**COMP3567 Game Specification Form Student ID: hdsq38**

|  |  |
| --- | --- |
| **Marking Criteria** | **Describe how your game matches the criteria (Description of each item is limited to 50 words)** |
| **Game design matching the ‘Covid-19 Fighter in the UK’ theme (5%)** | |
| Justification of the choice of game type: | I’ve played games for many years and some of my fondest memories have been playing movement-based first person shooters such as Team Fortress 2 or Doom Eternal. I’ve decided to make a fast-paced platformer fps in this style. |
| Game story: | You are a newly built memory B cell in a body that has just been infected with COVID-19! Use your blaster and grappling hook to navigate the body’s veins while fending off any dangerous pathogens that stand in your way.  As the game progresses, the levels become harder to navigate and the enemies become more numerous as the infection gets worse. In the final level, even the friendly T cells have turned against you in a cytokine storm! You’ll need your wits about you if you’re going to beat this. |
| **Core development and implementation (30%)** | |
| Game scene (visual representation [2D, 2.5D or 3D], internal data structure): | The scene is a Unity3D game scene where objects are stored as vector arrays by the Unity back end. I constructed the more complex objects using the Unity ProBuilder package.  For the COVID cells’ pathfinding (described in more detail in the “Advanced interaction” section), each flying nav mesh is a 3D array of voxel objects, where a voxel is a Boolean stating whether or not a terrain object exists within the voxel. |
| Game flow and how it is designed (e.g., navigation, screen scrolling, levels): | The game consists of 4 levels of increasing difficult and stakes. Each individual level can contain any number of checkpoints. When a player enters the invisible bounding box of a checkpoint this becomes their respawn point (unless the checkpoint is less advanced than their current one meaning they have backtracked through the level). When the player dies, they respawn at their respawn point.  Each level can also contain a number of “battle arenas”. Each battle arena can have a number of enemies attached to it, when a player enters a battle arena they too become attached to it. Once a player becomes attached to a battle arena, all enemies attached to it spawn. The player remains attached to the battle arena until all enemies attached to it are dead, at which point the battle arena’s termination code runs (the level ends, a door opens, etc.). If a player dies while attached to a battle arena, all enemies attached to it respawn (and their health is reset to full).  The general layout of a level consists of a number of battle arenas separated by parkour sections. Although, later in the game (particularly the third level) battle arenas and parkour sections are mixed, requiring the player to use all the skills they’ve learned simultaneously if they’re to stay alive. |
| Game interaction (e.g., action detection and response generation): | The basic controls of the game are that of a standard first person shooter: the player uses WASD to navigate, the mouse controls the direction which they’re looking, left clicking with the mouse fires a projectile that damages enemies. More complex movements include dashing (via pressing LSHIFT) and using the grappling hook (by right clicking).  Beyond the above explicit interaction, the game also uses more passive forms of interactions controlled by the position of the player within the game scene (rather than just being triggered by the user themselves). For example, an abstract AI class that I wrote myself includes two functions MoveToTarget() and CanShootPlayer() both of which are determined by the player’s position and various other states of the game scene (such as shoot cooldowns etc.). |
| Game object (e.g., use of sprite, 3D objects, animation, multimedia): | As mentioned above, the game objects consist of vertex arrays with some fairly complex terrain objects (unity terrain, stairs, arches, doorways, etc.). Beyond this, I also used some models I found online for bacteriophages and coronaviruses. I imported some animations for these enemies which I modified to match the speed and intensity of the game. Beyond this, I made some of my own animations for the player and UI elements as well as creating the model for T cells myself with blender.  Furthermore, I imported the unity particle effects package from the unity asset store. This provided a set of generic particle effects which I combined and modified to suit my needs. I developed particle effect for the big walker’s poison smoke from scratch myself. This required an understanding of unity’s animation system as well as the theory behind billboarding and sprite generation. |
| **Game mechanics development and implementation (30%)** | |
| Main game rules / logics to control game progression, difficulty and end game conditions: | The primary resource the player must keep track of is their health. When the player’s health reaches 0, all resources are set to their initial state meaning the player’s work is essentially undone.  To maintain their health the player must build up a mobility advantage. This is an intangible, abstract resource that measures the player’s current advantage in terms of position and mobility of their enemies. The greater the player’s mobility advantage the slower the enemies are able to deplete their health.  There are 4 uncontrollable, tangible enemy types: Floaters, Flyers, Big Walkers, and Walkers. When the level is started/restarted (e.g. by the player’s health reaching 0), a fixed number of generic enemy resources are created and a random gate converts them into one of the above types (with probability based on enemy difficulty). The more enemies there are (and the more difficult enemies there are) the faster the player’s mobility advantage is drained, which in turn reduces the player’s health.  The player wins the battle arena when all enemies are dead. When a battle arena is complete the player moves on to the next, and once all battle arenas are complete the player wins.  The first diagram shows an in depth perspective of a single game arena, in which the player must manage their mobility and health resources while attacking enemies to win the arena. The second diagram shows a high level view an entire level, demonstrating how the player must win several battle arenas and how each arena’s difficulty (probability of failure in the diagram) increases with the number of enemies. |
| Control of game object abilities: | In the first diagram (depicting a detailed view of a single battle arena), the player has 4 abilities: Jump, Dash, Grapple, and Attack. The first 3 directly affect the player’s Mobility Advantage: Jump increasing it by 8, Dash by 10 and Grapple by 20. The player can Jump whenever they please but the Dash and Grapple abilities have a cooldown and can only be used when one of their respective resources is in the pool. The player can have up to two dashes stored but only a single grapple. Once the player uses one of these abilities the resource is converted to mobility and after a few seconds is replenished to be used again.  At any point in time, the player may attack an enemy, an attack has a 10% chance of killing a Floater, a 20% chance of killing a Flyer, a 20% chance of killing a Big Walker, and a 50% chance of killing a Walker. If all enemies are dead and the player still has some health left, then they win the arena. But they should be careful, the asynchronous time mode means that the player must effectively divide their time between attacking enemies and gaining a mobility advantage.  In the second diagram (depicting a high level view of an entire level), the entirety of the first diagram (an attempt at winning a battle arena) is described with a single action: Attempt Arena. When the player attempts an arena they have a certain probability (50 - number of enemies)% chance of winning. Upon winning an arena, the player progresses to the next arena which has more enemies. Upon failing an attempt, the player and enemies respawn and they must try again. Once the player has defeated all arenas, the level is complete. |
| **Good use of game engine (12%)** | |
| Justification of the choice of game engine (pyGame, Unity) in terms of suitability of matching the theme and the expected target audience (game player): | Given the first-person nature of the game and the complex physics-based platforming, the most logical choice was Unity as it’s physics engine in more advanced and it’s simple to develop 3D games. |
| Types of user input supported (keyboard, mouse, joystick, etc.): | Through using Unity’s extensive Input API, the game supports the use of keyboard and mouse inputs. This generally the standard for such fast paced first person shooters as precision aiming is a must. |
| Types of game object interaction supported (e.g., event triggering, collision detection): | Game events can be triggered in several ways: dialogue is triggered by a player approaching a point of interest, battle arenas being cleared (all enemies killed) trigger specific events, the player coming within a certain distance of enemies triggers their AI etc.  The player collides with terrain objects and enemies, enemies collide with one another, COVID cells also implement collision avoidance via a push force that activates if they come within a certain distance of terrain objects or other enemies.  More specific interactions with game objects include evil T cells which will track the player, changing states based on the amount of time they have been able to follow the player (forcing the player to keep moving) with the final state being firing. The grapple hook also allows the player to interact with terrain and the game scene in different ways and the floating grapple points provide a unique way of navigating around obstacles. |
| Other game engine features used (e.g., asset, incorporation of external libraries): |  |
| **Demonstrate creativity (15%)** | |
| Effective use of multimedia content: | I imported models and animations for the enemies (stored as .fbx files) as well as created several of my own models from scratch in blender (such as for the T cells or blaster).  I also imported some textures from the unity asset store which I modified in photoshop so they look like the inside of body. These textures came with normal and bump maps which also required modification and provide a sense of depth and realism to the textures in game.  The game makes use of several sound effects which I imported from the unity asset store (stored as .wav files) which I trigger via direct or indirect interaction by the player.  Furthermore, as described above I used a significant amount of particle effects to provide realism and visual appeal to the game (some effects I imported and some I made myself from imported textures).  Finally, I was able to use Unity’s extensive game object construction features to build complex game environments with interactable terrain and other objects (such as stairs, doorways etc.) these more Unity-specific objects are stored as .asset files. |
| Advanced interaction implemented (e.g., game physics, object tracking, steering behaviour): | All movement of the player and enemies is based around physics, using character controllers that I coded from scratch. This allows for complex interactions between the player and other enemies, one such example of this is rocket jumping. The main mobility tool the player has access to is their rocket launcher. Shooting it at enemies will push them around and shooting the floor beneath your feet will propel you into the air, chaining rocket jumps together can result in quick movement over complex terrain. Beyond rocket jumping the player can use dashes which impart a force on them and a grapple hook which generates a spring connection between the player and another game object, chaining these moves together results in a huge amount of mobility for the player.  For the COVID cells I wanted them to fly in an organic and dynamic way. However, the Unity navmesh only supports agents traversing 2D terrain. To get around this, I implemented a hierarchical A\* search where the game scene is divided into different 3D navmeshes through which COVID cells navigate until they reach the player, I also added a heuristic where a path that can see the player (using raycasting) will immediately terminate the algorithm as from there the COVID cell can fly straight towards the player, although this can result in a sub-optimal path being used it does find a fairly good path and terminates much quicker than the standard algorithm which was necessary as 3D A\* is computationally intensive (even with the hierarchical improvement).  To make the COVID cells’ routing more natural, I also implemented collision avoidance and steering behaviour.  All enemies that shoot (including T cells) implement object tracking and detection. In particular, the T cells efficiently locate a nearby enemy to which an unobstructed vector can be generated. This is done using physics check spheres for efficient collision detection and raycasting to determine if the object can be seen directly. |

**\*Note:** Your work must be done by yourself and comply with the university rules about plagiarism and collusion. (https://www.dur.ac.uk/learningandteaching.handbook/6/2/4/)