AT Block 3 - Task 3
Old Skool FPS Report

Matteo Bolognesi
19025911

University of the West of England

July 19, 2023

The chosen task for block 3 was "Old Skool FPS". It was required to create a first-person shooter (FPS) complete with multiple levels, music, SFX and menus. As inspiration, the style of old games like Wolfenstein 3D, Doom and Duke Nukem 3D, originally released for DOS during the 90s. In addition, enemies and objects were required to be billboards and have a working AI. Basic level completion logic and interactions were also part of the requirements, with specific mention of key cards and bosses, iconic for these types of games at the time. As a stretch goal, a level editor was required, to allow players to create and play their own levels. This report aims to explain how the requirements were implemented and the decision process that led to it, using Unity as the game engine of choice.

1 Introduction

Since the game had to be developed in its entirety, the main problem was to identify and build a structure of managers able to handle all the functions needed for the game. Moreover, research on the cited games was conducted to understand the key elements of the genre and how to represent them in the game.

2 Related Work

Wolfenstein 3D (Fig.1), its brother Doom (Fig.2) and Duke Nukem 3D (Fig.3) were released during the 90s by **id Software** and **3D Realms** and were a huge success among both the critics and the public. They featured a wide range of levels, weapons and enemies, a simple and readable UI, engaging audio and visuals

Figure 1: *Wolfenstein 3D*

and a simple menu. Some of the key elements present in the games are key cards to access blocked parts of the levels, pick-ups to restore health, ammo and increase the player's arsenal, unique sound effects and visuals for each enemy type and bosses to mark the end of specific sections of the game and convey the story.

3 Method

The game is divided into 6 managers, each responsible for a different task. Each manager has a singleton pattern and is marked as DontDestoryOnLoad, allowing them to be always present in each scene without losing references where needed (Fig.4). This manager structure was adopted to have a simple way to set up multiple scenes quickly without incurring major data stream problems.

Figure 2: *Doom*

Figure 3: *Duke Nukem 3D*

```
1 reference
private void GameManagerSingleton()
{
    if (_gameManagerInstance == null)
    {
        _gameManagerInstance = this;
    }
    else if (_gameManagerInstance != this)
    {
        Destroy(gameObject);
    }

    DontDestroyOnLoad(gameObject);
}
```

Figure 4: *Game Manager's singleton function*

```
1 reference
private void SubscribeToEvents()
{
    SceneManager.sceneLoaded -= SetUpGame;
    SceneManager.sceneLoaded += SetUpGame;
}
1 reference
private void SetUpGame()
{
    ActiveScene = SceneManager.GetActiveScene();
    ActiveSceneName = SceneManager.GetActiveScene().name;
    SceneLoadedIndex = SceneManager.GetActiveScene().buildIndex;
    SetGameState();
    _victory = false;
    OnGMSetUpComplete?.Invoke();
    //Debug.Log("GameManager SetUp");
}
2 references
private void SetUpGame(Scene scene, LoadSceneMode mode)
{
    ActiveScene = SceneManager.GetActiveScene();
    ActiveSceneName = SceneManager.GetActiveScene().name;
    SceneLoadedIndex = SceneManager.GetActiveScene().buildIndex;
    SetGameState();
    OnGMSetUpComplete?.Invoke();
}
```

Figure 5: *Game Manager's Event Subscription and Setup Function*

3.1 Game Manager

The game manager is responsible to communicate to the other managers that the game has started and setting up the game state based on the scene loaded. To achieve this there is a function that gets called in both the Start method and when a new scene is loaded (Fig.5). This is due to the fact that Unity does not call the sceneLoaded event when the first scene is loaded. The singleton function called in the Awake method makes sure that no duplicate is created, avoiding multiple setup calls on scene transition. When the event is triggered, all the other managers' setup functions are called allowing the Game Manager to receive all the references and start the game with the correct values. This manager is meant to hold the references to all the important variables to allow any game object in the game to access them at any time. Notably, the Game Manager has no references to other managers to avoid circular logic problems.

3.2 Audio Manager

The Audio Manager holds the references to the music tracks and SFX needed for the game (Fig.6). It has functions to start and stop the music (Fig.7) and to play every SFX (Fig.8), allowing it to quickly be referenced to play each audio track any time is needed.

3.3 Camera Manager

The Camera manager simply activates the correct camera based on the game state. The game is really simple and necessitates just two cameras to function, one for the menu and one for the gameplay (Fig.9).

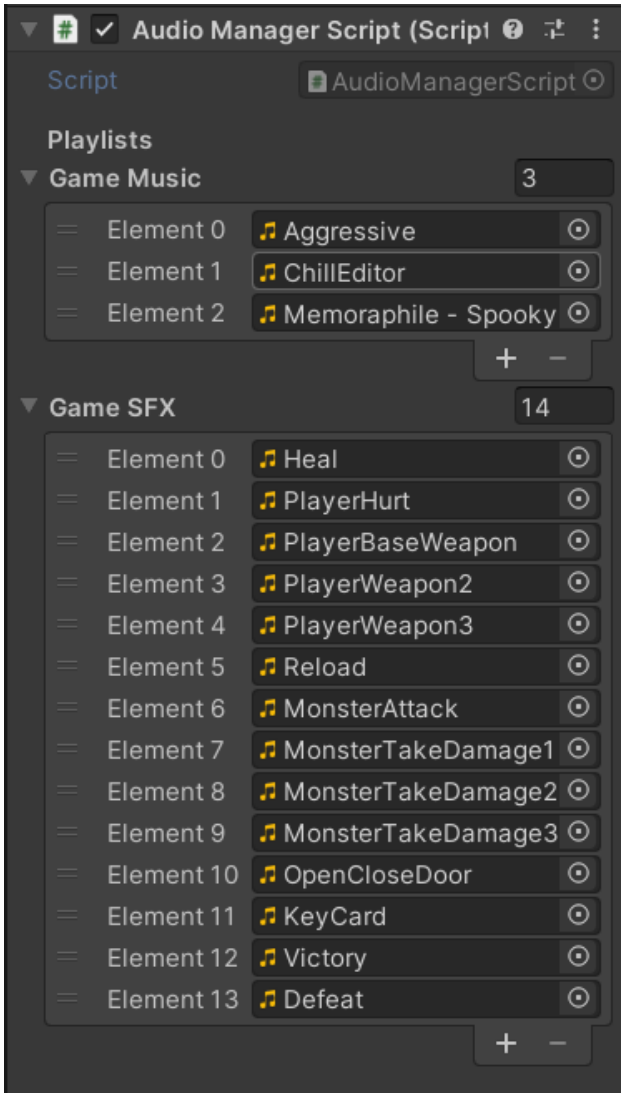

Figure 6: *Unity Inspector Audio Manager References*

```
3 references
public void PlayMusic(int clipIndex)
{
    if ((!_audioSourceInstance.isPlaying) ||
        (_audioSourceInstance.isPlaying &&
         _audioSourceInstance.clip != _gameMusic[clipIndex]))
    {
        _audioSourceInstance.Stop();

        _audioSourceInstance.clip = _gameMusic[clipIndex];
        _currentAudioClipLoaded = _gameMusic[clipIndex];

        _audioSourceInstance.Play();
        _audioClipPlaying = _audioSourceInstance.isPlaying;

        _gameManager.CurrentAudioClipLoaded = _currentAudioClipLoaded;
        _gameManager.AudioClipPlaying = _audioClipPlaying;
    }
}
2 references
public void StopMusic()
{
    _audioSourceInstance.Stop();

    _audioSourceInstance.clip = null;
    CurrentAudioClipLoaded = null;

    _audioClipPlaying = _audioSourceInstance.isPlaying;

    _gameManager.AudioClipPlaying = _currentAudioClipLoaded;
    _gameManager.AudioClipPlaying = _audioClipPlaying;
}
```

Figure 7: *Audio Manager Play and Stop Music*

```
0 references
public void PlaySFX(int clipIndex)
{
    AudioSource.PlayClipAtPoint(_gameSFX[clipIndex],
                                CameraManagerScript.CMInstance.ActiveCamera.transform.position);
}
2 references
public void PlayHealSFX()...
1 reference
public void PlayPlayerHurtSFX()...
1 reference
public void PlayWeaponBSFX()...
1 reference
public void PlayWeapon2SFX()...
1 reference
public void PlayWeapon3SFX()...
2 references
public void PlayReloadSFX()...
1 reference
public void PlayEnemyAtkSFX()...
2 references
public void PlayEnemyBaseTakeDamageSFX()...
1 reference
public void PlayEnemyStrongTakeDamageSFX()...
1 reference
public void PlayBossTakeDamageSFX()...
5 references
public void PlayDoorSFX()...
2 references
public void PlayKeyCardSFX()...
1 reference
public void PlayVictorySFX()...
1 reference
public void PlayDefeatSFX()...
```

Figure 8: *Audio Manager Play SFX*

```
2 references
private void AddCamerasToList()
{
    _cameraList.Add(gameObject.GetComponentInChildren<CinemachineVirtualCamera>());
    _cameraList.Add(_player.InGameCamera);
}

2 references
public void SetUpCameraManager()
{
    AddCamerasToList();
    if (_gameManager.SceneLoadedIndex == 4 ||
        _gameManager.SceneLoadedIndex == 5 ||
        _gameManager.SceneLoadedIndex == 6)
    {
        ActivateCameraCinemachine(1);
    }
    else
    {
        ActivateCameraCinemachine(0);
    }
    //Debug.Log("CameraManager CallFromGMEvent");
}

2 references
public void ActivateCameraCinemachine(int index)
{
    _cameraList[index].MoveToTopOfPrioritySubqueue();;
    _activeCamera = _cameraList[index];
    _gameManager.ActiveCamera = _cameraList[index];
}
```

Figure 9: *Camera Manager Functions*

```
2 references
public void SetUpStartingInputMap()
{
    if (_inputMap == null)
    {
        _inputMap = new NewInputMap();
        SubscribeUIInputs();
        PlayerScript.PlayerInstance.SubscribeGameInputs();
    }

    if (_gameManager.SceneLoadedIndex == 4 ||
        _gameManager.SceneLoadedIndex == 5 ||
        _gameManager.SceneLoadedIndex == 6)
    {
        ActivateInputMap(_inputMap.Game);
    }
    else
    {
        ActivateInputMap(_inputMap.UI);
    }
    //Debug.Log("InputManager Call from GMEvent: " + _activeInputMap);
}
1 reference
public void SubscribeUIInputs()
{
    _inputMap.UI.Move.started += OnMove;

    _inputMap.UI.ButtonWest.performed -= OnButtonWest;
    _inputMap.UI.ButtonNorth.performed -= OnButtonNorth;
    _inputMap.UI.ButtonEast.performed -= OnButtonEast;
    _inputMap.UI.ShoulderR.performed -= OnShoulderR;
    _inputMap.UI.ShoulderL.performed -= OnShoulderL;
    _inputMap.UI.StartButton.performed -= OnStartButton;

    _inputMap.UI.ButtonSouth.canceled -= OnButtonSouth;

    _inputMap.UI.Move.started += OnMove;
    //_inputMap.UI.ButtonSouth.started += OnButtonSouth;
    _inputMap.UI.ButtonWest.performed += OnButtonWest;
    _inputMap.UI.ButtonNorth.performed += OnButtonNorth;
    _inputMap.UI.ButtonEast.performed += OnButtonEast;
    _inputMap.UI.ShoulderR.performed += OnShoulderR;
    _inputMap.UI.ShoulderL.performed += OnShoulderL;
    _inputMap.UI.StartButton.performed += OnStartButton;

    //_inputMap.UI.Move.canceled += OnMove;
    _inputMap.UI.ButtonSouth.canceled += OnButtonSouth;
    //_inputMap.UI.ButtonWest.canceled += OnButtonWest;
    //_inputMap.UI.ButtonNorth.canceled += OnButtonNorth;
    //_inputMap.UI.ButtonEast.canceled += OnButtonEast;
    //_inputMap.UI.ShoulderR.canceled += OnShoulderR;
    //_inputMap.UI.ShoulderL.canceled += OnShoulderL;
    //_inputMap.UI.StartButton.canceled += OnStartButton;
}
4 references
public void ActivateInputMap(InputActionMap map)
{
    _inputMap.Disable();
    map.Enable();
    _activeInputMap = map;
}
```

Figure 10: *Input Manager Functions*

3.4 Input Manager

The Input Manager handles the subscription to input events for the menus. It is also responsible for creating and activating the correct input map based on the scene loaded (Fig.10).

3.5 Scene Manager

The Scene Manager handles the function to transition between scenes and has a function to load a scene based on its name.

3.6 UI Manager

The UI Manager is composed of two scripts. The first one handles the loading of the correct canvas based on the active scene. The second holds the references to all the images and text used for the UI present in the game to quickly swap visuals when needed(Fig.11).

```
6 references
public void LoadCanvas(int canvasIndex)
{
    if (ActiveCanvas != null)
    {
        ActiveCanvas.SetActive(false);
    }
    _canvasList[canvasIndex].SetActive(true);
    ActiveCanvas = _canvasList[canvasIndex];
    SetUpEventSystem();
    _gameManager.ActiveCanvas = ActiveCanvas;
}
```

Figure 11: *UI Manager Load Canvas Function*

```
2 references
private void SwapWeapon(int step)
{
    var newWeaponIndex = _activeWeaponIndex + step;
    if (newWeaponIndex > _weaponList.Length -1)
    {
        newWeaponIndex = 0;
    }
    else if (newWeaponIndex < 0)
    {
        newWeaponIndex = _weaponList.Length -1;
    }
    ActivateWeapon(newWeaponIndex);
}
1 reference
private void AutoSwapWeapon()
{
    var count = 0;
    for (int i = 0; i < _weaponList.Length; i++)
    {
        if (_weaponList[i].GetComponent<BulletScript>().Ammo > 0)
        {
            ActivateWeapon(i);
        }
        else
        {
            count += 1;
        }
    }
    if (count == 3)
    {
        Debug.Log("NoAmmo");
    }
}
7 references
public void ActivateWeapon(int index)
{
    if (_weaponList[index].GetComponent<BulletScript>().Ammo > 0)
    {
        _activeWeapon = _weaponList[index];
        _activeWeaponIndex = index;
        _fireDelay = _weaponList[index].GetComponent<BulletScript>().FireDelay;
        _weaponSprite.sprite = _weaponList[index].GetComponent<BulletScript>().WeaponSprite;
        LinkUI();
    }
}
```

Figure 12: *Player Script Weapon Swap Functions*

3.7 Player

The player script handles all the functions needed to set up the starting values for the player. It handles movement, shooting, spawning and despawning in addition to handling the inputs. Movement is achieved thanks to the **Character Controller** component attached to the player game object. Three functions manage the weapon swap selection, preventing the player from accessing weapons with no ammo and automatically swapping to another loaded weapon on fire input if the current one is empty (Fig.12).

3.8 Enemies

There are three types of enemies in the game. They are all part of the same script and behave the same way,

the main difference is in the numeric values like visuals, health, damage, etc. Two enumerators handle the type of enemy and the state during gameplay. To navigate the map, they use the **NavMeshAgent** component alongside a baked navigation map. Their behaviour is dictated by a function that calls the correct one based on the state (Fig.13). A set of functions is responsible for checking if the player is within the given range and if the enemy has the line of sight to shoot when within shooting range (Fig.14).

```
private void Behaviour()
{
    if (_currentState != EnemyState.Dead)
    {
        if (PlayerInRange(_ACTIVATION_RANGE))
        {
            if (PlayerInRange(_attackRange))
            {
                if (LoS())
                {
                    _currentState = EnemyState.Shoot;
                }
                else
                {
                    _currentState = EnemyState.Chase;
                }

            }
            else if (PlayerInRange(_aggroRange))
            {
                _currentState = EnemyState.Chase;
            }
            else
            {
                _currentState = EnemyState.Idle;
            }
        }
        else
        {
            _currentState = EnemyState.Inactive;
        }

        switch (_currentState)
        {
            case EnemyState.Idle:
                Idle();
                //Debug.Log("Idle Enemy");
                break;
            case EnemyState.Chase:
                Chase();
                //Debug.Log("Chase Enemy");
                break;
            case EnemyState.Shoot:
                Fire();
                //Debug.Log("Fire Enemy");
                break;
            case EnemyState.Inactive:
                _navMeshAgent.isStopped = true;
                break;
            default:
                _navMeshAgent.isStopped = true;
                Debug.Log("Inactive Enemy ERROR");
                break;
        }
    }

}
```

Figure 13: *Enemy Behaviour Function*

3.9 Pickup

Pickups can be of four types: weapons, ammo, health kit and key card. Similarly to the enemies, they are all handled by the same script and use enumerators to differentiate them (Fig.15). A collider set as a trigger handles the collision with the player and based on the pickup type, the correct function is called (Fig.16).

3.10 Billboard

As specifically required, both enemies and pickups have a script attached that allows them to turn their sprites based on the player's rotation. This gives a sense of tridimensionality to objects using simple 2D sprites (Fig.17).

3.11 Doors

Similarly to pickups and enemies, doors can be of multiple types but they all belong to the same script. Enumerators are used to handling the door types (Fig.18) and states and colliders are set as triggers to check if something is in its interaction range. Both the player and the enemies can open doors and as long as they remain within the trigger area, the doors remain open (Fig.19).

4 Evaluation

Although functional, the code presents some major issues considering its coding structure. The choices made were dictated mainly by time constraints but a more efficient and clean structure should be adopted. Managers should have a base manager script with all the functions common to all managers and then a child class should be used to handle each manager's own logic. Considering that the player is also a singleton, there might be room to consider making the player a child of the manager base class. Moreover, working with strings should be avoided and a more reliable way to pass data should be used. Coding patterns like the state machine could be implemented for both the player and the enemies, to make sure that only the needed code is run at any given time. Enemies and pickups should follow a structure similar to the manager's one instead of using an enumerator to handle the

```
private Vector3 PlayerDirection()
{
    Vector3 targetPos;
    Vector3 moveDirection;

    targetPos = _player.transform.position;
    moveDirection = (targetPos - _firePoint.transform.position).normalized;
    return moveDirection;
}
4 references
private bool PlayerInRange(float range)
{
    _distanceToPlayer = Vector3.Distance(transform.position, _player.transform.position);

    if (_distanceToPlayer <= range)
    {
        return true;
    }
    else
    {
        return false;
    }
}
2 references
private bool LoS()
{
    Ray ray = new Ray(transform.position, PlayerDirection());
    RaycastHit hit;
    if (Physics.Raycast(ray, out hit, _attackRange))
    {
        if (hit.collider.CompareTag("Wall") ||
            (hit.collider.CompareTag("Door") &&
            hit.collider.GetComponent<DoorScript>().CurrentDoorState == DoorScript.DoorState.Closed))
        {
            return false;
        }
    }
    return true;
}
```

Figure 14: *Enemy Orientation Functions*

```
5 references
public enum PickUpType
{
    Health,
    Ammo,
    Weapon,
    Keycard
}

7 references
public enum Weapons
{
    BaseWeapon,
    Weapon1,
    Weapon2,
}

4 references
public enum KeycardColour
{
    Red,
    Blue,
    Yellow
}
```

Figure 15: *Pickups Enumerators*

```
private void OnTriggerEnter(Collider other)
{
    if (other.GetComponent<PlayerScript>())
    {
        switch (_pickUpType)
        {
            case PickUpType.Health:
                RestoreHealth(other.GetComponent<PlayerScript>());
                _audioManager.PlayHealSFX();
                break;

            case PickUpType.Weapon:
                Reload(other);
                ActivateWeapon(other);
                _audioManager.PlayReloadSFX();
                break;

            case PickUpType.Ammo:
                Reload(other);
                _audioManager.PlayReloadSFX();
                break;

            case PickUpType.Keycard:
                CollectKeyCard(other);
                _audioManager.PlayKeyCardSFX();
                break;

            default:
                RestoreHealth(other.GetComponent<PlayerScript>());
                _audioManager.PlayHealSFX();
                Debug.Log("PickUp Error");
                break;
        }

        Destroy(gameObject);
    }
}
```

Figure 16: *Pickups OnTriggerEnter Function*

```
Unity Script (21 asset references) | 0 references
public class BillboardScript : MonoBehaviour
{
    [SerializeField] SpriteRenderer _sprite;
    Unity Message | 0 references
    void Start()
    {
        SetUpReferences();
    }

    Unity Message | 0 references
    void LateUpdate()
    {
        Rotate();
    }
    1 reference
    private void SetUpReferences()
    {
        _sprite = GetComponent<SpriteRenderer>();
    }
    1 reference
    private void Rotate()
    {
        _sprite.transform.rotation = PlayerScript.PlayerInstance.transform.rotation;
    }
}
```

Figure 17: *Billboard Script*

```
3 references
public enum DoorType
{
    NormalDoor,
    KeyCardDoor
}
4 references
public enum DoorColour
{
    Red,
    Blue,
    Yellow
}
12 references
public enum DoorState
{
    Open,
    Closed
}
```

Figure 18: *Door Enumerators*

```
private void OnTriggerEnter(Collider other)
{
    if (other.GetComponent<PlayerScript>() || other.GetComponent<EnemyScript>())
    {
        _objectsInTrigger.Add(other);
        switch (_doorType)
        {
            case DoorType.NormalDoor:
                _currentDoorState = DoorState.Open;
                _audioManager.PlayDoorSFX();
                break;

            case DoorType.KeyCardDoor:
                if (other.GetComponent<PlayerScript>())
                {
                    OpenKeyCardDoor(other);
                }
                break;

            default:
                Debug.Log("Door Error");
                break;
        }
    }
}
@ Unity Message | 0 references
private void OnTriggerExit(Collider other)
{
    if (other.GetComponent<PlayerScript>() || other.GetComponent<EnemyScript>())
    {
        _objectsInTrigger.Remove(other);
        if (_objectsInTrigger.Count == 0)
        {
            _currentDoorState = DoorState.Closed;
            _audioManager.PlayDoorSFX();
        }
    }
}
```

Figure 19: *Door OnTrigger functions*

various types, allowing also to have a cleaner inspector in Unity. The game works as intended except for a bug that could not be addressed and is not clear what is causing it. Upon level completion, the player is moved to the correct spawn position in the newly loaded level but the very next frame is placed at the same position where the endpoint was in the previous level. This bug is not consistent because at times the spawn upon level transition works as intended and some other times the bug happens. If each level is played individually from the Unity editor, the player spawns at the correct location.

5 Conclusion

The task required to create a complete game inspired by the old-school FPS games, featuring all the key elements of those games like keycards and billboard sprites. Despite not being fully functional due to a game-breaking bug, the game presents all the requested features although implemented at a simple level. The manager structure proved to be efficient to prototype and test the game and the frame rate proved to be stable during playtest sessions. Restructuring and refactoring the system would lead to better coding standards and better scalability of the project, also improving code stability and usability for designers.