Design Overview for Flappy Bird Game

Name: Nguyễn Minh Nhật
Student ID: 105680164

Summary of Program

This Flappy Bird Game is inspired by the original hit created by Dong Nguyen, and I was immersed in it. This program was developed with C# language. It is a side-scrolling game where the player controls a bird, navigating it through a series of pipes.

- ***Objects***: In order to achieve the highest possible score by flying the bird through the gaps between pairs of pipes without colliding with them or hitting the ground.
- **Game Controls**: The player will press a key (or click the mouse) to make the bird "flap," causing it to jump up a short distance. Without input, gravity will pull the bird into the ground.
- **End State**: The game ends when the bird collides with a pipe or the ground.
- **Language:** The project is built entirely in C#, focusing on the application of Object-Oriented Programming (OOP) principles to manage game components such as the character, obstacles, and game states.
- **Design Patterns:** The project is focused mainly on three different kinds of design patterns which are state, observer, factory patterns, not only it improve the system's modularity and complex logic code.

Required Roles

Describe each of the classes, interfaces, and any enumerations you will create. Use a different table to describe each role you will have, using the following table templates.

Table 1: Class Details

Role	Type	Notes
GameManager	Class	The central controller of the game. It manages the overall game state, score, the main game loop, and coordinates all other objects.
GameObject	Abstract Class	The base class for all physical entities in the game. It defines essential properties like position, boundingBox, and requires concrete implementations for the Update() method (polymorphism).
Bird	Class	Represents the player's character. This class handles all logic related to the bird's movement (flapping and gravity), position, and collision detection.
Pipe	Class	Represents a single pair of pipe obstacles (top and bottom). It is a simple component that primarily manages its own position as it moves across the screen.

PipeManager	Class	Responsible for creating and managing Pipe objects. It continuously spawns new pipes off-screen and removes old ones to create the endless level.
UIManager	Class	Manages all User Interface elements. This step includes displaying the score in real-time and showing the "Game Over" screen when the game ends.
Orb	Class	Represents a collectible item. It holds an effect type, such as health and power-up, that is triggered upon collection.
OrbManager	Class	Responsible for spawning and managing Orb objects in the game world.
OrbFactory	Class	Simple Factory Pattern which responsible for initilizing and modifiying orb effect type, seperating creating object from OrbManager
SoundManager	Class	Handles background music and plays specific sound effects for events like jumping, scoring, and collisions (PlayJump, BirdDie, PlayOrbSound)
HealthSystem	Class	Handles currentHealth, maxHealth, and logic for invincibility frames (ActivateInvincibility) and death (Kill).
ReadyState / PlayingState / GameOverState	Concrete State	Each class defines specific behaviors for HandleInput() and Update() corresponding to that game phase, replacing complex if-else logic in GameManager.

Table 2: Enumeration Table

Role	Type	Notes
GameState	Enum	Defines the specific states of the game, allowing the GameManager to control the flow and logic based on the current situation.
OrbEffectType	Enum	Defines the specific effects an orb can have, such as Immune, Health, Point.

GameState Table:

Value	Notes
Ready	The initial state before the game begins, where controls are typically inactive, waiting for the first player input to transition to **Playing**.
Playing	The main game state where physics, movement, spawning, and scoring are all active.
GameOver	The final state triggered after the player's defeat, stopping all gameplay mechanics and often displaying the final score/options.

Table 3: Interface

Role	Type	Notes
«interface» IDamageable	Interface	Defines a **contract** for objects that can **receive damage** (e.g., Bird). Enforces the **TakeDamage()** method.
«interface» ICollectable	Interface	Defines a **contract** for objects that can be **picked up** by the player (e.g., Orb). Enforces the **OnCollected()** method.
«interface» IPipe	Interface	Defines a **contract** for **pipe behavior**. Enforces the **Move()** method, allowing PipeManager to manage all pipe types abstractly.
«interface» IGameState	Interface	State Pattern is responsible for game state such as HandleInput(), Update(), OnEnterState()
«interface» ISubject	Interface	Observer Pattern takes the responsibility of receiving notifications such as Attach(), Detach(), and NotifyObservers().
«interface» IObserver	Interface	Observer Pattern is responsible for Observer, such as UIManager have to implement Update(score: int) in order to respond to the object changes

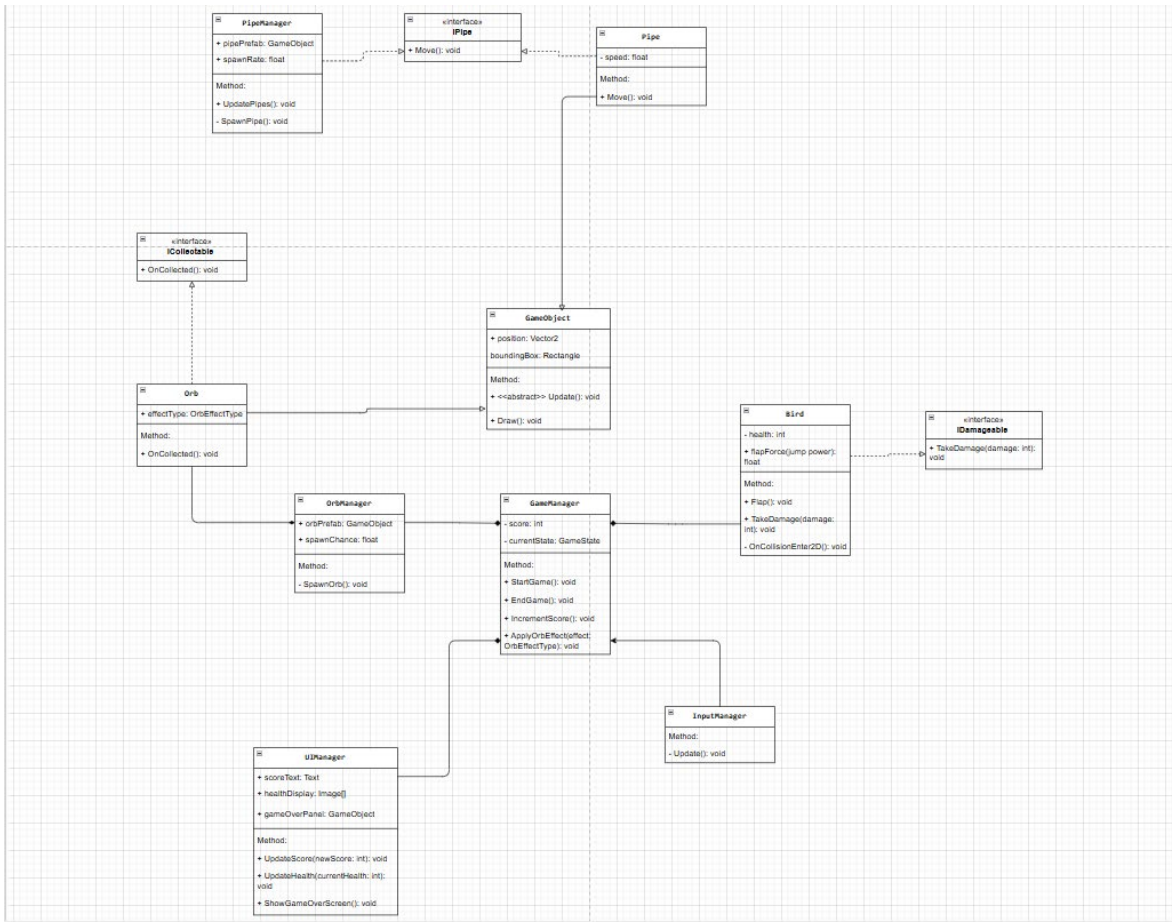
Table 4: Protocol enumeration

Value	Notes
Ready	The initial state before the game begins, waiting for the first player's input to start.
Playing	In the main state where gameplay is active, the bird is flying, and pipes are spawning.
GameOver	The final state after a collision occurs, where all gameplay is stopped and the final score is displayed.

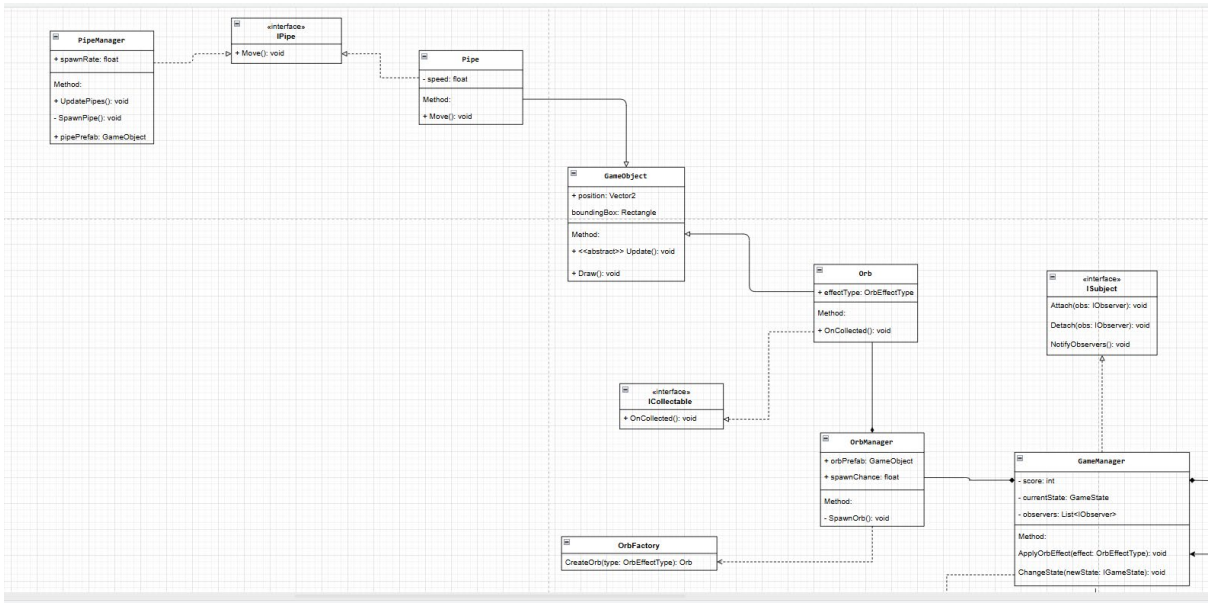
Health	Health Orb that can heal bird one time when collected
Immune	Immune Orb that last long 5 seconds where bird can move through pipes without losing health.
Point	Point Orb that can plus 2 scores when collected

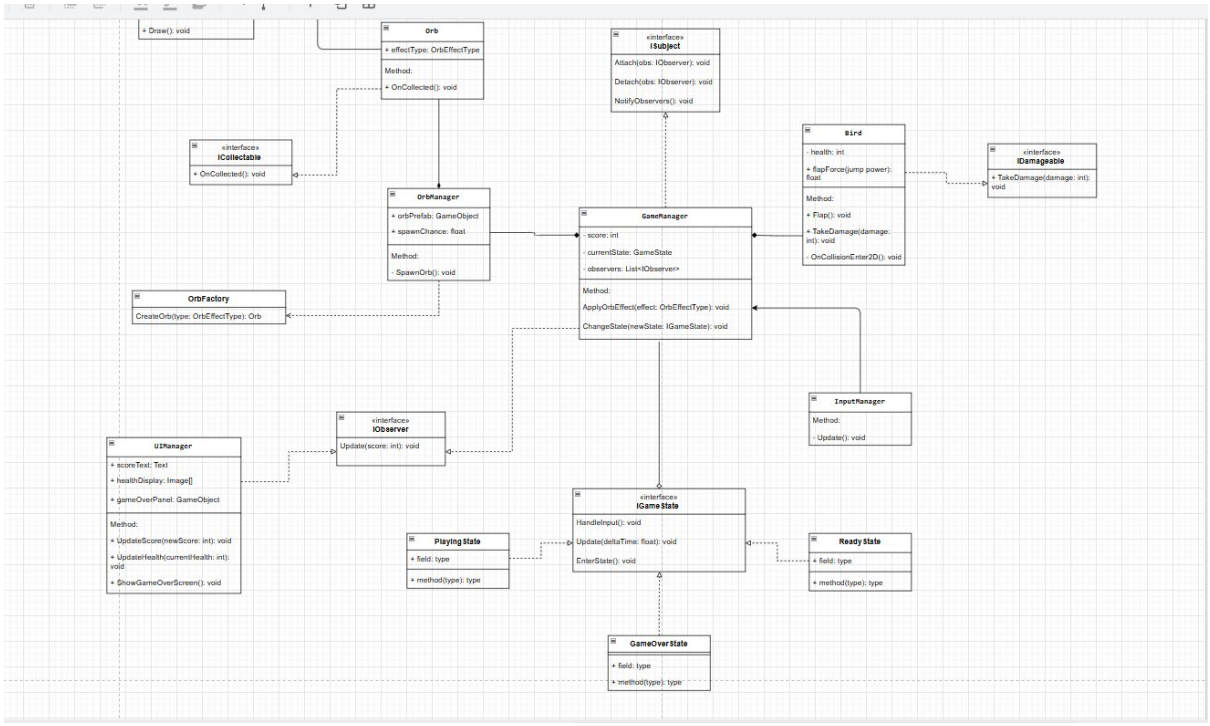
Class Diagram

Option 3 Class Diagram:

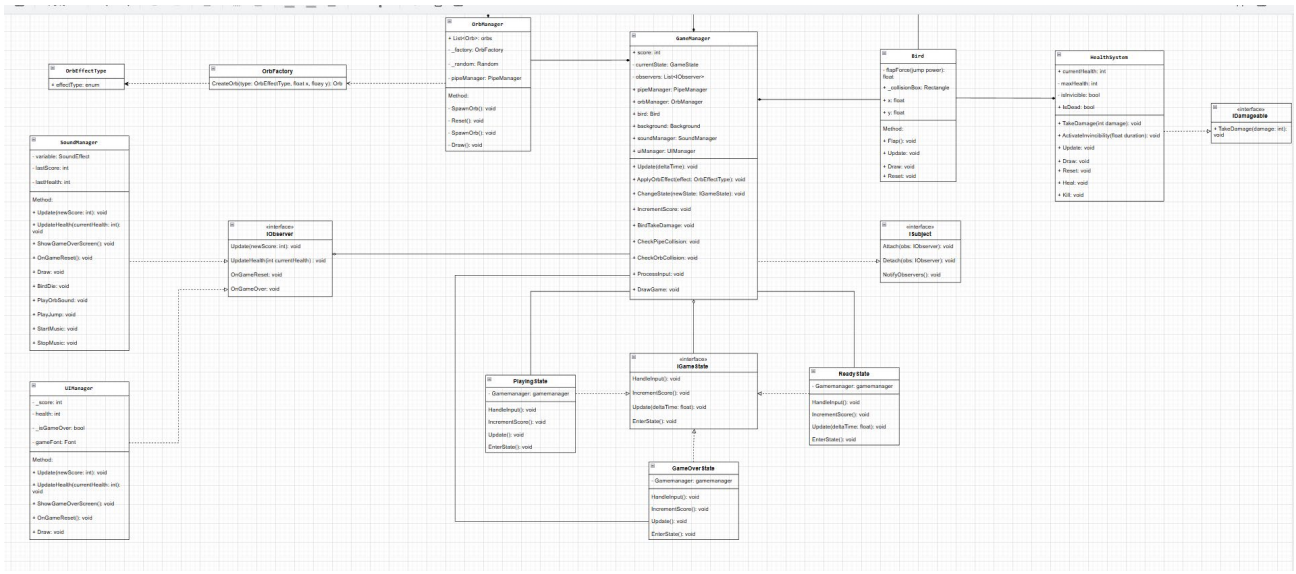


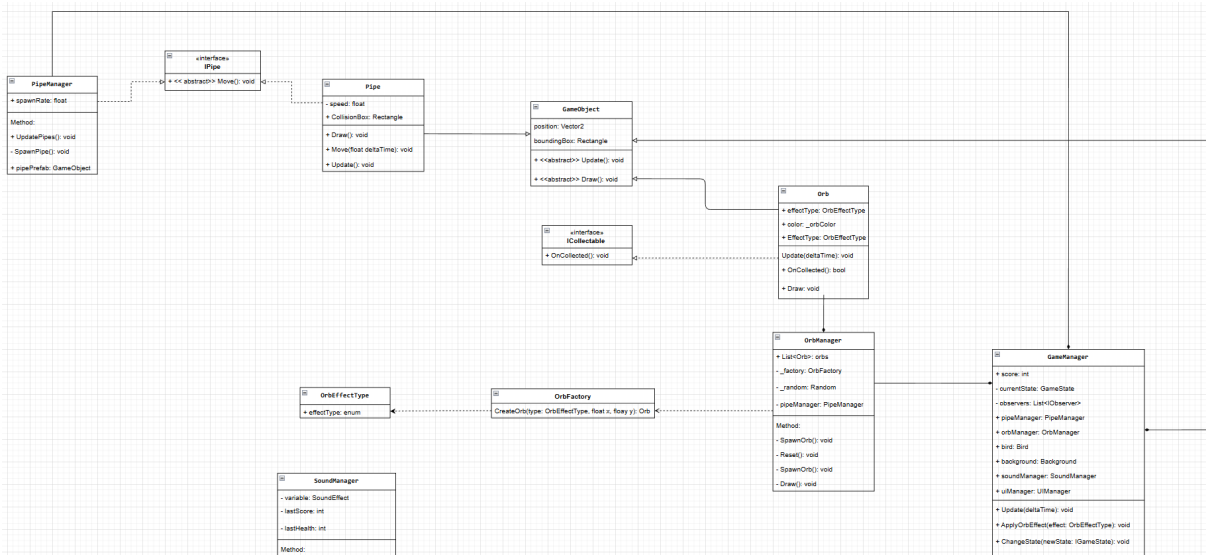
Option 4 Class Diagram:





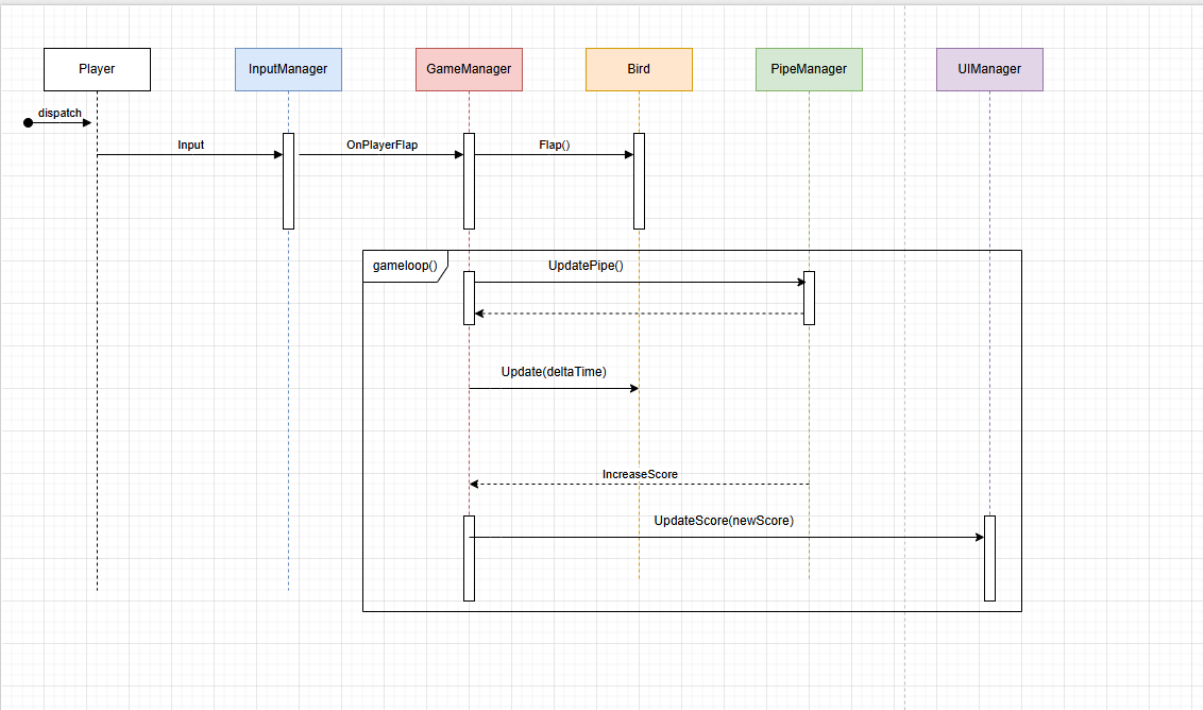
Final Diagram



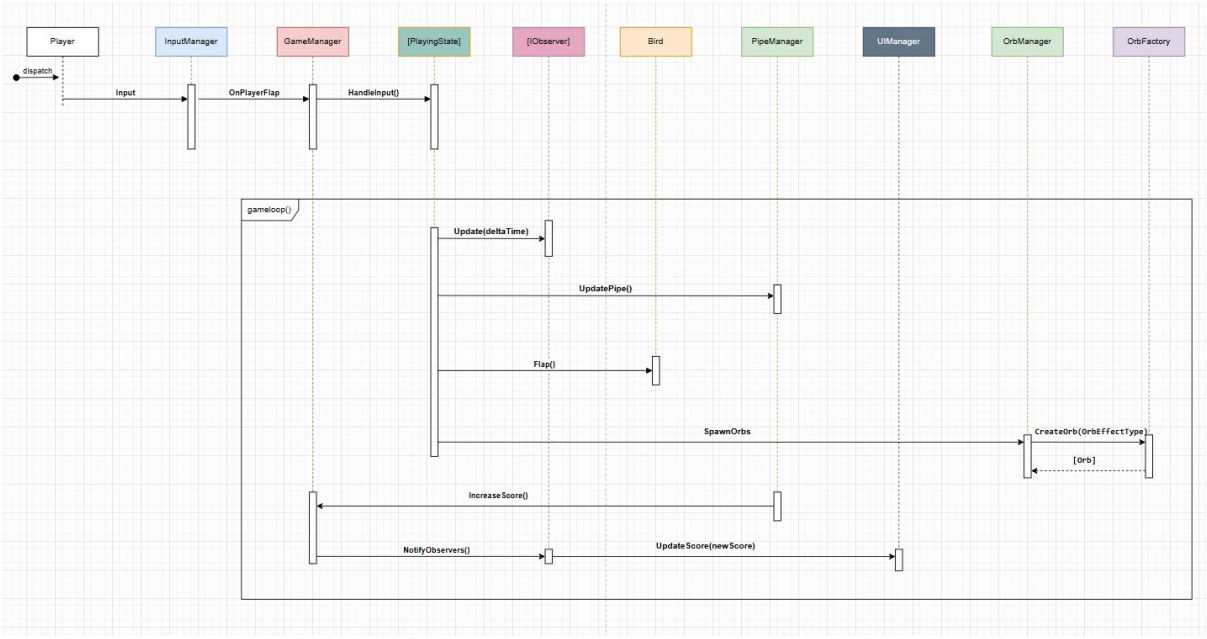

PipeManager
+ spawnRate: float
Method:
+ UpdatePipes(): void
- SpawnPipe(): void
+ pipePrefab: GameObject
«interface»
IPipe
+ << abstract>> Move(): void
Pipe
- speed: float
+ CollisionBox: Rectangle
+ Draw(): void
+ Move(float deltaTime): void
+ Update(): void
GameObject
position: Vector2
boundingBox: Rectangle
+ <<abstract>> Update(): void
+ <<abstract>> Draw(): void
Orb
+ effectType: OrbEffectType
+ color: _orbColor
+ EffectType: OrbEffectType
Update(deltaTime): void
+ OnCollected(): bool
+ Draw: void
«interface»
ICollectable
+ OnCollected(): void
OrbManager
+ List<Orb>: orbs
- _factory: OrbFactory
- _random: Random
- pipeManager: PipeManager
Method:
- SpawnOrb(): void
- Reset(): void
- SpawnOrb(): void
- Draw(): void
GameManager
+ score: int
- currentState: GameState
- observers: List<IObserver>
+ pipeManager: PipeManager
+ orbManager: OrbManager
+ bird: Bird
+ background: Background
+ soundManager: SoundManager
+ uiManager: UIManager
+ Update(deltaTime): void
+ ApplyOrbEffect(effect: OrbEffectType): void
+ ChangeState(newState: IGameState): void
OrbEffectType
+ effectType: enum
OrbFactory
CreateOrb(type: OrbEffectType, float x, floay y): Orb
SoundManager
- variable: SoundEffect
- lastScore: int
- lastHealth: int
Method:

Sequence Diagram

Option3:



Option 4:



By applying these design patterns, I can address Tight Coupling, Single Responsibility Principle (SRP) violation, and the lack of extensibility due to Open/Closed Principle violation that existed on Option 3.

State Pattern:

- ➔ Resolve the complexity of if/else statement blocks within the **GameManager**
- ➔ Separating the game logic by dividing distinct classes that implement the IgameState interface, including (Ready, Pause, GameOver).

Benefits when using State Pattern:

- ➔ In option 3, when adding a new state like PauseState requires modifying the whole GameManger code, triggering Open/Closed Principle. In option 4, adding a new state only requires creating a new class without affecting **GameManager** code

Observer Pattern:

- ➔ Eliminating Tight Coupling between GameManager and UIManager
- ➔ It establishes a notification between **ISubject** and **Iobserver. GameManager** acting as a subject does not call UIManager directly. Instead, it calls NotifyObservers() to announce score changes.

Benefits when using State Pattern:

- ➔ In option 3, GameManager is directly dependent on UIManager, violiating Open/Closed Principle. In option 4, when adding a new class that notifiying an events such as AchievementManager, SoundManager. We only need to register that class as an Observer without impacting GameManager. As a result, it reduces dependency and increases the system's modularity.

Factory Pattern:

- ➔ Upholds the Single Responsibility Principle (SRP) within **OrbManager**.
- ➔ By separating complex object creation and configuration logic for Orbs from OrbManager. The **OrbFactory** class takes the responsibility of creating the Orb object based on the required OrbEffectType.

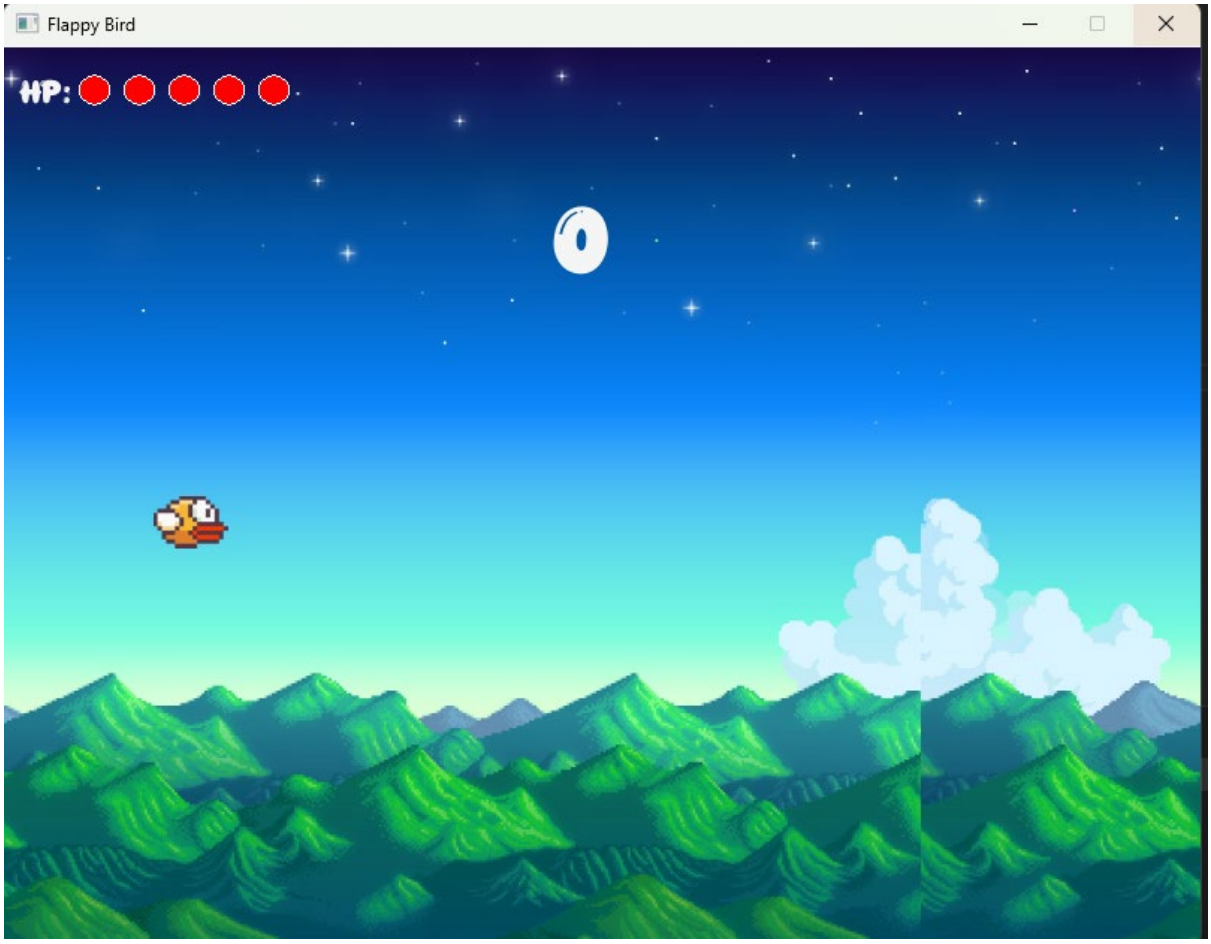
Benefits when using Factory Pattern:

- ➔ In Option 3, OrbManager handles both the spawning logic and the creation logic. In option 4, OrbManager focuses on spawning, while OrbFactory handles creation. As a result, when we are adding a new Orb effects type, only the OrbFactory code needs to be modified, leaving OrbManager code modifying.

Game Overview:

Ready State:

- You will see the bird, background, score, health, and music behind. At this stage the background will move from right to left. When player press space it will change to playing state.



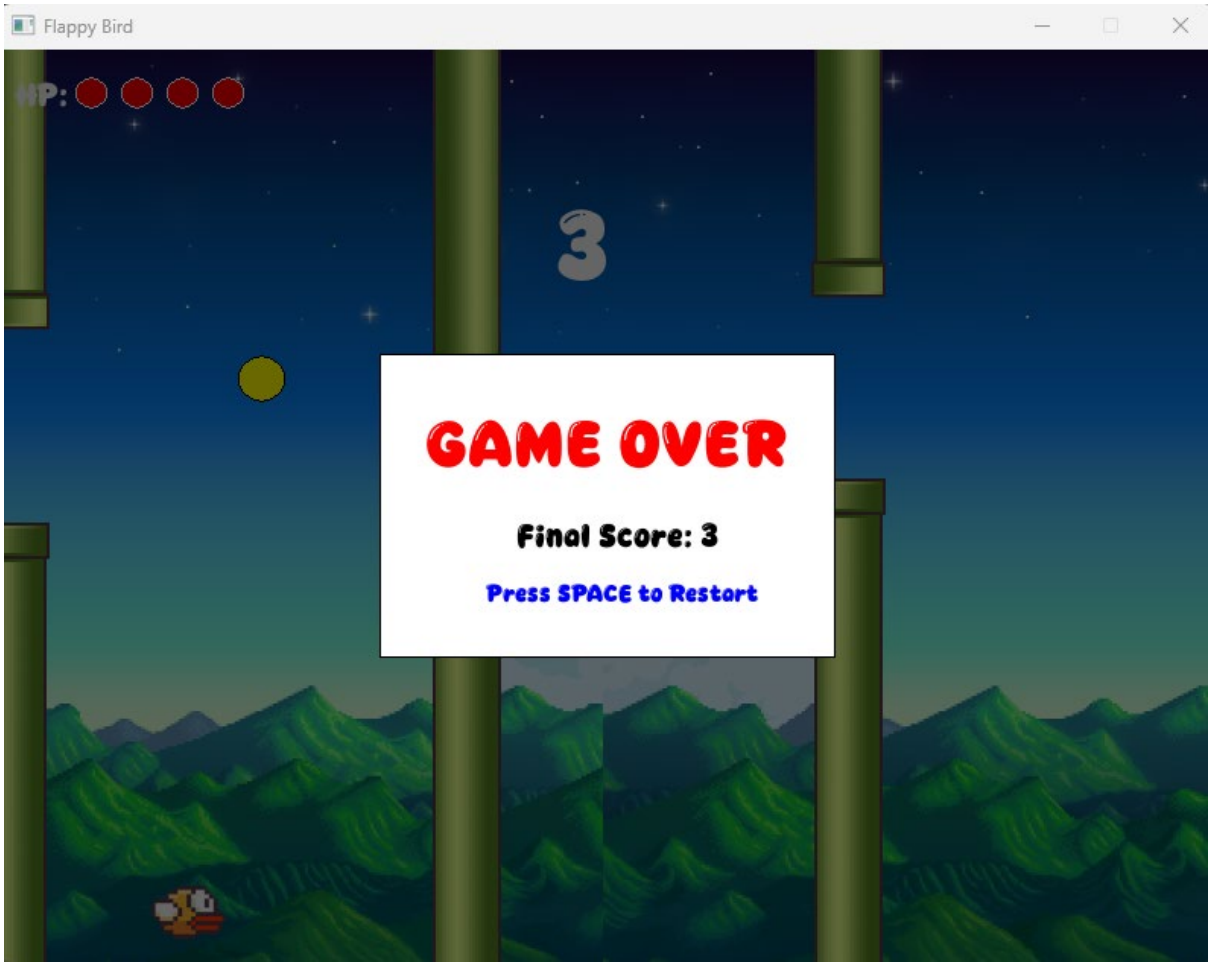
Playing State:

- At this stage when user use space the bird will jump, notice the bird will fall fastly upon the timer is counting. Also there are 4 pipes spawn respectively, these move from right to left. Also sometimes there will be a chance that dropping orb

which are Health, Immune, and Point.

GameOver State:

- When the bird is drop out the screen or there are no more health left, a Game Over screen will be popped up, it will print the total scores. When player press space again it will return to playing state.



I have used three Design Patterns:

- State Pattern
  Reference: https://refactoring.guru/design-patterns/factory-method

- Observer Pattern
  Reference: https://refactoring.guru/design-patterns/observer

- Simple Factory Pattern
  Reference: https://refactoring.guru/design-patterns/state