# Exercise 3 – Unity Project

Harry Findlay

***The process described is done so in-order of implementation.***

I started by creating a base-line player script which contained basic information regarding player speed, move-to and starting positions (checkpoints), and a simple Move() function.  
  
The Move() function contained a simple conditional check to ensure the player has a checkpoint(s) to move to, otherwise the a simple “You Won!” debug was displayed on the in-engine console. The updating of checkpoints is controlled by a simple index integer which is incremented until it meets the length of the list of checkpoints.

From here, two more scripts were developed: ObstacleScript and ObstacleManager.

The Obstacle script contains a simple tag update – updating the tag to any obstacles containing the script to “Obstacle”. This is important as this comes later on to check against player collisions. A rigid body was also added to the prefab, ensuring to de-active gravity and constrain the obstacles rotations and position.

The ObstacleManager Was then developed, and has the sole purpose of producing and handling the Instantiated obstcales. To insert more detail, lets talk about this.

The ObstacleManager initialises all the data required, some data including: the prefab of the obstacle, a reference to the manager game object for parenting the objects, spawning min and max (used in generating a random position within this range) and, lastly, distance checking values for a more polished generation of obstacles.

The scrips functionality starts by finding the parent /manager game object before continuing on and instantiating 100 randomly positioned obstacle game objects. Later, as polish, after instantiation, these obstacles were made children off the manager game object (for in-engine management) as well as scaling all obstacles to the size variable, which designers can use in engine (public) to control the obstacle local scale.

The random positions of the obstacles are generated by a function: private Vecror3 GenerateRandomPos() which uses the thresholds set in the initialisation stage of the script, then returning the random vector 3 for use elsewhere. In this case, within the generation loop.

Once this was implemented and tested, I then revisited the player script with the intention of adding a win and lose function. Within the WinGame() function, called if the list of vectors length is met, a simple audio clip is played (NOTE: Beware this audio file, the volume does not seem to scale with it properly – protect your ears. Test with headphones off initially). This function then extends to emit a green particle effect and congratulating the player on the successful movement.

The LoseGame() function is the same as the WinGame() function however, is called if the player and the obstacle rigid bodies collide. The main differences include a red particle system and the player is Destroyed after a 2 second delay (IEnumerator and Courantine).

Lastly, within the player, collision checks were added. This simply looks for the “Obstacle” tag when a collision happens and when this is registered – LoseGame() is triggered.