Newcastle University

CSC2034

Project Report

MAY 2022
Cameron Main
200422552

Contents

Abstract

This project details exactly how I implemented my own Pac-Man style game within the Unity game engine. After being provided with a basic ‘starter-kit’ scene containing limited scripts, I was presented with the task of finishing the game with the following feature requirements:

Task

The Mellow Yellow Fellow game is a currently 'unfinished' replica of an existing game. Using the provided Unity project as a starting point, enhance the game to include multiple new features:

- The Fellow and the Ghosts should be able to use the tunnel at the sides of the game arena to 'teleport' from side to side.
- Ghosts should return to the centre of the maze to be restored after having been 'eaten' by a powered-up Fellow.
- The player should have multiple lives, with the game resetting the location of the Fellow and Ghosts at the start of each 'life'.
- Rather than ending, the game should reset, allowing the player to complete a new 'level'.
- The game should have a user interface, informing the player of the current score, best high score, life count, current level, and so on.
- New high scores should be managed, so that they can be saved for later plays of the game.

Figure 1 Project Task Requirements

I was able to implement each of these requirements successfully in my game. Furthermore, I expanded upon the base requirements and opted to complete some further extension tasks. These tasks included the creation of a minigame, adding a stylised theme to the game and giving the ghost characters distinct personalities.

What Was Done & How

Character Teleportation

I began with the first requirement of enabling “teleportation” from one side of the level to the other. The “teleporting” was simply a case of changing the transform properties of an object. Two box colliders were used with an OnTriggerEnter() function so when the player or a ghost enters, they move from one side to the other while maintaining their velocity.

```
[SerializeField]
Transform destination;

Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Fellow"))
    {
        Vector3 position = destination.position;
        position.z = other.transform.position.z;
        other.transform.position = position;
    }
    else if (other.CompareTag("Ghost"))
    {
        // Disable NavMeshAgent for accurate teleport
        other.gameObject.GetComponent<NavMeshAgent>().enabled = false;

        Vector3 position = destination.position;
        position.z = other.transform.position.z;
        other.transform.position = position;

        other.gameObject.GetComponent<NavMeshAgent>().enabled = true;
    }

}
```

Figure 2 Code Snippet: Character Teleportation

As figure 2 shows, the Teleport.cs script determines whether it is the player teleporting or a ghost. I originally did not make this distinction and attempted to transform both in the same manner. However, since the ghosts are NavMeshAgents, a bug occurs where they do not arrive at the specified location after a transform. Therefore, the ghosts have the NavMeshAgent component temporarily disabled during the transform, then enabled again after.

Returning Ghosts

Next to be implemented:

- Ghosts should return to the centre of the maze to be restored after having been 'eaten' by a powered-up Fellow.

I originally implemented this by the same “teleportation” methods as previously mentioned. However, I preferred the mechanics of the original Pac-Man game where the ghosts slowly return to the ghost house while in a ‘death phase’. This creates a less jarring playing experience as objects are not popping in and out over the player’s screen.

```
public void GhostDied()
{
    isDead = true;
    //change to dead material
    GetComponent<Renderer>().material = deadMaterial;

    //allow player to pass through ghost
    Physics.IgnoreCollision(GetComponent<CapsuleCollider>(),
        player.GetComponent<SphereCollider>(), true);

    //speed up phase
    agent.speed = 8f;
    agent.acceleration = 10f;
}


2 references
public bool HasRespawned()
{
    return respawned;
}

Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (isDead && other.gameObject.CompareTag("GhostHouse"))
    {
        isDead = false;
        hiding = false;
        respawned = true;
        GetComponent<Renderer>().material = normalMaterial;

        //reset speed values
        agent.speed = startSpeed;
        agent.acceleration = 8f;


        //enable collision
        Physics.IgnoreCollision(GetComponent<CapsuleCollider>(),
            player.GetComponent<SphereCollider>(), false);
    }
}
```

Figure 3 Code Snippet: Returning Ghosts

Figure 3 demonstrates how this was achieved. In conjunction with a GhostManager to serve as an interface, each of the ghosts are now able to have multiple distinct states. They can be

alive (false isDead boolean), dead, hiding or respawned with the above logic determining when each can enter the next state.

Breaking the game down into having player and ghost managers to serve as an interface between them and the game object was centrally important to creating a modular game that I could easily build upon. For example, creating additional ghosts was simple as they could each inherit the same states but make small modifications to behaviour to develop the distinct personalities. Details of this regarding the first extension task are later discussed.

Player Lives

Another requirement for the game was:

- The player should have multiple lives, with the game resetting the location of the Fellow and Ghosts at the start of each 'life'.

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("Ghost"))
    {
        if (PowerupActive() && !collision.gameObject.GetComponent<GhostManager>().HasRespawned())
        {
            collision.gameObject.GetComponent<GhostManager>().GhostDied();
            score += 200;
            scoreText.GetComponent<TextMeshProUGUI>().text = score.ToString();
        }
        else
        {
            GameObject[] ghosts = GameObject.FindGameObjectsWithTag("Ghost");
            Vector3 ghosthouse;
            ghosthouse = new Vector3(7.5f, 0.1f, 6.5f);

            foreach (GameObject ghost in ghosts)
            {
                ghost.transform.position = ghosthouse;
            }

            Debug.Log("Player hit ghost");
            gameObject.transform.position = startPos;


            lives--;

            livesUI[lives].SetActive(false);

            if (lives <= 0)
            {
                gameObject.SetActive(false);
                game.StartGameOver();

            }
        }

    }
}
```

Figure 4 Code Snippet: Player Lives

Figure 4 shows code derived from the Fellow script and illustrates the logic that occurs when the player collides with a ghost. If the player does not currently have a powerup, a life is lost and the player is repositioned to their start position. The final if statement determines when the game lose condition for the player and runs the game over UI screen.

Implementing UI

It was at this point in the project I began to implement my own UI system so the player's lives could be displayed on screen. The UI already in the project was rather basic and used Unity Legacy text that looked too blurry. Upon searching the Unity Manual, I discovered TextMeshPro (TMP). The standard in quality between the two is exemplary so opted to use TMP for the rest of my project. Figure 5 shows a comparison of the two.

Figure 5 Screenshot: Comparison of Default and TextMeshPro UI

Despite this not being a specified requirement, having clear and legible text not only improves legibility, but also facilitates a higher standard of quality and adds polish to the game. From this, I have now developed a proficiency in creating UI with TMP and confidently created the remaining UI for the game. This included a screen for the main menu, pausing, game over, restart, save score and leader board.

Before moving onto these however, I created the heads-up display (HUD) that overlays on the game to display the players current score, the all-time high score, the level and the number of lives remaining. As per the requirement of:

- The game should have a user interface, informing the player of the current score, best high score, life count, current level, and so on.

Figure 6 Screenshot: Labelled Illustration of In-Game UI

Also opting for the Roboto font for its large bold lettering and modern aesthetic. Each of the following UI screens were created with TMP and operate upon a Canvas with the 'Screen Space – Overlay' render mode.

Sticking with a consistent theme was a deliberate effort to adhere to one of Nielsen's ten usability heuristics for good UI design [1]. In conjunction with 'visibility of system status' by adding hover animations to buttons, the system provides immediate visual feedback to the user.

Collection of Game Screenshots

Figure 7 Screenshot: Main Menu

Figure 8 Screenshot: Leaderboard

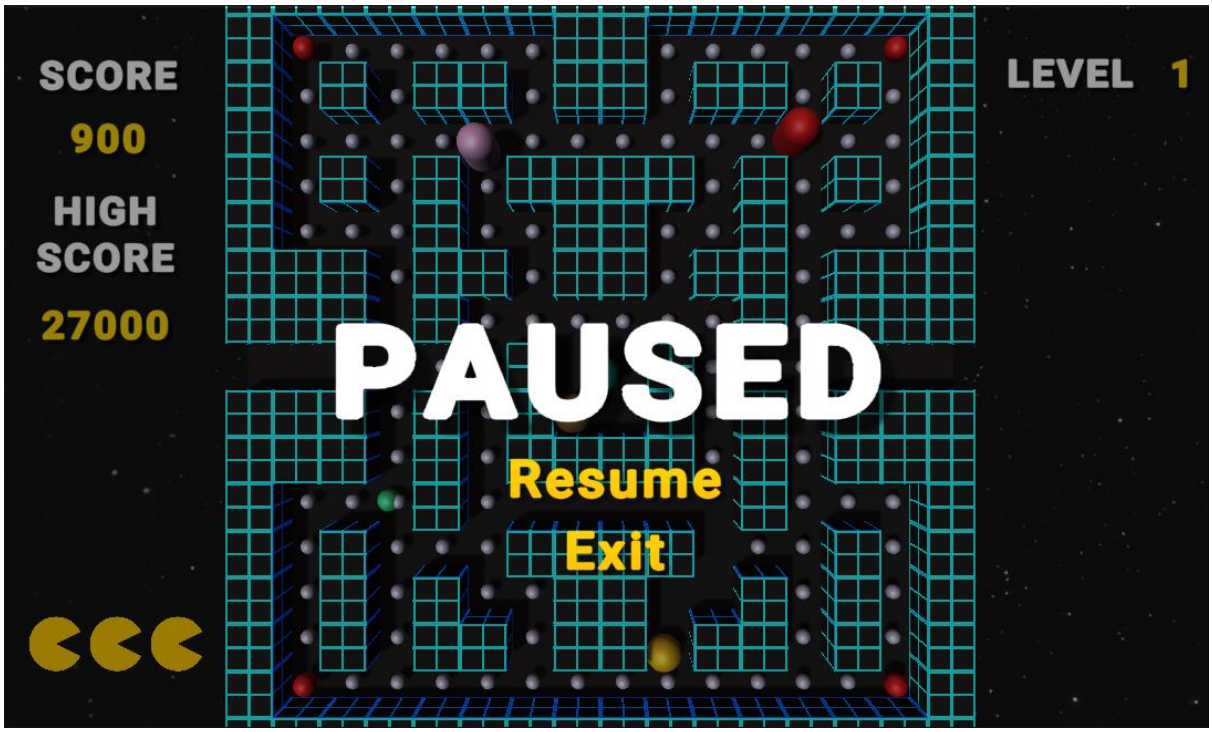

Figure 9 Screenshot: Pause Menu

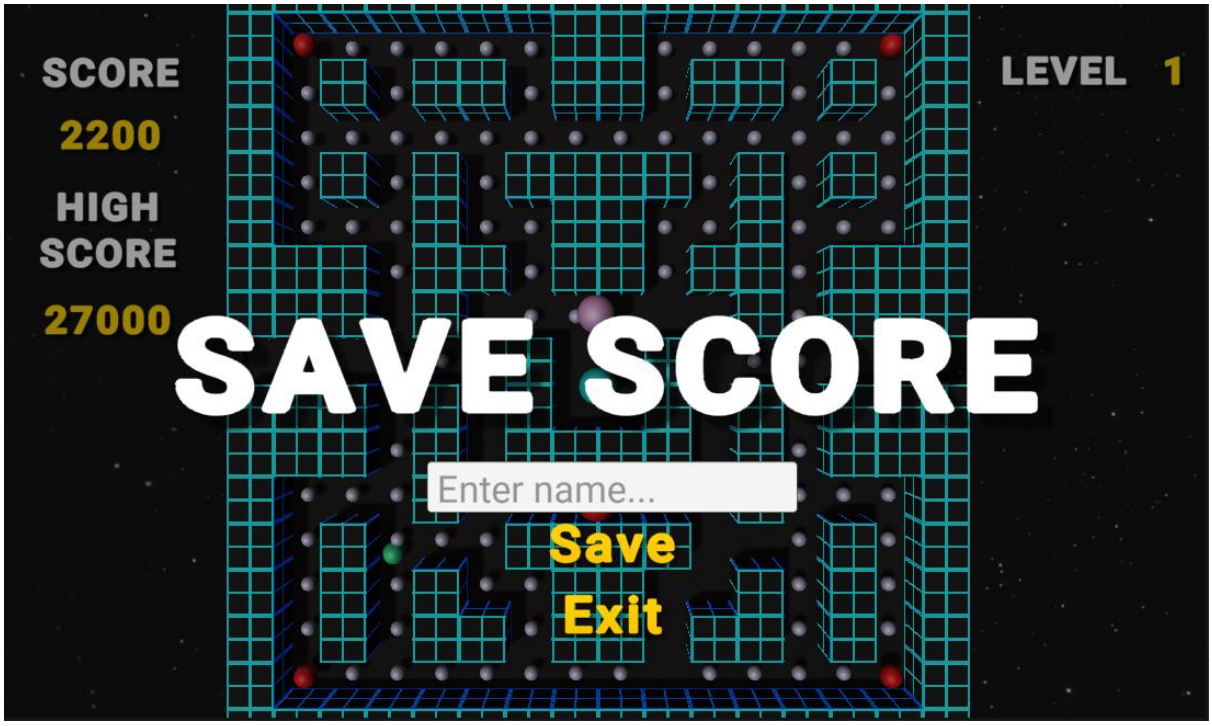

Figure 10 Screenshot: Save Score Menu

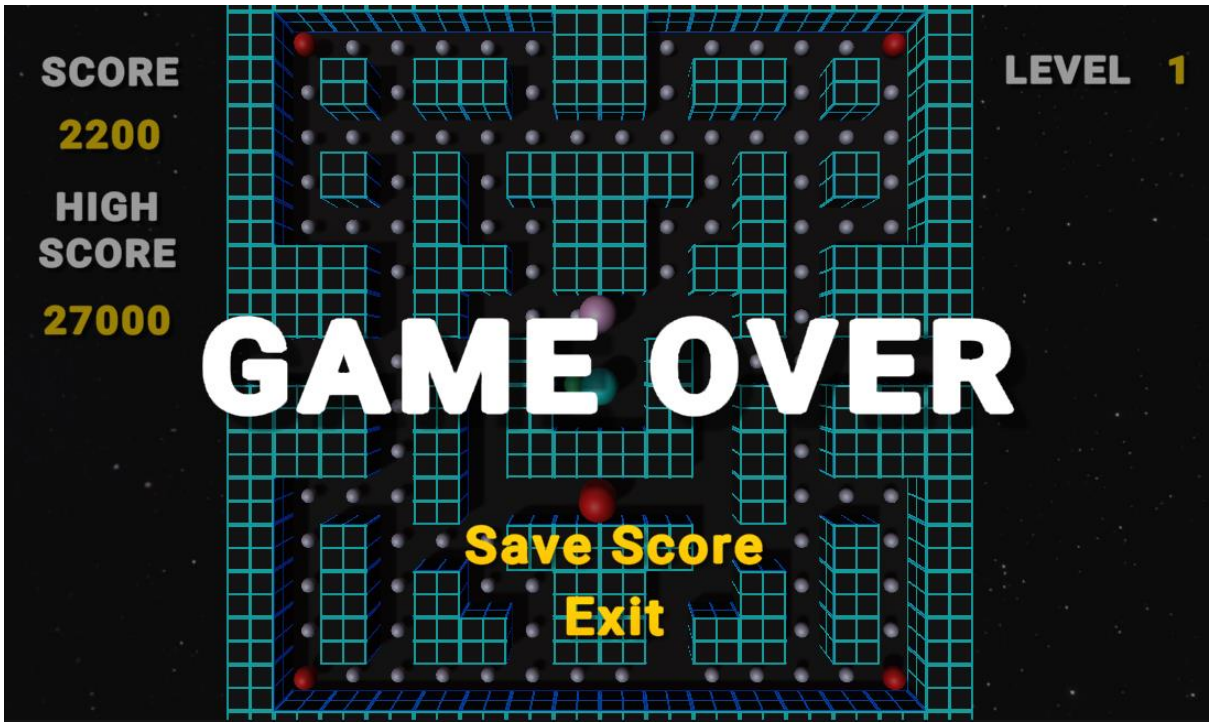

Figure 11 Screenshot: Game Over

UI Logic

The logic to determine which screen is displayed exists withing the main game update function of YellowMellowFellow.

```
switch(gameState)
{
    case GameState.MainMenu:    UpdateMainMenu(); break;
    case GameState.HighScores:  UpdateHighScores(); break;
    case GameState.InGame:      UpdateMainGame(); break;
    case GameState.Paused:      UpdatePauseMenu(); break;
}
```

Figure 12 Code Snippet: Game State Switch Case

Through the use of a switch statement, I broke the game down into further states to run separate functions depending on the current game state. Thus, whenever the game state is changed, the corresponding function is called where the correct UI can be displayed or logic executed.

For example, here is the functions for starting and enabling the Pause menu:

```
1 reference
public void StartPauseMenu()
{
    gameState = GameState.Paused; //change gamestate

    gameTrack.Pause(); //pause current song
    pauseUI.gameObject.SetActive(true); //display pause UI

    Time.timeScale = 0; //pause the game
}

1 reference
public void RemovePauseMenu()
{
    gameState = GameState.Playing; //change gamestate

    gameTrack.UnPause();  //resume music
    pauseUI.gameObject.SetActive(false); //remove pause UI

    Time.timeScale = 1; //resume gameplay
}

1 reference
void UpdatePauseMenu()
{
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            RemovePauseMenu();
        }

}
```

Figure 13 Code Snippet: Pause Menu Logic

Displaying & Saving Scores

The final requirement for the game was to have the ability to save scores:

- New high scores should be managed, so that they can be saved for later plays of the game.

I interpreted this as to have a system that writes scores to a local save file, allowing every score to be viewed in game as a player ordered on a leaderboard.

To begin with, I created a TMP Scroll View seen on figure 14 to house the leaderboard text. The text was read from the score file to the game, sorted by highest to lowest and displayed into the Scroll View.

```
1 reference
void CreateHighScoreText()
{
    for (int i = 0; i < allScores.Count; ++i)
    {

        GameObject o = new GameObject();
        o.transform.parent = transform;

        Text t = o.AddComponent<Text>();
        t.text = allScores[i].name + "\t\t" + allScores[i].score;
        t.font = scoreFont;
        t.fontSize = fontSize;

        o.transform.localPosition = new Vector3(0, -(i) * 6, 0);

        o.transform.localRotation = Quaternion.identity;

        o.GetComponent<RectTransform>().sizeDelta = new Vector2(400, 100);
    }
}

2 references
public void ReadScores()
{
    using (TextReader file = File.OpenText(highscoreFile))
    {
        string text = null;
        while ((text = file.ReadLine()) != null)
        {
            Debug.Log(text);
            string[] splits = text.Split(' ');
            HighScoreEntry entry;
            entry.name = splits[0];
            entry.score = int.Parse(splits[1]);
            allScores.Add(entry);
        }
    }
}

2 references
public void SortScores()
{
    //sort scores by highest to lowest
    allScores.Sort((x, y) => y.score.CompareTo(x.score));
}
```

Figure 14 Code Snippet: Read & Display Scores Logic

```
//write save to local file
1 reference
public void WriteSave(string name, int score)
{
    using StreamWriter file = new StreamWriter(highscoreFile, true);

    file.WriteLine(name + " " + score.ToString());
    file.Close();
}
```

Figure 15 Code Snippet: Saving Scores

The writing to the file is done via the WriteSave function that takes the name and score from the main game script, and writes a new line in the existing highscore file before closing it. As a result, saved scores are permanent after a player decides to save their score across multiple compiles.

Extension Tasks

My Pac-Man replica game was now in a somewhat finished state so I began work on some extension tasks. One such extension was that the game should include multiple ghosts of distinct “personalities”. Further reading of The Pac-Man Dossier by Jamey Pittman revealed the true behaviours of the Pac-Man ghost A.I.. Firstly, ghosts are in one of three mutually exclusive modes of behaviour during play: chase, scatter and frightened [2].

In chase mode, the ghost is given the task of attempting to find and ‘attack’ the player throughout the maze. Scatter mode causes them to dissert chasing the player to temporarily go to specified “home” corner locations in the maze before returning to chase mode again. Finally, ‘frightened’ occurs when the player consumes a powerup and the ghosts become vulnerable to the player and wander aimlessly.

Two of these modes were already implemented so only an additional scatter mode was required. I achieved a scatter mode adding empty game objects to the scene and using their transforms to act as checkpoints for each of the ghosts.

```
1 reference
public void GetScatterMarkers()
{
    int numMarkers = GameObject.Find("Maze/ScatterMarkers/Blinky").transform.childCount;
    scatterMarker = new Transform[numMarkers];
    for (int i = 0; i < numMarkers; i++)
    {
        scatterMarker.SetValue(GameObject.Find("Maze/ScatterMarkers/Blinky").transform.GetChild(i), i);
    }
}
```

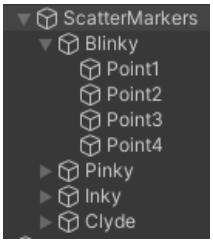

Figure 16 Code Snippet: Logic behind retrieving 'scatter markers'

Each time the level loads, these points are stored in an array for each ghost. Therefore, when in scatter mode, each one is sequentially selected as the A.I. agent’s target position and the ghost can loop them until the scatter timer expires and behaviour state changes.

```
if (agent.isActiveAndEnabled)
{
    if (isDead)
    {
        //return to ghost house if dead
        agent.destination = ghostHouse.transform.position;
    }
    else if (player.PowerupActive() && !isDead && !respawned)
    {
        Debug.Log("POWERUP ACTIVE");
        if (!hiding || agent.remainingDistance < 0.5f)
        {
            hiding = true;
            GetComponent<Renderer>().material = scaredMaterial;
            agent.destination = PickHidingPlace();
        }
    }
    else if (!player.PowerupActive() && respawned)
    {
        //powerup ended
        respawned = false;
    }
    else
    {
        //when hiding
        if (hiding)
        {
            GetComponent<Renderer>().material = normalMaterial;
            hiding = false;
        }

        //when in scatter mode
        if (scatterTime > 0.0f)
        {
            //revert to scatter points
            if (!agent.pathPending && agent.remainingDistance < 0.3f)
            {
                destPoint = (destPoint + 1) % scatterMarker.Length;
                agent.destination = scatterMarker[destPoint].position;
                GetComponent<Renderer>().material = normalMaterial;
                hiding = false;
            }
        }
        //when in chase
        else if (chaseTime > 0.0f)
        {
            agent.destination = player.transform.position;
        }

        isDead = false;
    }
    //personality trait of the 5% speed boost once pellet threshold reached
    if ((agent.speed != (startSpeed * 1.05f)) && (player.PelletsEaten() >=
        (game.GetCurrentTotalPellets() / 3)) &&
        (player.PelletsEaten() < (game.GetCurrentTotalPellets() * 2 / 3)))
    {
        agent.speed = startSpeed * 1.05f;
    }
    else if ((agent.speed != (startSpeed * 1.1025f)) && (player.PelletsEaten() >=
        (game.GetCurrentTotalPellets() * 2 / 3)))
    {
        agent.speed = startSpeed * 1.1025f;
    }
    else if (player.PelletsEaten() < (game.GetCurrentTotalPellets() / 3))
    {
        agent.speed = startSpeed;
    }
}
else
{
    agent.speed = 0f;
}

Vector3 PickHidingPlace()
{
    Vector3 directionToPlayer = (player.transform.position -
        transform.position).normalized;

    NavMeshHit navHit;
    NavMesh.SamplePosition(transform.position -
        (directionToPlayer * 8.0f), out navHit, 8.0f, NavMesh.AllAreas);
```

Figure 17 Code Snippet: Ghost 'Blinky' logic

This is the core logic behind the red ghost known as ‘Blinky’.

Almost all of the logic is shared among the ghosts except for key personality traits such as how the ghost moves to the player.

Blinky also gets a 5% speed boost when the player reaches a specified pellet threshold.

Ghost ‘Pinky’ documentation determines that the A.I. should aim to move four “tiles” ahead of the players facing direction.

My implementation of this was not exact as this project is a 3D game that is not built upon a 2D tile system in the same manner of the original game. However, using 4 Unity units withing Vector3 sufficed.

```
if (chaseTime > 0.0f)
{
    //pinky locates player ahead of their position by 4 units
    Vector3 targetPos = Vector3.zero;
    Vector3 playerPos = player.transform.position;
    if (player.GetDirection().Equals("left"))
    {
        targetPos = new Vector3(playerPos.x - 4, playerPos.y, playerPos.z);
    }
    else if (player.GetDirection().Equals("right"))
    {
        targetPos = new Vector3(playerPos.x + 4, playerPos.y, playerPos.z);
    }
    else if (player.GetDirection().Equals("up"))
    {
        targetPos = new Vector3(playerPos.x, playerPos.y, playerPos.z + 4);
    }
    else if (player.GetDirection().Equals("down"))
    {
        targetPos = new Vector3(playerPos.x, playerPos.y, playerPos.z - 4);
    }

    agent.destination = targetPos;
}
```

Figure 18 Code Snippet: Ghost 'Pinky' logic

‘Inky’ is perhaps the most interesting ghost behaviour as it utilises two positions. Inky behaves like Pinky but instead four positions ahead, its only two. The difference of this value with Blinky’s current position is then used to determine where Inky should move towards.

```
if (chaseTime > 0.0f)
{
    //inky targets the difference between the position 2 units infront of player
    //and blinky's current position
    Vector3 blinkyPos = GameObject.Find("Blinky").transform.position;
    Vector3 aheadPlayerPos = Vector3.zero;
    Vector3 playerPos = player.transform.position;

    if (player.GetDirection().Equals("left"))
    {
        aheadPlayerPos = new Vector3(playerPos.x - 2, playerPos.y, playerPos.z);
    }
    else if (player.GetDirection().Equals("right"))
    {
        aheadPlayerPos = new Vector3(playerPos.x + 2, playerPos.y, playerPos.z);
    }
    else if (player.GetDirection().Equals("up"))
    {
        aheadPlayerPos = new Vector3(playerPos.x, playerPos.y, playerPos.z + 2);
    }
    else if (player.GetDirection().Equals("down"))
    {
        aheadPlayerPos = new Vector3(playerPos.x, playerPos.y, playerPos.z - 2);
    }

    Vector3 targetPos = (aheadPlayerPos - blinkyPos) * 2;

    agent.destination = targetPos;
}
```

Figure 19 Code Snippet: Ghost 'Inky' logic

'Clyde' is the final ghost. This ghost's behaviour is simple as it only chases the player when more than eight units away.

```
if (chaseTime > 0.0f)
{
    //clyde chases only when more than eight units away from player
    Vector3 playerPos = player.transform.position;
    Vector3 ghostPos = this.transform.position;
    float unitsAway = Vector3.Distance(playerPos, ghostPos);

    //begin chase
    if (unitsAway > 8)
    {
        agent.destination = player.transform.position;
    }
    else
    {
        if (!agent.pathPending && agent.remainingDistance < 0.5f)
        {
            destPoint = (destPoint + 1) % scatterMarker.Length;
            agent.destination = scatterMarker[destPoint].position;
            GetComponent<Renderer>().material = normalMaterial;
            hiding = false;
        }
    }
}
```

Figure 20 Code Snippet: Ghost 'Clyde' logic

An advanced extension task I completed was that of implementing a strong visual style through the combination of textures, material, sound, skybox and UI system.

As previously illustrated, the UI system of the game facilitates a clear system design where a bold and strong aesthetic is carried throughout the game. Furthermore, the amalgamation of UI coinciding with the addition of a space skybox sets the game in an interesting but relevant setting. The traditional game has no background but making the creative decision of add a 3D skybox paid dividends to the theming.

I downloaded and imported the free skybox through the Unity Asset Store and, with some minor tweaking, was operating well. Importing packages this way is a new skill developed from this and I am now confident in importing assets from the Unity store for future projects.

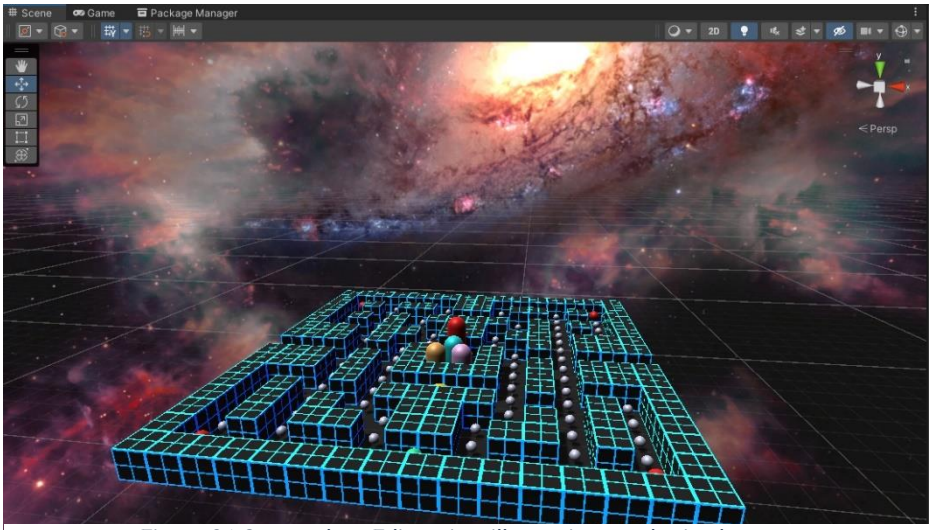

Figure 21 Screenshot: Editor view illustrating aesthetic changes

In addition, different music tracks were added to the game to contribute to the game's atmosphere. Two copyright free electronic songs were used to fit the theme and feel of the game. A steady one for the main menu and a more tense sounding track for when in game to match the gameplay.

Another change I intended was to replace the ghosts default capsules with models but I was unsuccessful in finding appropriate free models. Thus, I simply created a material and painted each one their corresponding colour. In regards to the maze, a custom material derived from a shader was used on the wall prefab for the level. The intent was to have a 'Tron'-like grid shader to fit with the dark space theming. I opted for a shader over standard textures due to the modularity gained from seamless colour customisation or modifying line frequency and thickness. Reading The Book of Shaders by Patricio Gonzalez Vivo offered significant insight to shaders and furthered my understanding of the topic. Having already used Unity prior to this module, deciding to still push myself to new territory regarding shaders made sure I was always learning and developing new skills.

Likewise, continuing to push myself further meant I was looking to tick more extensions off the list. I decided upon attempting the remaining advanced extension; creating a minigame. For this, I wanted to build it from the ground up in a new scene. The game I settled on was a clone of the 2014 hyper-casual mobile game Flappy Bird. In order for it to fit the Mellow Fellow styling, I continued to use the same UI, Skybox and materials. The scene is loaded with the SceneManager when the player clicks the 'Flappy Fellow' button.

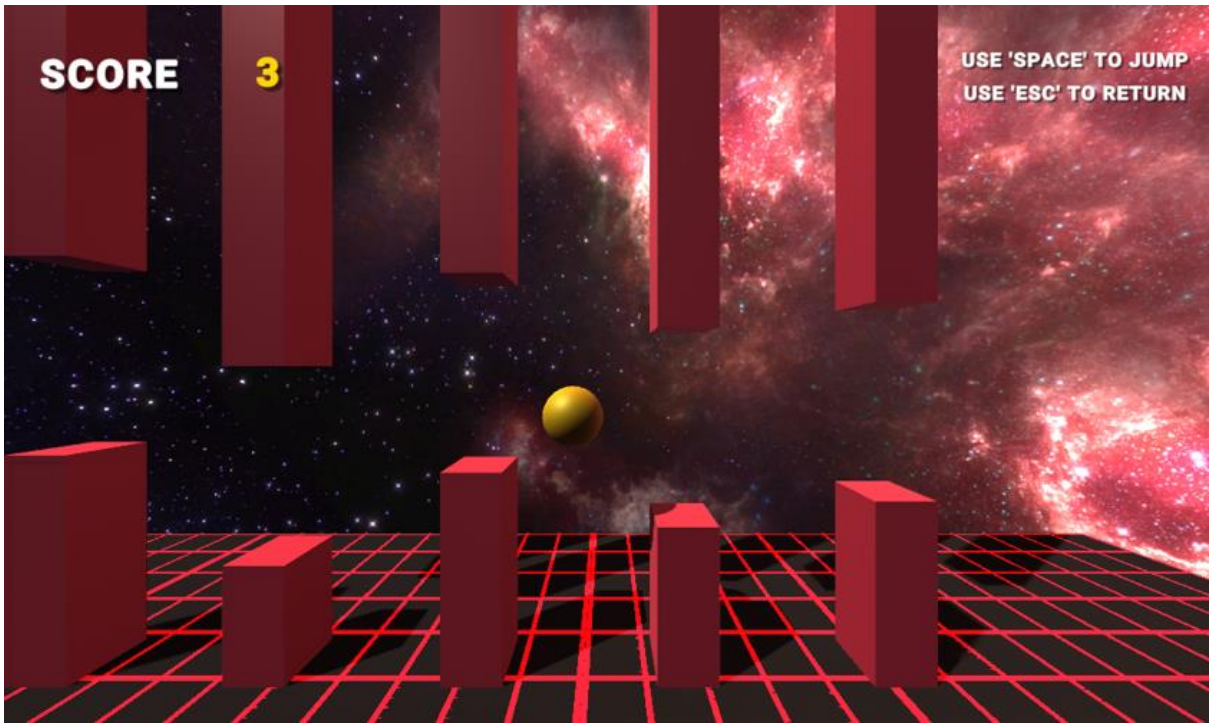

Figure 22 Screenshot: In-game view of Flappy Fellow minigame

In order to simulate continuous movement, the ground plane prefab is instantiated and gradually moved along from right to left of the camera while the player jumps stationary on y axis. After a specified number of seconds, the plane is then despawned off camera while a new instance replaces it at the other end of the conveyor belt. Despawning the objects was imperative otherwise a memory leak would occur as they objects would be produced infinitely, hindering performance or even causing a crash.

The pillars prefab operates in a comparable fashion except it utilises randomness to spawn the gate height. Having the pillars spawn at regular intervals but random height is what generates the fun of the game as no two levels are ever the same. Consequently, promoting replayability and enjoyment for the player as they aim to beat their high score.

Figure 23 shows the code used to instantiate each pillar to a random transform within a suitable range that the player can realistically get through.

```
void Update()
{
    sendTimer -= Time.deltaTime;
    if(sendTimer <= 0)
    {
        position = Random.Range(3f, 6.5f);
        transform.position = new Vector3(2.5f, position, 3.7f);
        Instantiate(myPillar, transform.position, transform.rotation);
        sendTimer = frq;
    }

    if (mainCharacter != null) Time.timeScale = 1;
    else Time.timeScale = 0;
}
```

Figure 23 Code Snippet: Pillar Spawning Logic

One issue that arose with this minigame regarded the movement of the player. I originally added a force to the rigidbody of the player in an upward direction. Although this worked, the physics were unrealistic. After jumping, the player slowly glided down in a slow linear motion, where as in the original Flappy Bird, the player falls at an exponential rate.

Online tutorials and the Unity Manual documentation explained what linear interpretation (Lerp) is in Unity. Further reading lead me to understand lerps mathematical function to return a value between A and B at a point on a linear scale [3].

Utilising Lerp is a pivotal tool for a game developer relating to aspects of a game, so having this now under my belt will definitely be a huge benefit.

```
void Update()
{
    //ORIGINAL METHOD
    //if (Input.GetKeyDown(KeyCode.Space))
    //      {
    //          rb.velocity = Vector3.zero;
    //          rb.AddForce(new Vector3(0, force, 0));
    //      }


    //NEW METHOD
    transform.Translate(0, yForce, 0);
    yForce = Mathf.Lerp(yForce, yTarget, .025f);

    if (Input.GetKeyDown(KeyCode.Space))
    {
        yForce = .05f;
    }

    if (Input.GetKeyDown(KeyCode.Escape))
    {
        SceneManager.LoadScene("MainGame");
    }

}
```

Figure 24 Code Snippet: Two physics logic used for player movement in the minigame

Results & Evaluation

As the screenshots, code snippets and final artefact demonstrate, I have been able to produce not only a fully playable game going beyond the initial specification, but one that is immersive and enjoyable. The player is greeted with the main menu, where they are presented with a series of options to either: play, see leaderboard, play the minigame or exit, see figures 7-11.

The main game plays exactly like the traditional Pac-Man game with each ghost having their own behaviour states and unique characteristics. The only difference being it is one single ‘level’ that is repeated once the pellets are cleared by the player with all lives remaining, as per requirement three. Characters can ‘teleport’ from one side of the level to another through the ‘tunnel’. After three deaths, the player is given the option to save their score to a local leaderboard. All the while, key game data is displayed on screen with appropriate UI.

The ‘Flappy Fellow’ minigame provides an entirely different gameplay experience to the player. Since this extension task was completed in the latter stages of the project, it is not as well polished as the main game. The core mechanics and scoring work but a leaderboard similar to the main game was not implemented due to time constraints.

Bugs do occur however. Occasionally concerning the respawning of the characters but single developer playtesting can only diagnose so many irreproducible bugs. Overall, the artefact meets each of the necessary criteria as well as the one easy extension and both advanced extension criteria.

Conclusion

A comprehensive summary of the work done; clear suggestions for future work. Demonstrates reflective thinking on how the project went, including positive and negative aspects.

For the most part, the project was tremendously successful as the artefact met all the necessary requirements. This alone developed my proficiency with Unity and C# scripting immensely. I have since gained confidence and knowledge in an assortment of tools regarding game development. Both in the context of Unity specific game engine tools, such as creating prefabs, as well as the broader game development aspects like level design.

I then stretched myself further with the more demanding extension tasks. Being fully independent particularly for these tasks was a challenge as most of the work required was never covered during tutorial sessions. Implementing these displayed that I was up to the challenge, willing and capable of finding external sources of information to research the relevant topics.

Despite all the positives, there are plenty of mistakes I made that I am able to learn from. One such error was failing to organise the project well enough. I decided to introduce a Kanban board late into the project and it did not offer much value. Mostly, it was difficult to keep track of what bugs were not fixed or what I was last working on since it would often be some days between working on the project. Crucially, I falsely believed that since it was a relatively small single-developer project, I could manage since team communication wouldn't be an issue. Although the project turned out well, if the deadline was tight the project could have failed from my own underestimations. Going forward, I will prioritise planning with the upmost importance.

References

[1] J. Nielsen, "Ten Usability Heuristics," 2005.

[2] J. Pittman, "The Pac-Man Dossier," Game Developer, 23 Feb 2009. [Online]. Available: https://www.gamedeveloper.com/design/the-pac-man-dossier. [Accessed May 2022].

[3] J. French, "The right way to Lerp in Unity," gamedevbeginner, 13 Apr 2020. [Online]. Available: https://gamedevbeginner.com/the-right-way-to-lerp-in-unity-with-examples/. [Accessed May 2022].

[4] P. G. Vivo, The Book of Shaders, 2015.