PAC-SNAKE

Technical Design Document

Version: 1.0
Date: September 10, 2024
Lead Designer, Developer: Haruki Uda

Haruki Uda

Table of Contents

1. Introduction

This Technical Design Document (TDD) outlines the architecture, design patterns, and algorithms that will be used to implement Pac-Snake. The document includes descriptions of the logic behind game mechanics, technical algorithms, debugging tools, and a set of testing criteria. Pac-Snake combines mechanics from Pac-Man and Snake, and this TDD details how these mechanics will be realized through code.

---

2. Technical Overview

Platform: Pac-Snake is developed using the Unity engine (C#), targeting PC and Mobile platforms (iOS and Android). Unity's built-in physics engine, asset management, and 2D toolsets will be leveraged for fast and efficient development.

Primary Systems:

1. **Player Control System**: Manages player movement, body growth, and collision detection.
2. **Ghost AI System**: Controls the behavior of enemy ghosts, pathfinding, and reactions to player power-ups.
3. **Collision Detection System**: Handles player-ghost, player-body, and player-wall interactions.
4. **Growth and Power-Up Systems**: Extend the player's body when ghosts are eaten and manage the temporary effects of power-ups.
5. **HUD and Score Management**: Tracks the player's score, power-up timers, and body length.

---

3. Logic and Technical Algorithms

Player Movement

- **Input Mapping**: On PC, arrow keys or WASD will be mapped for movement. On mobile, swipe gestures will trigger directional changes.
- **Smooth Movement**: The player moves grid-based, but transitions smoothly between grid cells to give a continuous movement effect.
- **Algorithm**:

```
Vector2 direction;

if (Input.GetKey(KeyCode.UpArrow)) direction = Vector2.up;

if (Input.GetKey(KeyCode.DownArrow)) direction = Vector2.down;

if (Input.GetKey(KeyCode.LeftArrow)) direction = Vector2.left;

if (Input.GetKey(KeyCode.RightArrow)) direction = Vector2.right;

transform.position += direction * moveSpeed * Time.deltaTime;
```

Ghost AI

- **Pathfinding**: Ghosts use a simplified form of A* or BFS to track the player character, adjusting their path dynamically based on the player's position.
- **Avoidance**: When a power-up is active, ghosts will switch to a state of fleeing, moving randomly to avoid the player.
- **Algorithm**:

```
if (powerUpActive)

  MoveAwayFromPlayer();

else

  MoveTowardsPlayer();
```

Collision Detection

- **Player-to-Wall Collision**: When the player hits a wall, movement in that direction is blocked. Collision detection will use Unity's 2D Collider system with basic checks:

```
if (Physics2D.OverlapCircle(transform.position + direction, collisionRadius))

  return; // Don't move if collision detected.
```

- **Player-to-Body Collision**: The player dies if their face collides with their own body. A linked list tracks the positions of body segments, and a simple collision detection compares the player's head position with each body segment.

```
foreach (Vector2 segment in bodySegments)

{

  if (headPosition == segment)
```

```
    PlayerDeath();

}
```

Growth Mechanic

- **Body Segmentation**: Each time the player eats a ghost, a new segment is added to the player's body. The body segments follow the head's previous positions, stored in a queue.

```
void AddBodySegment()

{

  Vector2 newSegment = bodySegments.Last(); // Get last segment position

  bodySegments.Enqueue(newSegment); // Add a new segment at the tail.

}
```

Power-Up Mechanic

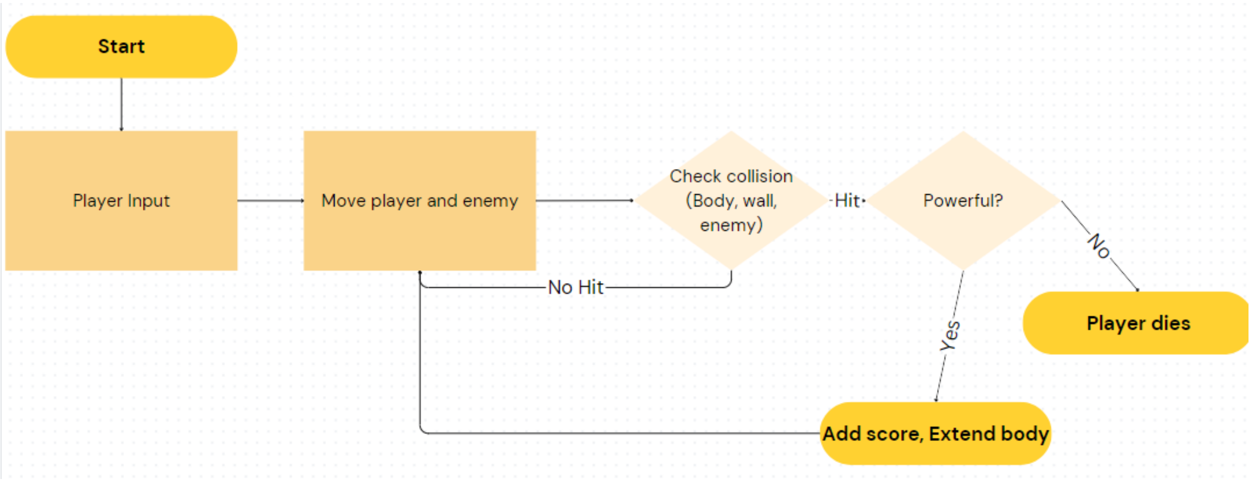
- **Power-Up Duration**: The power-up effect lasts for a predefined time (10 seconds). A coroutine handles the power-up duration and reverts the player's state after the timer expires.

```
IEnumerator PowerUpTimer()

{

  powerUpActive = true;

  yield return new WaitForSeconds(10);

  powerUpActive = false;

}
```

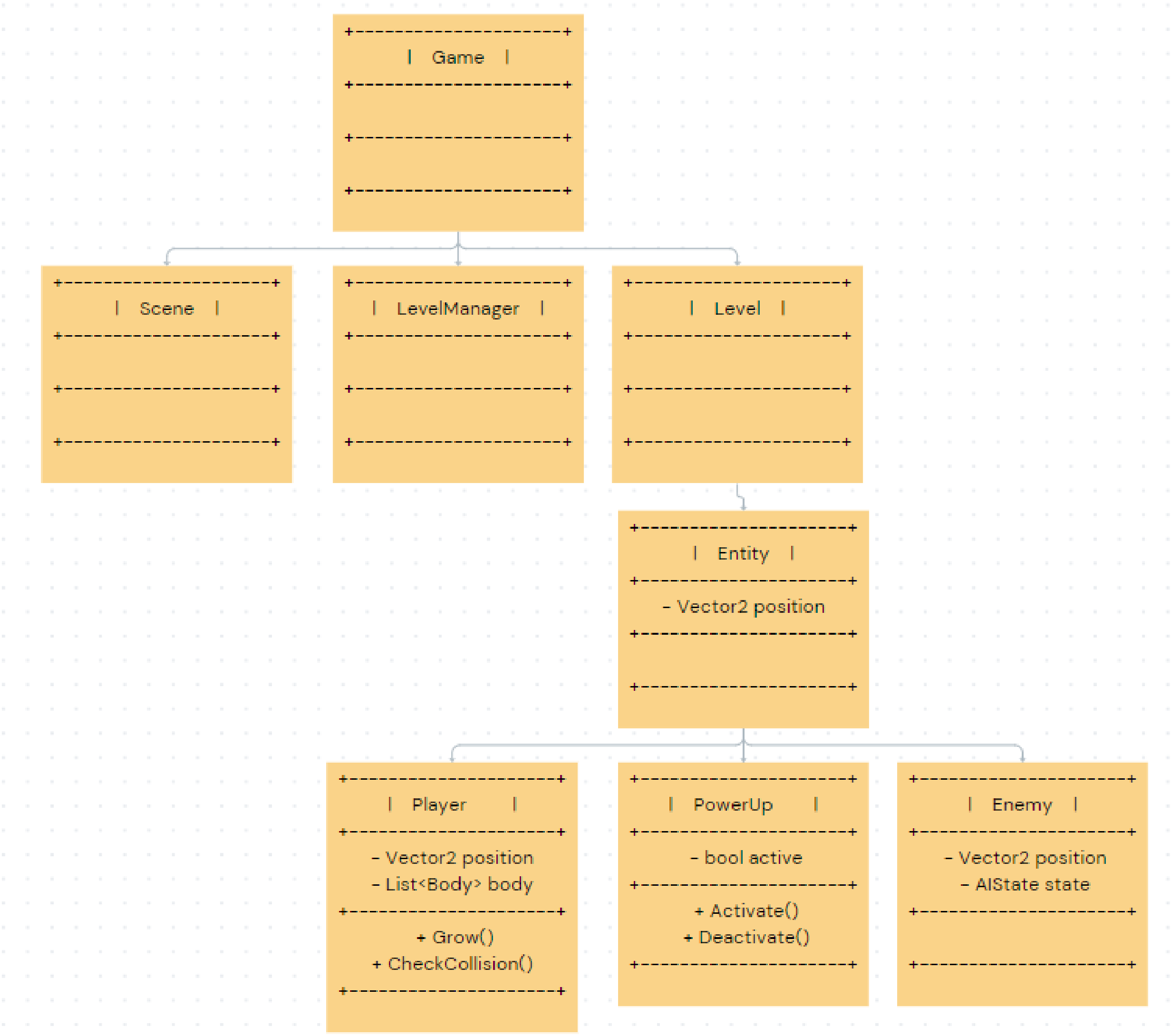
---

4. Flowchart and UML Diagrams

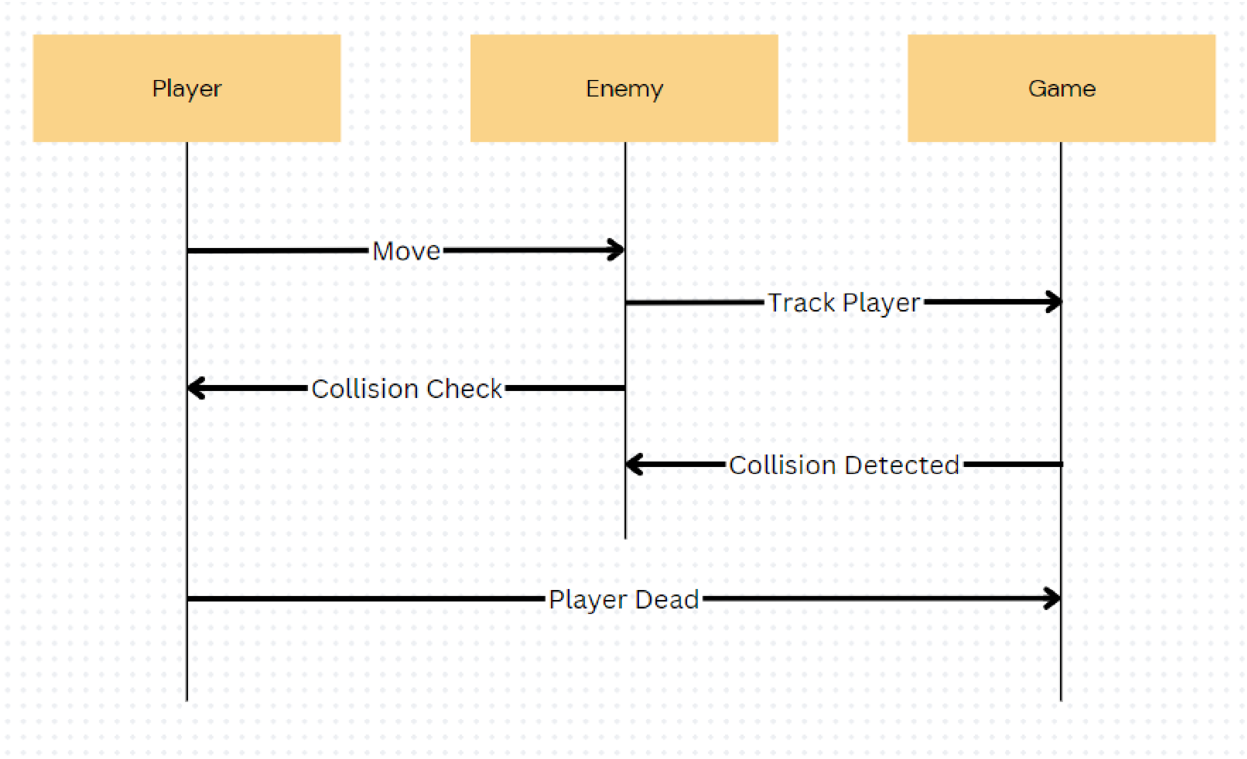
Flowchart of Game Mechanics



UML Class Diagram



Sequence Diagram (Player Collision with Ghost)



5. Debug Features

In-Game Debug Mode:

- **Show Colliders**: A debug option to visualize player and ghost colliders in real-time, useful for collision detection troubleshooting.
- **Ghost AI Path**: Shows the intended path of ghosts during their movement, allowing developers to track AI behavior.
- **Manual Power-Up Activation**: Toggle power-ups on and off for testing purposes.
- **Invincibility Toggle**: Turn on invincibility to test levels without worrying about collisions.

6. Coding Standards and Naming Schemes

- **Variables**: camelCase for local variables and ALL_CAPS for constants.
- **Classes**: Each class represents a game entity (e.g., Player, Ghost, PowerUp) with descriptive names.

---

7. File Formats

- **Sprites**: .png files for 2D sprites (player, ghost, coins).
- **Sound Effects**: .wav or .ogg for sound effects like coin collection and power-up activation.
- **Music**: Background music stored in .mp3 format.

---

8. Acceptance Test Plan

1. Can the player move in all four directions (up, down, left, right)?
2. Are ghosts correctly avoiding the player when a power-up is active?
3. Does the player’s body grow when a ghost is consumed after a power-up?
4. Are coins and power-ups respawning every 10 seconds?
5. Is the player’s face the only vulnerable part for collision with ghosts?
6. Do power-ups last exactly 10 seconds before deactivating?
7. Does the player die if their face collides with their own body?
8. Are HUD elements (score, power-up timer, body length) correctly updating in real-time?
9. Does the game return to the main menu when the player dies?
10. Are all sprites (player, ghosts, coins, power-ups) rendering correctly on the screen?