effect could enhance realism.
- **In Sling Shot:** Birds being aimed and prepared for launch. Includes a sound effect (e.g., a stretching sound).
- **Used:** Birds that have been launched and either hit a target or missed.
- **No. of Available Birds:** Tracks how many birds are left for launching.

2. **Pigs:**

The targets in the game, with two primary states:

- **Active:** Pigs are alive and require elimination. They might display idle animations and sound effects.
- **Popped:** Pigs disappear with an animation and a "pop" sound when hit with sufficient force.

3. **Blocks:**

Structures that protect the pigs. They can be destroyed with force and have the following characteristics:

- **Active:** Undamaged and in position.
- **Broken:** Destroyed, typically via collisions with birds or other blocks.
- **Broken-Sound:** A unique sound effect plays upon destruction.
- **Types:** Rectangular and Square blocks offer variation in gameplay strategy.

4. **Sling Shot:**

The launching mechanism for the birds. States include:

- **Inactive:** When not in use.
- **Active:** When the player interacts with it to aim and launch a bird. Includes an "active sound" for stretching rubber.

Game Over

This state signifies the end of the game if the player loses:

1. **Game Over Sound:** A single sound effect to signal failure.
2. **Game Over Text:** Displays a message such as “Game Over” on the screen.
3. **Restart Button:** Allows the player to restart the current level.
4. **Quit Button:** Provides an option to exit the game or return to the main menu.

Game Win

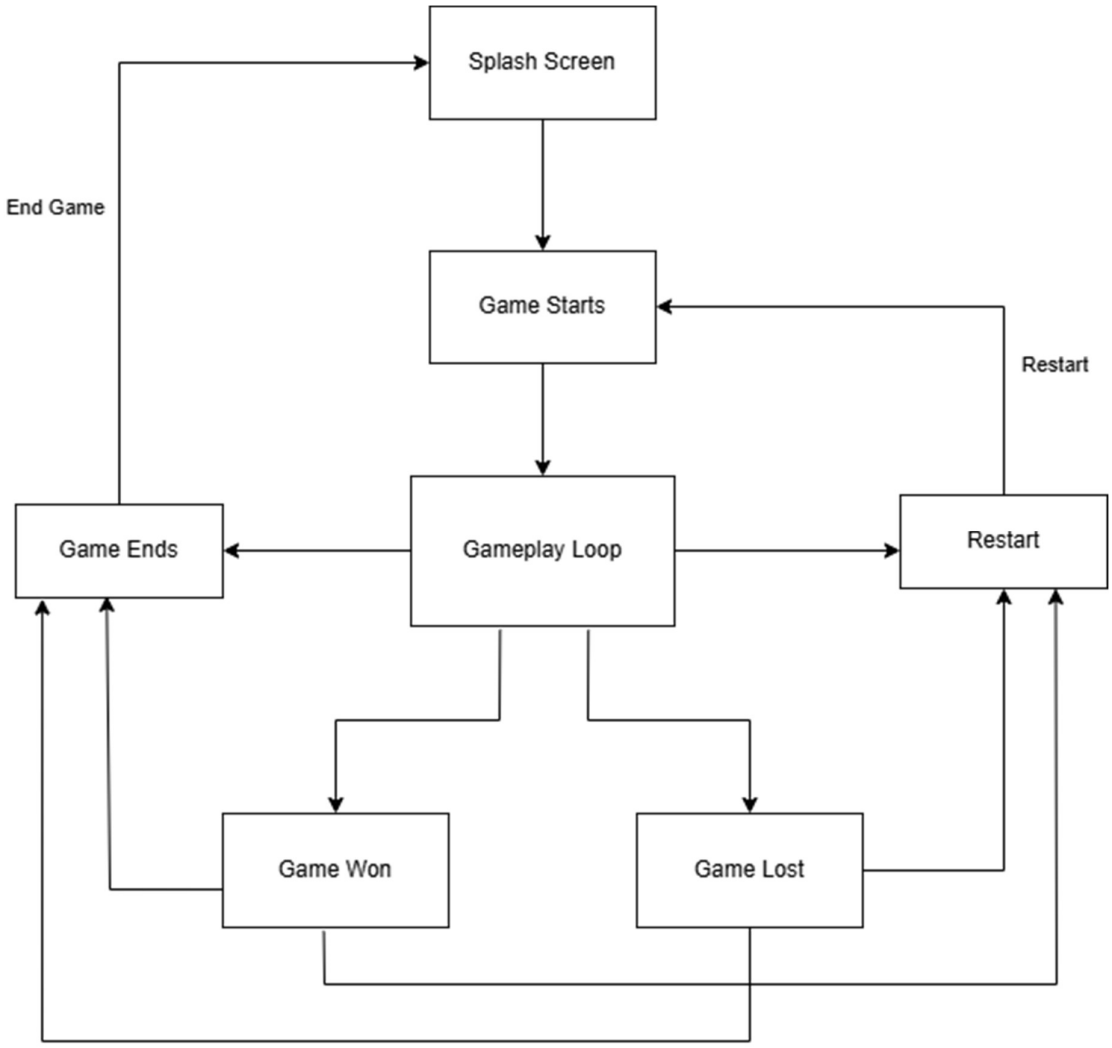
This state is triggered when the player successfully eliminates all active pigs:

1. **Game Win Sound:** Plays celebratory audio upon victory.
2. **Game Win Text:** Displays a congratulatory message.
3. **Restart Button:** Enables the player to replay the level for better performance.
4. **Next Level Button:** Progresses to the subsequent level (if implemented).
5. **Quit Button:** Allows the player to exit the game or return to the main menu.

Game Progression Overview

To further decompose the problem, I have decided to go over how the player will progress through every game module. Because I have two development success criteria (Bare features and Advanced features), I have designed a progression overview that covers the scope of each of these criteria. To reiterate, Bare Features serve as the foundation for core gameplay while Advanced Features add complexity and enhance player engagement. A modular design approach enables incremental development and testing of each component.

Bare Features Game Progression



Game Start

- The game begins with a simple splash screen.
- Player clicks "Start" to enter the first level.

Gameplay Loop

Camera Movement

- The camera centres on the slingshot and active bird.
- Tracks the bird during its flight, ensuring focus remains on the action.

Bird Launch

- Player interacts with the slingshot to aim and launch the bird.

Slingshot Animation:

- Stretches dynamically based on drag distance and angle.
- Upon release, the bird obeys basic physics (ignoring air resistance).

Target Interaction

Birds collide with blocks and pigs:

- Blocks: May fall or break depending on the force applied.
- Pigs: Pop when hit with sufficient force.

Level Conditions

- Player wins the level if all pigs are eliminated.
- Player loses if all birds are used without clearing the pigs.

Game End States

Game Win Screen

- Triggered when all pigs are eliminated.
- Displays a congratulatory message and a "Restart" button.

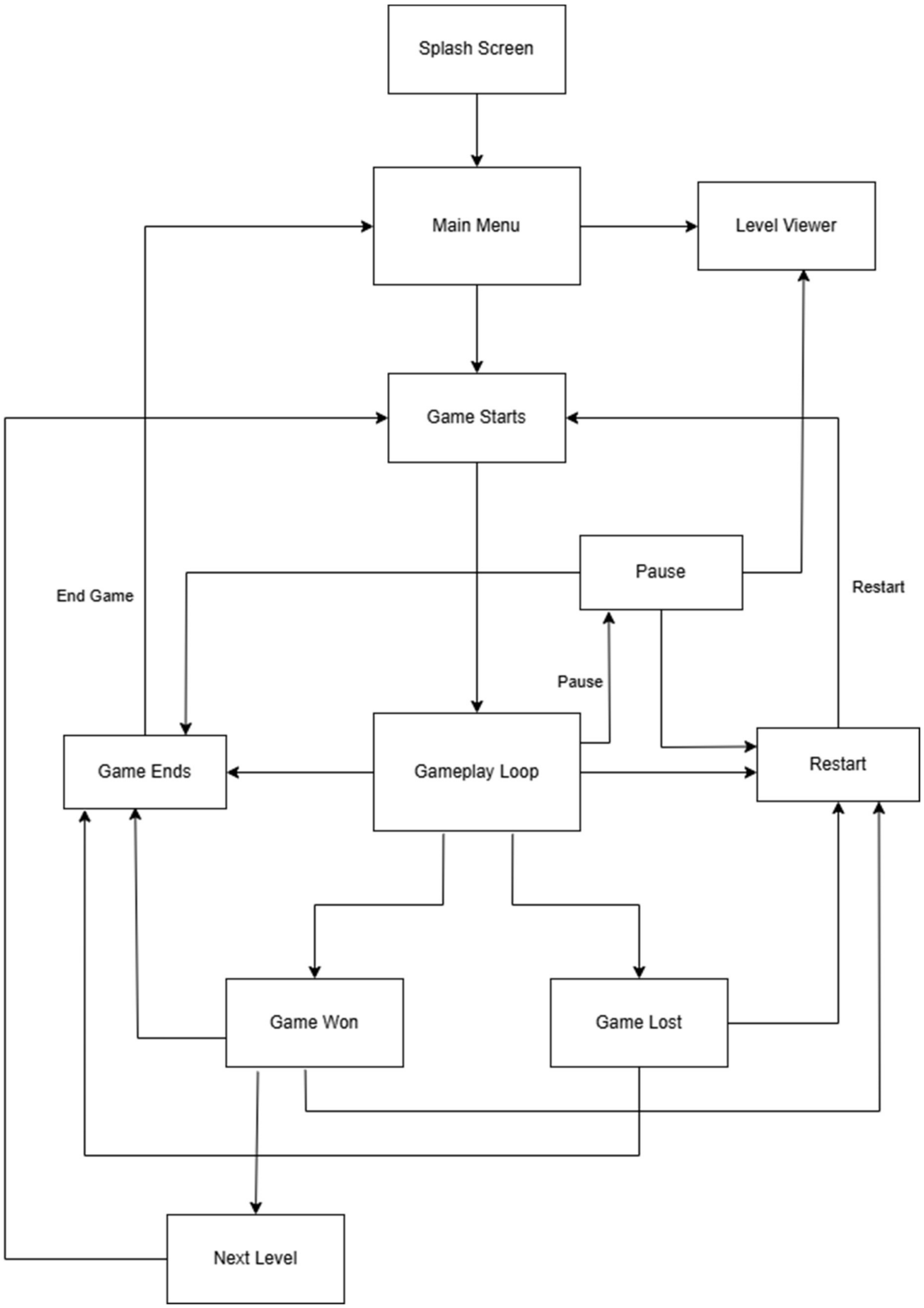
Game Over Screen

- Triggered when birds are depleted, and pigs remain.
- Displays "Game Over" and provides "Restart" and "Quit" buttons.

Conditions

- Win Condition: All pigs are popped.
- Lose Condition: All birds are used, and pigs remain.
- Restart: Reinitializes the current level.
- Quit: Exits to the splash screen.

Advanced Features Game Progression



Main Menu

The player starts at a menu page with options:

- Start Game: Launches the first level or the last unlocked level.
- Level Viewer: Displays a list of available levels and their statuses.
- Quit: Exits the game.

Loading Screen

Briefly displayed during transitions between menu and gameplay, improving immersion.

Gameplay Loop

Pause Feature:

- Player can pause the game.
- Options include "Resume," "Restart," or "Quit to Menu."

Score System:

- Tracks points for each level based on performance (e.g., number of birds left, destruction).
- Adds replay value as players aim for high scores.

Powerups:

- Introduces temporary abilities for strategic advantage.
- Examples include stronger birds, explosive impacts, or extra birds.

Camera Movement:

- Enhanced tracking with zoom-in/zoom-out mechanics for better visualization.

Advanced Bird Types:

- Birds with unique abilities (e.g., split into smaller birds, explosive impact).
- Adds variety and strategy to the game.

Levels

- Includes 10 unique levels with increasing complexity
- Level Viewer Screen tracks progress, showing completed and locked levels.
- Each level introduces different structures, pig placements, and challenges.
- Extra block types (e.g., stone, glass) behave differently under impact, increasing strategy.

Game End States

Game Win:

- Unlocks the next level if available.
- Provides "Restart," "Next Level," and "Quit" options.

Game Over

- Prompts replay with the same options as the basic version.

High Score System

- Tracks best scores per level across sessions, encouraging competition.

Conditions

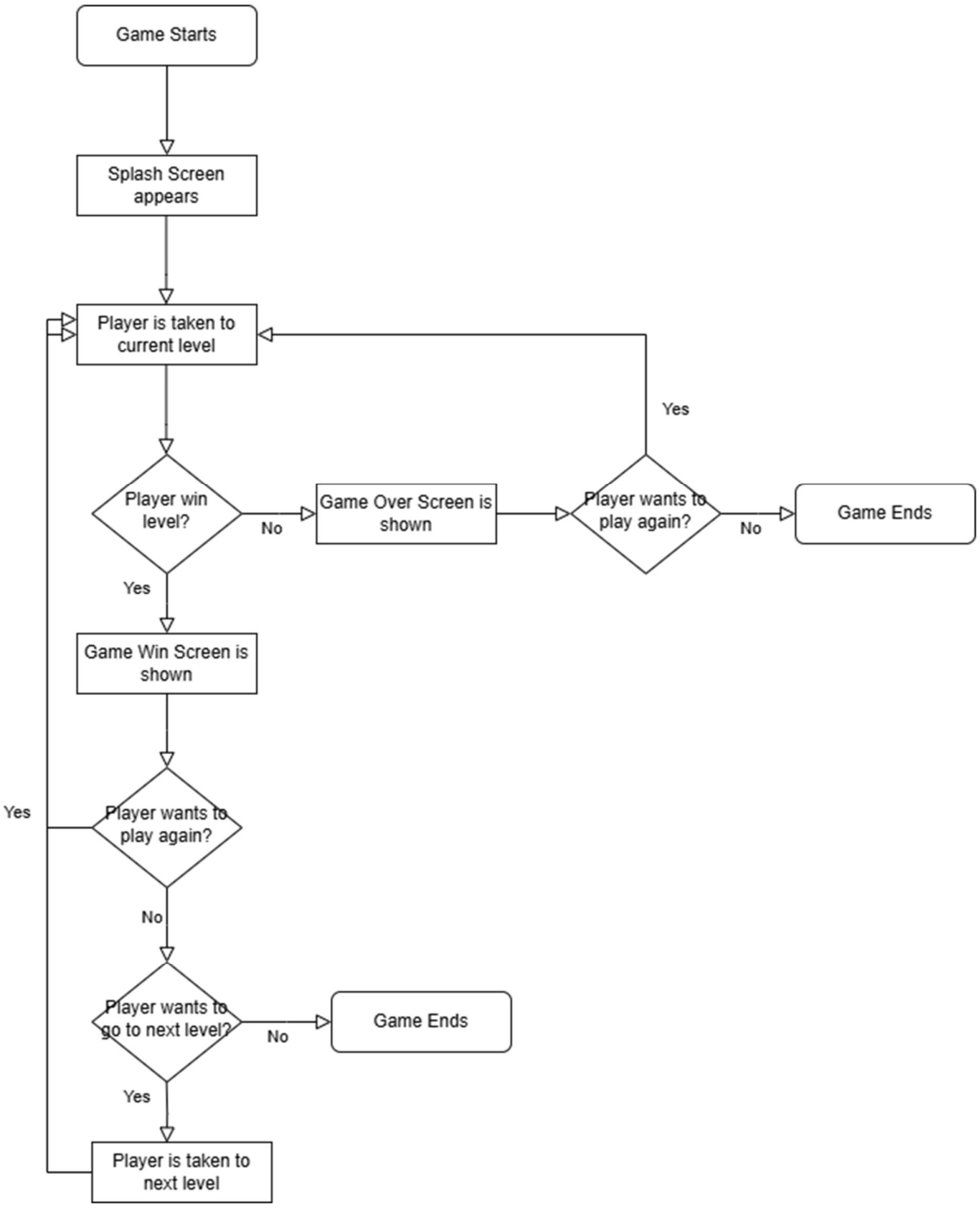
- Win Condition: All pigs are eliminated before using all birds.
- Lose Condition: All birds are used, and pigs remain.
- Next Level: Available if the current level is cleared.
- High Score: Stored if a new record is set.
- Powerup Usage: Players must manage powerups strategically, as their availability may be limited per level.

Algorithms

Gameplay Flow

This algorithm outlines the step-by-step process of a game flow. The game starts by displaying a splash screen and then takes the player to the current level. The game runs in a loop where it checks if the player wins the level. If the player wins, a game win screen is shown, and the player is given the option to play again. If the player chooses to play again, they can either proceed to the next level or end the game. If the player does not win the level, a game over screen is shown, and the player is given the option to play again from the current level or end the game. This flow is a structured way to manage game progression, player choices, and game state transitions.

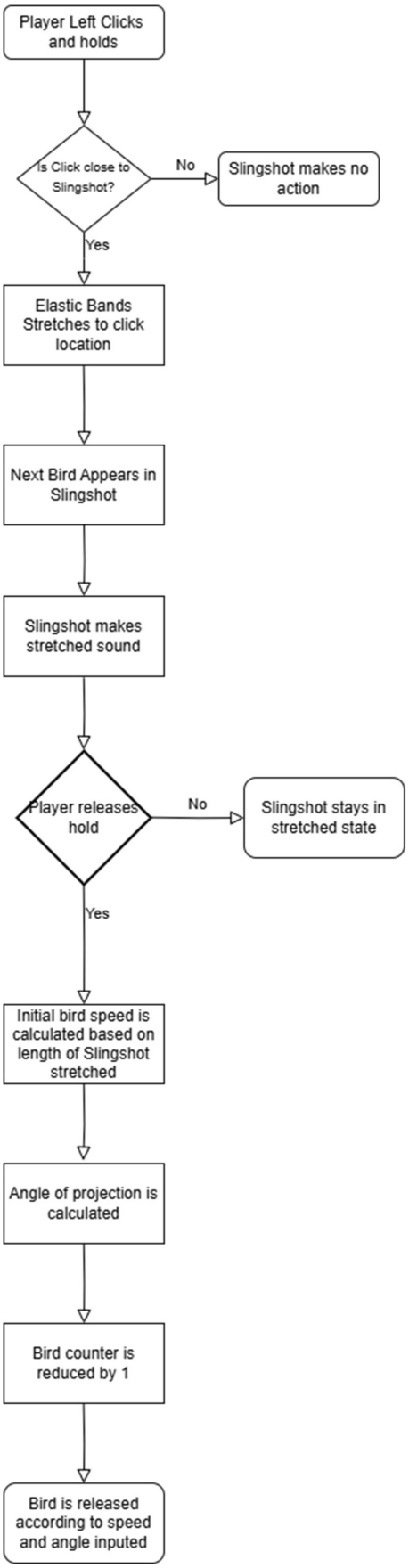
```
START
    Display Splash Screen
    Take Player to Current Level

    WHILE game is running
        IF Player wins level THEN
            Display Game Win Screen
            IF Player wants to play again THEN
                IF Player wants to go to next level THEN
                    Take Player to Next Level
                ELSE
                    END GAME
            ELSE
                END GAME
        ELSE
            Display Game Over Screen
            IF Player wants to play again THEN
                Take Player to Current Level
            ELSE
                END GAME
    END WHILE
END
```


Game Starts
Splash Screen appears
Player is taken to current level
Player win level?
No
Game Over Screen is shown
Player wants to play again?
No
Game Ends
Yes
Yes
Game Win Screen is shown
Player wants to play again?
Yes
No
Player wants to go to next level?
No
Game Ends
Yes
Player is taken to next level

Detecting if the Player wants to handle the Sling Shot

The process begins when the player left-clicks and holds the mouse button. The system checks if the click is close to the slingshot. If it is, the elastic bands stretch to the click location, and the next bird appears in the slingshot. A stretched sound is played, and the system waits for the player to release the hold. Once released, the initial bird speed is calculated based on the length of the slingshot stretch, and the angle of projection is determined. The bird counter is then reduced by one, and the bird is released according to the calculated speed and angle.

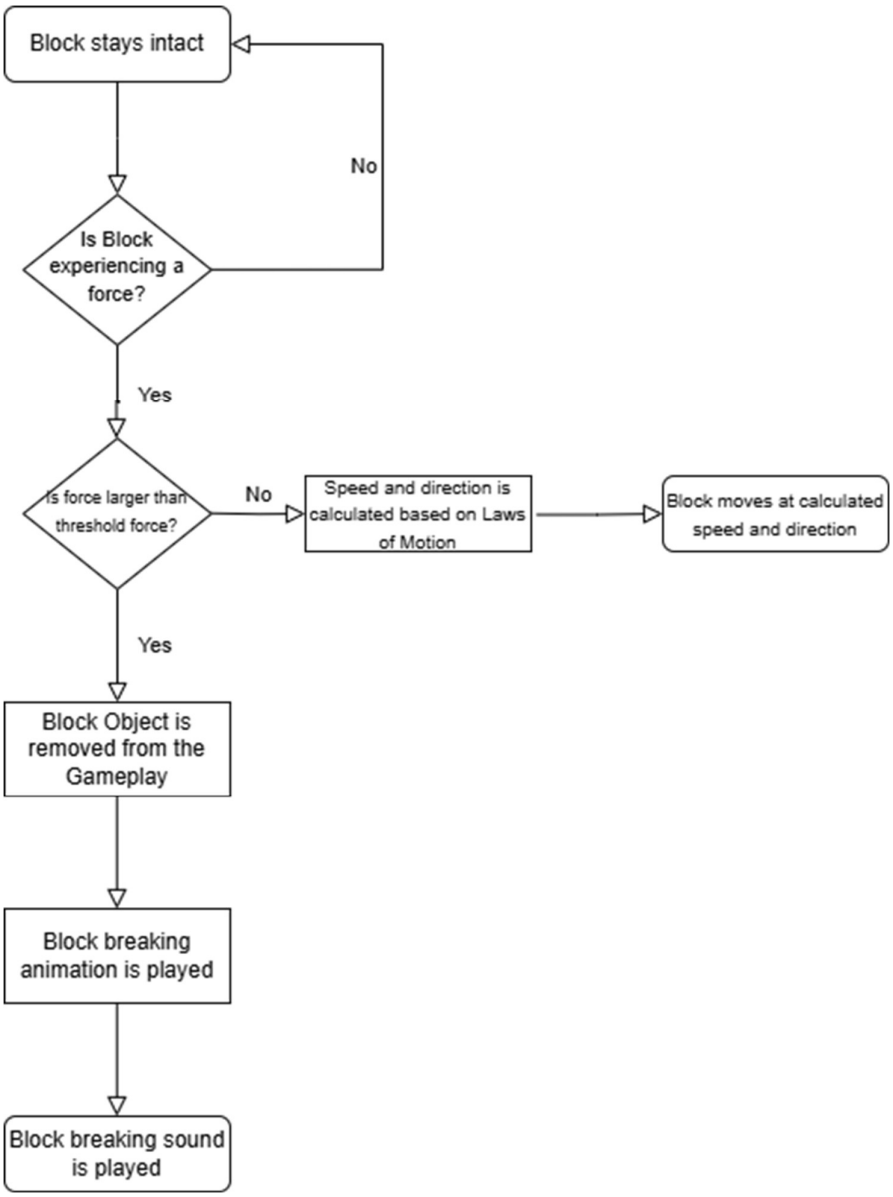
```
START
    Player initiates action by left-clicking and holding

    IF click is close to slingshot THEN
        Stretch elastic bands to click location
        Next bird appears in slingshot
        Play stretched sound

        WHILE player has not released hold
            Slingshot stays in stretched state
        END WHILE

        Calculate initial bird speed based on slingshot stretch length
        Calculate angle of projection
        Reduce bird counter by 1
        Release bird according to speed and angle
    ELSE
        Slingshot makes no action
    END IF
END
```


Player Left Clicks and holds
Is Click close to Slingshot?
No
Slingshot makes no action
Yes
Elastic Bands Stretches to click location
Next Bird Appears in Slingshot
Slingshot makes stretched sound
Player releases hold
No
Slingshot stays in stretched state
Yes
Initial bird speed is calculated based on length of Slingshot stretched
Angle of projection is calculated
Bird counter is reduced by 1
Bird is released according to speed and angle inputed

Detecting if a Block is Broken

The process begins with the block staying intact. During the game, the system continuously checks if the block is experiencing a force. If a force is detected, the system then checks if the force is larger than a predefined threshold. If the force exceeds the threshold, the block object is removed from the gameplay, and a breaking animation and sound are played. If the force is below the threshold, the block's speed and direction are calculated based on the laws of motion, and the block moves accordingly.

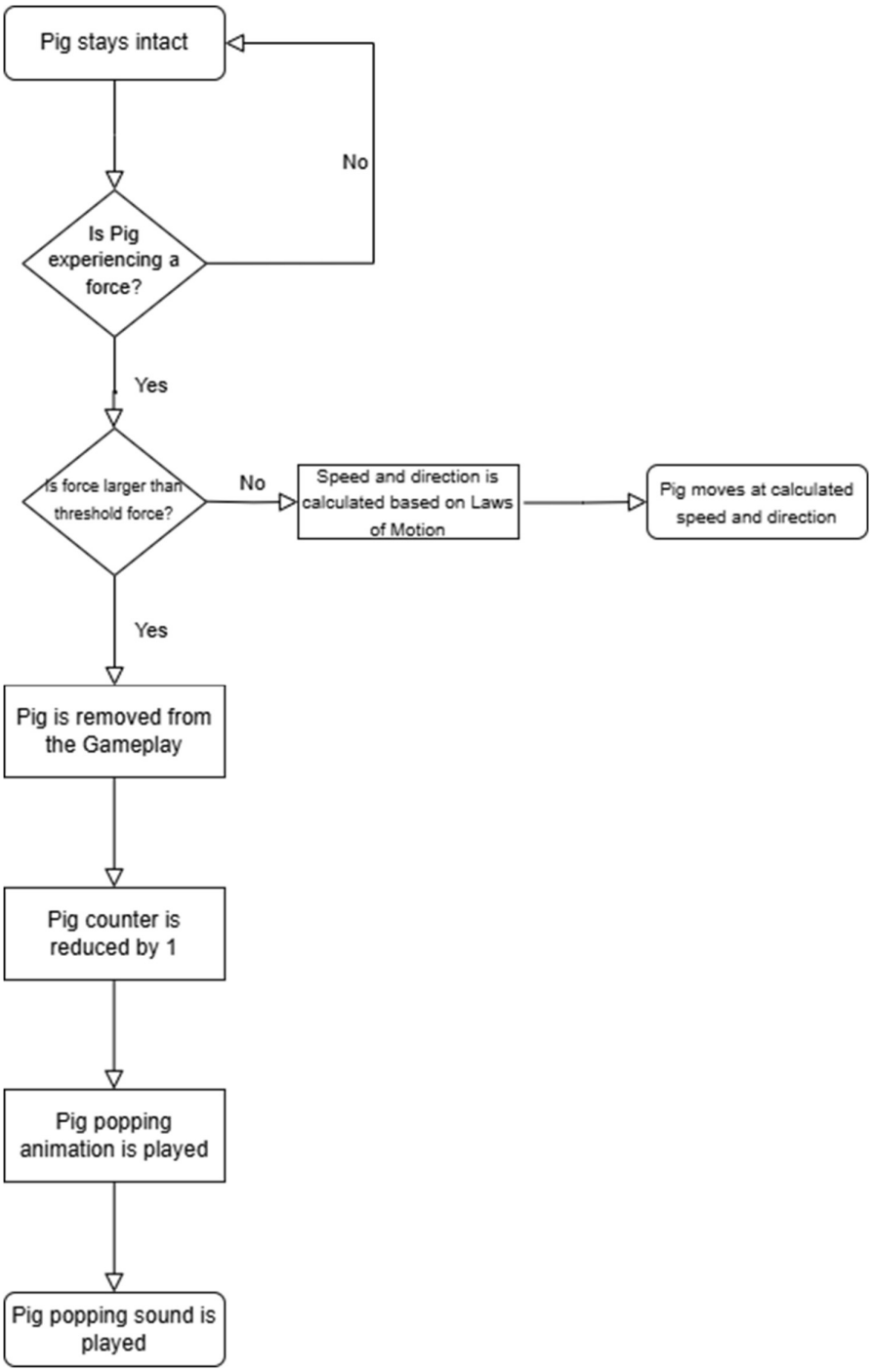
```
START
    Block stays intact

    WHILE game is running
        IF Block is experiencing a force THEN
            IF force is larger than threshold force THEN
                Block Object is removed from the Gameplay
                Play Block breaking animation
                Play Block breaking sound
            ELSE
                Calculate speed and direction based on Laws of Motion
                Block moves at calculated speed and direction
            END IF
        ELSE
            Block stays intact
        END IF
    END WHILE
END
```



Detecting if a Pig has been popped

Similar to the Block Breaking Algorithm, the process begins with the pig staying intact. During the game, the system continuously checks if the pig is experiencing a force. If a force is detected, the system then checks if the force is larger than a predefined threshold. If the force exceeds the threshold, the pig is removed from the gameplay, the pig counter is reduced by one, and a popping animation and sound are played. If the force is below the threshold, the pig's speed and direction are calculated based on the laws of motion, and the pig moves accordingly.

```
START
    Pig stays intact

    WHILE game is running
        IF Pig is experiencing a force THEN
            IF force is larger than threshold force THEN
                Pig is removed from the gameplay
                Reduce pig counter by 1
                Play pig popping animation
                Play pig popping sound
            ELSE
                Calculate speed and direction based on Laws of Motion
                Pig moves at calculated speed and direction
            END IF
        ELSE
            Pig stays intact
        END IF
    END WHILE
END
```

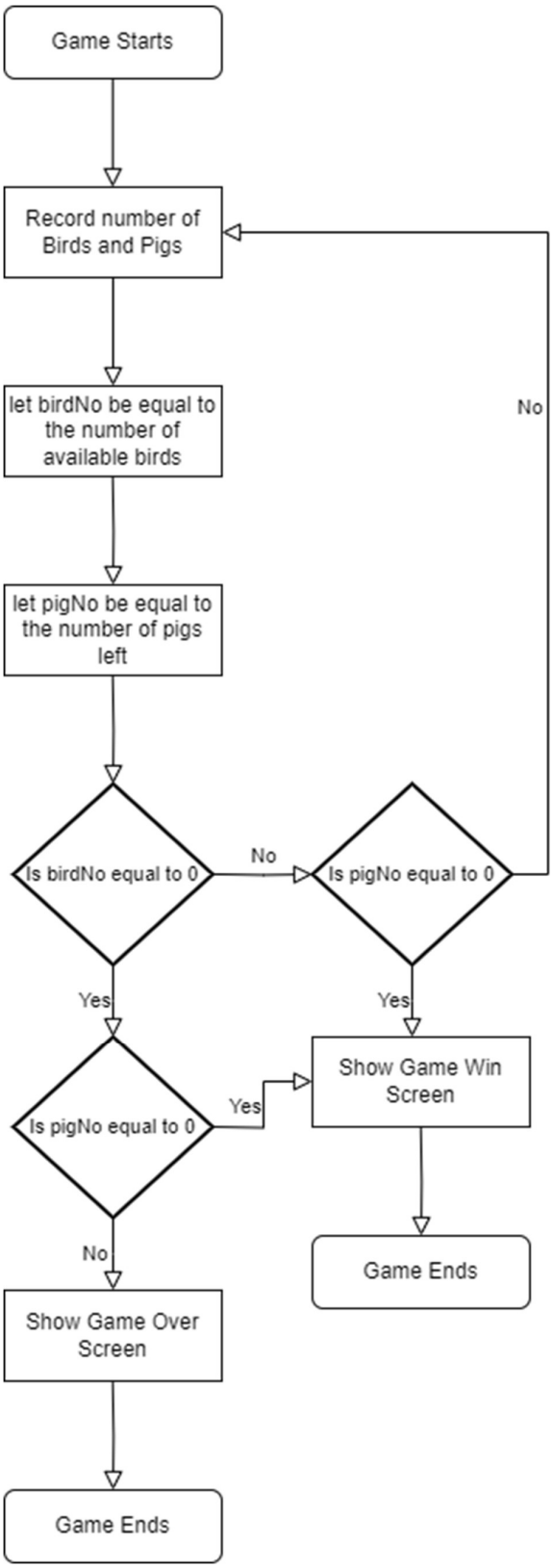


Detecting if a is Game Over

This algorithm outlines the game flow logic, focusing on the number of birds and pigs. The game starts by recording the number of birds and pigs and setting the variables `birdNo` and `pigNo` to the number of available birds and pigs left, respectively. During the game, the system continuously checks if `birdNo` is equal to 0. If it is, the “game over” screen is shown, and the

game ends. If `birdNo` is not 0, the system then checks if `pigNo` is equal to 0. If it is, the game win screen is shown, and the game ends. If neither condition is met, the game continues. This process is crucial for managing game progression and determining the game's outcome based on the player's performance.

```
START
    Game Starts
    Record number of Birds and Pigs
    birdNo = number of available birds
    pigNo = number of pigs left

    WHILE game is running
        IF birdNo == 0 THEN
            Show Game Over Screen
            END GAME
        ELSE
            IF pigNo == 0 THEN
                Show Game Win Screen
                END GAME
            ELSE
                // Continue game logic
            END IF
        END IF
    END WHILE
END
```


Game Starts
Record number of Birds and Pigs
let birdNo be equal to the number of available birds
let pigNo be equal to the number of pigs left
Is birdNo equal to 0
No
Is pigNo equal to 0
No
Yes
Yes
Is pigNo equal to 0
Yes
Show Game Win Screen
No
Game Ends
Show Game Over Screen
Game Ends

Test Data

Slingshot		
Tests	**Expected output**	**Explanation**
The Slingshot animates when being attempted to launch. If the slings is past the slingshot area it restrict the player	The slings pull back when left clicked and follow the cursor till the player releases	This test ensures that the slingshot behaves realistically by pulling back smoothly when the player clicks and drags it. The restriction prevents unrealistic stretching beyond the designated area, maintaining fair gameplay and a natural look.
Slingshot launches bird when the player releases the slingshot after being pulled back	The bird is flung out of the slingshot with a specific force	This test verifies that the bird is correctly launched when the player releases the slingshot. The force applied should be proportional to how far back the slingshot was pulled, ensuring consistency in gameplay.

Bird		
Tests	**Expected output**	**Explanation**
Interaction between Bird and Block	The bird alters the block based on its force and direction.	When the bird collides with a block, the impact should cause movement or destruction depending on the bird's speed and angle of collision. This test ensures that the physics engine properly calculates these interactions.
Interaction between Bird and Pig	The bird hits the pig, and pops them when they are hit with a certain force	The game should recognize when the bird has hit a pig with enough force to eliminate it. If the impact is too weak, the pig remains in play, maintaining the challenge for the player.
Bird obeys laws of physics of look realistic	Bird has an acceleration due to gravity and acts like a normal projectile when launched (excluding air resistance)	The bird should follow a parabolic trajectory with acceleration due to gravity. This means after being launched, it should rise and then fall realistically, without floating or stopping mid-air.

Blocks		
Tests	**Expected output**	**Explanation**
Physics Govern how blocks act	The bird alters the block based on its force and direction. The block falls in response to gravity	Blocks should move or topple when struck with sufficient force. They must also fall naturally due to gravity if unsupported, ensuring a dynamic environment. designated area, maintaining fair gameplay and a natural look.

Pigs		
Tests	**Expected output**	**Explanation**
Pigs interaction with Bird	The pig pops when hit with a certain force or else it stays a in the game	The pig should pop when hit with a strong enough impact but remain in play if the force is insufficient. This ensures a balance in difficulty and realism in gameplay.

Background		
Tests	**Expected output**	**Explanation**
The Game has a Static Background	The Game has a Static Background that supports the other sprites in the game.	The background should not move or interfere with the gameplay, providing a consistent visual experience for the player.

Gameplay		
Tests	**Expected output**	**Explanation**
Determines if Game has been won or lost	The game monitors the number of Bird and Pigs every time a bird has been used and determines if the Game has been won or lost.	The game continuously checks the number of remaining pigs after each shot. If all pigs are eliminated, the player wins. If all birds are used and pigs remain, the player loses.
Ability to Restart the Game when it has been won or lost	Game restarts or has the option to restart when the game is over.	After the game ends, either by victory or defeat, the player should have the option to restart, ensuring replayability.
Core Gameplay Mechanic: The Bird tries to knock out all the Pigs with a limited amount of Birds	The Game lets you aim and pop pigs. If you can't pop all the pigs with a limited amount of Birds you lose the game else you win.	The player should be able to aim and strategize how to take out all the pigs with a limited number of birds. If successful, they win; otherwise, they lose, maintaining the challenge of the game.

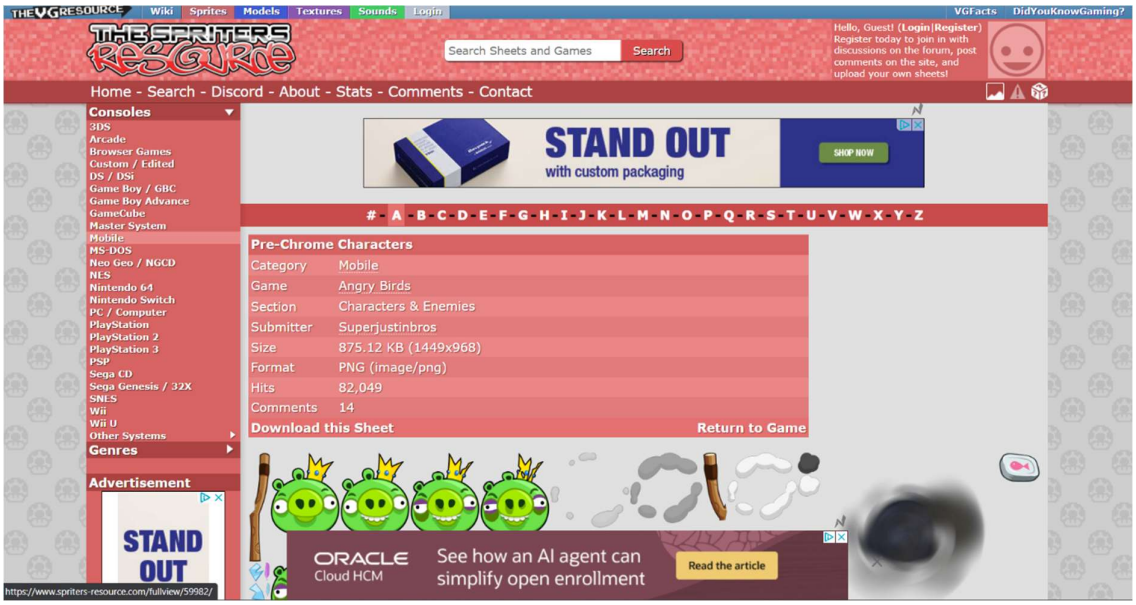
DEVELOPMENT

Developing the Coded Solution

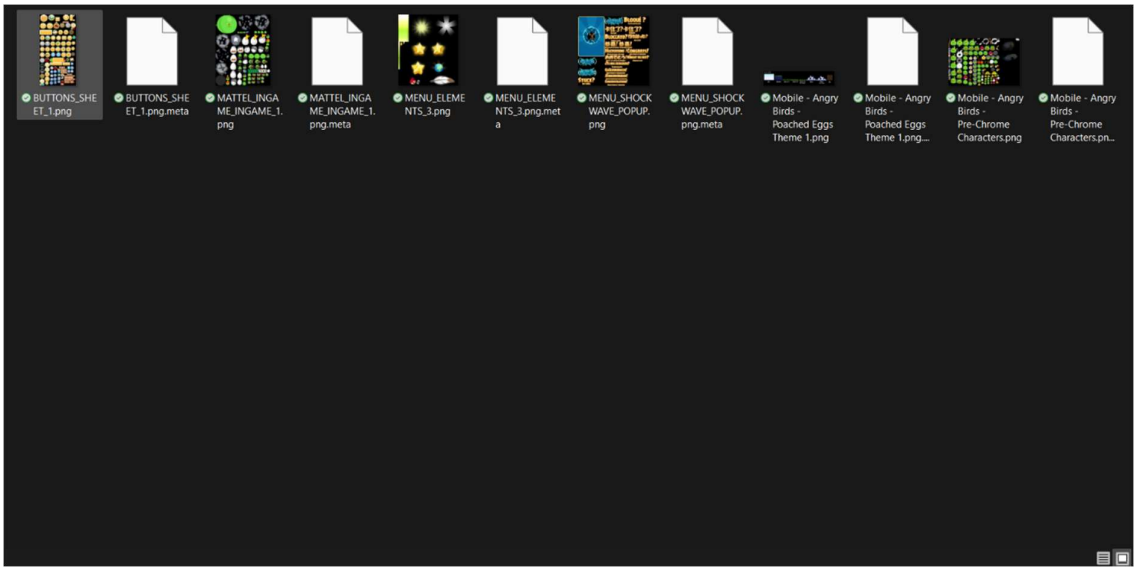
All code referred to can be found in the Project Appendix at the end of this document.

Creating the Environment [20/02/2025]

I first decided to install all the necessary sprites and create a background for the game. To begin, I searched the internet for appropriate pixel-style images that would fit the aesthetic of my game, such as birds, pigs, blocks, and a slingshot, along with a suitable background image. Though I found many assets suitable for the game, my stakeholder preferred to use assets that were as similar to the original game as possible. So I found some original Rovio (The company that made Angry Birds) assets for free on a website called the Spriters Resource.



The Resource used to find Sprites

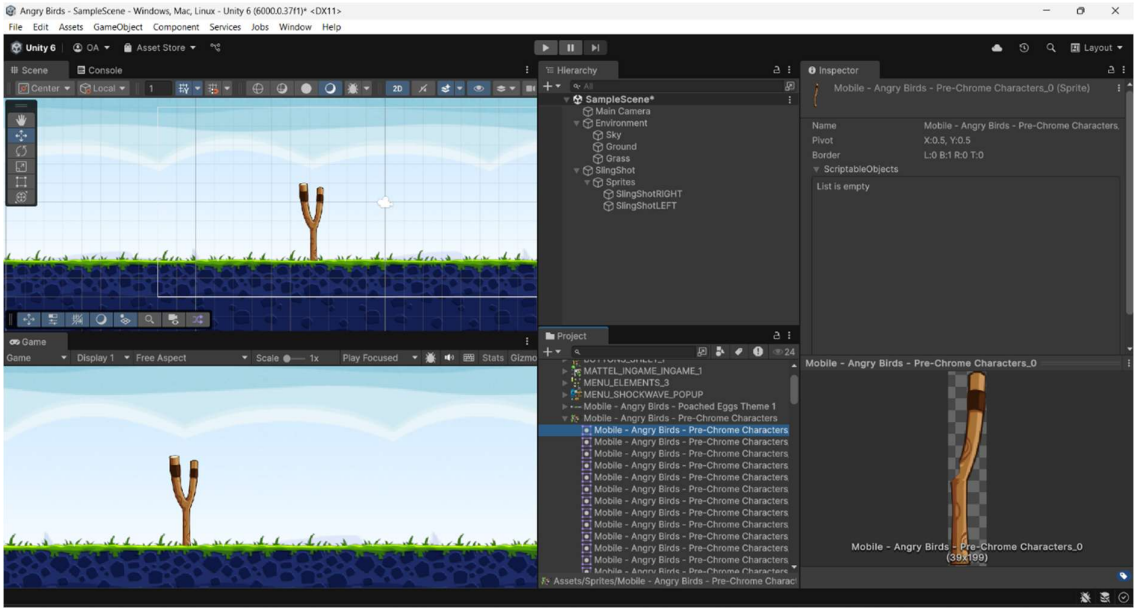


Stored all Assets in a Folder

Once I had gathered the required assets, I used Unity Sprite Editor to assemble and edit some of the sprites where necessary, ensuring they matched the overall visual style of the game and that their hit boxes weren't too big. Due to the relatively simple nature of the sprites, no other specialised software was required for this step.

Editing the sprites on Unity Editor

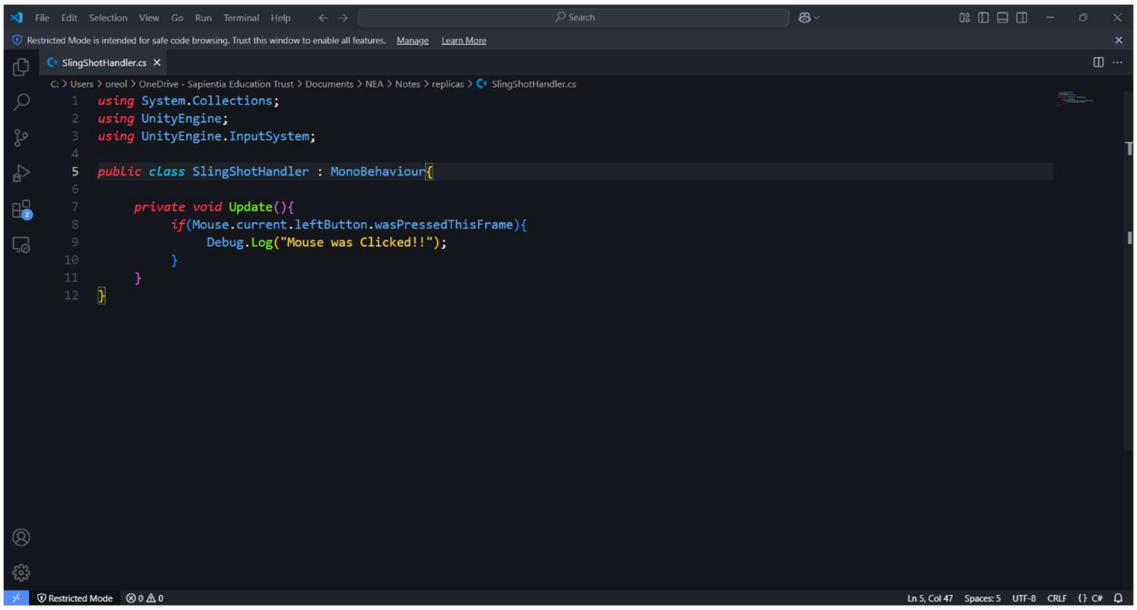
Once all sprites were properly resized and formatted, I set up the game background. This involved selecting and positioning a sky and ground texture that would serve as the foundational environment for the gameplay. Alongside this, I also designed the slingshot which will be launching the bird as an interactable part of the environment.



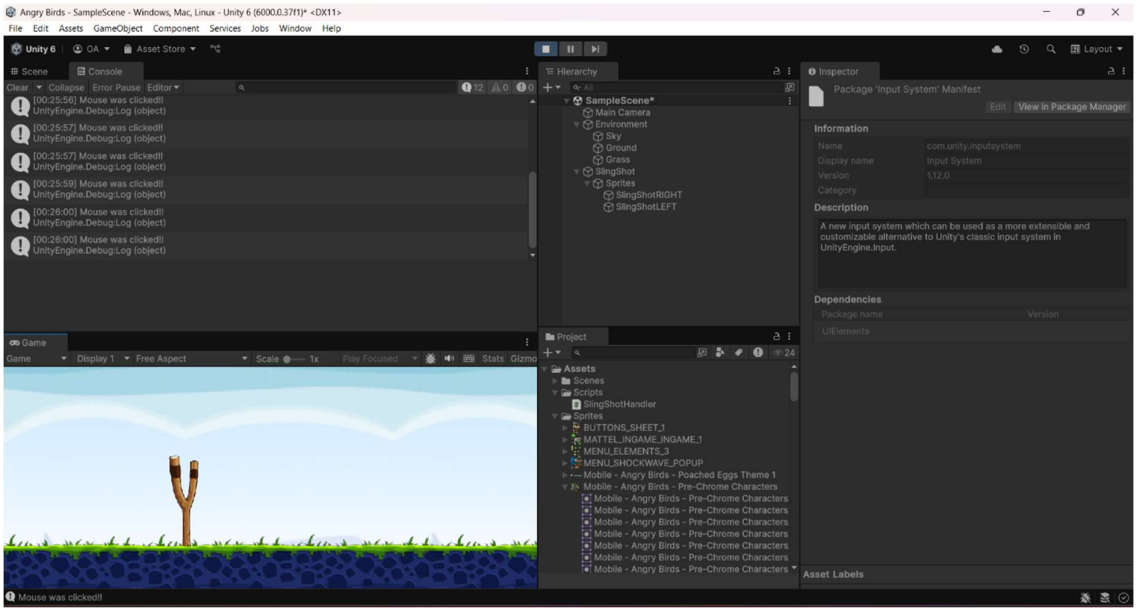
Creating Script for Slingshot [21/02/2025]

After positioning and assembling the environment and the slingshot, I decided to begin scripting the basic functionality for the slingshot using VS Code, specifically focusing on the mechanics that would allow it to stretch when pulled. Since the slingshot is a core element of the gameplay, it was important to ensure that its movement felt both natural and intuitive for the player.

Before implementing the full functionality, I first needed to test whether the game's input system could properly detect user interactions. To do this, I wrote a simple console log test to verify that input events were being recorded correctly within my script.

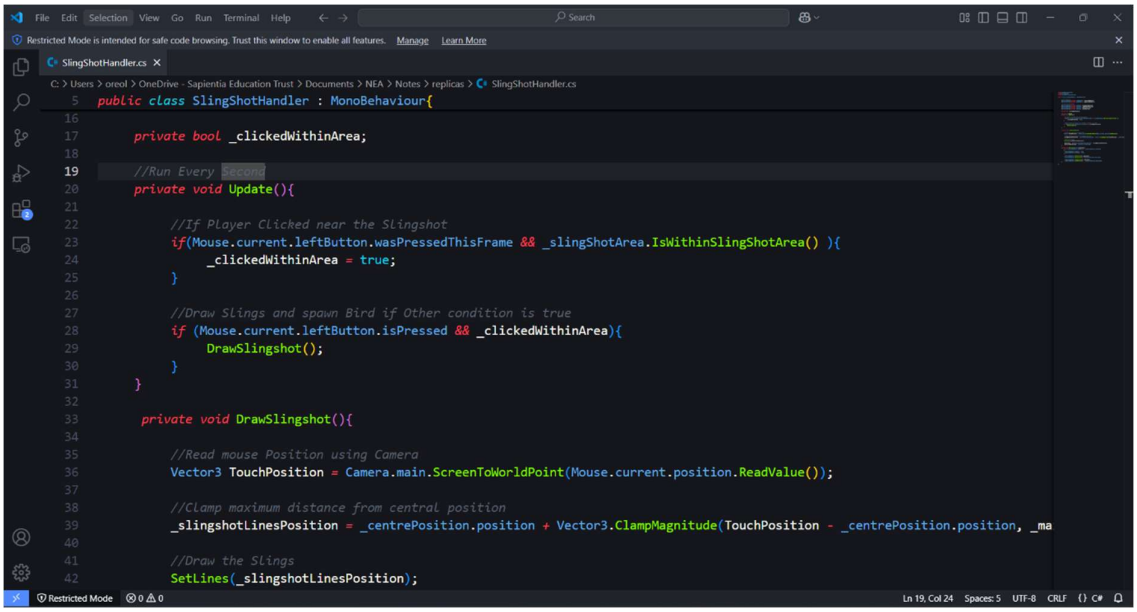


Code Tested



Results after testing

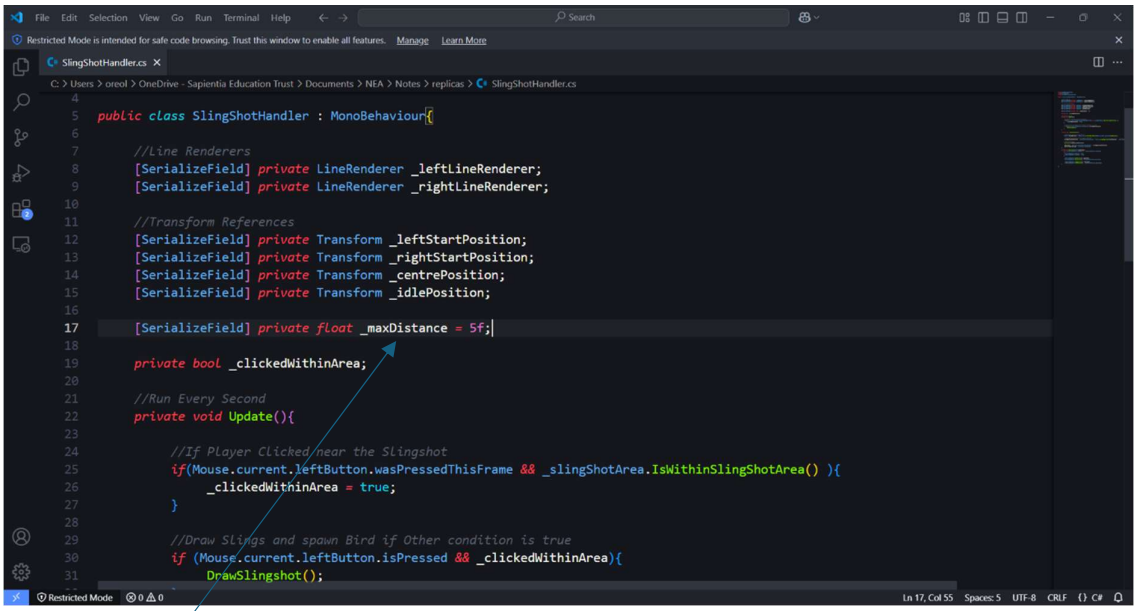
Once I confirmed that input detection was working as expected, I proceeded with integrating the slingshot's control system. For aiming and launching, I chose to use the mouse as the primary input method. I found that the mouse offered a more intuitive and precise control scheme compared to the keyboard, making it easier for the player to adjust the trajectory of their shots. Implementing mouse control also streamlined the coding process, as it allowed for direct tracking of the cursor's movement in relation to the slingshot's position.



```
public class SlingShotHandler : MonoBehaviour{

    private bool _clickedWithinArea;

    //Run Every Second
    private void Update(){

        //If Player Clicked near the Slingshot
        if(Mouse.current.leftButton.wasPressedThisFrame && _slingShotArea.IsWithinSlingShotArea() ){
            _clickedWithinArea = true;
        }

        //Draw Slings and spawn Bird if Other condition is true
        if (Mouse.current.leftButton.isPressed && _clickedWithinArea){
            DrawSlingshot();
        }
    }

     private void DrawSlingshot(){

        //Read mouse Position using Camera
        Vector3 TouchPosition = Camera.main.ScreenToWorldPoint(Mouse.current.position.ReadValue());

        //Clamp maximum distance from central position
        _slingshotLinesPosition = _centrePosition.position + Vector3.ClampMagnitude(TouchPosition - _centrePosition.position, _ma

        //Draw the Slings
        SetLines(_slingshotLinesPosition);
```

Code written to adjust the Slings

After further testing, the Slings followed the cursor everywhere it went on the screen. I then moved on to defining the slingshot's stretching mechanics. To maintain realism and ensure a smooth user experience, I imposed a maximum stretching limit of 20 pixels from the slingshot's central position. This constraint prevented excessive stretching while still allowing the player enough flexibility to aim effectively. Additionally, this restriction contributed to the overall aesthetic of the game by ensuring that the slingshot animation remained visually consistent and natural.



```
public class SlingShotHandler : MonoBehaviour{

    //Line Renderers
    [SerializeField] private LineRenderer _leftLineRenderer;
    [SerializeField] private LineRenderer _rightLineRenderer;

    //Transform References
    [SerializeField] private Transform _leftStartPosition;
    [SerializeField] private Transform _rightStartPosition;
    [SerializeField] private Transform _centrePosition;
    [SerializeField] private Transform _idlePosition;

    [SerializeField] private float _maxDistance = 5f;

    private bool _clickedWithinArea;

    //Run Every Second
    private void Update(){

        //If Player Clicked near the Slingshot
        if(Mouse.current.leftButton.wasPressedThisFrame && _slingShotArea.IsWithinSlingShotArea() ){
            _clickedWithinArea = true;
        }

        //Draw Slings and spawn Bird if Other condition is true
        if (Mouse.current.leftButton.isPressed && _clickedWithinArea){
            DrawSlingshot();
```

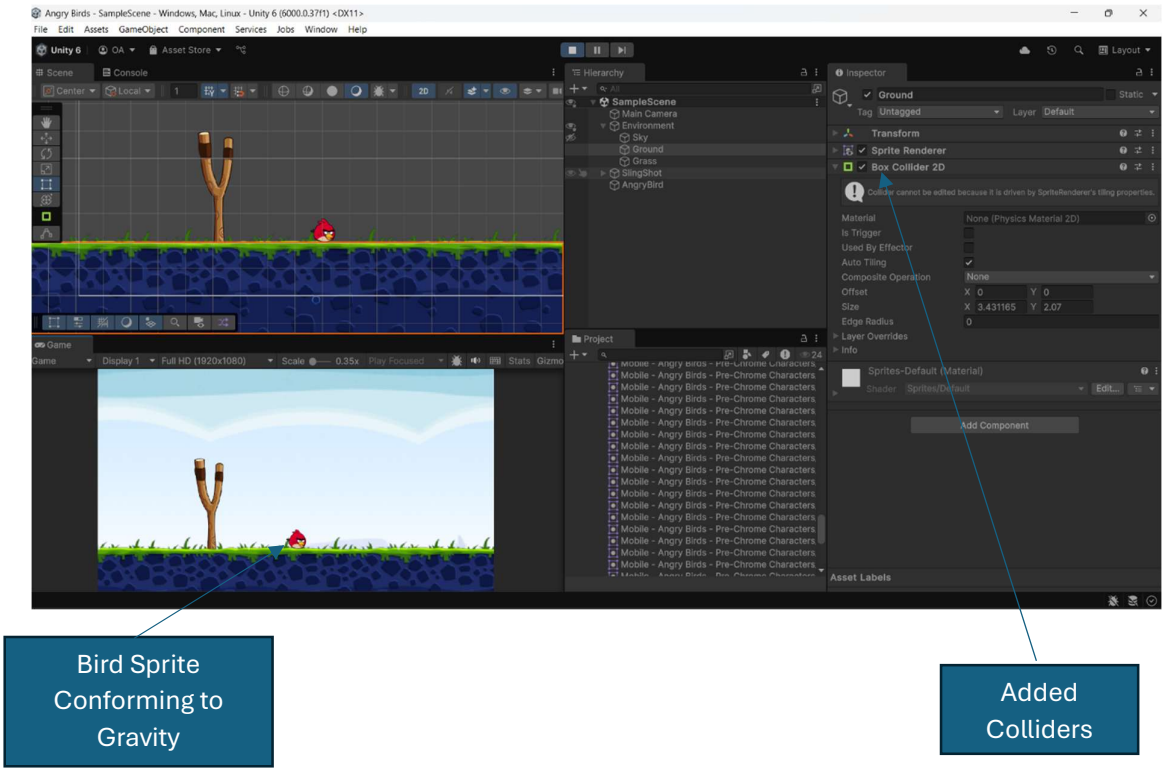
Set MaxDistance Variable

This initial implementation laid the foundation for more advanced slingshot mechanics, such as elasticity effects and the physics-based launching system, which I planned to refine in later stages of development.

Started Scripting the Bird [22/02/2025]

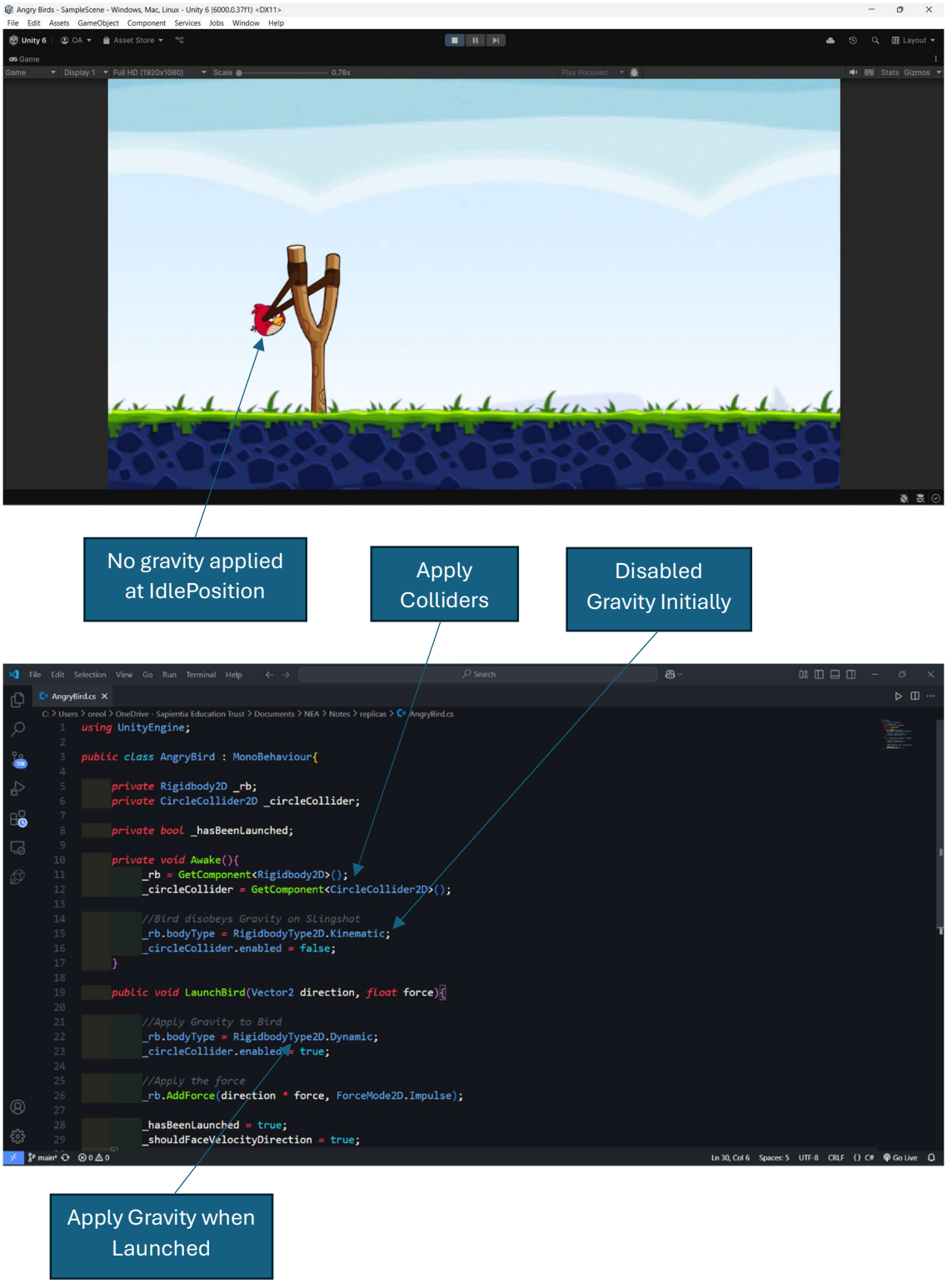
Today, I significantly updated the game by adding another sprite to the environment; the bird character.

To begin, I visually placed the bird sprite within the game world, ensuring it was positioned correctly in relation to the slingshot and the ground. Once the sprite was integrated, I applied 2D vector gravity to simulate realistic falling motion. In addition to gravity, I assigned a collider component to the bird, enabling it to interact with the ground and other objects in the scene. This ensured that the bird would properly respond to collisions, preventing it from passing through obstacles unnaturally.



Testing Colliders and Gravity Effects on Bird Sprite

However, after testing the physics implementation, I realized that gravity should only affect the bird after it has been launched from the slingshot. If gravity were constantly applied, the bird would fall out of place before the player had a chance to aim and shoot. To address this, I modified the script to delay the activation of gravity until the bird is launched. This was achieved by programming a condition that only enables gravity once the launch function is called.



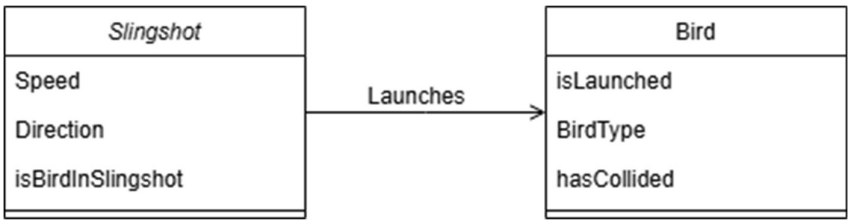

```
using UnityEngine;

public class AngryBird : MonoBehaviour{

    private Rigidbody2D _rb;
    private CircleCollider2D _circleCollider;

    private bool _hasBeenLaunched;

    private void Awake(){
        _rb = GetComponent<Rigidbody2D>();
        _circleCollider = GetComponent<CircleCollider2D>();

        //Bird disobeys Gravity on Slingshot
        _rb.bodyType = RigidbodyType2D.Kinematic;
        _circleCollider.enabled = false;
    }

    public void LaunchBird(Vector2 direction, float force){

        //Apply Gravity to Bird
        _rb.bodyType = RigidbodyType2D.Dynamic;
        _circleCollider.enabled = true;

        //Apply the force
        _rb.AddForce(direction * force, ForceMode2D.Impulse);

        _hasBeenLaunched = true;
        _shouldFaceVelocityDirection = true;
```

Next, I implemented the launch function itself. This function retrieves the direction and speed from the slingshot's pull-back distance and applies an impulse force to the bird upon release.

By using an impulse-based force system, I was able to accurately simulate the slingshot's effect on the bird, allowing it to arc through the air naturally based on the player's input.

With these mechanics in place, the bird is now fully functional as a projectile, responding dynamically to the player's aim and launch strength while maintaining realistic physics interactions within the game world.

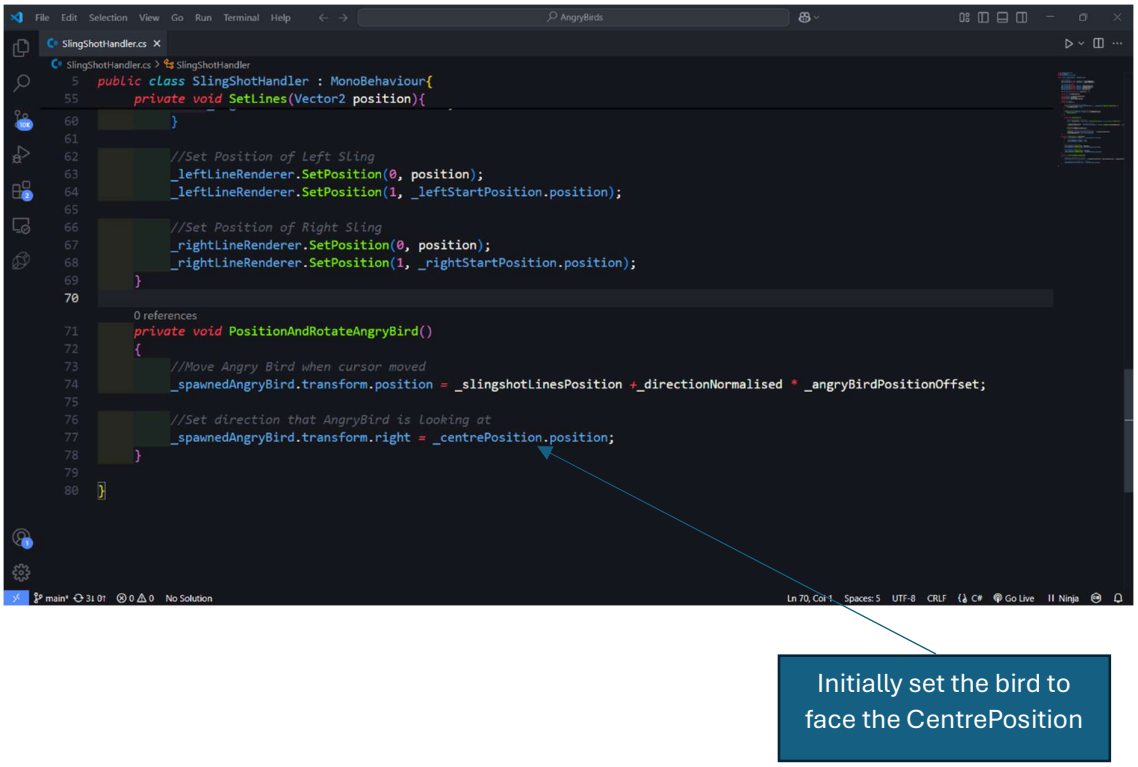
So far, I have developed methods for the Slingshot Object and the Bird Object. As explained in my analysis, dividing these functions into modules has made development way easier and faster. Here is a diagrammatic representation of the Objects and the methods I have developed currently.

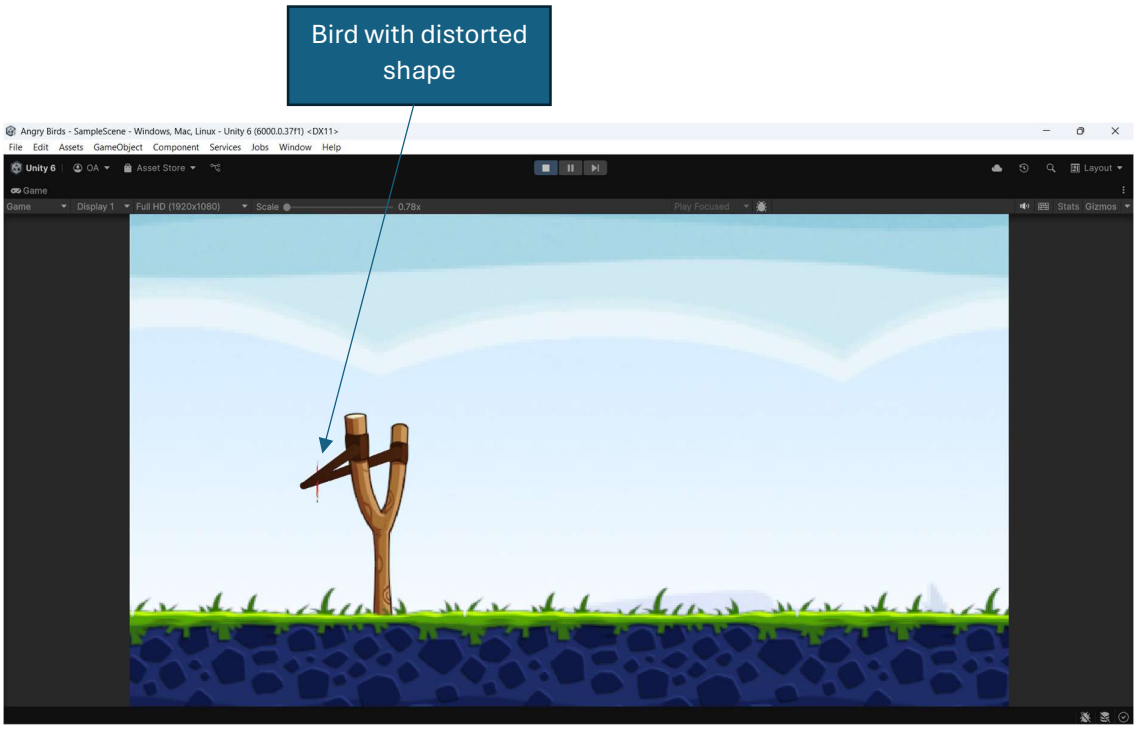


Developing Launch Mechanics[28/02/2025]

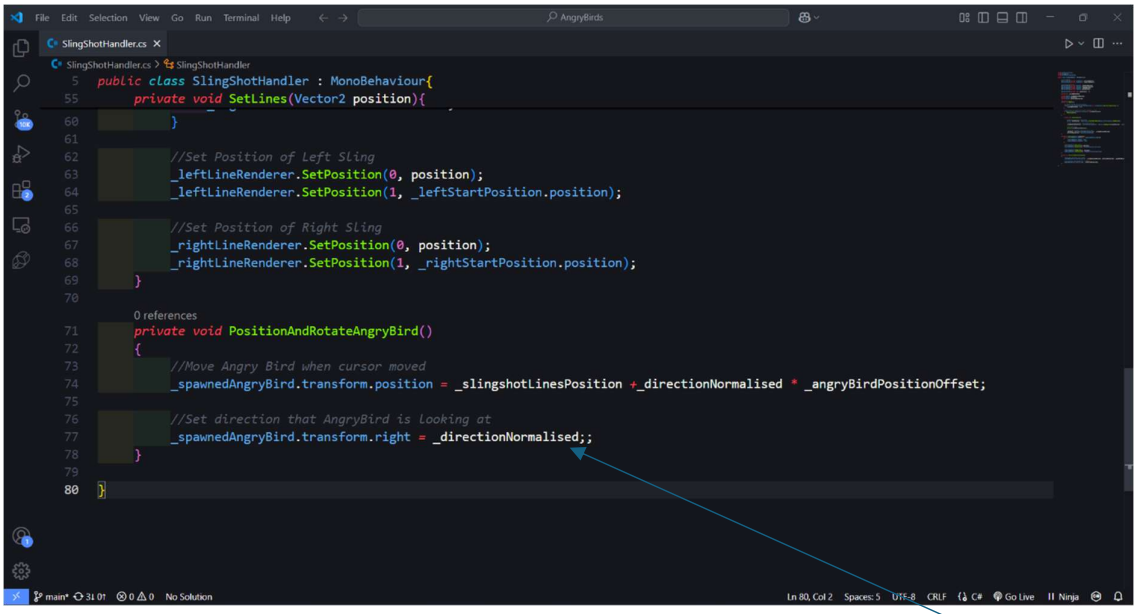
Today, I wanted to focus on refining the bird's launching mechanics. But before implementing the launch itself, I needed to address an issue with the bird's facing direction. My stakeholder pointed out that the bird should always be facing the direction in which it will be launched, both for aesthetic purposes and to provide better visual feedback to the player. This adjustment would improve the overall game experience by making aiming more intuitive.

Initially, I attempted to simply set the bird's rotation to face the centre of the slingshot. However, this caused an unexpected distortion in the bird's shape because I hadn't accounted for its size and position offset. To fix this, I recalculated the facing direction using a properly normalized direction vector that considered the bird's actual dimensions. This direction vector updates dynamically each time the `DrawSlingshot()` method is called, meaning that every time the player pulls back the slingshot, the bird's orientation is adjusted accordingly. With this fix, the bird now correctly faces its launch direction without any visual distortion.





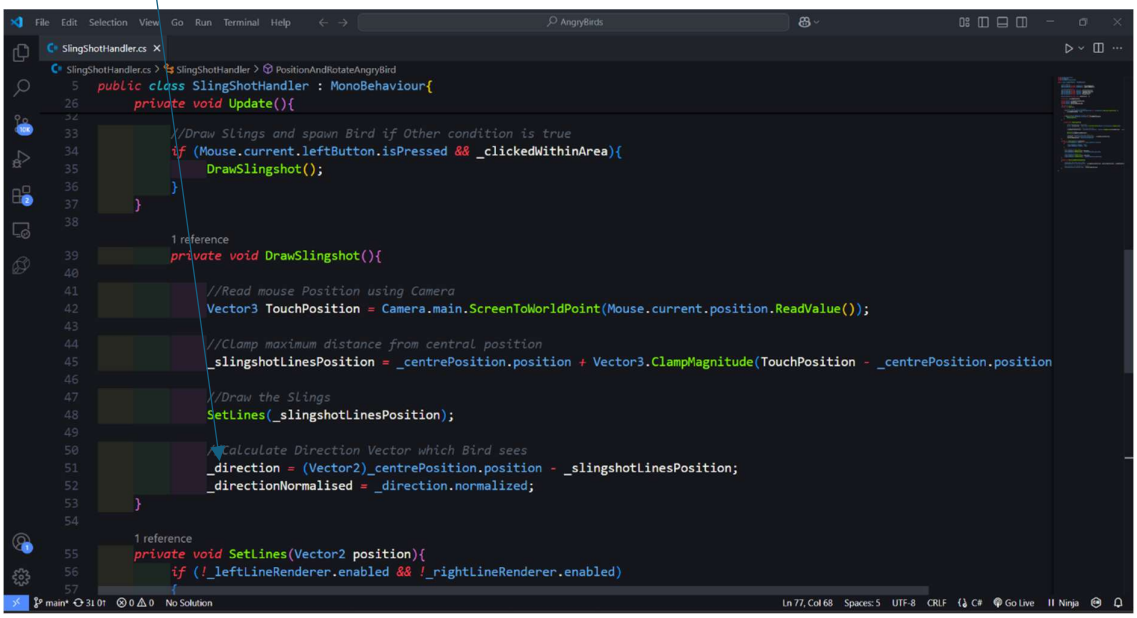
Bird Sprite with a distorted shape after implementing code



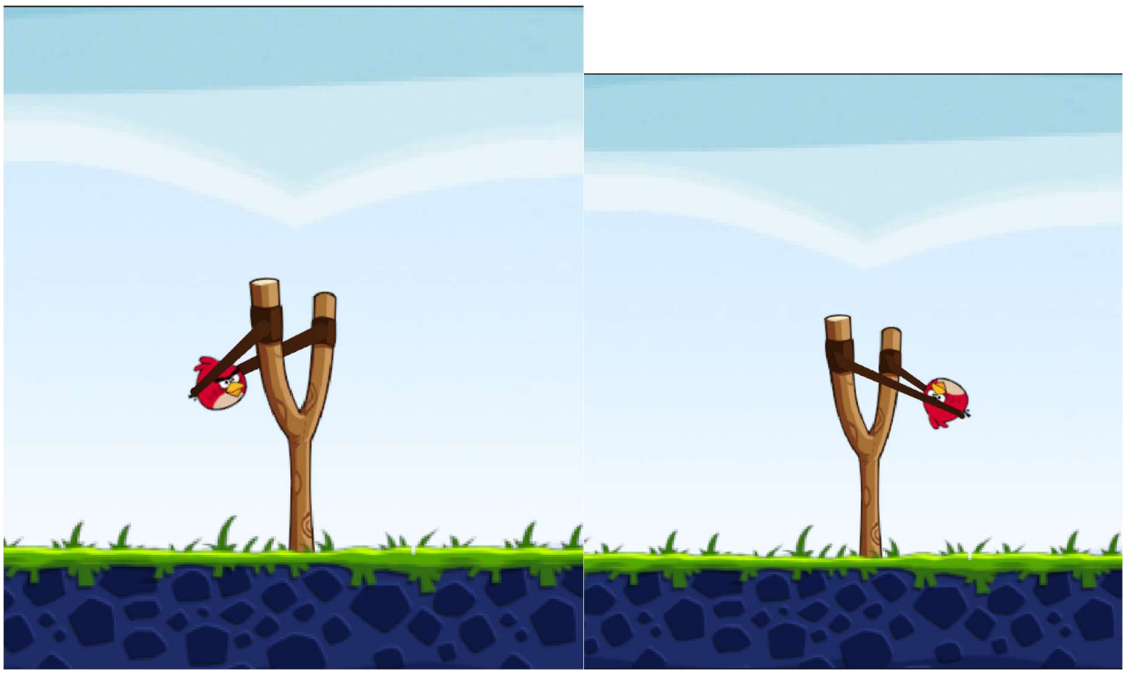
```
public class SlingShotHandler : MonoBehaviour{
    private void SetLines(Vector2 position){
        }

        //Set Position of Left Sling
        _leftLineRenderer.SetPosition(0, position);
        _leftLineRenderer.SetPosition(1, _leftStartPosition.position);

        //Set Position of Right Sling
        _rightLineRenderer.SetPosition(0, position);
        _rightLineRenderer.SetPosition(1, _rightStartPosition.position);
    }

    0 references
    private void PositionAndRotateAngryBird()
    {
        //Move Angry Bird when cursor moved
        _spawnedAngryBird.transform.position = _slingshotLinesPosition +_directionNormalised * _angryBirdPositionOffset;

        //Set direction that AngryBird is Looking at
        _spawnedAngryBird.transform.right = _directionNormalised;;
    }

}
```

Changed direction to a calculated Vector

Vector Calculation



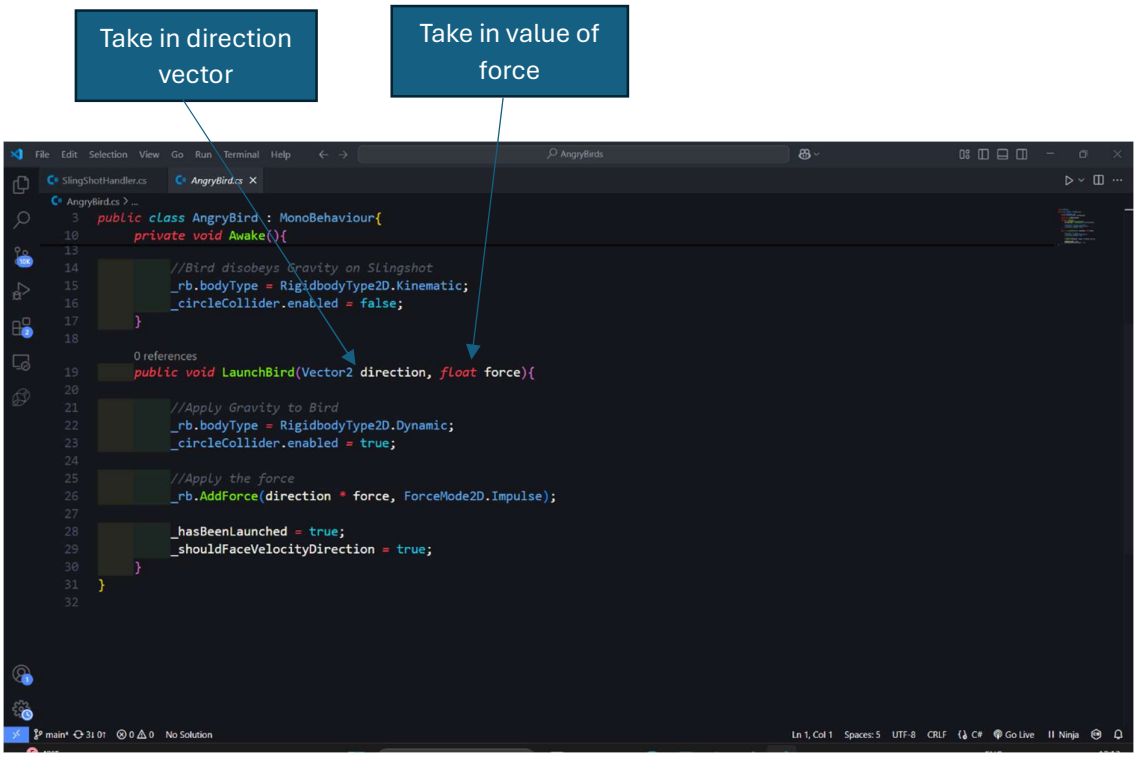
```
public class SlingShotHandler : MonoBehaviour{
    private void Update(){

        //Draw Slings and spawn Bird if Other condition is true
        if (Mouse.current.leftButton.isPressed && _clickedWithinArea){
            DrawSlingshot();
        }
    }

        1 reference
        private void DrawSlingshot(){

            //Read mouse Position using Camera
            Vector3 TouchPosition = Camera.main.ScreenToWorldPoint(Mouse.current.position.ReadValue());

            //Clamp maximum distance from central position
            _slingshotLinesPosition = _centrePosition.position + Vector3.ClampMagnitude(TouchPosition - _centrePosition.position

            //Draw the Slings
            SetLines(_slingshotLinesPosition);

            //Calculate Direction Vector which Bird sees
            _direction = (Vector2)_centrePosition.position - _slingshotLinesPosition;
            _directionNormalised = _direction.normalized;
    }

    1 reference
    private void SetLines(Vector2 position){
        if (!_leftLineRenderer.enabled && !_rightLineRenderer.enabled)
```



Bird Now constantly rotates to the centre of the Slingshot

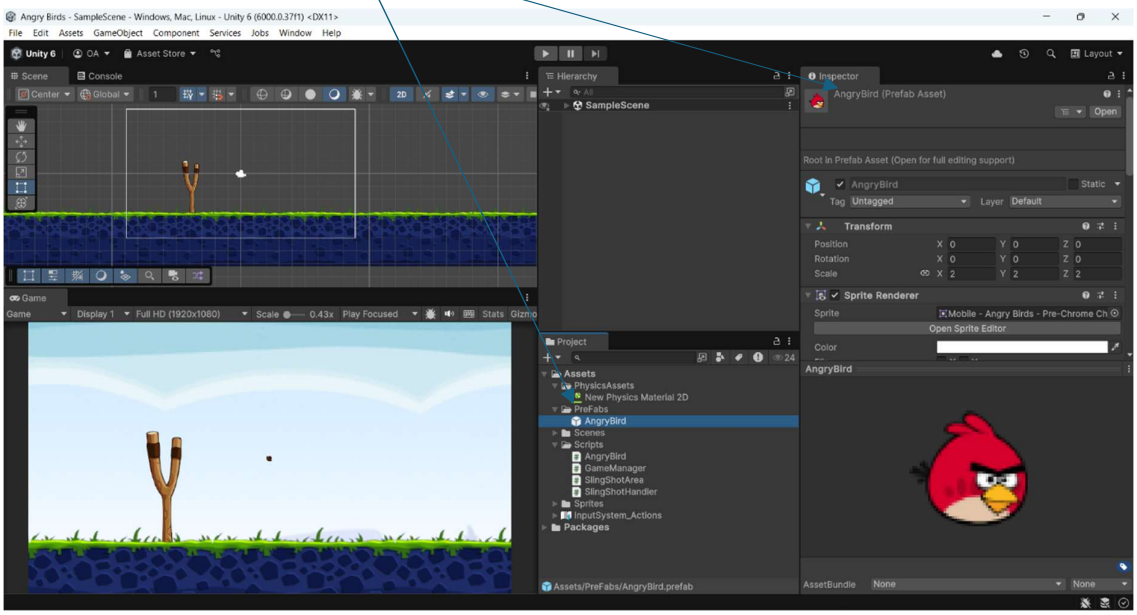
Developing Launch Mechanics[01/03/2025]

Once that issue was resolved, I moved on to implementing the actual launch mechanics using the `launchBird()` function I had previously scripted. The function takes in the calculated direction and the distance from the bird's idle position to determine the appropriate force to apply. To simplify initial testing, I temporarily set the slingshot's force as a hard-coded constant. This allowed me to focus on fine-tuning the launch physics without worrying about variable force calculations.



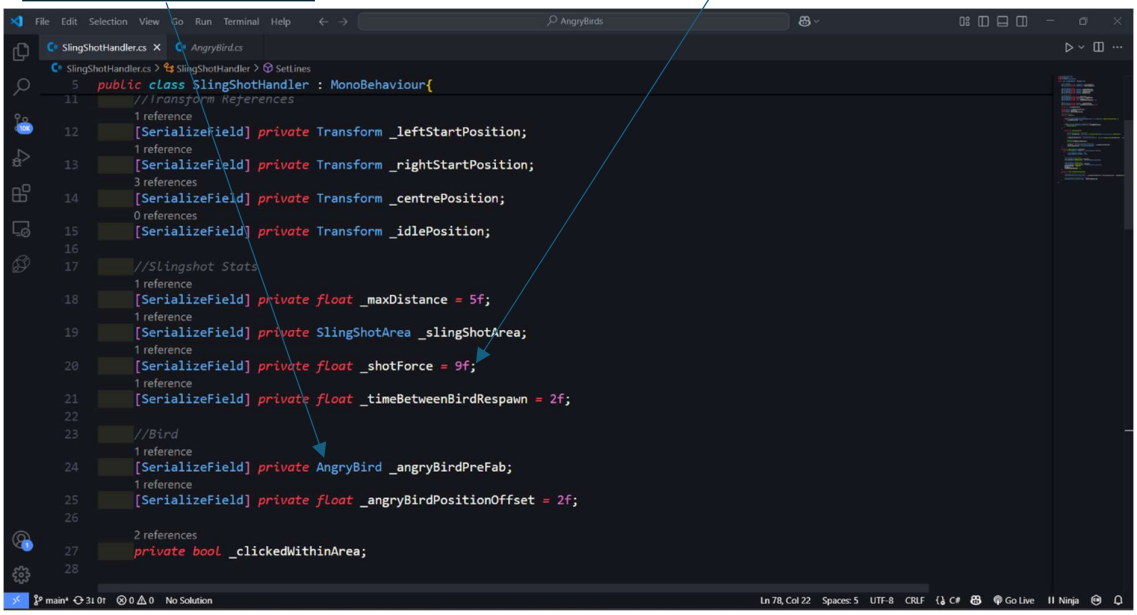
This was also the first time that two objects in the game directly interacted with each other, meaning I needed to import the `AngryBird` object into the slingshot module to ensure proper reference handling. Recognizing that multiple birds will need to be created throughout the game, I also decided to convert the `AngryBird` object into a Prefab. This allows for the efficient reproduction of birds whenever a new one is required, ensuring that the slingshot can seamlessly cycle through multiple birds in a single level.

Made Bird a Pre-Fabricated Asset



Created a data type for AngryBird

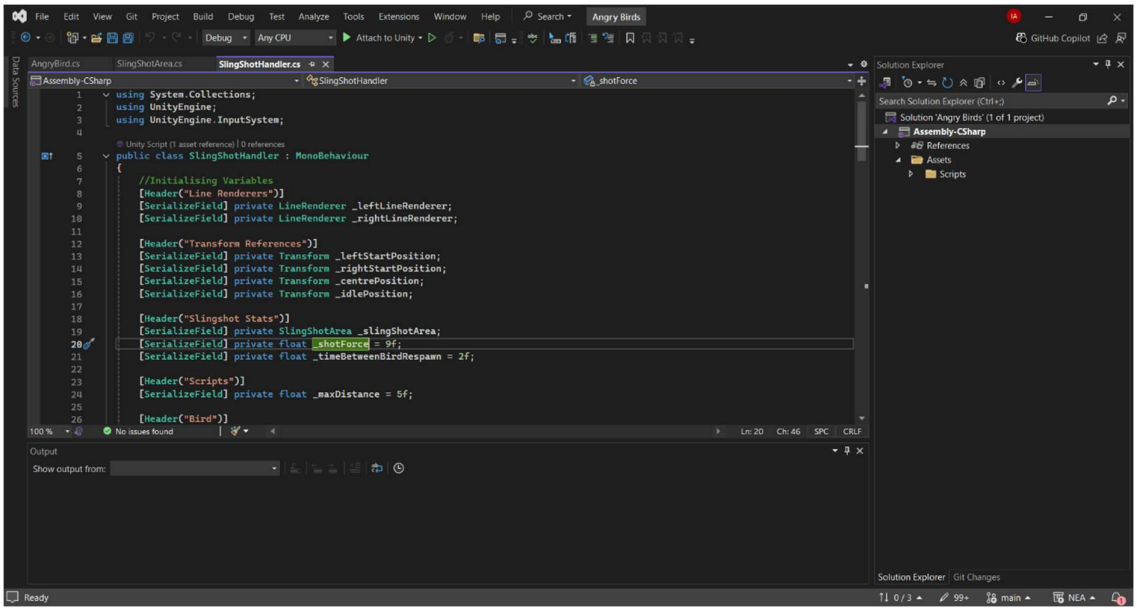
Temporarily hard coded shot force



With these improvements in place, the bird now correctly orients itself, interacts with the slingshot, and launches with force applied in the right direction.

Developing Launch Mechanics[05/03/2025]

First, I decided to switch my development environment from VS Code to Visual Studio. Although Visual Studio is a heavier program, it is more specialised for languages like C#, providing better debugging tools and a more streamlined workflow for Unity development. I found that running tests and debugging my code was significantly easier in Visual Studio compared to VS Code, making it the better choice for this project.

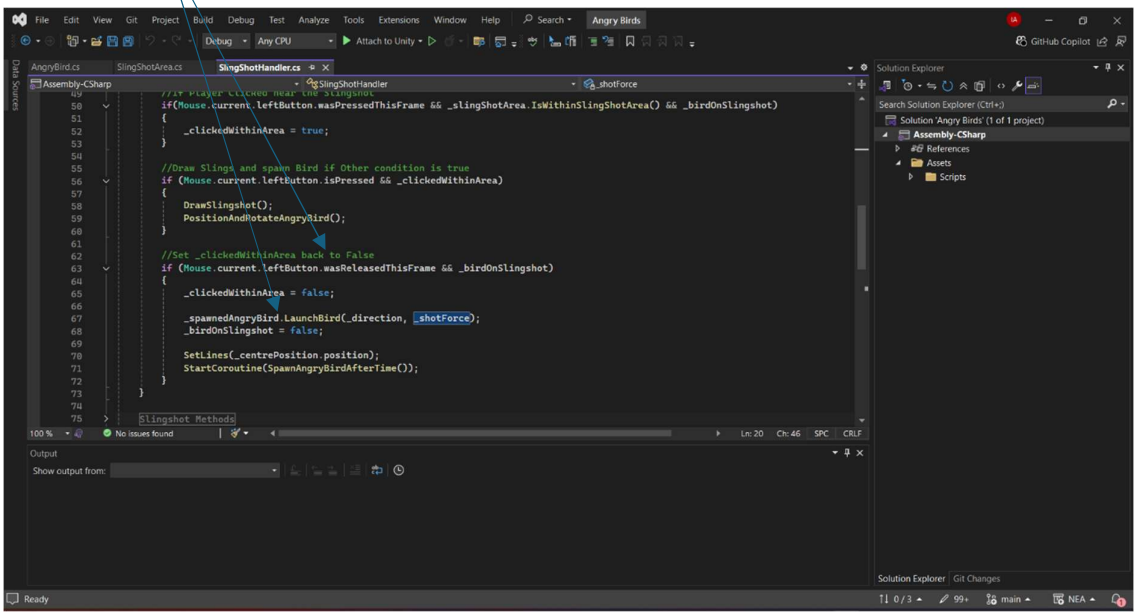


Visual Studio (My new IDE)

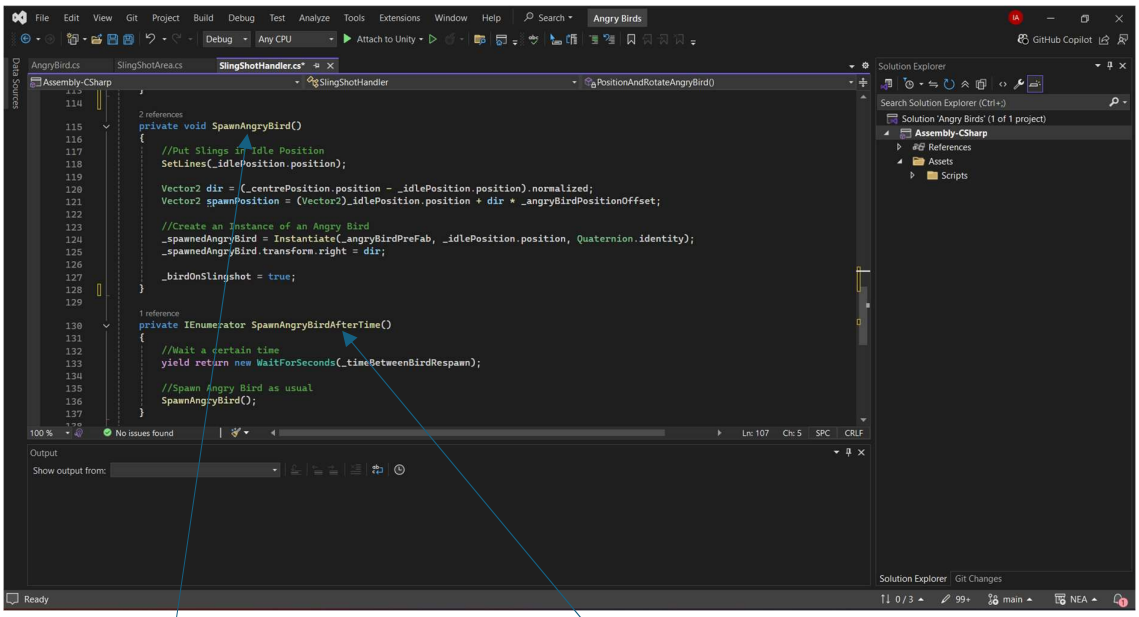
With my development environment optimized, I moved on to finalizing the launch mechanics for the bird. I wrote the script that ensures the bird launches precisely in the direction it is facing, aligning the motion with the player's aim. This was a crucial step in making the launch feel accurate and responsive. Now, when the player releases the slingshot, the bird is propelled forward based on its orientation.

Additionally, I implemented a respawn system for the birds. Once a bird is launched, the slingshot automatically spawns a new bird after a 2-second delay, allowing continuous gameplay without interruptions. This ensures that players can smoothly transition between shots without having to manually reset the slingshot.

Launch Bird when player is no longer clicking



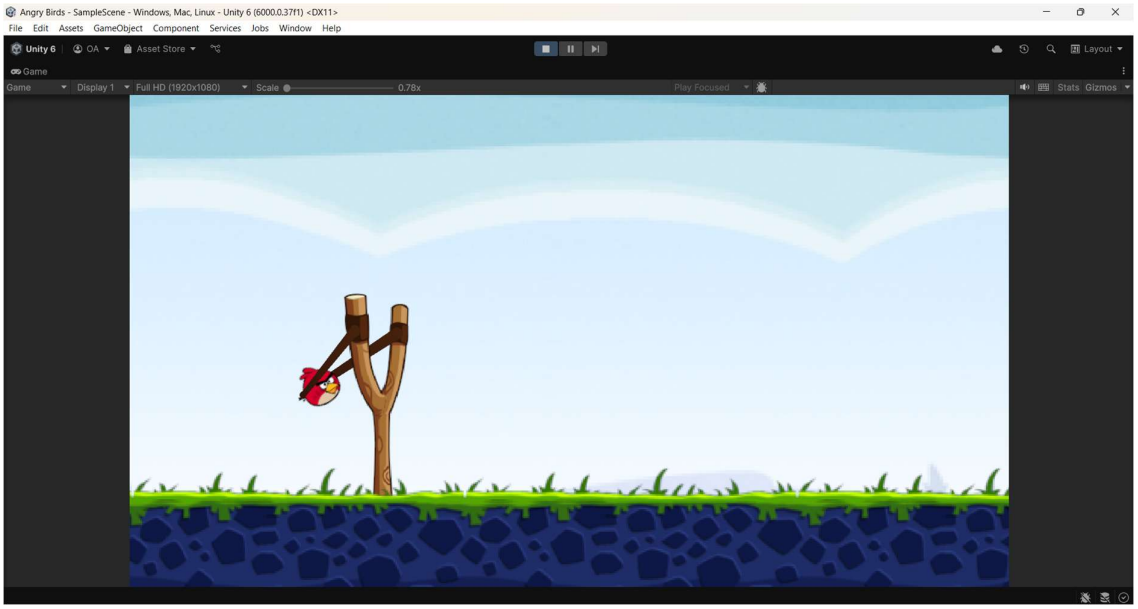
```
//If Player Clicked near the Slingshot
if(Mouse.current.leftButton.wasPressedThisFrame && _slingShotArea.IsWithinSlingShotArea() && _birdOnSlingshot)
{
    _clickedWithinArea = true;
}

//Draw Slings and spawn Bird if Other condition is true
if (Mouse.current.leftButton.isPressed && _clickedWithinArea)
{
    DrawSlingshot();
    PositionAndRotateAngryBird();
}

//Set _clickedWithinArea back to False
if (Mouse.current.leftButton.wasReleasedThisFrame && _birdOnSlingshot)
{
    _clickedWithinArea = false;

    _spawnedAngryBird.LaunchBird(_direction, _shotForce);
    _birdOnSlingshot = false;

    SetLines(_centrePosition.position);
    StartCoroutine(SpawnAngryBirdAfterTime());
}
}

Slingshot Methods
```



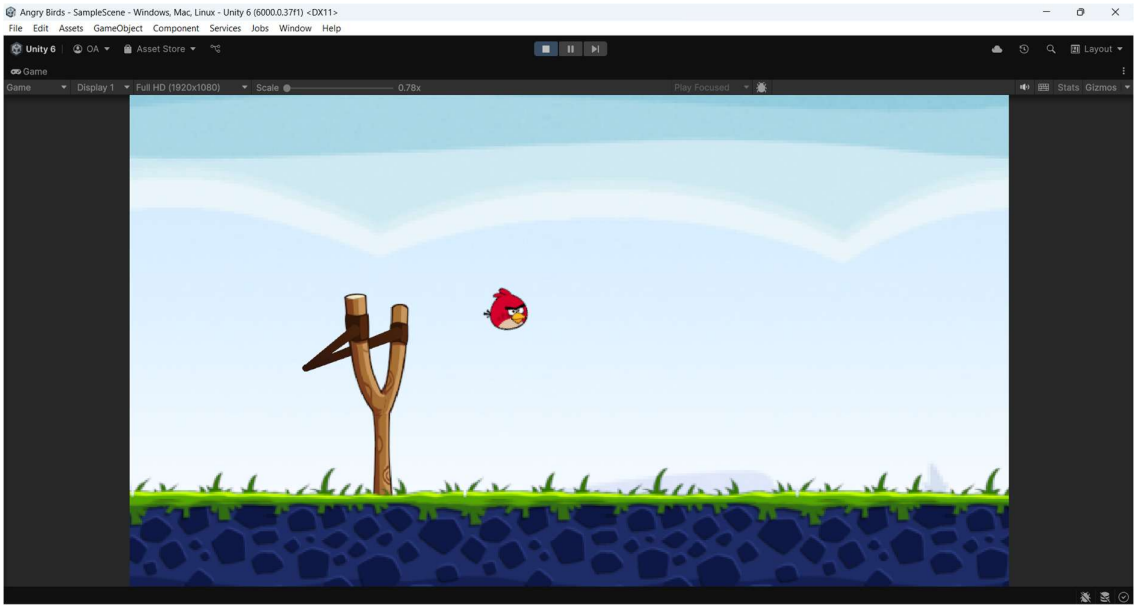
```
2 references
private void SpawnAngryBird()
{
    //Put Slings in Idle Position
    SetLines(_idlePosition.position);

    Vector2 dir = (_centrePosition.position - _idlePosition.position).normalized;
    Vector2 spawnPosition = (Vector2)_idlePosition.position + dir * _angryBirdPositionOffset;

    //Create an Instance of an Angry Bird
    _spawnedAngryBird = Instantiate(_angryBirdPreFab, _idlePosition.position, Quaternion.identity);
    _spawnedAngryBird.transform.right = dir;

    _birdOnSlingshot = true;
}

1 reference
private IEnumerator SpawnAngryBirdAfterTime()
{
    //Wait a certain time
    yield return new WaitForSeconds(_timeBetweenBirdRespawn);

    //Spawn Angry Bird as usual
    SpawnAngryBird();
}
```

Spawn AngryBird function

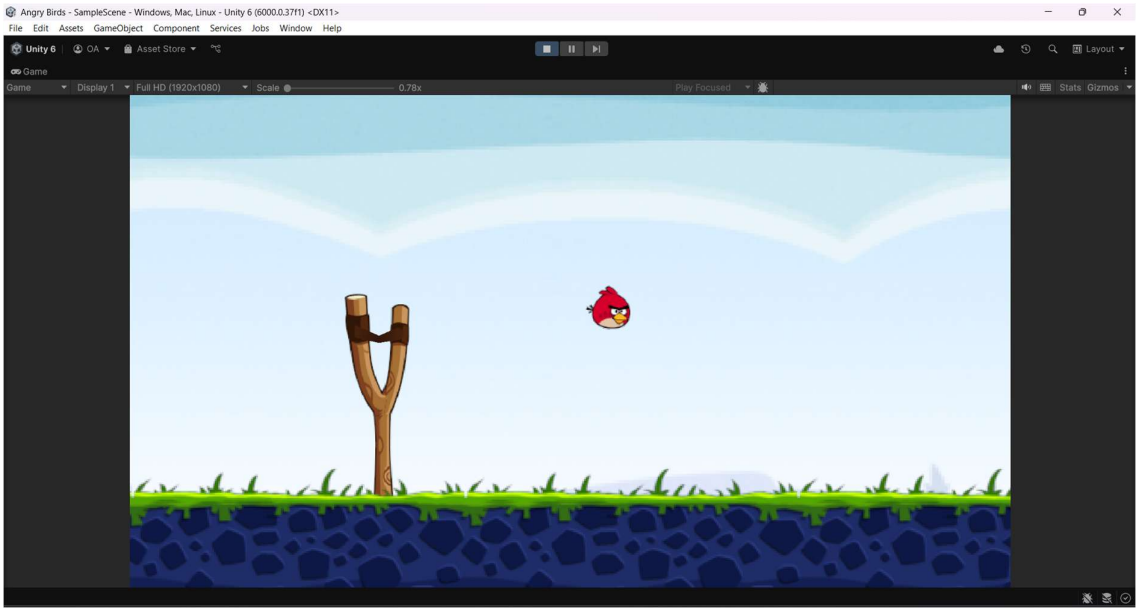
Call the SpawnAngryBird() function after a set amount of time



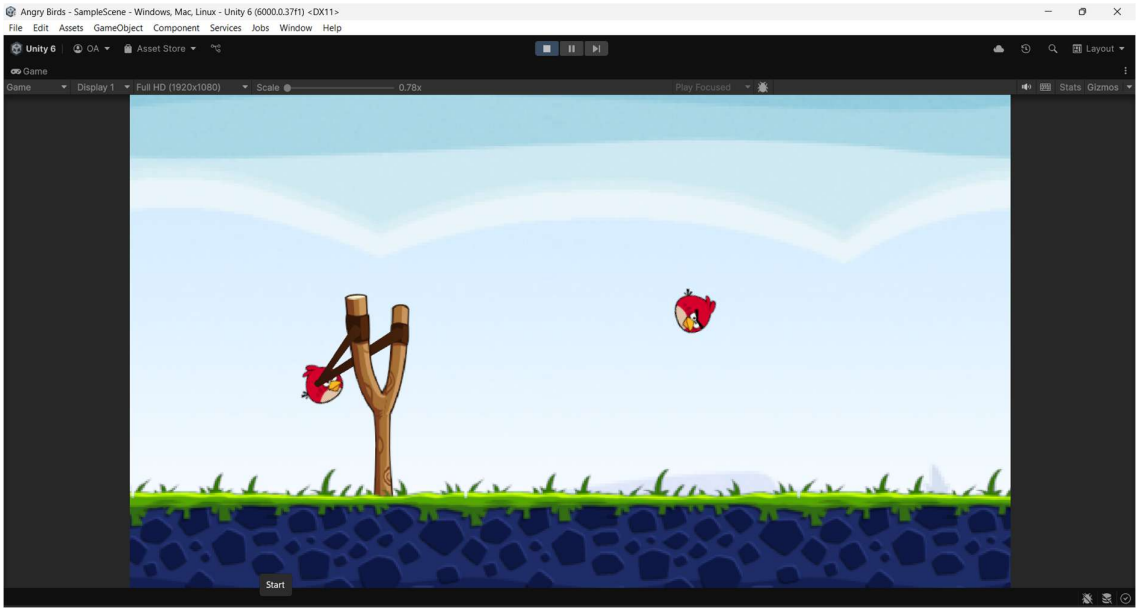
Testing Launch Mechanics (1)



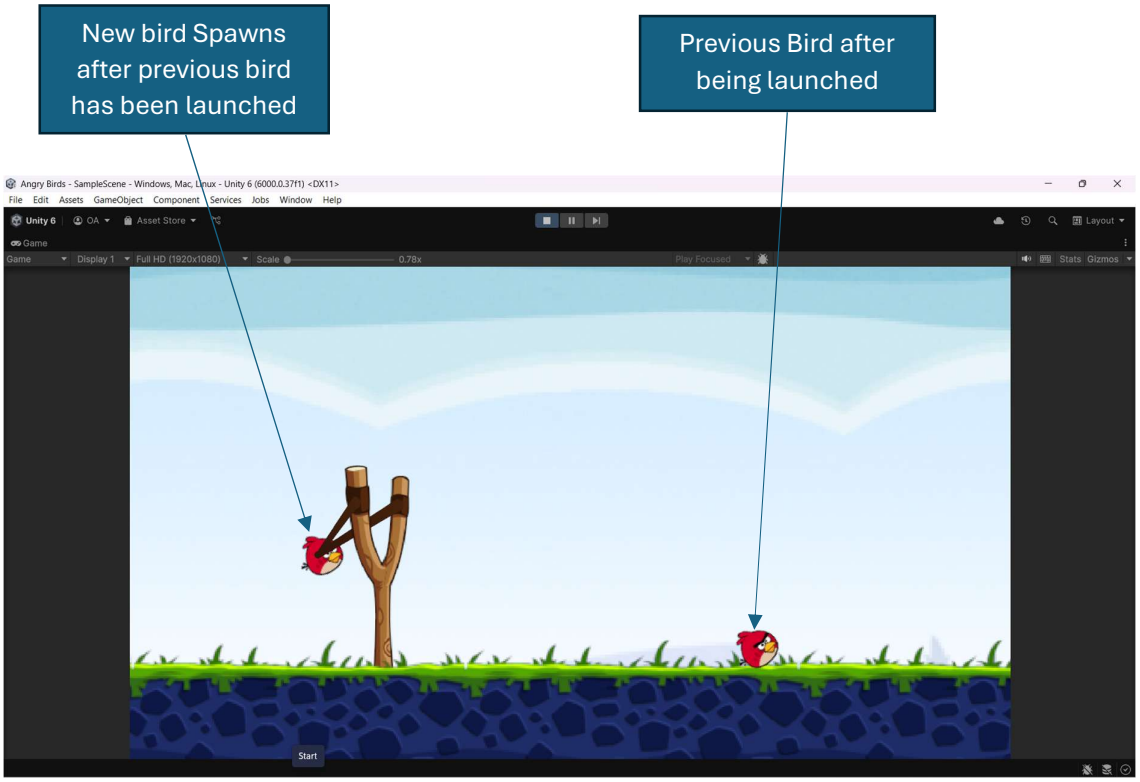
Testing Launch Mechanics (2)



Testing Launch Mechanics (4)



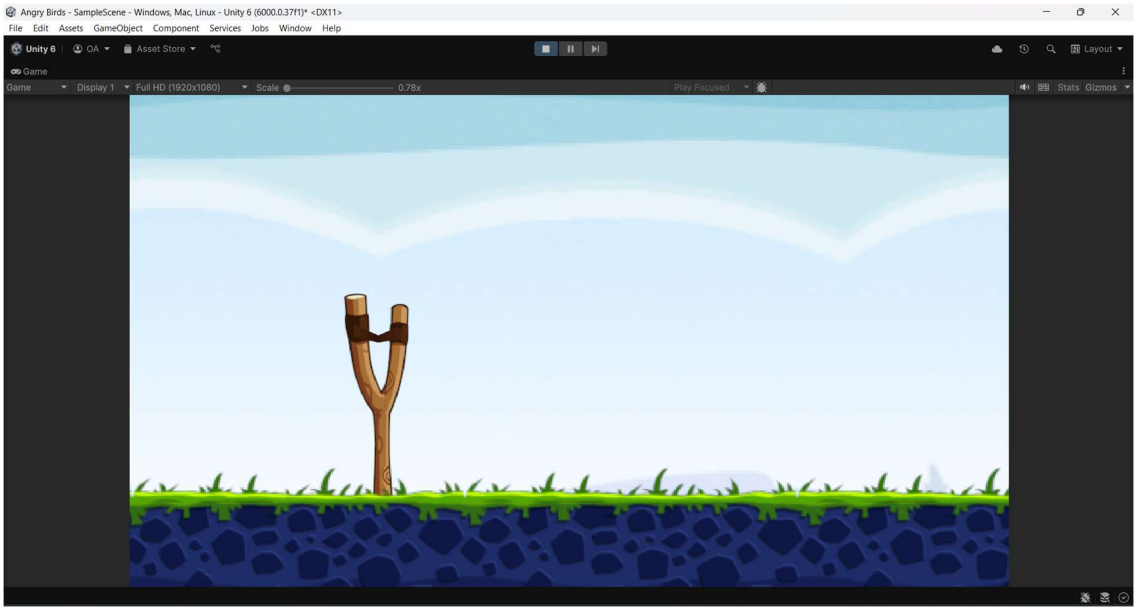
Testing Launch Mechanics (5)



Testing Launch Mechanics (6)

Limiting Number of Birds Shot[08/03/2025 - 10/03/2025]

Today, I implemented a Game Manager script to control the number of birds that can be regenerated during gameplay. This script is responsible for tracking the total number of shots taken by the player and determining whether the number of shots has exceeded the maximum allowed for the level. By enforcing this limitation, the game becomes more challenging and competitive, aligning with my stakeholder's preference for a more strategic experience.



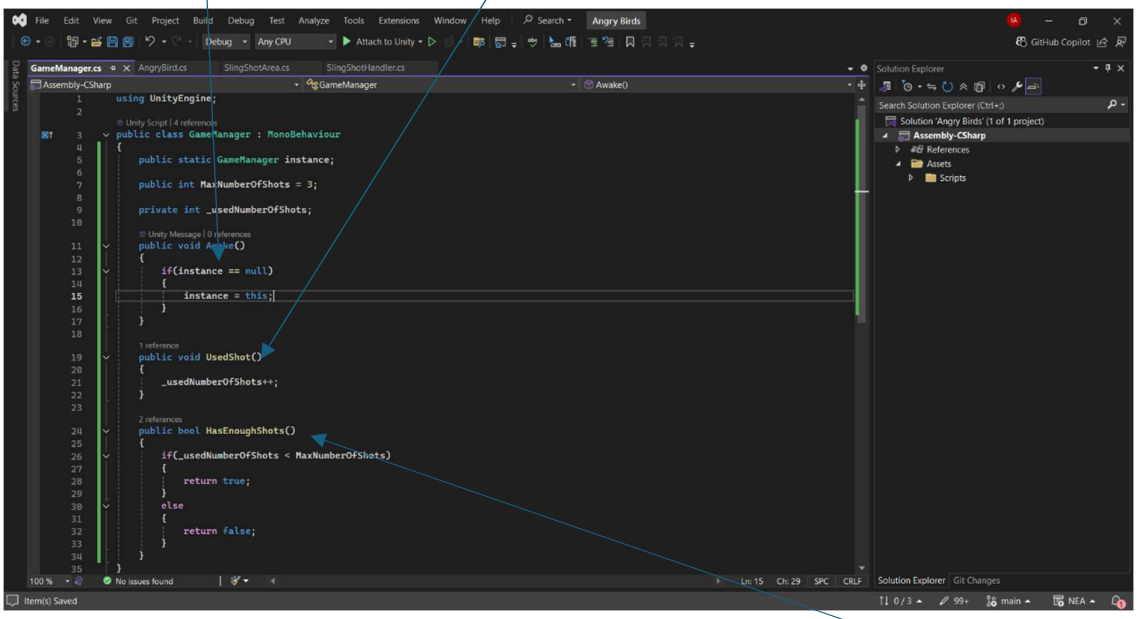
No more Birds available after they have all been used

Instead of following my previous approach of manually importing scripts via serialized variables, I opted to implement the Singleton pattern for the Game Manager. Unlike other objects, such as the Angry Birds, which require multiple instances, the Game Manager is a global entity that should have only one instance throughout the entire game. By using a Singleton, I ensured that the Game Manager remains accessible without requiring manual imports in each script.

Within the Slingshot script, I integrated the Game Manager using its Singleton instance. This allowed me to directly call its methods to track the number of birds launched and continuously check whether the maximum shot limit has been reached. If the player exhausts all available shots before eliminating the pigs, the Game Manager will trigger the Game Over state.

Singleton Method

Method to count the number of Shots

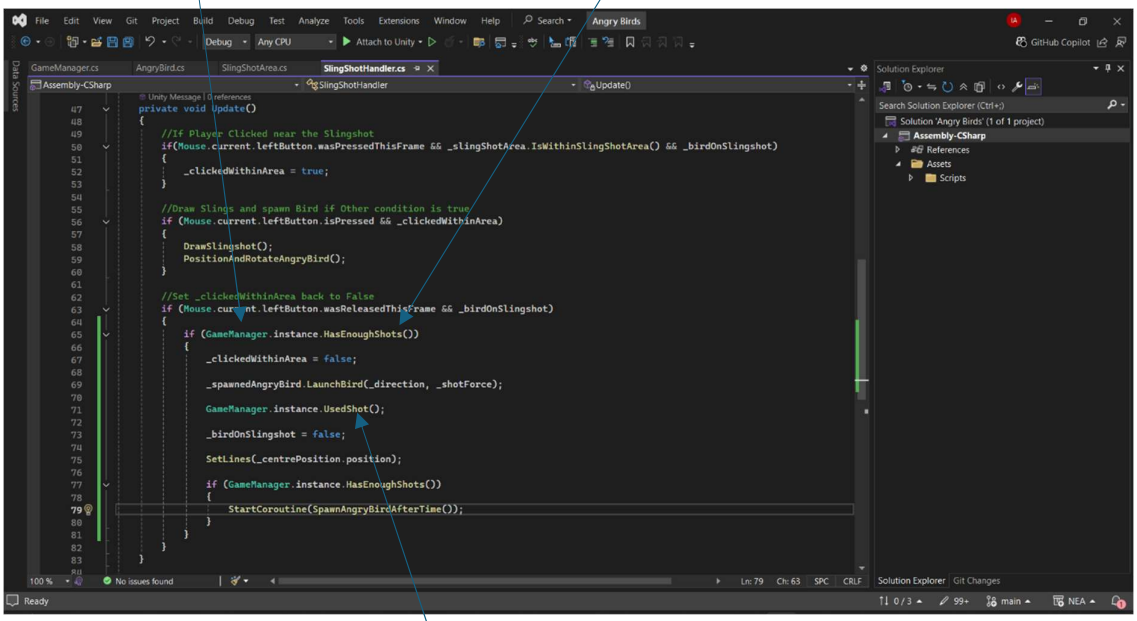


```
using UnityEngine;

public class GameManager : MonoBehaviour
{
    public static GameManager instance;

    public int MaxNumberOfShots = 3;

    private int _usedNumberOfShots;

    public void Awake()
    {
        if(instance == null)
        {
            instance = this;
        }
    }

    public void UsedShot()
    {
        _usedNumberOfShots++;
    }

    public bool HasEnoughShots()
    {
        if(_usedNumberOfShots < MaxNumberOfShots)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Method to check if a player has enough Shots

Calling GameManager methods in the Slingshot script

Checking if the Player has enough shots before launching



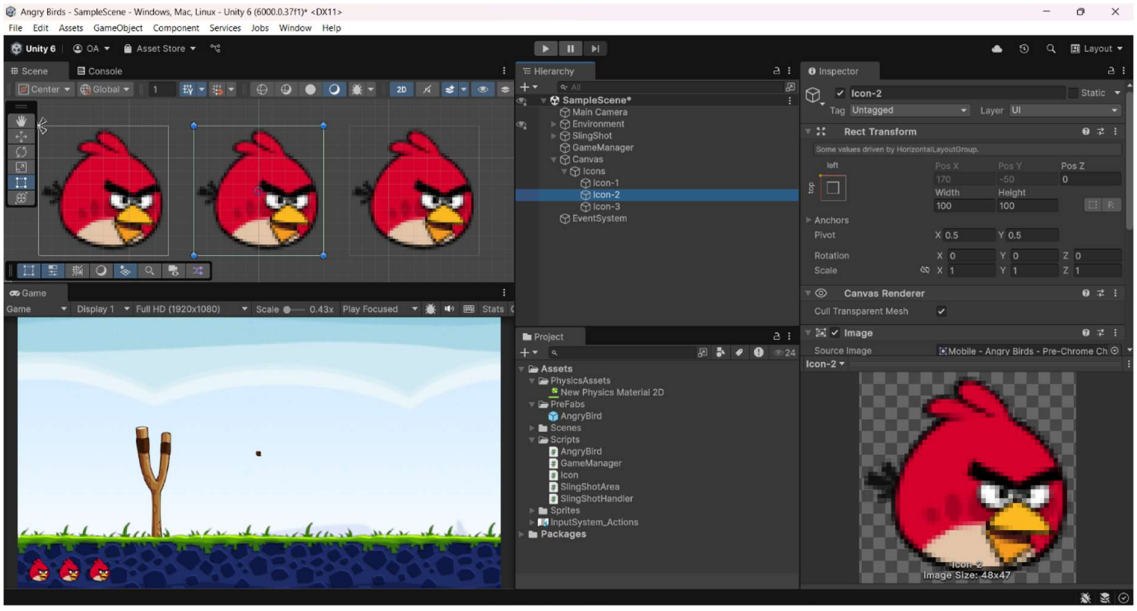
```
private void Update()
{
    //If Player Clicked near the Slingshot
    if(Mouse.current.leftButton.wasPressedThisFrame && _slingShotArea.IsWithinSlingShotArea() && _birdOnSlingshot)
    {
        _clickedWithinArea = true;
    }

    //Draw Slings and spawn Bird if Other condition is true
    if (Mouse.current.leftButton.isPressed && _clickedWithinArea)
    {
        DrawSlingshot();
        PositionAndRotateAngryBird();
    }

    //Set _clickedWithinArea back to False
    if (Mouse.current.leftButton.wasReleasedThisFrame && _birdOnSlingshot)
    {
        if (GameManager.instance.HasEnoughShots())
        {
            _clickedWithinArea = false;

            _spawnedAngryBird.LaunchBird(_direction, _shotForce);

            GameManager.instance.UsedShot();

            _birdOnSlingshot = false;

            SetLines(_centrePosition.position);

            if (GameManager.instance.HasEnoughShots())
            {
                StartCoroutine(SpawnAngryBirdAfterTime());
            }
        }
    }
}
```

Using Shot when launched

With this implementation, the game now has a structured and efficient way to manage bird regeneration, enforce shot limits, and enhance the overall difficulty curve.

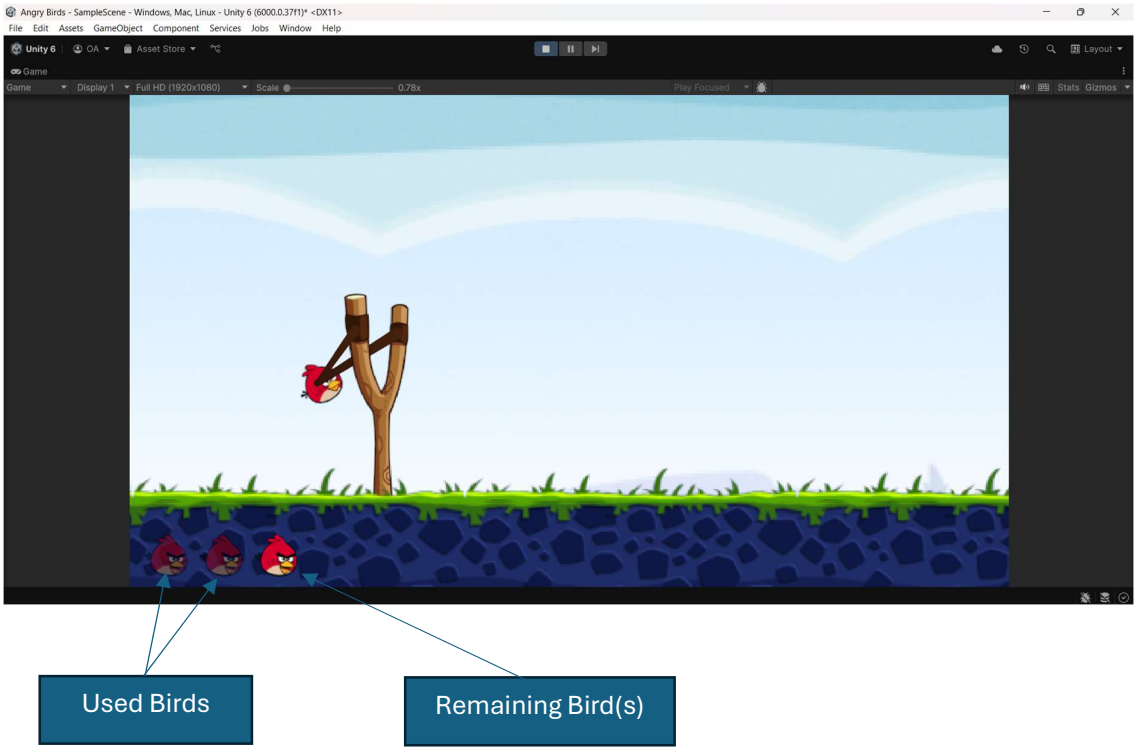
Displaying Number of Birds Shot[12/03/2025]

After implementing the system to limit the number of birds, I realized it would only be fair to provide the player with a visual indicator of how many birds they have left. To achieve this, I designed an icon-based system to represent the remaining birds in play.



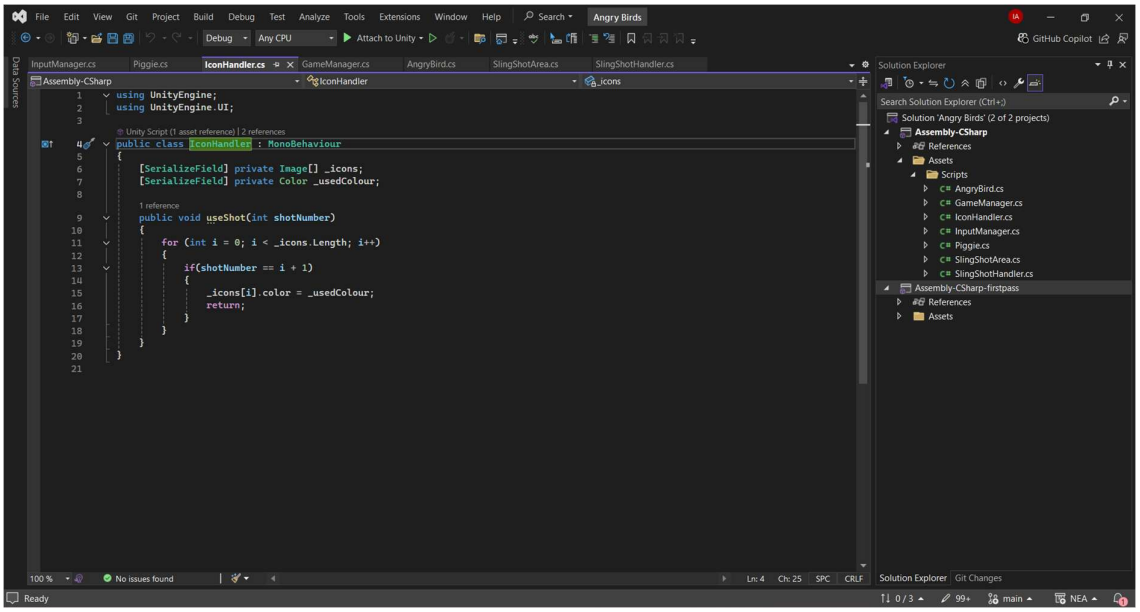
Designing an Icon

During the analysis phase, my stakeholder mentioned a preference for a card-based system similar to Angry Birds 2 rather than the classic Angry Birds approach, where birds are physically queued in the slingshot. To align with this request, I opted for a Bird Icon System, which visually displays the number of available birds in a way that mimics the card selection style.



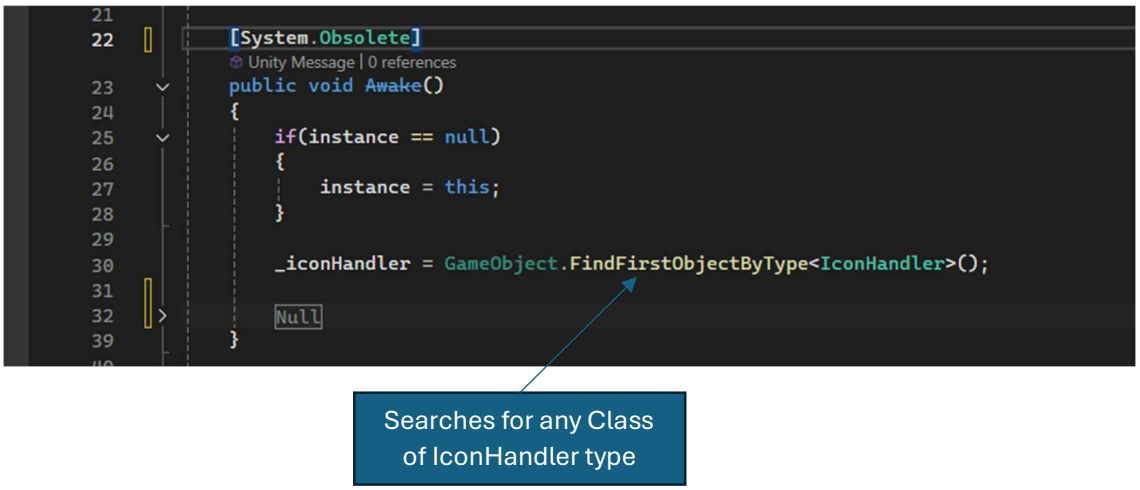
The system consists of a row of bird icons positioned at the bottom of the screen. Each icon represents a bird that can be launched. When the player uses a bird, the corresponding icon dims to indicate that it is no longer available. This provides real-time feedback on bird availability while keeping the interface clean and intuitive.

To achieve this functionality, I created an IconHandler script that manages the appearance of these icons. The script's main task is to detect when a bird has been launched and adjust the corresponding icon's colour to appear dimmed. This ensures that players can easily track how many birds they have left at a glance.

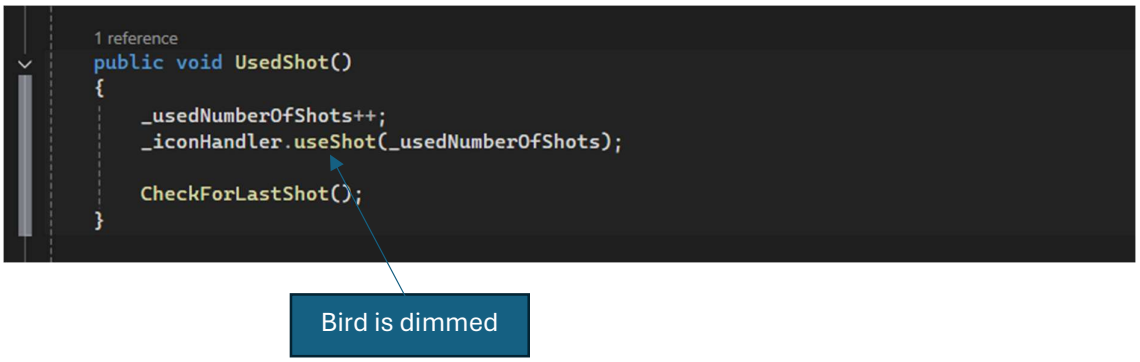


IconHandler Script

To integrate this system with the rest of the game, I referenced the IconHandler script within the Game Manager. Instead of manually assigning the reference, I used a search method to locate the script dynamically. This approach keeps the code more efficient and flexible, allowing it to work regardless of how the objects are structured in the game hierarchy.



Finally, I implemented a function within IconHandler that is triggered whenever the Shot Number increases. Each time a bird is launched, this function updates the icon display, dimming the appropriate icon to reflect the bird's usage. This seamless interaction between the Game Manager and IconHandler ensures that the UI remains responsive and accurate throughout gameplay.



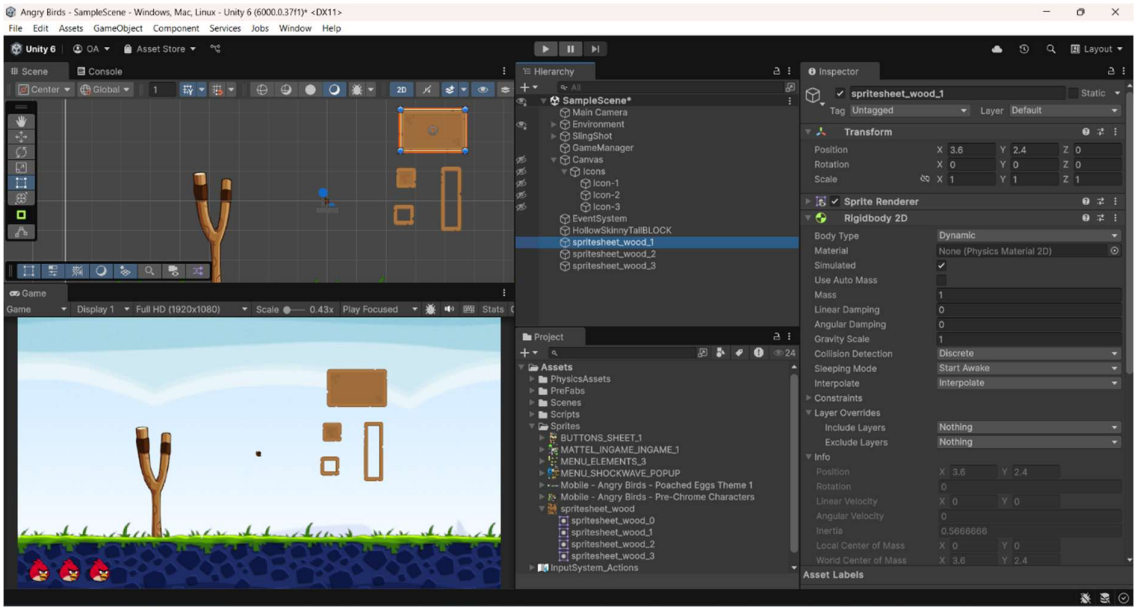
With this implementation, players can easily track their remaining birds, making the game more strategic and user-friendly while also meeting my stakeholder's design preferences.

Designing the Blocks [15/03/2025]

To create a functional and engaging level, I first installed the block sprites that I believed would best fit the game. I wanted to ensure that the visual style closely resembled the classic game, as preferred by my stakeholder, while still incorporating some differences to make it unique. To achieve this, I carefully selected sprite resources from Spriters Resource.

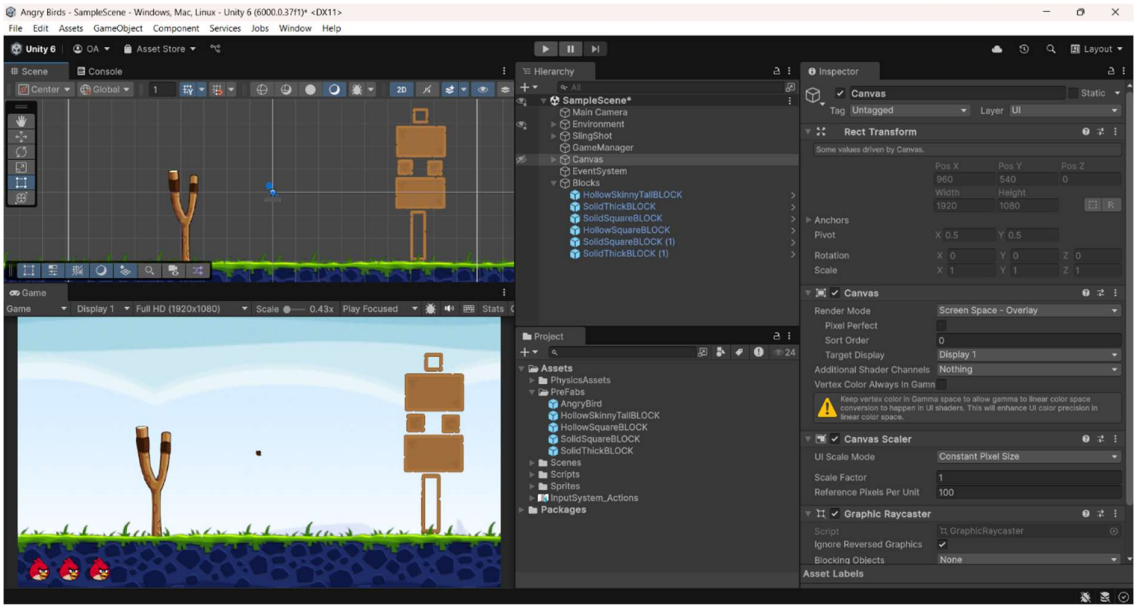
While choosing the sprites, I focused on maintaining a consistent art style that would blend well with the overall aesthetic of the game. The blocks needed to be visually distinct, easily recognizable, and stand out against the background to ensure clear gameplay visibility. I experimented with different colours, shapes, and textures to find the best fit, ensuring that they not only looked good but also complemented the gameplay mechanics.

By selecting these specific sprites, I was able to strike a balance between nostalgic familiarity and fresh design elements, making the game appealing to both fans of the original and new players alike.



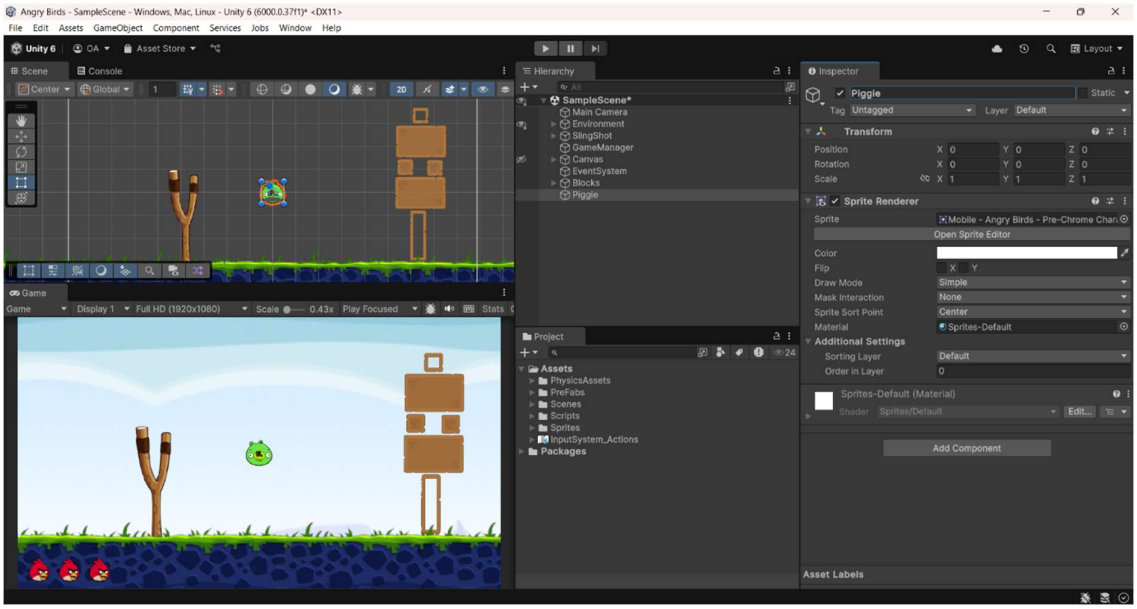
The Blocks I chose

Once the block sprites were completed, I proceeded to build a simple structure using these blocks. The structure serves as an obstacle for the player to overcome, requiring strategic aiming and precise shots to bring it down effectively. I arranged the blocks in a way that would create a balanced level—challenging enough to be engaging but not so difficult that it becomes frustrating.



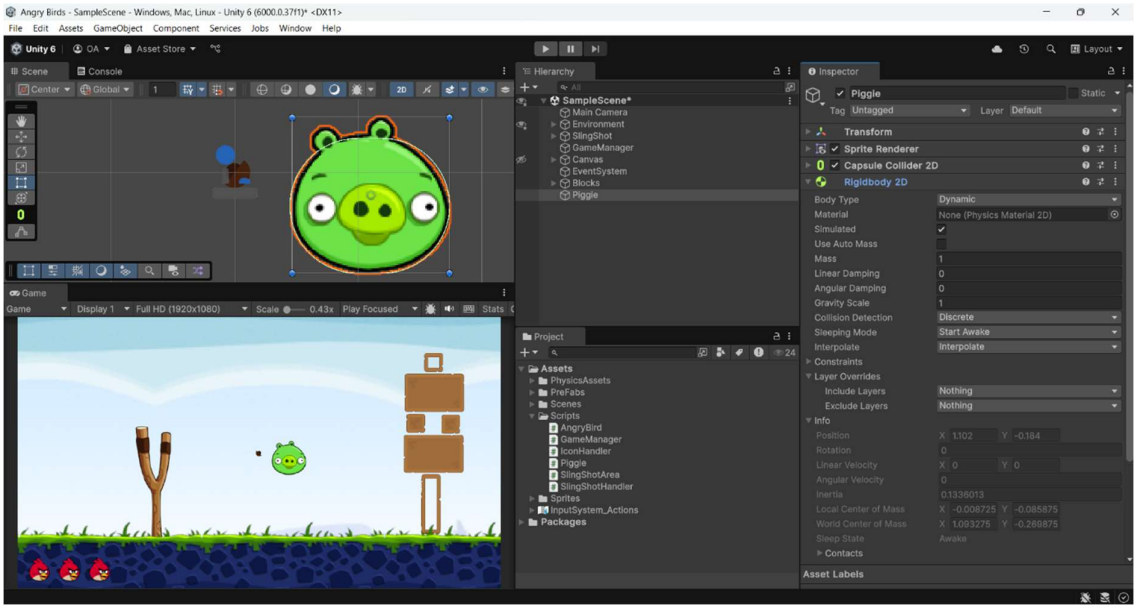
Adding the Pig [16/03/2025]

After finalizing the block sprites, I moved on to designing the sprite for the pig. Since the rest of the game had been developed in a style reminiscent of the classic version, I decided to keep the pig's design very similar to the original as well. This maintained visual consistency and aligned with my stakeholder's preference for a game that felt familiar while still being unique in certain aspects.



Adding the Pig Sprite

Once the sprite was ready, I needed to ensure that the pig could properly interact with the game environment. To achieve this, I added colliders to the pig. These colliders allow the game to detect when the pig makes contact with other objects, such as the bird, blocks, or the ground. By implementing this, I ensured that my code would be notified whenever a collision occurs, making it possible to track impacts and apply damage accordingly. This step was essential for creating a functional damage system and making sure that the pig responded realistically to hits.



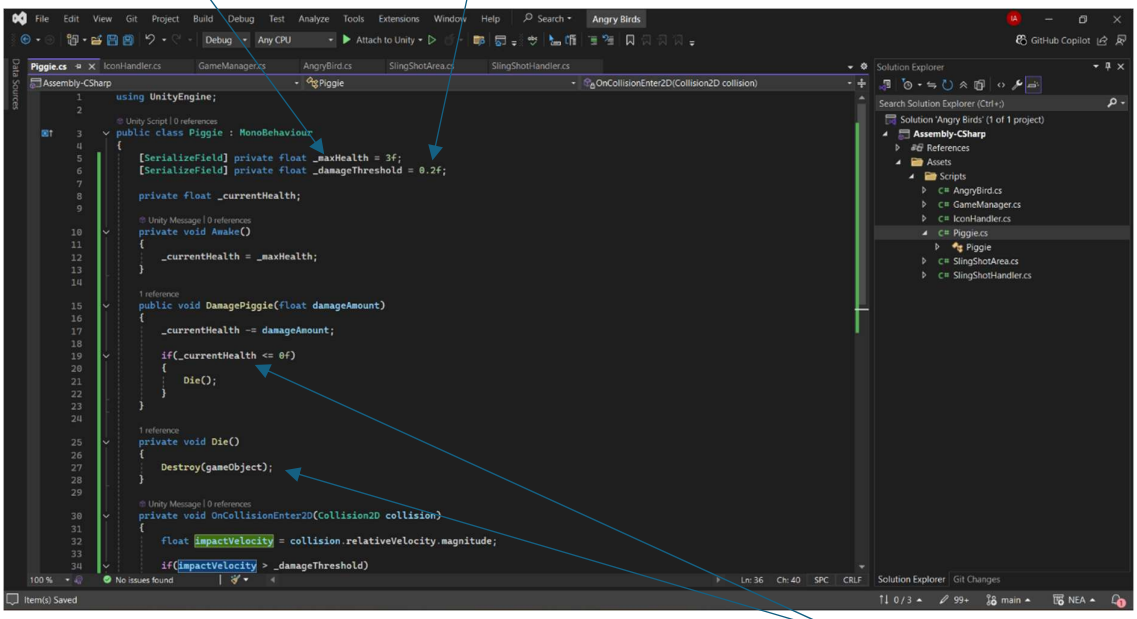
Shaping the Colliders

With the colliders in place, I wrote a script to define the pig's behaviour. I implemented a health system, ensuring that each pig starts with a set maximum health at the beginning of the game. Whenever the pig collides with an object, it loses health based on the impact. Initially, I noticed that the pig would take damage from even the smallest collisions, which didn't feel natural or fair. To address this, I introduced a damage threshold, preventing the pig from losing health due to insignificant impacts.

I originally considered using momentum to determine the damage received, as it would provide a more physics-based approach. However, since the objects in the game do not have assigned masses, I opted to calculate damage based purely on velocity instead. This method still ensures that stronger impacts cause greater damage while preventing unrealistic health loss from minor movements.

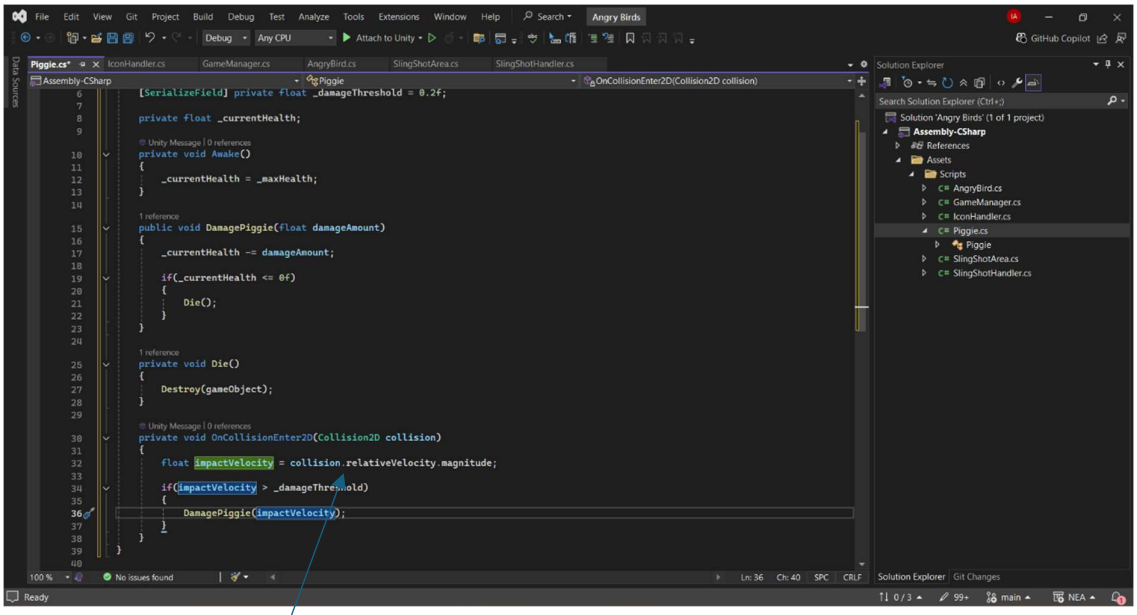
Maximum Health

Pig attains no damage if less than this



```
using UnityEngine;

Unity Script | 0 references
public class Piggie : MonoBehaviour
{
    [SerializeField] private float _maxHealth = 3f;
    [SerializeField] private float _damageThreshold = 0.2f;

    private float _currentHealth;

    Unity Message | 0 references
    private void Awake()
    {
        _currentHealth = _maxHealth;
    }

    1 reference
    public void DamagePiggie(float damageAmount)
    {
        _currentHealth -= damageAmount;

        if(_currentHealth <= 0f)
        {
            Die();
        }
    }

    1 reference
    private void Die()
    {
        Destroy(gameObject);
    }

    Unity Message | 0 references
    private void OnCollisionEnter2D(Collision2D collision)
    {
        float impactVelocity = collision.relativeVelocity.magnitude;

        if(impactVelocity > _damageThreshold)
```

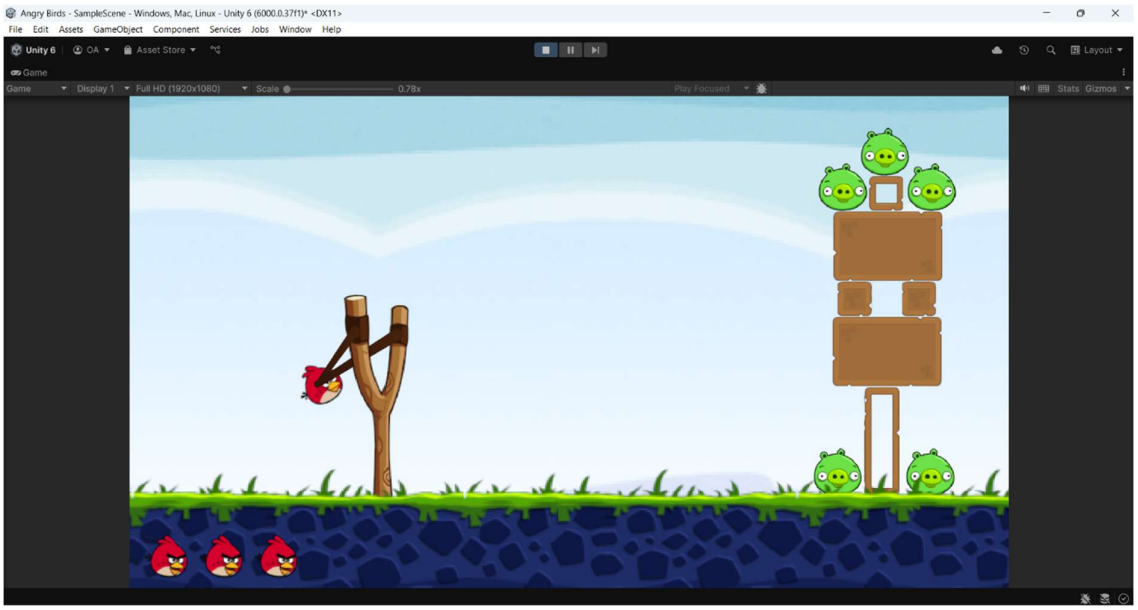
Kill Pig if health is less than zero by removing it from the game



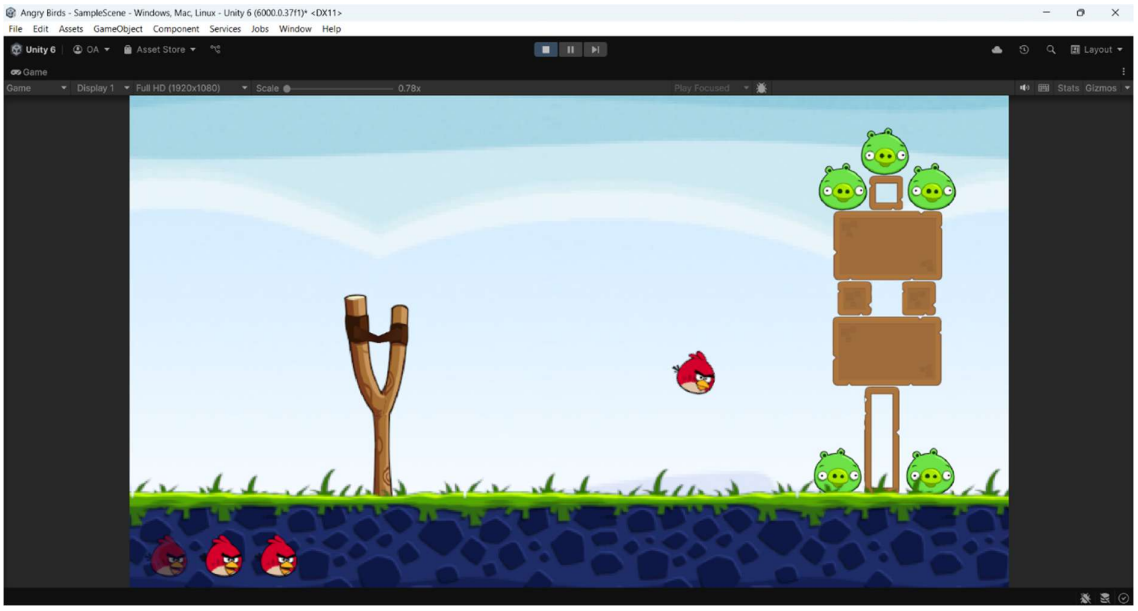
```
    [SerializeField] private float _damageThreshold = 0.2f;

    private float _currentHealth;

    Unity Message | 0 references
    private void Awake()
    {
        _currentHealth = _maxHealth;
    }

    1 reference
    public void DamagePiggie(float damageAmount)
    {
        _currentHealth -= damageAmount;

        if(_currentHealth <= 0f)
        {
            Die();
        }
    }

    1 reference
    private void Die()
    {
        Destroy(gameObject);
    }

    Unity Message | 0 references
    private void OnCollisionEnter2D(Collision2D collision)
    {
        float impactVelocity = collision.relativeVelocity.magnitude;

        if(impactVelocity > _damageThreshold)
        {
            DamagePiggie(impactVelocity);
        }
    }
}
```

Calculate damage taken using Velocity

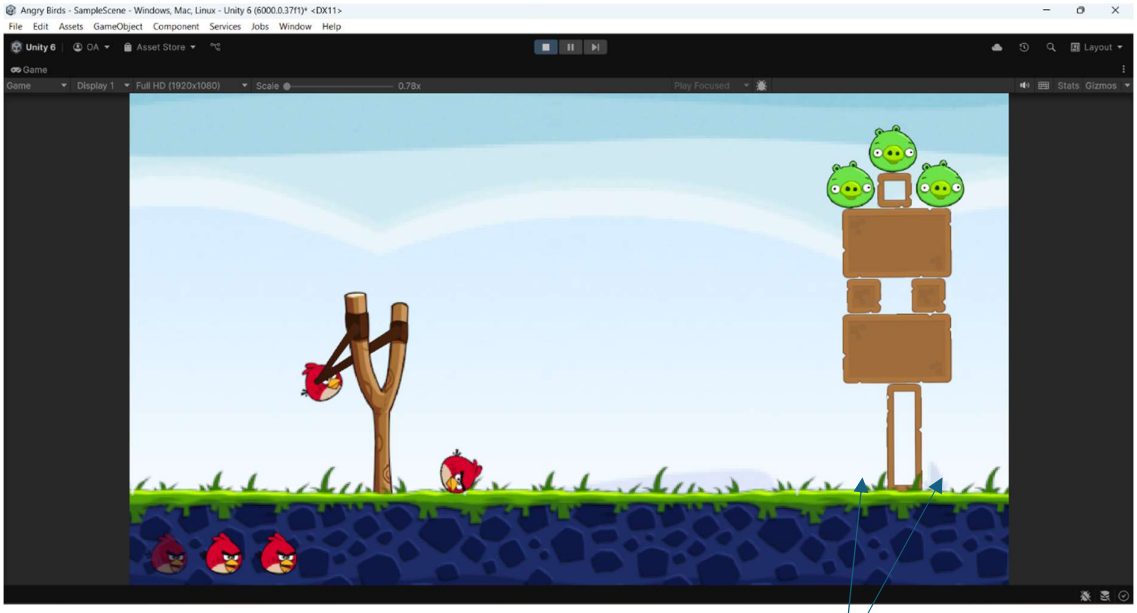
Testing the Pig [16/03/2025]



Testing to see if Pig disappears (1)



Testing to see if Pig disappears (2)

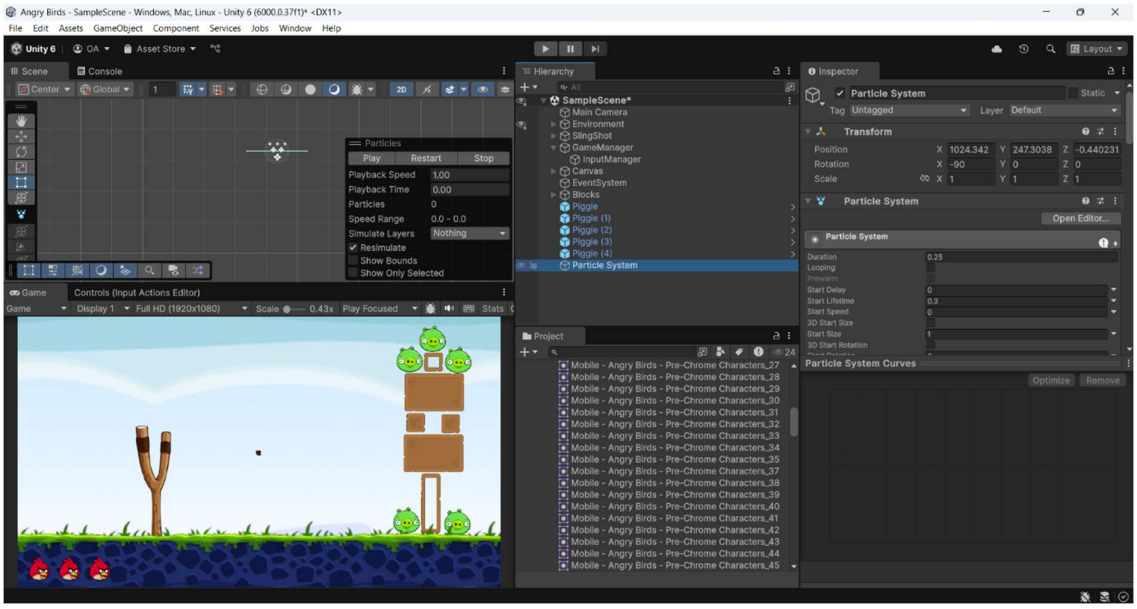


Testing to see if Pig disappears (3)

Adding Animation to Pig Popping [16/03/2025]

I decided to animate the pig's popping effect by adding visual effects to make the game feel more polished and engaging. Initially, my stakeholder did not express a strong preference for this feature, but later, they considered the popping animation to be important as the absence of it made the game look "unreal" and less satisfying. Adding a visual effect would enhance the player's experience by providing clear feedback when a pig is eliminated.

To achieve this, I first explored different particle effects to determine which one would be most suitable. I experimented with various effects, including small explosion animations and fading effects, but I ultimately decided on using a classic smoke effect, similar to the one seen in the original game. This effect provides a subtle yet clear indication that the pig has been removed from play while maintaining the aesthetic of the game.

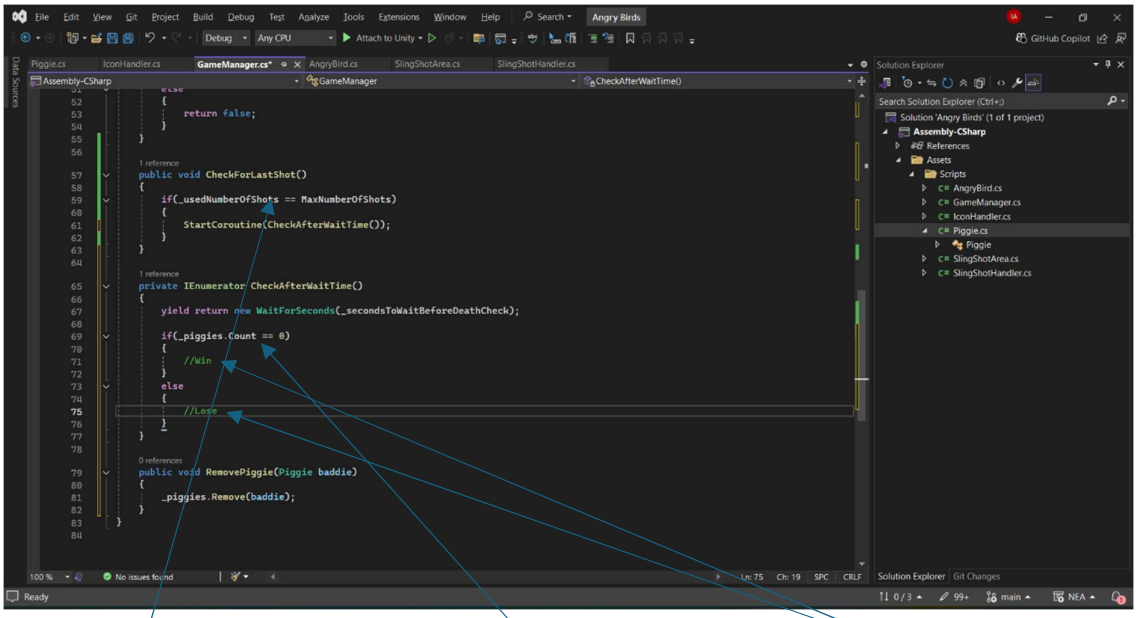


Selecting a Particle animation

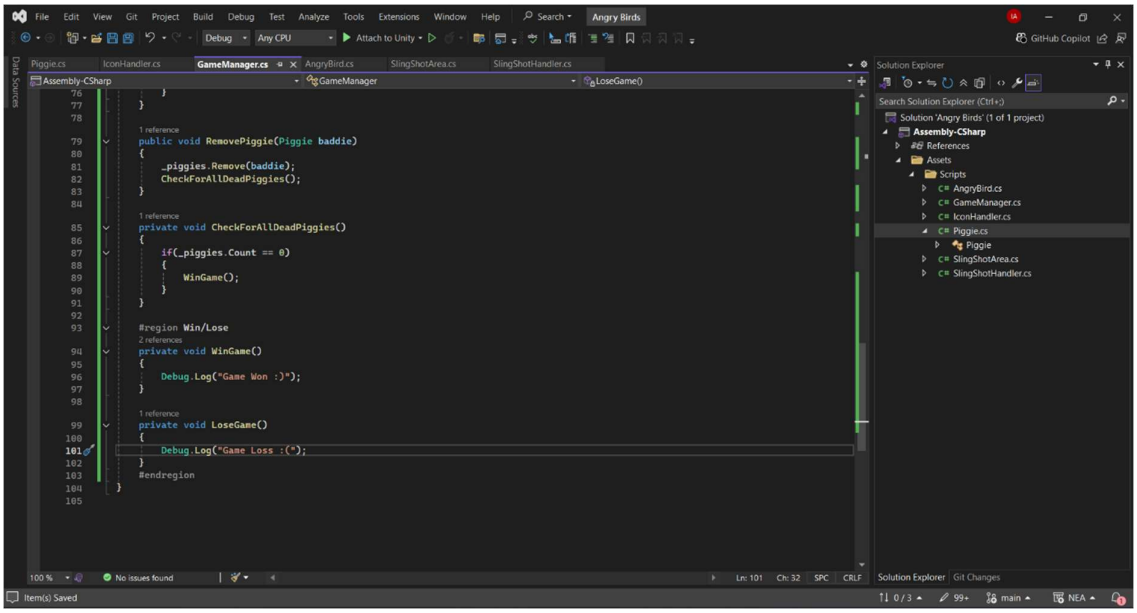
Once I had chosen the effect, I implemented it into the pig popping sequence ensuring that the animation plays immediately when the pig's health reaches zero. This way, the player receives instant visual feedback upon successfully hitting and eliminating a pig. The final implementation can be seen in the gameplay footage displayed later on in the evaluation as a reference.

Determining Win/Loss [18/03/2025]

The next task was to determine whether the player had won or lost the game. To do this, I created a game management script that handles the win/loss conditions. The script works by checking if the number of pigs remaining in the level is zero after all the birds have been used. If there are no pigs left, then the game has been won. If there are still pigs remaining and the birds are gone, then the game has been lost.



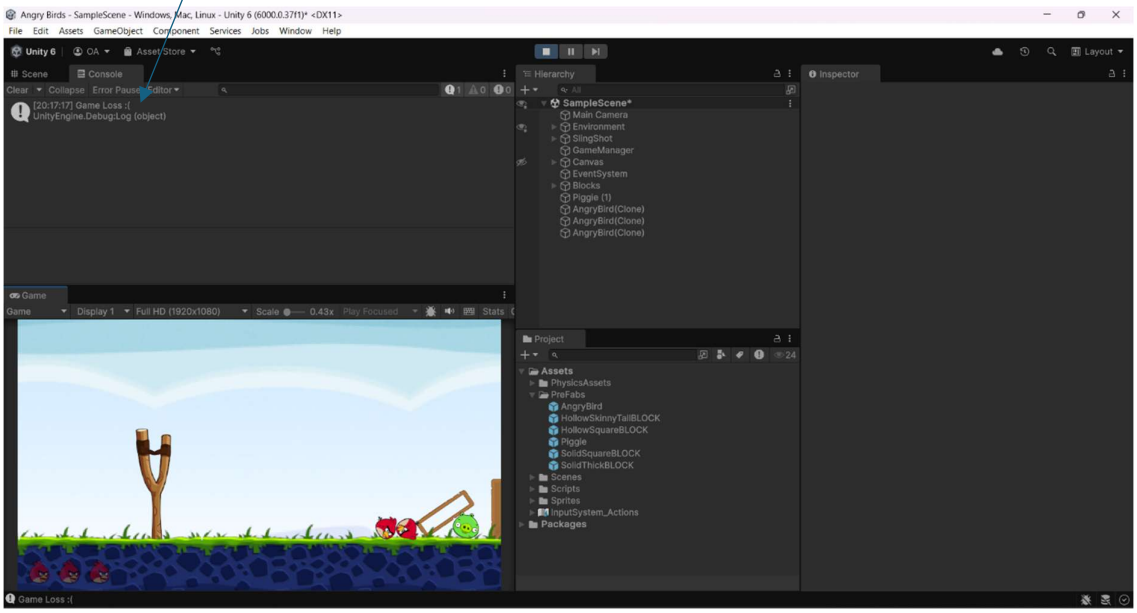
I also made sure the game manager continuously checks the win condition every time a bird is launched. This way, the game is always monitoring the status and can immediately identify whether the player has won or lost as soon as the conditions are met. To test that everything was functioning correctly, I did a simple log test. This involved adding a log in the script to print out whether the game thought the player had won or not based on the current state of the pigs and birds. This log helped me confirm that the win/loss check was happening as expected.



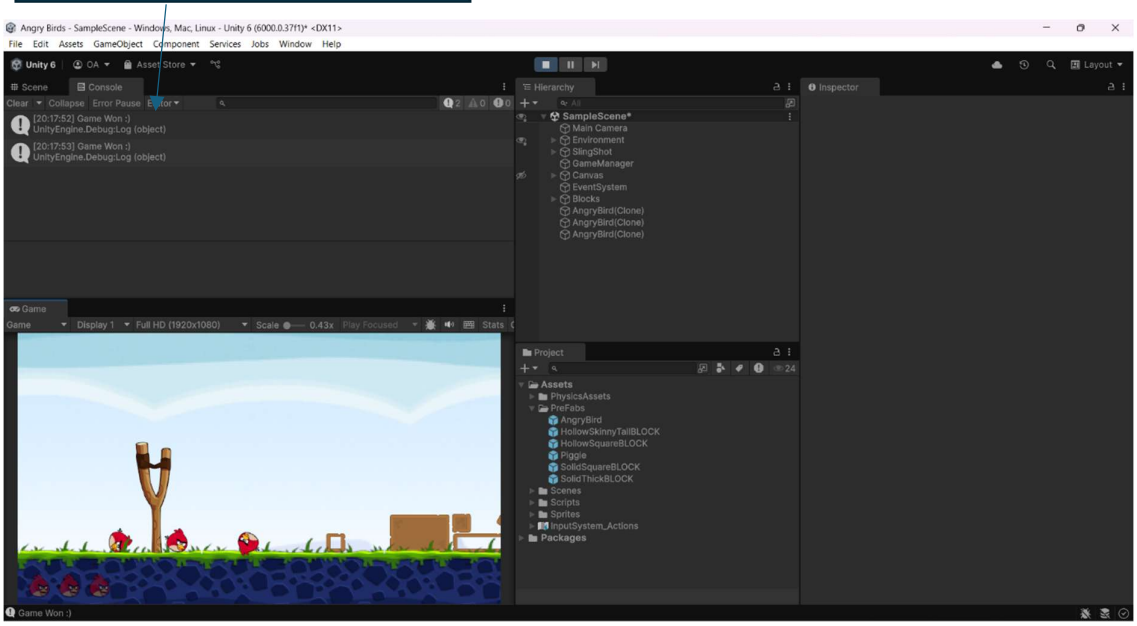
Coded log test

`

Log when game is lost



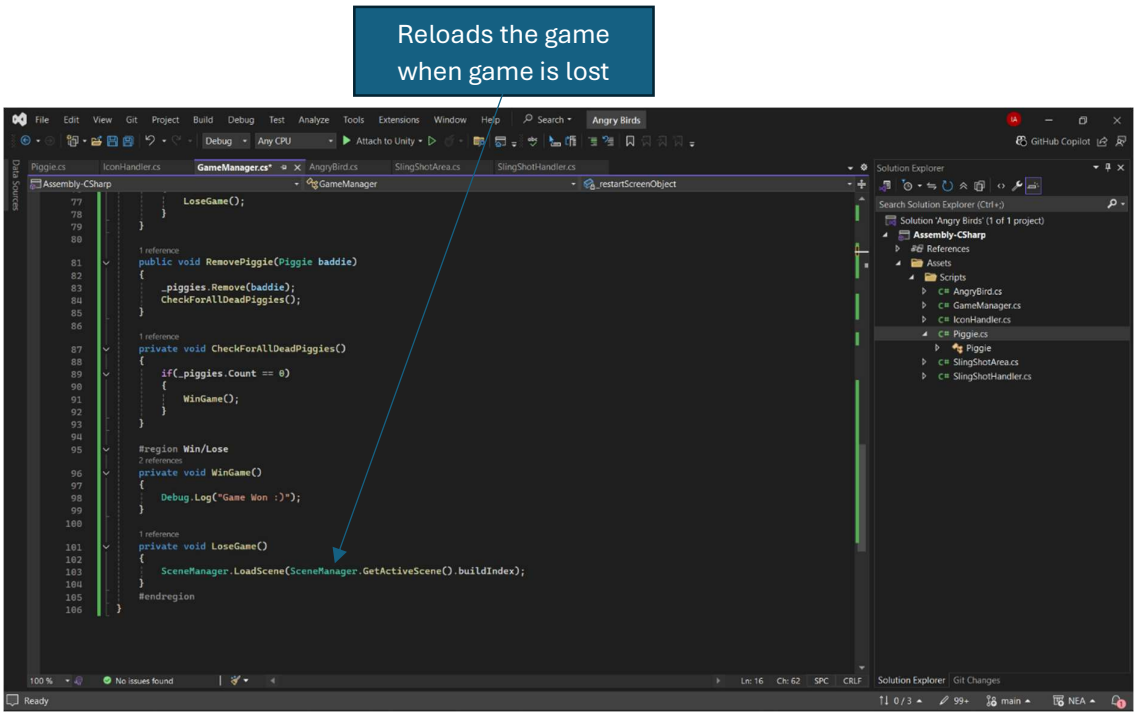
Log when game has been won



Programming Win/Loss Conditions[19/03/2025]

After writing the script that determines the win and loss conditions, my next step was to implement the corresponding actions that would take place in each scenario. To keep things simple and ensure that the core functionality was in place, I opted for basic implementations at this stage, with the intention of refining them later if time allowed.

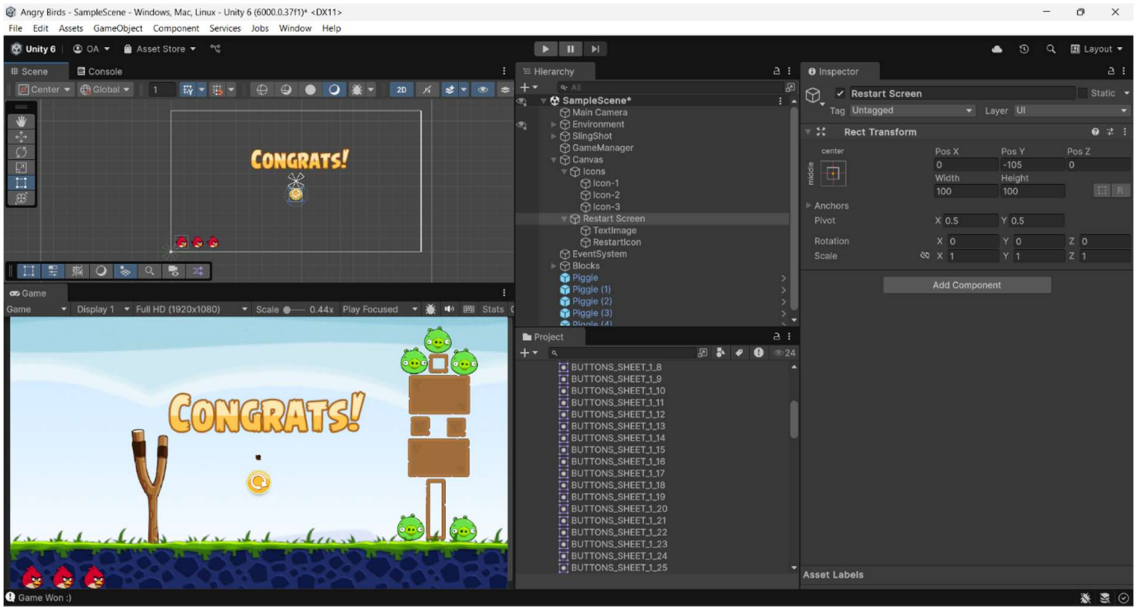
I started with the loss condition, as it was the simpler of the two. Since there is currently only one level in the game, I implemented a straightforward script that reloads the scene when the player loses. This ensures that the player can immediately retry the level without the need for additional menus or interactions.



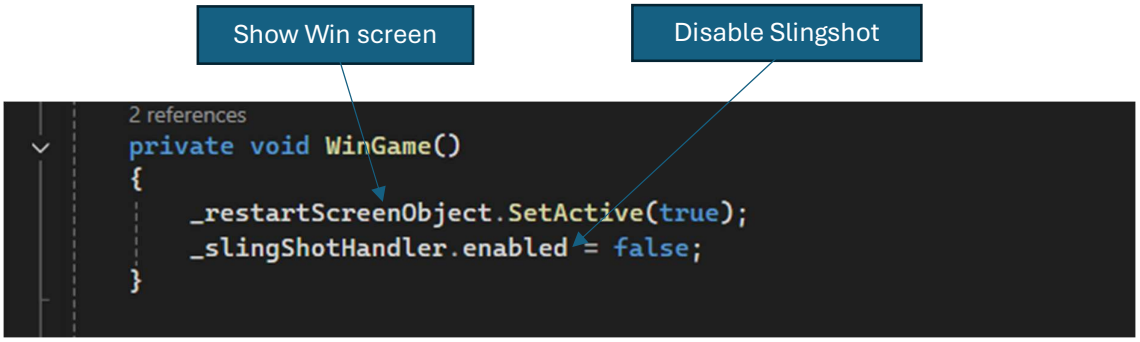
Loss Function

For the win condition, I needed to design a proper restart screen to provide clear feedback to the player. I created a simple yet effective win screen that displays a message indicating success along with an option to restart the game. Once the design was complete, I wrote a script that activates the win screen when the game manager detects that all pigs have been eliminated.

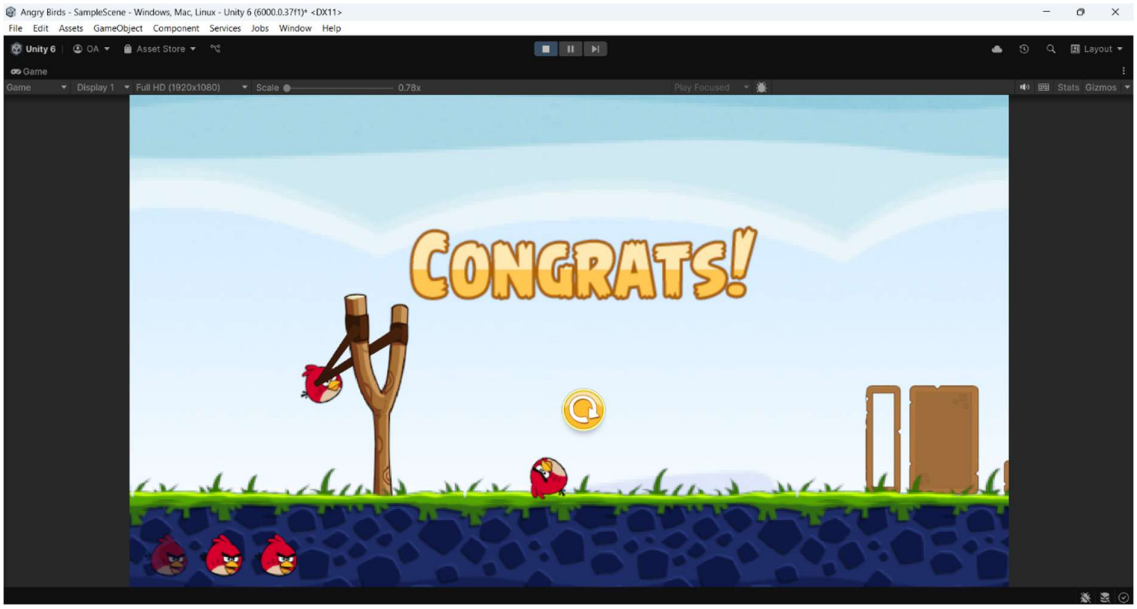
Additionally, I implemented a function to disable the slingshot once the game has been won. This prevents the player from continuing to interact with the game world after achieving victory, ensuring a more polished and structured experience.



Creating Win Screen



Win Function

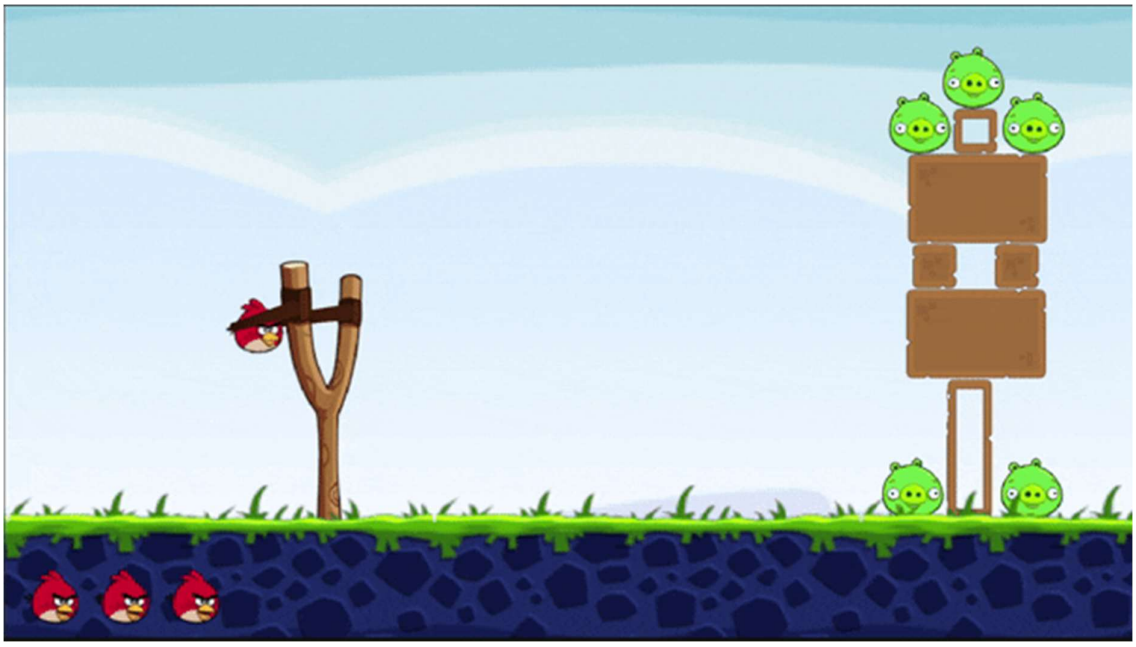


Testing Win screen

With these implementations, the basic game requirements have now been met, and the core functionality is in place.

EVALAUTION

Below is a recording of the full gameplay:

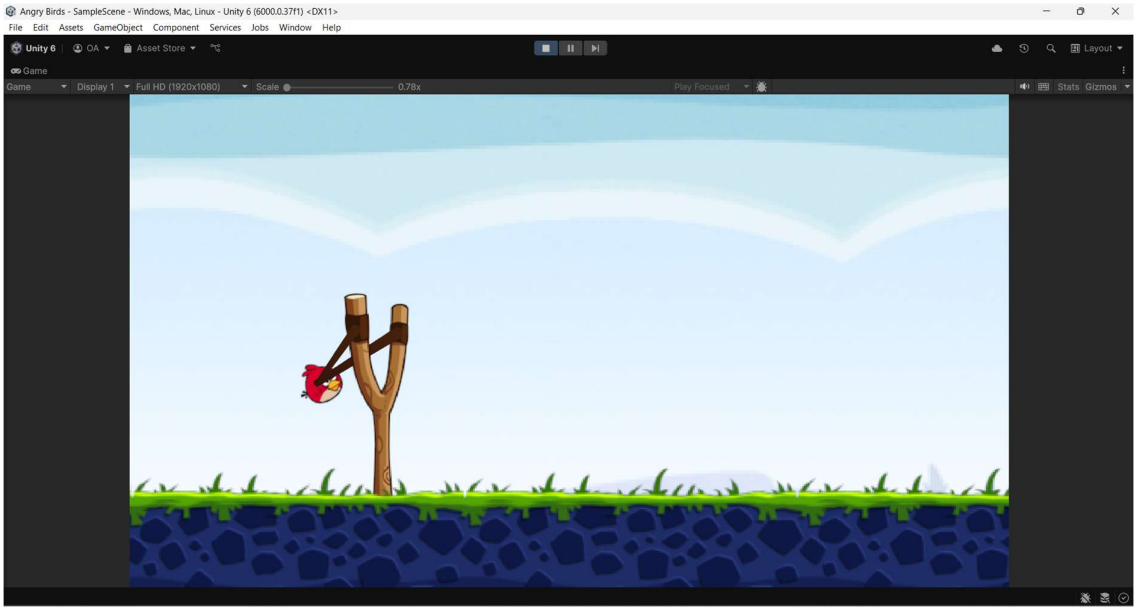


Cross Referencing Solution with Success Criteria

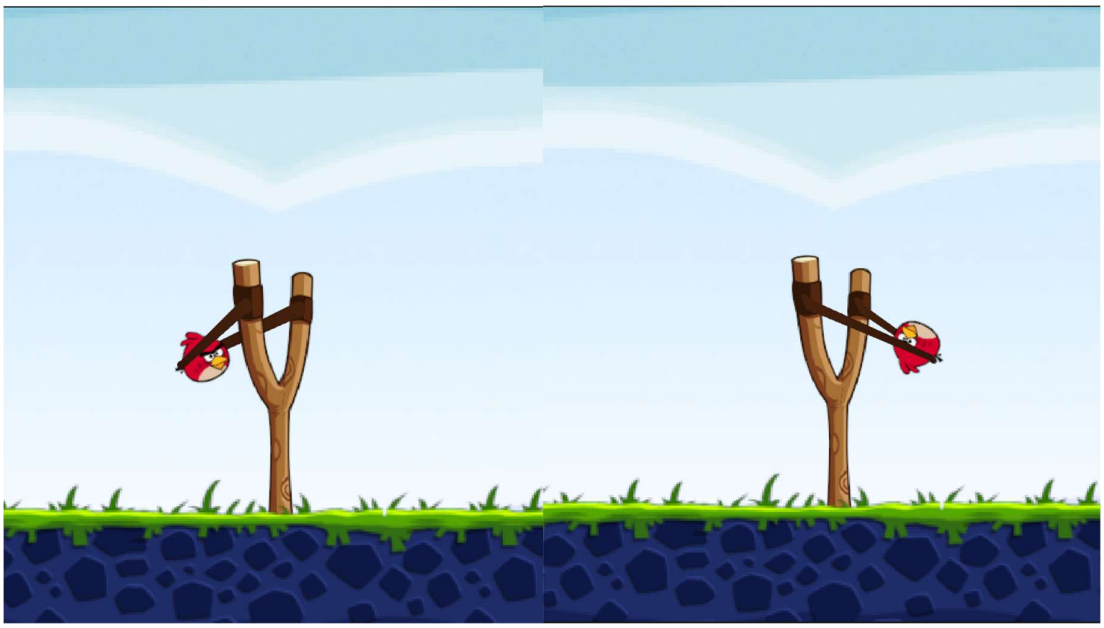
As referenced in my design, here is the test I conducted after creating the game.

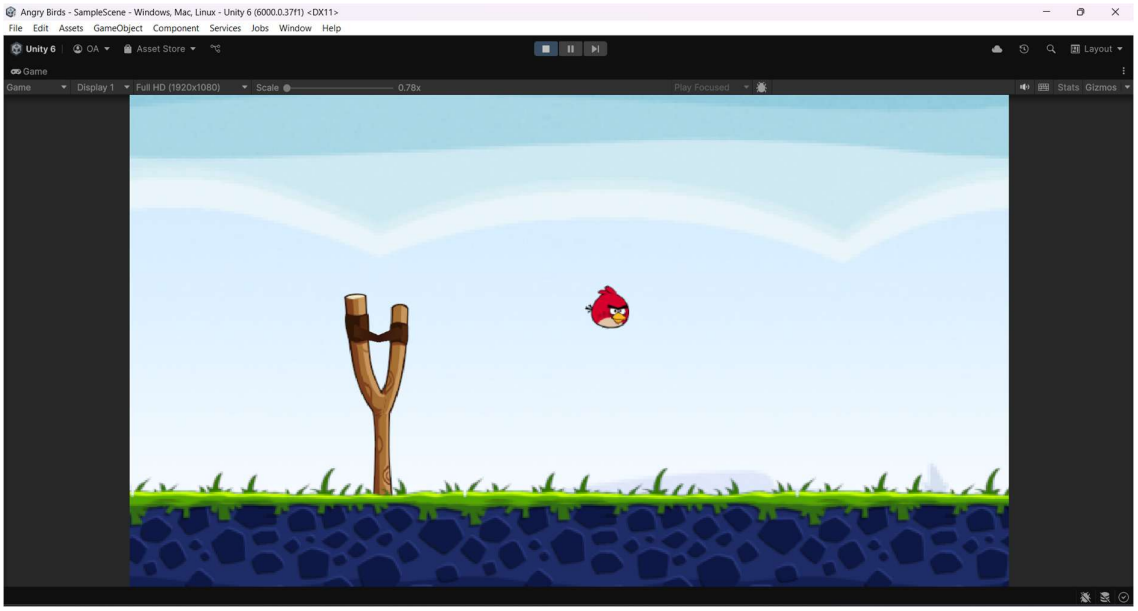
Slingshot				
Test No.	**Tests**	**Expected output**	**Evidence**	**Pass/Fail**
1	The Slingshot animates when being attempted to launch. If the slings is past the slingshot area it restrict the player	The slings pullback when left clicked and follow the cursor till the player releases	▪ See video evidence below ▪ See screenshots below ▪ See full gameplay video above	Pass
2	Slingshot launches bird when the player releases the slingshot after being pulled back	The bird is flung out of the slingshot with a specific force	▪ See Screenshots below	Pass

			▪ See full gameplay video above	

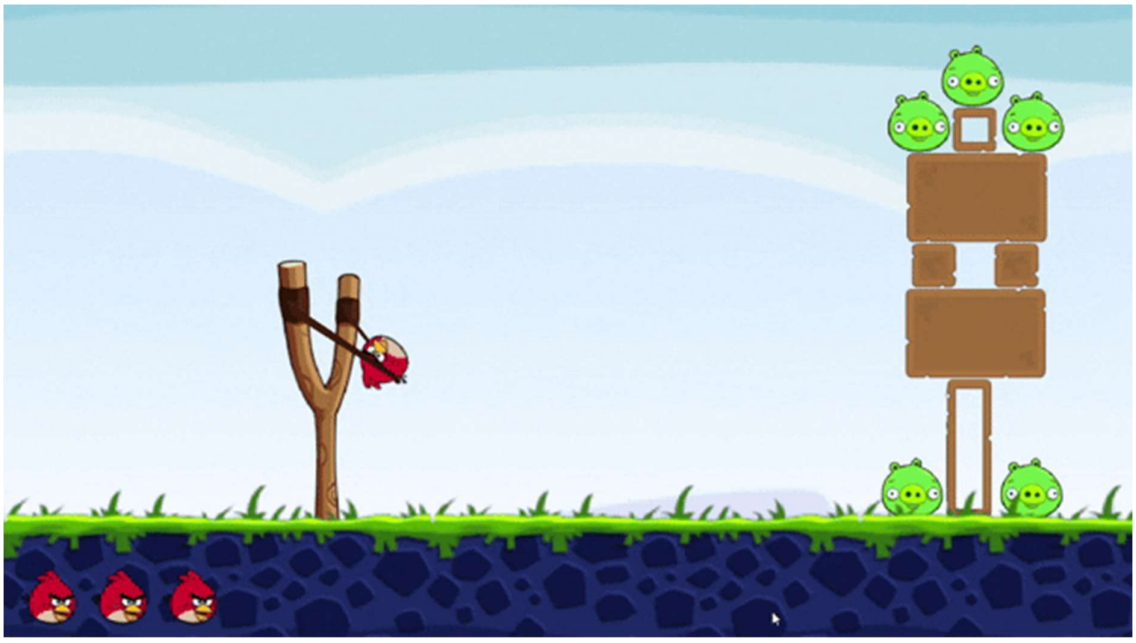


Slingshot draws back



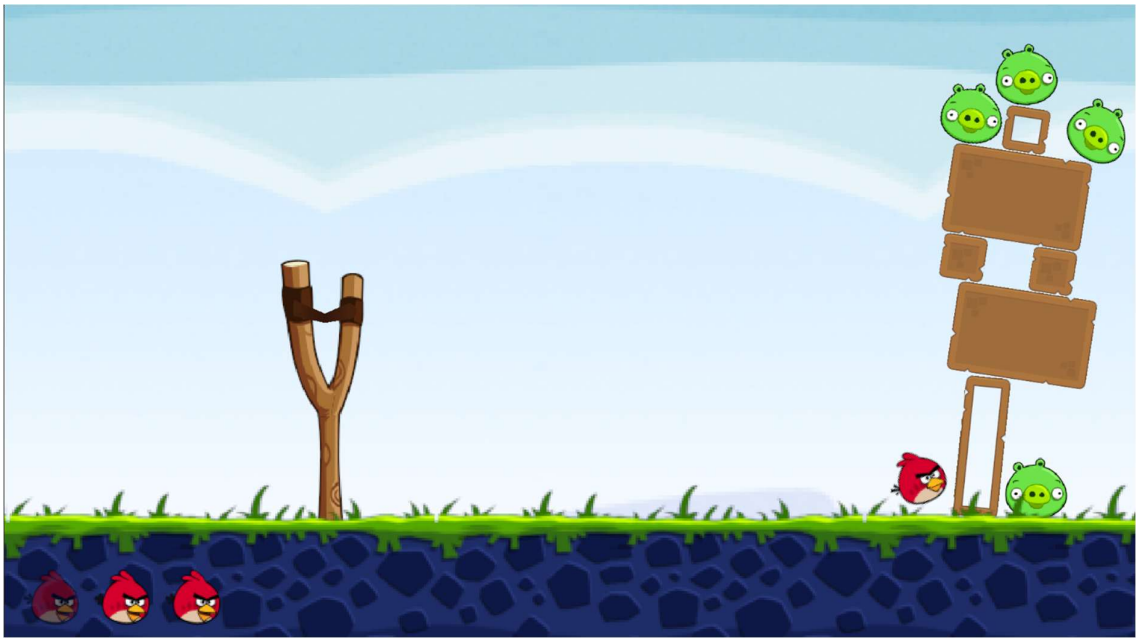


Bird in Air after being launched

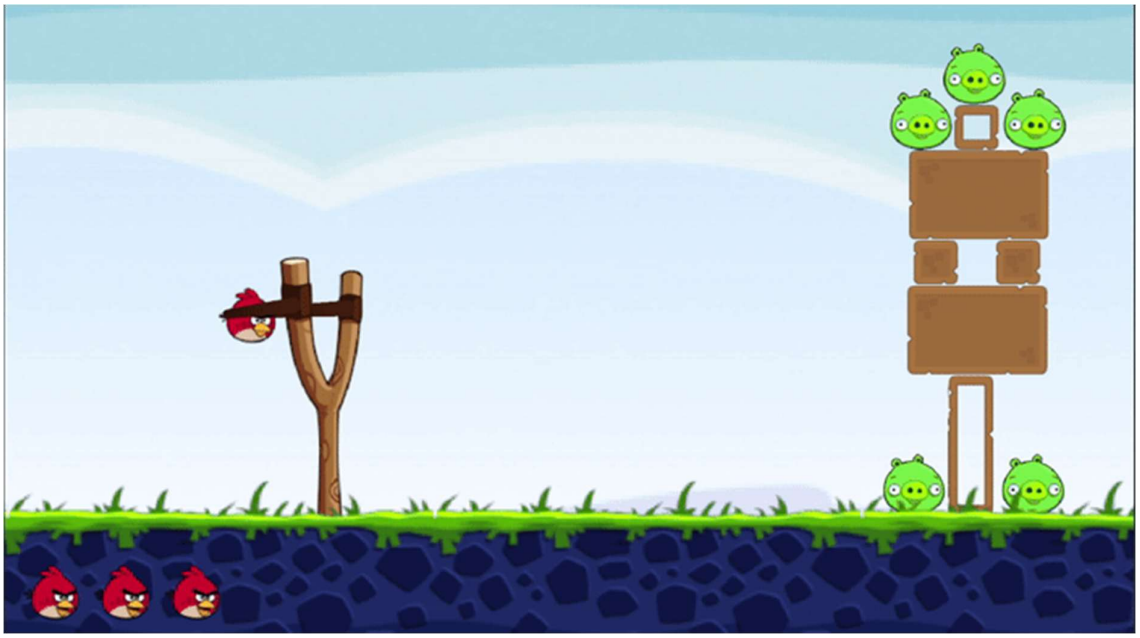


Slingshot animates when dragged and launches when released

Bird				
Test No.	**Tests**	**Expected output**	**Evidence**	**Pass/Fail**
1	Interaction between Bird and Block	The bird alters the block based on its force and direction.	▪ See video evidence below ▪ See screenshots below ▪ See full gameplay video above	Pass
2	Interaction between Bird and Pig	The bird hits the pig, and pops them when they are hit with a certain force	▪ See video evidence below ▪ See full gameplay video above	Pass
3	Bird obeys laws of physics of look realistic	Bird has an acceleration due to gravity and acts like a normal projectile when launched (excluding air resistance)	▪ See video evidence below ▪ See full gameplay video above	Pass



Bird Pushing Blocks after being hit



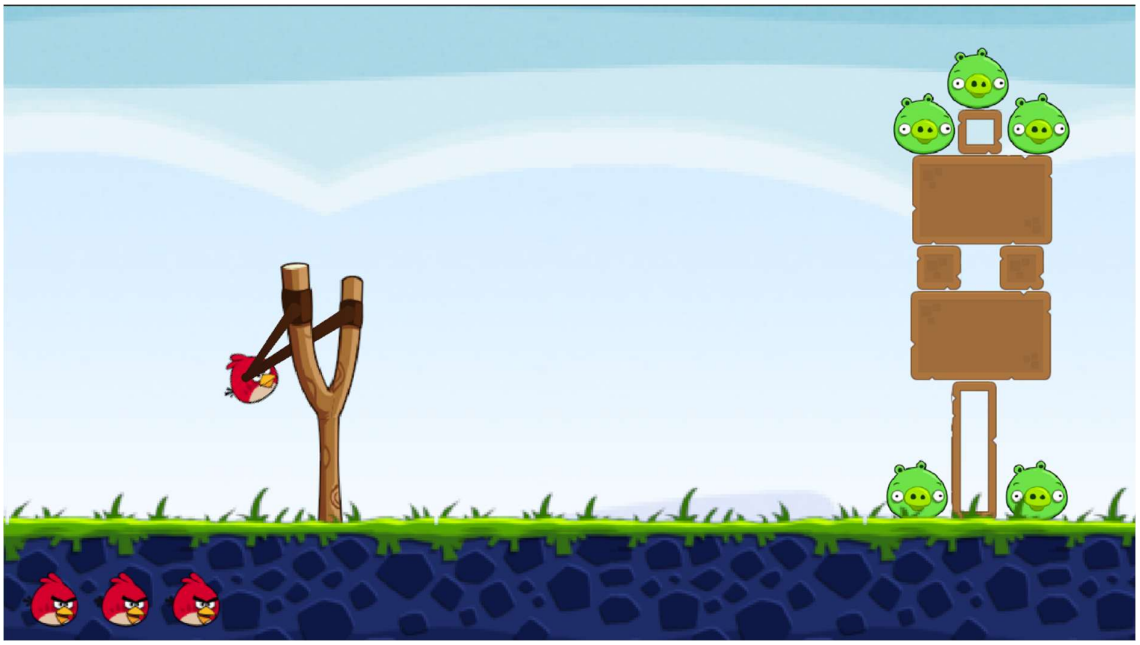
[1.] Bird Pushing Blocks and Popping Pig while obeying laws of physics

[2.] Blocks responding the a force applied to it while having a gravitational force applied to it

Blocks				
Test No.	**Tests**	**Expected output**	**Evidence**	**Pass/Fail**
1	Physics Govern how blocks act	The bird alters the block based on its force and direction. The block falls in response to gravity	▪ See video evidence above ▪ See full gameplay video above	Pass

Pigs				
Test No.	**Tests**	**Expected output**	**Evidence**	**Pass/Fail**
1	Pigs interaction with Bird	The pig pops when hit with a certain force or else it stays a in the game	▪ See video evidence above ▪ See full gameplay video above	Pass

Background				
Test No.	**Tests**	**Expected output**	**Evidence**	**Pass/Fail**
1	The Game has a Static Background	The Game has a Static Background that supports the other sprites in the game.	▪ See screenshot below ▪ See full gameplay video above	Pass



The Game and its Background.

Gameplay				
Test No.	**Tests**	**Expected output**	**Evidence**	**Pass/Fail**
1	Determines if Game has been won or lost	The game monitors the number of Bird and Pigs every time a bird has been used and determines if the Game has been won or lost.	▪ See full gameplay video above	Pass
2	Ability to Restart the Game when it has been won or lost	Game restarts or has the option to restart when the game is over.	▪ See full gameplay video above	Pass

3	Core Gameplay Mechanic: The Bird tries to knock out all the Pigs with a limited amount of Birds	The Game lets you aim and pop pigs. If you can't pop all the pigs with a limited amount of Birds you lose the game else you win.	▪ See full gameplay video above	Pass

After thoroughly testing each component of the basic game mechanics, I can confidently say that the core requirements of the game have been successfully implemented. Every essential feature, including the slingshot mechanics, bird launching, collision detection, and win/loss conditions, has been tested to ensure that they function as intended. The game now meets the fundamental objectives outlined in my initial plan.

With the basic structure complete, I will now focus on testing for the additional features that I originally wanted to include. These extra features were mentioned earlier in my analysis as potential improvements that could make the game more engaging and polished.

Extra Features			
Test No.	**Criteria**	**Success**	**Evaluation**
1	Score System	Not Met	A score system was not implemented due to time constraints. In future iterations, I plan to develop a system where points are assigned based on whether a pig or a block is hit. Additionally, the system will reward players for completing a level using fewer birds, encouraging strategic gameplay.
2	High Score System	Not Met	Since a score system was not implemented, a high score system was also not included. The original plan was to use a lightweight database like SQLite to store scores locally. I also intended to implement an online leaderboard, allowing players to compare high scores with friends. However, dealing with network sockets and setting up a server and domain proved too complex for the scope of this project.
3	Levels	Not Met	Due to limited time, I was unable to create multiple levels. This would be straightforward to implement by

			arranging blocks and pigs in more challenging configurations. Expanding the number of levels will be a priority in future updates, as it will significantly enhance the game's replayability.
4	Sound Effects	Not Met	Sound effects were not included because sourcing suitable audio files proved difficult. Without proper recording equipment, I was unable to create my own sounds, and my search for appropriate royalty-free sound effects was unsuccessful. Moving forward, I aim to integrate sounds to enhance player immersion.
5	Splash Screen	Partially Met	The final game includes a splash screen, but it is the default Unity splash screen rather than a custom design. Additionally, it does not function as intended based on the original design specification. In future updates, I plan to create a personalized splash screen that aligns with the game's theme and integrates seamlessly into the overall experience.
6	Game Over Screen	Partially Met	Instead of a dedicated game over screen, the game currently restarts automatically when the player loses. Since there is only one level, a restart function was more practical. However, as additional levels are introduced, a proper game over screen will become essential and will be implemented accordingly.
7	Game Win Screen	Fully M et	The win screen is functional and operates as intended. However, I plan to enhance it by adding animations, an in-game currency system to allow players to purchase an extra bird, and a next-level function once additional levels are created. These features will improve the overall game experience and provide more incentive for players to continue progressing.

Next Steps…

I plan to continue developing the program by implementing the additional features I initially outlined, such as sound effects, a scoring system, and additional levels. These features will enhance the gameplay experience, making the game more engaging and immersive for players.

One of the original ideas was to include a global high-score ranking system. However, after conducting further research, I realized that the complexity and financial costs associated with this feature may outweigh its benefits. Implementing a global leaderboard would require setting up and maintaining an online server, managing network sockets, and ensuring real-time data updates, all of which introduce significant technical challenges. Additionally, maintaining an online server could lead to recurring costs that may not be justifiable for a small-scale project. As a result, I have decided to pivot towards an offline approach, using a local database to store player statistics, such as high scores and completed levels. This will still allow players to track their progress without the added complications of online connectivity.

Beyond feature development, I have also considered how the game will be maintained over time. Since the game was built with a modular structure, debugging and updating different components will be relatively straightforward. If any bugs are discovered or improvements need to be made, I can modify specific sections of the code without affecting the entire project. However, I recognize that I and my stakeholder may not be able to identify every potential issue or enhancement on our own.

To address this, I plan to create a website alongside the game launch, which will include a feedback system. This will allow players to submit opinions, report bugs, and suggest improvements. By gathering feedback from real users, I will be able to make targeted updates, ensuring that the game continues to improve based on community input. Maintaining an open line of communication with players will be essential in refining the game and keeping it engaging over time.

Stakeholder Remarks:

"" Overall, I would say that the game has met my requirements. It is simple, engaging, and fun to play, which is exactly what I was looking for. The core mechanics function as expected, and the game provides a satisfying experience.

While I am happy with how the game turned out, I do think there is still room for improvement. Adding more sound effects, animations, and additional levels could make the gameplay even more immersive. However, the foundation is solid, and I appreciate the effort that went into making sure the game was both enjoyable and easy to understand.

Overall, I am pleased with the outcome, and I look forward to seeing how the game evolves with future updates and improvements. ""

Project Appendix

```
#region Slingshot Methods


    private void DrawSlingshot()
    {
        // Convert the mouse position to world coordinates
        Vector3 TouchPosition = Camera.main.ScreenToWorldPoint(InputManager.MousePosition);

        // Calculate how far the slingshot has been pulled back within the allowed distance
        _slingshotLinesPosition = _centrePosition.position +
Vector3.ClampMagnitude(TouchPosition - _centrePosition.position, _maxDistance);

        // Update the slingshot lines to the new position
        SetLines(_slingshotLinesPosition);

        // Determine the launch direction
        _direction = (Vector2)_centrePosition.position - _slingshotLinesPosition;
        _directionNormalised = _direction.normalized;
    }

    private void SetLines(Vector2 position)
    {
        // Enable the slingshot lines if they were hidden
        if (!_leftLineRenderer.enabled && !_rightLineRenderer.enabled)
        {
            _leftLineRenderer.enabled = true;
            _rightLineRenderer.enabled = true;
        }

        // Set the position of the slingshot lines
        _leftLineRenderer.SetPosition(0, position);
        _leftLineRenderer.SetPosition(1, _leftStartPosition.position);

        _rightLineRenderer.SetPosition(0, position);
        _rightLineRenderer.SetPosition(1, _rightStartPosition.position);
    }

    #endregion
```

```
#region AngryBird Methods


    private void PositionAndRotateAngryBird()

    {

        // Move the bird according to slingshot stretch and align its rotation

        _spawnedAngryBird.transform.position = _slingshotLinesPosition + _directionNormalised *
_angryBirdPositionOffset;

        _spawnedAngryBird.transform.right = _directionNormalised;

    }



    private void SpawnAngryBird()

    {

        // Reset slingshot visuals to idle position

        SetLines(_idlePosition.position);



        Vector2 dir = (_centrePosition.position - _idlePosition.position).normalized;

        Vector2 spawnPosition = (Vector2)_idlePosition.position + dir *
_angryBirdPositionOffset;



        // Instantiate a new Angry Bird and align it correctly

        _spawnedAngryBird = Instantiate(_angryBirdPreFab, _idlePosition.position,
Quaternion.identity);

        _spawnedAngryBird.transform.right = dir;



        _birdOnSlingshot = true;

    }



    private IEnumerator SpawnAngryBirdAfterTime()

    {

        // Wait for the specified delay before spawning a new bird

        yield return new WaitForSeconds(_timeBetweenBirdRespawn);



        SpawnAngryBird();

    }



    #endregion
```

```
#region Animate Slingshot

private void AnimateSlingShot()
    {
        // Move the elastic transform to simulate slingshot snapback
        _elasticTransform.position = _leftLineRenderer.GetPosition(0);
        float dist = Vector2.Distance(_elasticTransform.position, _centrePosition.position);
        float time = dist / _elasticDivider;
        _elasticTransform.DOMove(_centrePosition.position, time).SetEase(_elasticCurve);

        StartCoroutine(AnimateSlingshotLines(_elasticTransform, time));
    }

    private IEnumerator AnimateSlingshotLines(Transform trans, float time)
    {
        float elapsedTime = 0f;
        while (elapsedTime < time)
        {
            elapsedTime += Time.deltaTime;
            SetLines(trans.position);
            yield return null;
        }
    }

    #endregion
```

```
using UnityEngine;
using UnityEngine.UI;

public class IconHandler : MonoBehaviour
{
     // Array of UI Image components representing the shots available
     [SerializeField] private Image[] _icons;

     // Color to indicate a used shot
     [SerializeField] private Color _usedColour;

     /// <summary>
     /// Updates the shot icon to indicate that a shot has been used.
     /// </summary>
     /// <param name="shotNumber">The number of the shot that was
used.</param>
     public void useShot(int shotNumber)
     {
          // Loop through all icons and find the one that corresponds to the
used shot
          for (int i = 0; i < _icons.Length; i++)
          {
               if (shotNumber == i + 1) // Check if the shotNumber matches
the current index (+1 since array is 0-based)
               {
                    // Change the icon color to indicate that the shot has
been used
                    _icons[i].color = _usedColour;
                    return; // Exit the loop early since the required shot
has been updated
               }
          }
     }
}
```

```
using UnityEngine;

public class Piggie : MonoBehaviour
{
    // Maximum health of the pig (can be adjusted in the Inspector)
    [SerializeField] private float _maxHealth = 3f;

    // Threshold for the damage to register when the pig collides with an object
    [SerializeField] private float _damageThreshold = 0.2f;

    // Particle effect to show when the pig is popped
    [SerializeField] private GameObject _piggiePoppedParticle;

    // Current health of the pig (starts at _maxHealth)
    private float _currentHealth;

    private void Awake()
    {
        // Initialize the pig's health to the maximum value when it spawns
        _currentHealth = _maxHealth;
    }

    /// <summary>
    /// Reduces the pig's health by the given damage amount.
    /// If health falls to zero or below, the pig dies.
    /// </summary>
    /// <param name="damageAmount">The amount of damage to apply to the pig.</param>
    public void DamagePiggie(float damageAmount)
    {
        _currentHealth -= damageAmount;

        // If health reaches zero or below, call Die() method
        if (_currentHealth <= 0f)
        {
            Die();
        }
    }

    /// <summary>
    /// Handles the death of the pig. It removes the pig from the game, instantiates the "popped" particle
effect,
    /// and destroys the pig game object.
```

```
/// </summary>
     private void Die()
     {
          // Remove the pig from the game (called by GameManager)
          GameManager.instance.RemovePiggie(this);

          // Instantiate the particle effect at the pig's position
          Instantiate(_piggiePoppedParticle, transform.position, Quaternion.identity);

          // Destroy the pig's game object after death
          Destroy(gameObject);
     }

     /// <summary>
     /// Detects collisions with other objects and applies damage if the impact velocity exceeds the damage
threshold.
     /// </summary>
     /// <param name="collision">The collision data of the object the pig collides with.</param>
     private void OnCollisionEnter2D(Collision2D collision)
     {
          // Calculate the velocity at which the pig collided with an object
          float impactVelocity = collision.relativeVelocity.magnitude;

          // If the collision impact is above the threshold, apply damage
          if (impactVelocity > _damageThreshold)
          {
               DamagePiggie(impactVelocity);
          }
     }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    // Singleton instance of the GameManager
    public static GameManager instance;

    // Maximum number of shots a player can take in the game
    public int MaxNumberOfShots = 3;

    // Keeps track of the number of shots used by the player
    private int _usedNumberOfShots;

    // Time to wait before checking if the game should end after the last shot
    [SerializeField] private float _secondsToWaitBeforeDeathCheck = 3f;

    // UI object to display when the game is over and the player needs to restart
    [SerializeField] private GameObject _restartScreenObject;

    // Reference to the SlingShotHandler to disable it upon game win
    [SerializeField] private SlingShotHandler _slingShotHandler;

    // IconHandler for updating shot icons when a shot is used
    private IconHandler _iconHandler;

    // List of all the piggies in the game
    private List<Piggie> _piggies = new List<Piggie>();

    // This method is called when the game starts (or when the scene is loaded)
    [System.Obsolete]
    public void Awake()
    {
        // Set the instance of GameManager if it doesn't exist
        if(instance == null)
        {
            instance = this;
```

```
    }

    // Find the IconHandler to update shot icons during the game
    _iconHandler = GameObject.FindFirstObjectByType<IconHandler>();

    // Find all Piggie objects in the scene and add them to the _piggies list
    Piggie[] piggies = FindObjectsOfType<Piggie>();
    for (int i = 0; i < piggies.Length; i++)
    {
        _piggies.Add(piggies[i]);
    }
}

/// <summary>
/// Increments the number of shots used and updates the shot icons.
/// Checks if the last shot was used and triggers game end checks.
/// </summary>
public void UsedShot()
{
    _usedNumberOfShots++;  // Increment the used shots counter
    _iconHandler.useShot(_usedNumberOfShots);  // Update the shot icon UI

    // Check if the player has used all available shots
    CheckForLastShot();
}

/// <summary>
/// Returns whether the player has enough shots remaining.
/// </summary>
public bool HasEnoughShots()
{
    return _usedNumberOfShots < MaxNumberOfShots;
}

/// <summary>
/// Checks if the player has used the last shot.
/// If so, it starts a coroutine to check if the game is won or lost.
/// </summary>
public void CheckForLastShot()
```

```
{
        if (_usedNumberOfShots == MaxNumberOfShots)
        {
            StartCoroutine(CheckAfterWaitTime());
        }
    }

    /// <summary>
    /// Waits a specified amount of time before checking if the game is won or lost.
    /// </summary>
    private IEnumerator CheckAfterWaitTime()
    {
        // Wait for the specified time before performing the check
        yield return new WaitForSeconds(_secondsToWaitBeforeDeathCheck);

        // If no piggies are left, the player wins; otherwise, the game restarts
        if (_piggies.Count == 0)
        {
            WinGame();
        }
        else
        {
            RestartGame();
        }
    }

    /// <summary>
    /// Removes a pig from the game and checks if all piggies have been defeated.
    /// </summary>
    public void RemovePiggie(Piggie baddie)
    {
        _piggies.Remove(baddie);  // Remove the pig from the list
        CheckForAllDeadPiggies();  // Check if all piggies are dead
    }

    /// <summary>
    /// Checks if all piggies are dead and if so, triggers the win condition.
    /// </summary>
    private void CheckForAllDeadPiggies()
```

```
{
        if (_piggies.Count == 0)
        {
            WinGame();
        }
    }

    #region Win/Lose

    /// <summary>
    /// Triggered when the player wins the game. Displays the restart screen and disables the
slingshot.
    /// </summary>
    private void WinGame()
    {
        _restartScreenObject.SetActive(true);  // Show the restart screen
        _slingShotHandler.enabled = false;    // Disable the slingshot handler to stop gameplay
    }

    /// <summary>
    /// Restarts the game by reloading the current scene.
    /// </summary>
    public void RestartGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);  // Reload the current
scene
    }

    #endregion
}
```