Specification Compliance Report

+30%: **Physics**

- +5%: Appropriate use of Newtonian physics
  - ☑ Usage of Rigid Bodies
    - Rigid Bodies can be found on the player and enemies, and cannonballs.

  - ☑ Correct application of impulses to bodies
    - The player can press ‘space’ to jump, applying upward force to the player.
    - When the player collides with an enemy, they are knocked backward using impulses to the player’s Rigidbody

  - ☑ Objects have appropriate mass quantities
    - Player and enemies have appropriate mass quantities, with non-cannon enemies knockback force being calculated depending on their mass

  - ☑ Game mechanics is physics-driven (e.g., immediate response to collisions).
    - The player’s jump and collision with enemy mechanics are physics-driven, as physics are applied to the player using Rigidbodies and force

- +5%: Advanced Physics (multiple gravity areas / changing mass / *et ceteram*)
  - ☑ Physics properties are changed via scripts
    - The cannon enemy shoots cannonballs from its cannon. The mass and scale of the regular cannonballs are 10 with a scale of 1. Every third shot, I increase the mass and scale of the cannonball to 15 and to 1.5 scale via script.

  - ☑ Mass/Physics is a gameplay mechanic
    - The mass of the player and non cannon enemy affect the impact of the knockback. Heavier enemies push the player back further.

  - ☑ Additional forces beyond simple motion-driven accelerations are provided (projectile trajectories, gravity interfering with the velocity).
    - The CannonController script takes the gravity and velocity of the cannonballs into account when calculating the trajectory of the cannonball, changing the arc accordingly.

  - ☑ AI uses calculations to determine projectile forces.

        - The cannon uses calculations to try and predict the player’s position in the future using the distance between the cannon and the player, as well as the player’s velocity.

- +5%: Basic Collision Volumes
    - [x] There is at least one collision volume
        - I have box colliders on the environment and enemies, I have a capsule collider on the player, and a sphere collider on the coins.
    - [x] There is more than one collision volume
        - I use multiple types of colliders, my player has a capsule collider, enemies have box colliders, and coins use sphere colliders.
    - [x] The collision volume is appropriate and matches the Game Object’s mesh.
        - The capsule collider on my player matches the Game Object’s mesh, the enemies box colliders match their mesh, and my coins closely match their mesh.

- +5%: Advanced collision volumes.
    - [x] A single Game Object has multiple colliders.
        - The enemies in my game have two colliders, one for the body, and one for the top of the head
    - [x] Colliders are enabled/disabled via scripts.
        - When the player jumps on top of an enemy’s head, they defeat the enemy and disable their game object model, including the colliders they had attached
        - I have a rock at the beginning of the level that has a moving collider that also disables and re-enables its collider every 5 seconds
    - [x] Colliders can change their position programmatically.
        - I have a rock at the beginning of the level that has a moving collider, the rock moves its box collider along the x axis, pushing the player left or right when interacted with
    - [x] Trigger volumes are used as part of player mechanics.
        - I use trigger volumes on my star and coin collectables. Using the method ‘OnTriggerEnter’ to handle the interactions between player and

collectable.

- +5%: Appropriate collision response and feedback.
    - ☑ Rigidbody responds to collisions realistically
        - When colliding with an enemy, the player's rigidbody responds realistically, as force is applied to knock the player backwards, simulating the effect of the collision
    - ☑ An object makes use of OnCollisionEnter / Exit
        - To apply the knockback behaviour between player and non-cannon enemy, I use the OnCollisionEnter method.
    - ☑ Collision Layers are used to separate out collision types
        - In the PlayerMovement script, I have a ground check that checks if the player is in contact with the "Ground" or "EnemyLand" layers, ensuring that the player can only jump when on these layers.
    - ☑ Physics materials are used
        - Different types of physics materials with varying properties are applied to the player depending on the layer that they are colliding with. When the player collides with game objects with the 'Water' tag, they are assigned the water physics material. I have applied the same with ground and enemyland.

- +5%: Advanced collision response and feedback
    - ☑ Multiple physics materials appearing in the game.
        - The script 'PlayerMaterialManager' assigns the player different physics materials based on what they ayre interacting with (touching water, ground or enemy land)
    - ☑ Physics materials are changed at run-time via script
        - In the same script 'PlayerMaterialManager' my player's physics material is changed during run-time through the AssignMaterial function, using collision detection and trigger events.
    - ☑ Trigger volumes are used to trigger gameplay events
        - When the player collides with coins or stars via trigger volumes, the player 'collects' them and increases the amount of coins / stars they have by 1.
        - When the player collides with the body of water, it uses OnTriggerEnter to check if the player has collided and emits particles

+20%: **Graphics**

- +10%: Appropriate Use of Graphical Elements
  - ☑ Multiple textures appearing in game
    - Multiple textures can be found on the walls, ground, mountain, and caves
  - ☑ Appropriate use of lighting
    - The outdoor section is well lit and set in daytime, whereas the cave sections are darker due to the absence of the sun and shadow of the cave
  - ☑ GameObjects moving & rotating via script
    - Coins and Stars rotate via script, and the enemy chases the player via script
  - ☑ A navigable camera moving in 3D
    - The player can fully move the camera around the player in both the x and y axis

- +10%: Advanced Graphics
  - ☑ Environment appears to extend infinitely
    - I have extended the environment past the play area so that the player cannot see the edge of the map. I have added a skybox, mountain backdrops, and invisible walls to add depth but keep the player within the level.
  - ☑ A body of realistic looking water
    - There is a body of realistic looking water in the first level
  - ☑ Scripted lighting/effects (e.g. weather, day/night cycle)
    - I have a day and night cycle that cycles from day to night, lasting for 120 seconds.
  - ☑ Change object appearance via script
    - Player model changes from ‘standing’ to ‘crouched’ via script when the player presses ‘left ctrl’, making the player’s model smaller.
  - ☑ Geometry that changes over time (e.g. plants growing)

- In the water plane, there is an ice cube in the water that slowly melts in the water, decreasing in size over a duration of 3 minutes.

+10%: Pathfinding

- [x] NavMeshAgents are used
    - The enemies in my game use NavMeshAgents for movement

- [x] NavMeshObstacles are used
    - In the navmesh where my enemies roam, there are rocks that are assigned as NavMeshObstacles, they avoid and walk around these obstacles when moving.

- [x] Custom pathfinding code, or external library with some modifications.
    - The enemies in use waypoints to navigate around their environment, travelling between set waypoints. If the enemies gain sight of the player they will chase the player until they lose them, returning to the last waypoint they were previously moving towards before chasing the player.

- [x] AI makes decisions based on pathfinding
    - My enemies use NavMeshAgents which involves pathfinding in order to navigate between waypoints. Additionally, when chasing the player, it uses NavMeshAgent.SetDestination() to follow the player. When they lose sight, they rotate clockwise and counter clockwise to simulate looking for the player, before returning to their waypoint patrol.

+30%: Artificial Intelligence

- +10%: State Machines
  - ☑ Usage of simple state-machines
    - For my enemies, I use basic state-driven behaviour, transitioning between searching, chasing, and patrolling based on the player’s position
  - ☑ Usage of boolean or state-driven state machines
    - My enemies' AI is dictated by booleans as well as state-driven logic, as it checks if the player has been spotted or lost before deciding which state to transition to.
  - ☑ Usage of object-encapsulation for modelling states
    - I have encapsulated each state into its own class. For example, Chasing has its own state, as does Patrolling and Searching.
  - ☐ Usage of hierarchical state machines or usage of external tools for generating state machines.
    - I did attempt to implement hierarchical state machines by organising the NPC behaviours into their own states (Patrolling, Chasing, and Searching) and had them partially inherit from the base state class but did not manage to fully implement the hierarchical structure.
  - ☑ State machines are triggered by external events or timeouts.
    - The state transitions for my enemies are triggered by external events, such as gaining sight of the player, making them chase the player, or losing sight of the player, triggering the lost state.
  - ☑ Usage of probabilistic/stochastic state transitions.
    - I have two probabilistic transitions for my enemies. When the enemy reaches a waypoint, it has a 20% chance to pick a random waypoint, and has a 50% chance to transition to the Searching state.

- +10%: Advanced AI
  - ☐ Usage of Planning techniques (Real Planners, GOAPs)
  - ☐ Usage of Non-Cooperative Game Strategies (Min-Max trees, α-βpruning)
  - ☐ Usage of basic Reinforcement Learning techniques.

- +10%: Structuring NPCs
  - [x] NPCs are not orchestrated, and mainly react directly to the player or the environment.
    - My enemies react directly to the player by stopping whatever they were doing to chase the player until they defeat or lose sight of the player, before searching and returning to their patrol afterward
  - [ ] NPCs are coordinated as a group or in tandem, to fulfil the same goal or task.
  - [ ] An orchestrator or game manager is used to handle multiple and contrasting behaviours, thus including the generation of random events.

Advanced Features

- [x] **[+2%]** Appropriate usage of Prefabs.
    - I have created prefabs for reusable elements in my scene such as mountains, trees, coins, rocks, and enemies

- [x] **[+2%]** Levels/Menus are separated into distinct scenes.
    - I have separated my menu, hub area and my first level area into 3 distinct scenes, where interacting with the menu will load the next two.

- [x] **[+3%]** Evidence of code for limiting expensive computations (e.g., *raycasting*).
    - In EnemyBehaviour.cs, there is a raycastTimer variable that adds a time delay between each raycast check. This limits the amount of times the PlayerInSight method is called to reduce processing cost. The addition of a Coroutine checks whether the player is seen at intervals instead of every frame, which helps limit this computation.

- [ ] **[+2%]** Flocking techniques.

- [x] **[+2%]** Usage of Vector Fields.
    - I have implemented vector fields at the beginning of the level in the form of wind, pushing leaves and whatever walks into it upwards

- [x] **[+2%]** Usage of Particle Systems.
    - Particles can be found emitting from the star, and when the player collides with the water

- [ ] **[+2%]** Implementation of custom AI tools (AverageMinMax, *et ceteram*)