Game Dev Project Report

1. Description of the Game

The 2D platformer game tasks the player with surviving **60 seconds** per level. Each time they survive, the game bumps up difficulty by adjusting **spawn rates** and **enemy health**. Collectibles like coins and healing potions help manage resources, while a medieval-fantasy art style sets the overall tone. Player death triggers a *Game Over* screen, with the option to restart or quit. As I built the game, I found it crucial to integrate an **orchestration** system where the **GameManager** and supportive managers (UI, Audio, ObjectPool) each handle specific functions, making the codebase cleaner

---

2. Preliminary Setup and Asset Choices

Asset Selection (All from Unity Asset Store) - *Hero Knight (Free)*: I wanted a sprite that already had animations for attack, hurt, idle, etc. This saved me time since I didn't have to create animations from scratch. - *Pixel Art Enemies & Tilemap*: I started with a free pixel mob asset, then expanded it. For my ground tiles, I used pixel art day/grass blocks from the Unity Asset Store. - *Collectibles (Potion & Gold)*: I leveraged a 2D RPG treasure collection pack. Only the coin (gold) and potion assets were used initially, but the system is flexible for future expansions. - *Platformer Music Pack Lite & Sound Effects*: I grabbed a free background track for the main theme and coin/potion pickup sounds for a more polished feel.

---

3. High-Level Architecture

3.1 Manager Classes

1. **GameManager**
    - Manages global **GameState**: *MainMenu, Playing, Paused, GameOver.*
    - Tracks `coins`, `score`, `enemiesKilled`, and handles transitions like `PauseGame()`, `ResumeGame()` or `StartNextLevel()`.
    - *I realized* centralizing state logic in a single class simplified scene changes but risked creating a super-object. I tried to keep it lean by delegating UI and audio tasks to **UIManager** and **AudioManager** respectively.

```
public enum GameState
{
    MainMenu,
    Playing,
    Paused,
    GameOver
}
```

    I used an enum for easy checks—like if (currentState == GameState.Playing) etc throughout to orchestrate the game flow.

2. **UIManager**
    - Updates text (timer, score, coins) and displays damage/heal popups.
    - Shows/hides **Pause Menu** and **Game Over** screens.
    - *I discovered* references to UI elements can break if you load multiple scenes, so I used `DontDestroyOnLoad(gameObject)` in `Awake()` [1], ensuring the UI persists across scene changes.

3. **AudioManager**
    - Holds background music and SFX in a dictionary for easy `PlaySFX("clipName")`. New clips can be added without changing the code, just by dragging in elements in the Unity Editor.

4. **ObjectPool**
    - Reuses pooled objects (enemies, collectibles) to avoid repeated `Instantiate()` calls.
    - *I had to ensure each object was clean before re-pooling it. This was done by calling the reset enemy which changes up enemy properties to default values, as if a new gameobject was instantiated*

---

4. Challenges and How I Overcame Them

4.1 Object Pooling Performance **Problem**: I was getting poor performance **Solution**: I implemented `ObjectPool.Instance.SpawnFromPool(...)`. Rather than `Instantiate` new enemies, I re-use existing ones from a queue, then move them to new positions.I took inspiration from the lecture notes and discussions regarding this topic

```
public GameObject SpawnFromPool(string tag, Vector3 position, Quaternion rotation)
{
    if (!poolDictionary.ContainsKey(tag))
    {
        Debug.LogWarning($"Pool with tag {tag} doesn't exist");
        return null;
    }

    var pool = poolDictionary[tag];
    if (pool.Count == 0)
    {
        // dynamically create a new object if queue is empty
    }

    GameObject objectToSpawn = pool.Dequeue();
    objectToSpawn.SetActive(true);
    objectToSpawn.transform.position = position;
    objectToSpawn.transform.rotation = rotation;

    // Add it back to queue so it can be reused next time
    pool.Enqueue(objectToSpawn);

    return objectToSpawn;
}
```

4.2 State Management & Level Progression **Problem**: Each **60-second** wave ended in abrupt level transitions. I needed to ensure all spawners paused and existing enemies were either cleared or re-initialized when a new wave began. **Solution**: *In `GameManager.cs`, I track a countdown (`remainingTime`) and call `StartNextLevel()` once it hits zero. Then, I forcibly remove all enemies from the field, awarding players a fresh start.*

```
// [1] GameManager.cs
private void Update()
{
    if (currentState == GameState.Playing)
    {
        remainingTime -= Time.deltaTime;
        if (remainingTime <= 0)
```

```
        {
            StartNextLevel();
            return;
        }
        UIManager.Instance.UpdateTimer(remainingTime);
    }
}
```

4.3 Weighted Probability & Raycasting in Spawners **Problem**: I needed to **randomly spawn** different enemy or collectible types, but not always at the same rate. I also didn't want them appearing inside walls. **Solution**: *I implemented a* ***weighted probability*** *approach in* ***EnemySpawner*** *and* ***CollectibleSpawner****. Each prefab has a 'spawnWeight,' which influences how frequently it appears. For valid spawn positions, I do a downward raycast from above the potential point to detect ground.*

```
// EnemySpawner.cs
private GameObject SelectEnemyToSpawn()
{
    float totalWeight = 0f;
    foreach (var enemy in enemies)
        totalWeight += enemy.spawnWeight;

    float random = Random.Range(0f, totalWeight);
    float weightSum = 0f;
    foreach (var enemy in enemies)
    {
        weightSum += enemy.spawnWeight;
        if (random <= weightSum)
            return enemy.prefab;
    }
    return enemies[0].prefab;
}
```

Debugging the raycast was tricky. Sometimes the spawn point was at the edge of a platform. I loop up to 30 times, searching for a valid point—if none is found, I spawn the object near the player or log a warning.

```
// EnemySpawner.cs
private Vector2 GetSpawnPoint()
    {
        if (player == null) return Vector2.zero;

        for (int i = 0; i < 30; i++)
        {
            float angle = Random.Range(0f, 360f);
            Vector2 direction = Quaternion.Euler(0, 0, angle) * Vector2.right;
            float distance = Random.Range(minSpawnRadius, maxSpawnRadius);
            Vector2 potentialPoint = (Vector2)player.position + direction * distance;

            Vector2 rayStart = potentialPoint + Vector2.up * 10f;
            RaycastHit2D hit = Physics2D.Raycast(rayStart, Vector2.down, 20f, LayerMask.GetMask("Ground

            if (hit.collider != null)
            {
                return hit.point + Vector2.up * 1f; // adjusted spawn offset above ground
            }
```

```
        }

        Debug.Log("could not find a valid enemy spawn point.");
        return player.position + Vector3.up * 3f;
    }
```

4.4 Level Progression **Problem**: Every 60-second wave ended abruptly, and old enemies sometimes remained on screen. I also needed a neat transition to the next level. **Solution**: 1. Tracked a countdown in `GameManager.Update()`. When it hits zero, I call `StartNextLevel()`. 2. In `StartNextLevel()`, I loop through all active enemies, dealing them `int.MaxValue` damage to push them back into the pool. 3. The `LevelSystem.OnLevelStart()` method ramps up difficulty by lowering spawn intervals and boosting enemy health multipliers.

```
// LevelSystem.cs
public void OnLevelStart()
{
    float spawnMultiplier = 1 + (currentLevel * 0.2f);
    float healthMultiplier = 1 + (currentLevel * 0.1f);

    enemySpawner.ReduceSpawnInterval(spawnMultiplier);
    enemySpawner.IncreaseEnemyHealth(healthMultiplier);

    Debug.Log($"level {currentLevel} started with spawn x{spawnMultiplier}, health x{healthMultiplier}"
}
```

This design gave me a consistent difficulty curve: each new level spawns slightly faster and stronger enemies, making survival progressively harder.

4.5 UI Reliability & Scene Transitions **Problem**: Reloading or switching scenes broke UI references—panels, text elements, and popups sometimes vanished or used stale references. **Solution**: - *I used a* ***singleton*** *pattern with* `UIManager`*, along with* `DontDestroyOnLoad(gameObject)` *in* `Awake()`*. This ensures a single, persistent UI across scenes. Each time a new scene loads, UI elements remain intact. - Additionally, I carefully* ***delegated*** *show/hide logic for Pauses, Game Over, etc. so that* `GameManager` *just tells* `UIManager` *to display or hide the relevant panel. This method was also applied to the canvas passed in via a PersistentCanvas script*

4.6 Animations & Combat Collisions **Problem**: Attacks needed precise collision checks so enemies would be hit at the correct animation frames. **Solution**: - *In* `PlayerController.ExecuteAttack()`*, I used an OverlapCircle in front of the player. The offset depends on the player's facing direction,* `isFacingRight`*. - Each new attack triggers an animation parameter, e.g.,* `Attack1`*,* `Attack2`*,* `Attack3` *for combos. While basic, this was enough to chain combos—especially if I wanted to integrate the* `ComboSystem.cs` *fully.*

```
// PlayerController.cs
private void ExecuteAttack()
{
    animator.SetTrigger($"Attack{currentCombo}");
    AudioManager.Instance.PlaySFX("sword");
    Vector2 attackPosition = transform.position;

    // Offset the OverlapCircle to the correct side
```

```
    if (!isFacingRight)
        attackPosition -= Vector2.right * attackRange;
    else
        attackPosition += Vector2.right * attackRange;

    var hitEnemies = Physics2D.OverlapCircleAll(attackPosition, attackRange, LayerMask.GetMask("Enemy"))
    // etc
}
```

5. Additional Features & Suggestions

1. **ComboSystem**
   - *I started drafting a `ComboSystem.cs`, which detects sequential key presses (e.g., Attack -> Attack -> Attack in time) for special damage. If I expand combat, implementing this would add deeper fighting mechanics.*
   - Potential pitfalls:
     - *Input buffering and timing windows can be tricky.*
     - *Ensuring the correct animation or damage multiplier triggers only within a short input window.*
2. **PowerUpSystem**
   - Currently commented out code. I wanted to add timed power-ups like `GiantPowerUp` and `GhostModePowerUp`. In my coroutine, I tested scaling the player to 2.0x size for 10 seconds.
   - Future expansions could include:
     - Stacking multiple power-ups.
     - Visual effects (e.g., a glow or aura).
     - UI indicators for remaining power-up time.
3. **Skilltree**
   - *I considered a skill tree system where players could spend coins on permanent upgrades like health, attack power, or speed. This would add a layer of progression beyond just surviving waves, and an actual utility to the coins other than a metric of progress*