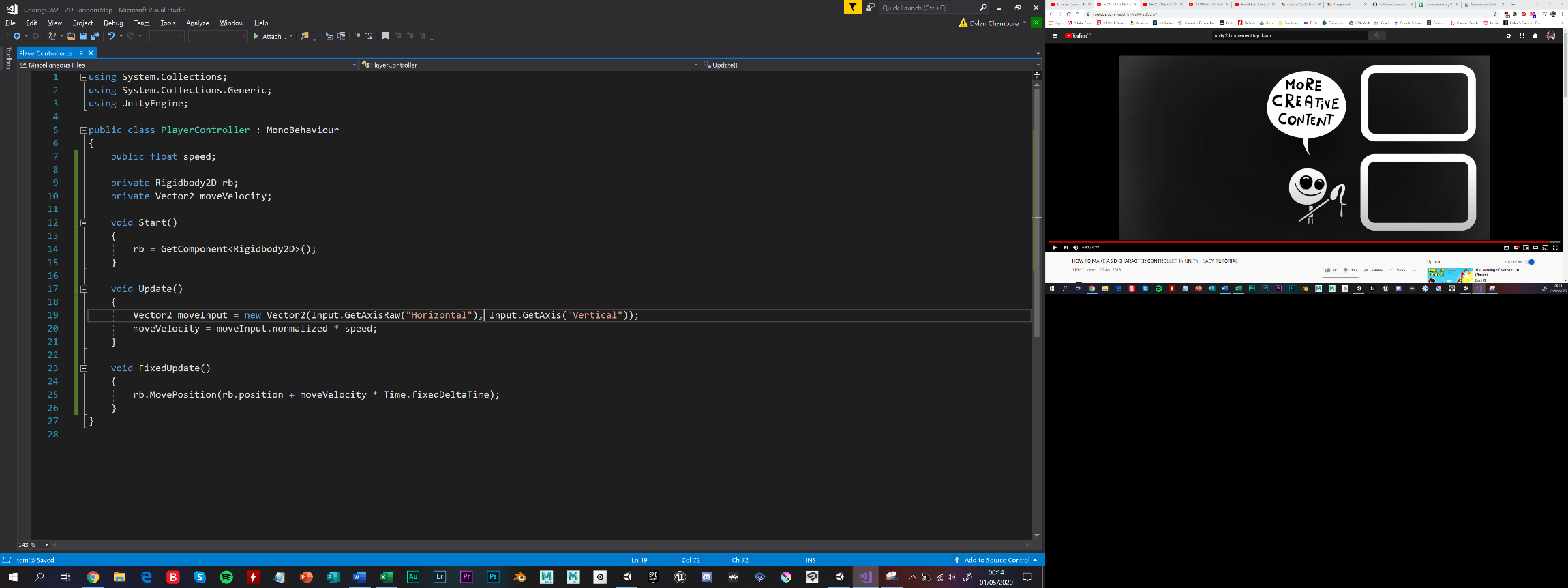
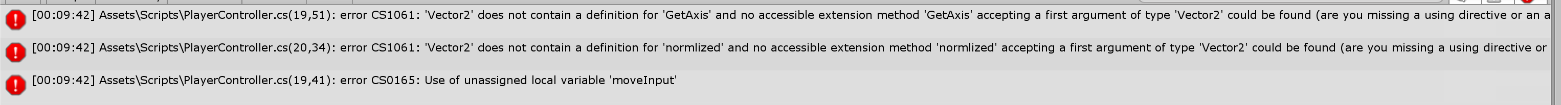
**Learning Journal**

**2D Movement Controller:**

The first package I started with was making a 2D movement controller. I created the *‘Player Controller’* C# script and added the dynamic body type with **Rigidbody2D** for the **Vector2** to move the object from different directions with a 2D viewpoint and with the right amount of speed.

The player object sprite with **moveVelocity** for the **moveInput** which requires the sprite object to attach the controls for the up-down movement. In the scene contains the circle sprite representing the player and it showcases the 2D controller movement and letting the player control in any direction with 'speed'. While making the script, I received errors for making spelling mistakes in the script and had to correct the errors [the image below] to get it working which make a smooth movement with a strong stop with the **GetAxisRaw.**

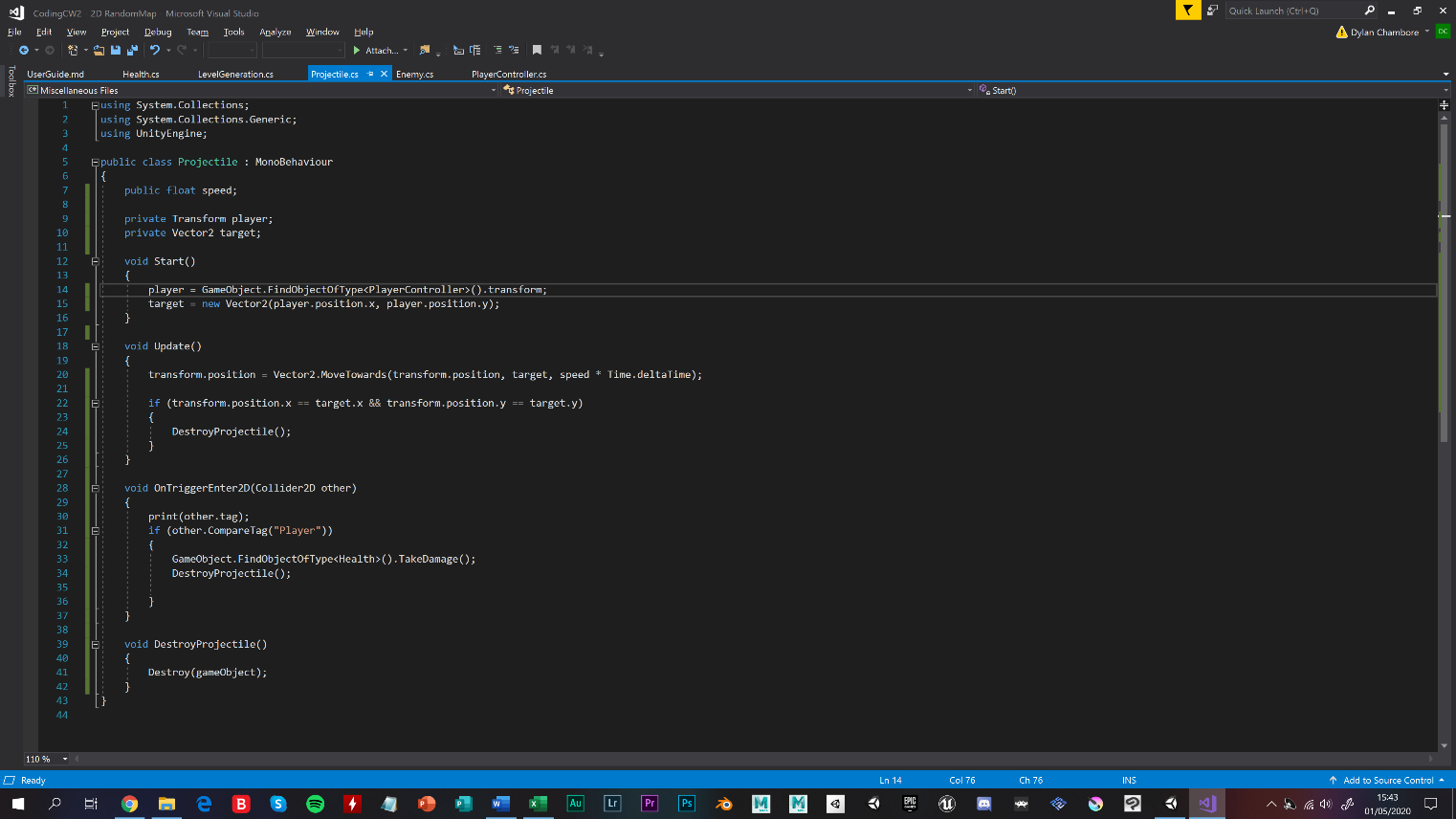
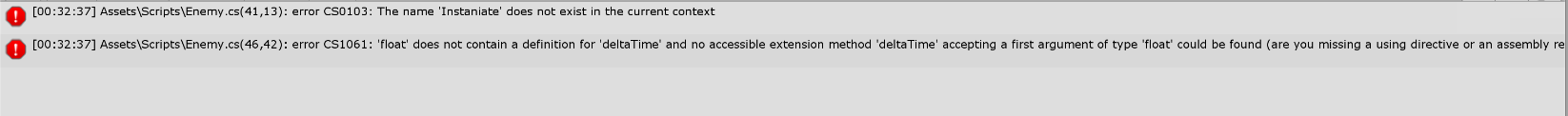
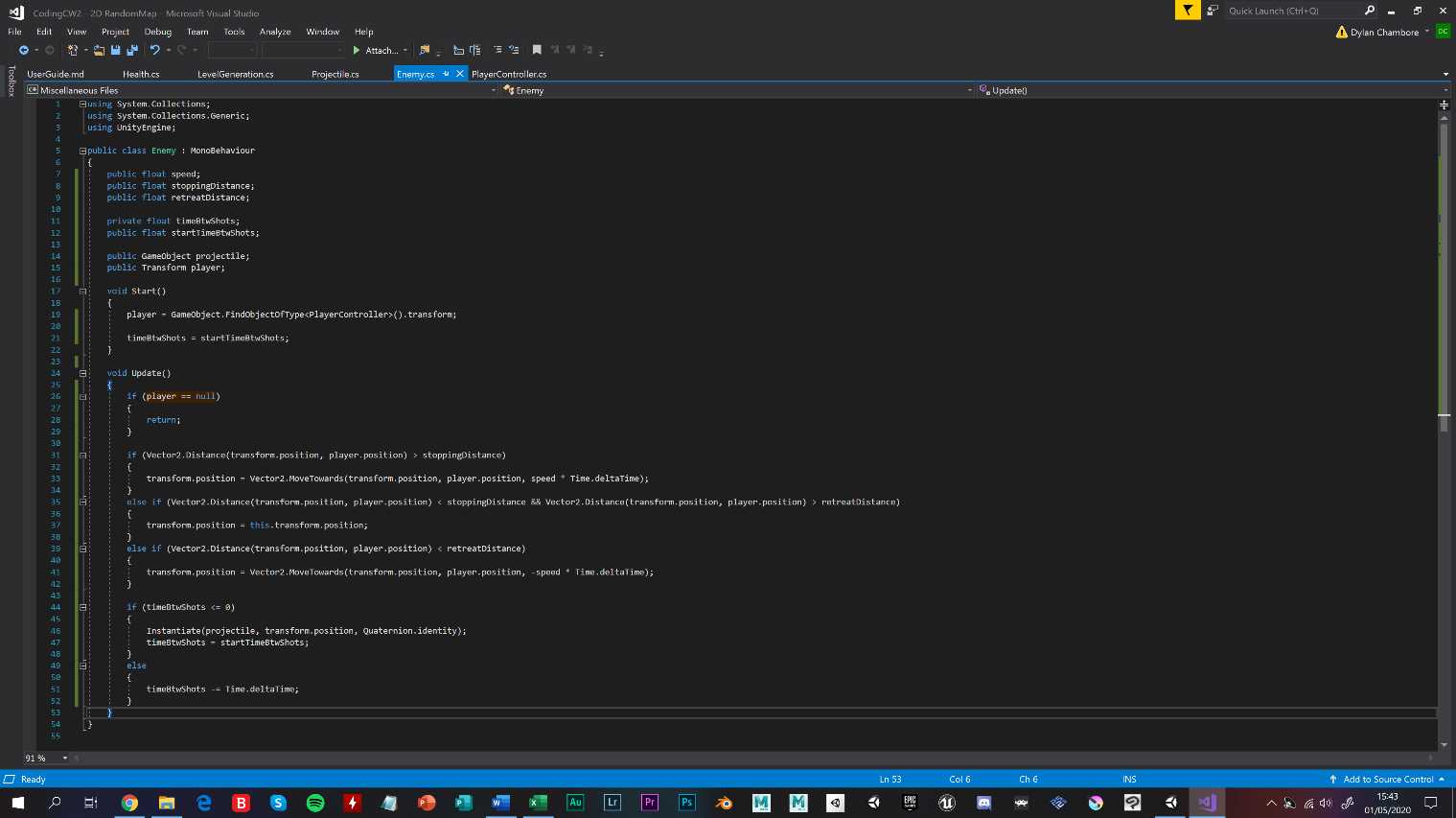


**Enemy AI:**

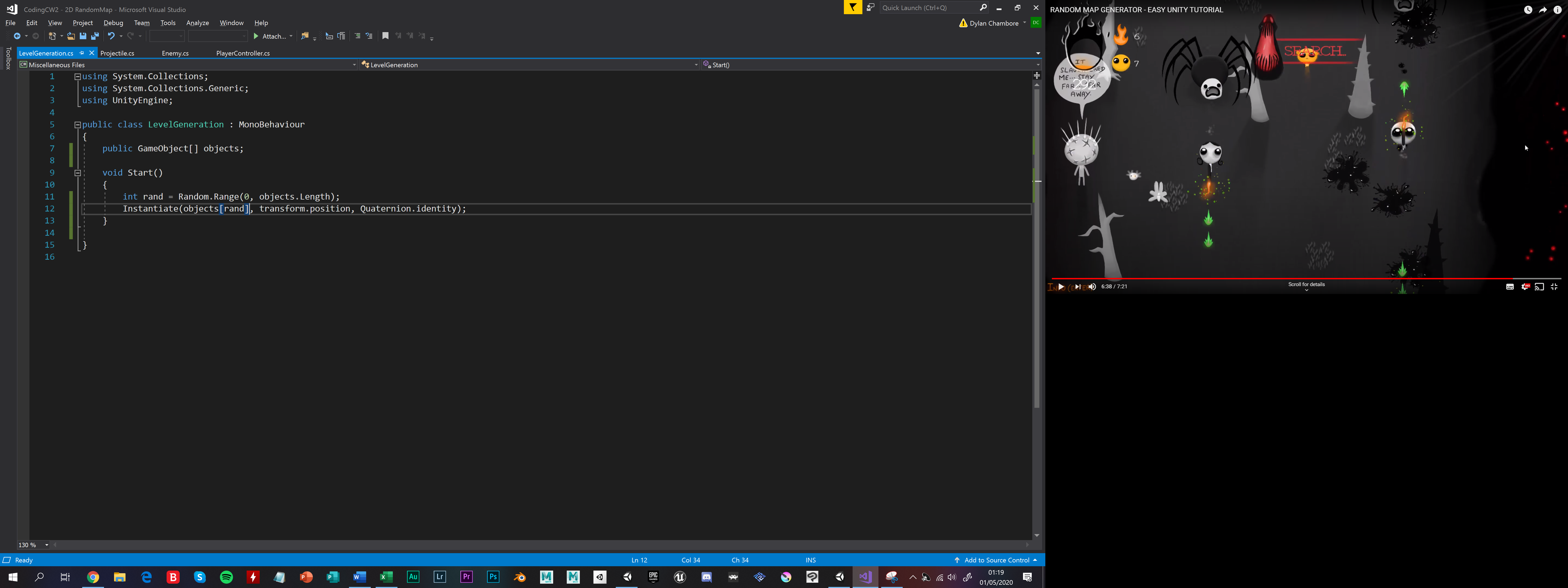
The second package I made next was making Enemy AI functional which is where the player is near the enemy game object and uses the *'projectile'* to fire at the player. When making this behaviour script was using the **FindObjectOfType<PlayerController>().transform;** when it connects with the values of destroying objects. The aim was the player controller gets near the enemy, which it distances itself and comes closer to the player and activates the projectile.

This behaviour tries to get the projectile with the Collider2D with the enemy timing the firing rate depending on the player's movement in the package. It uses the target variable with the Vector2 values while the camera remains stationary. The enemy component enables the firing function by linking the **target, speed \* Time.deltaTime.**

This test demonstrates just the Enemy and Projectile scripts. It contains the circle sprite representing the player and a square sprite presenting as the enemy and positioned at each side. When running the scene, the player moves and the enemy retreats slowly at certain space while firing the projectile at the player. I did have a few errors when making the script and it took a while to solve the error and must restructure the assets as prefabs and managed to clear the errors and got it working.



**Random Map Generator:**



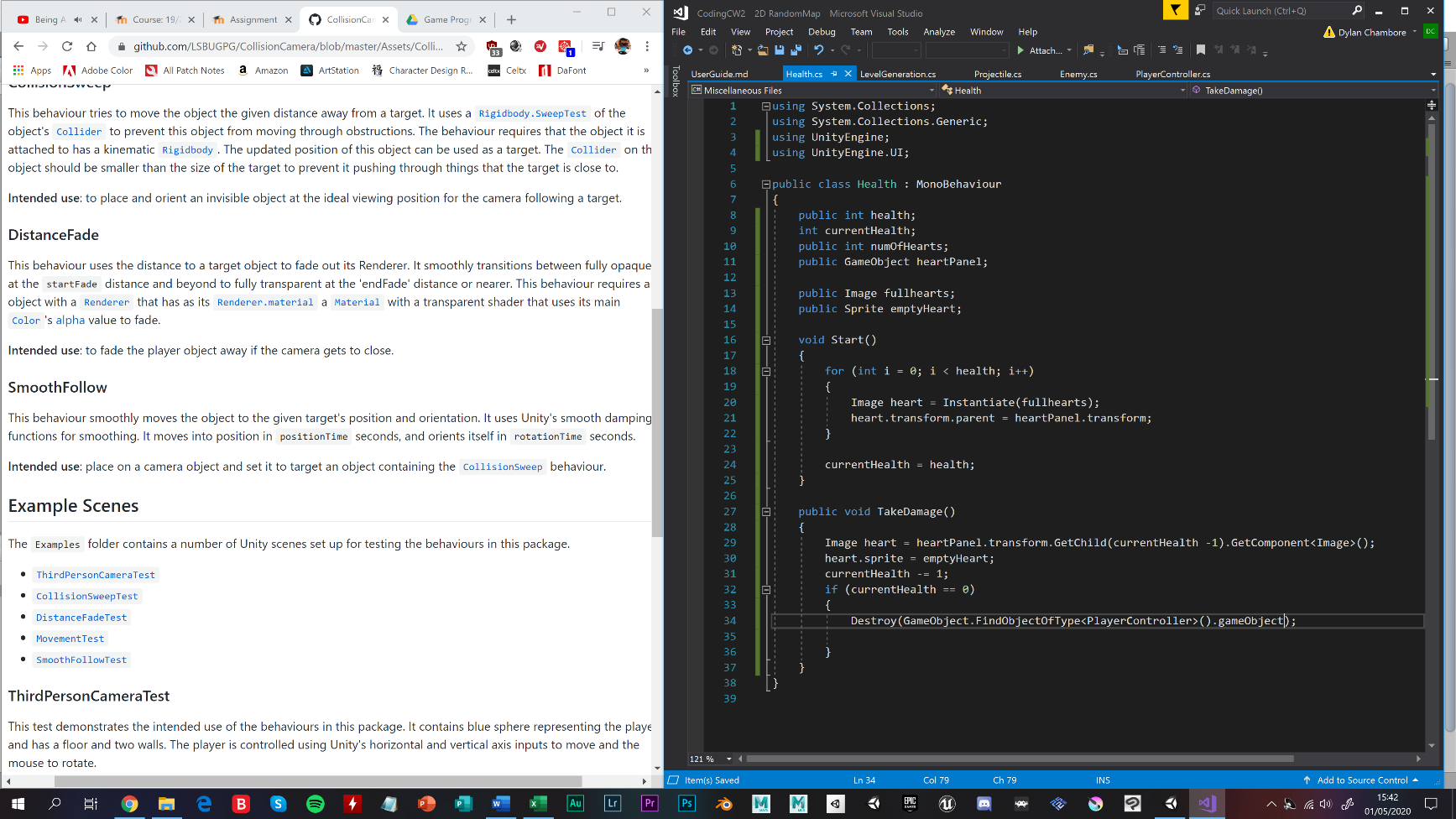
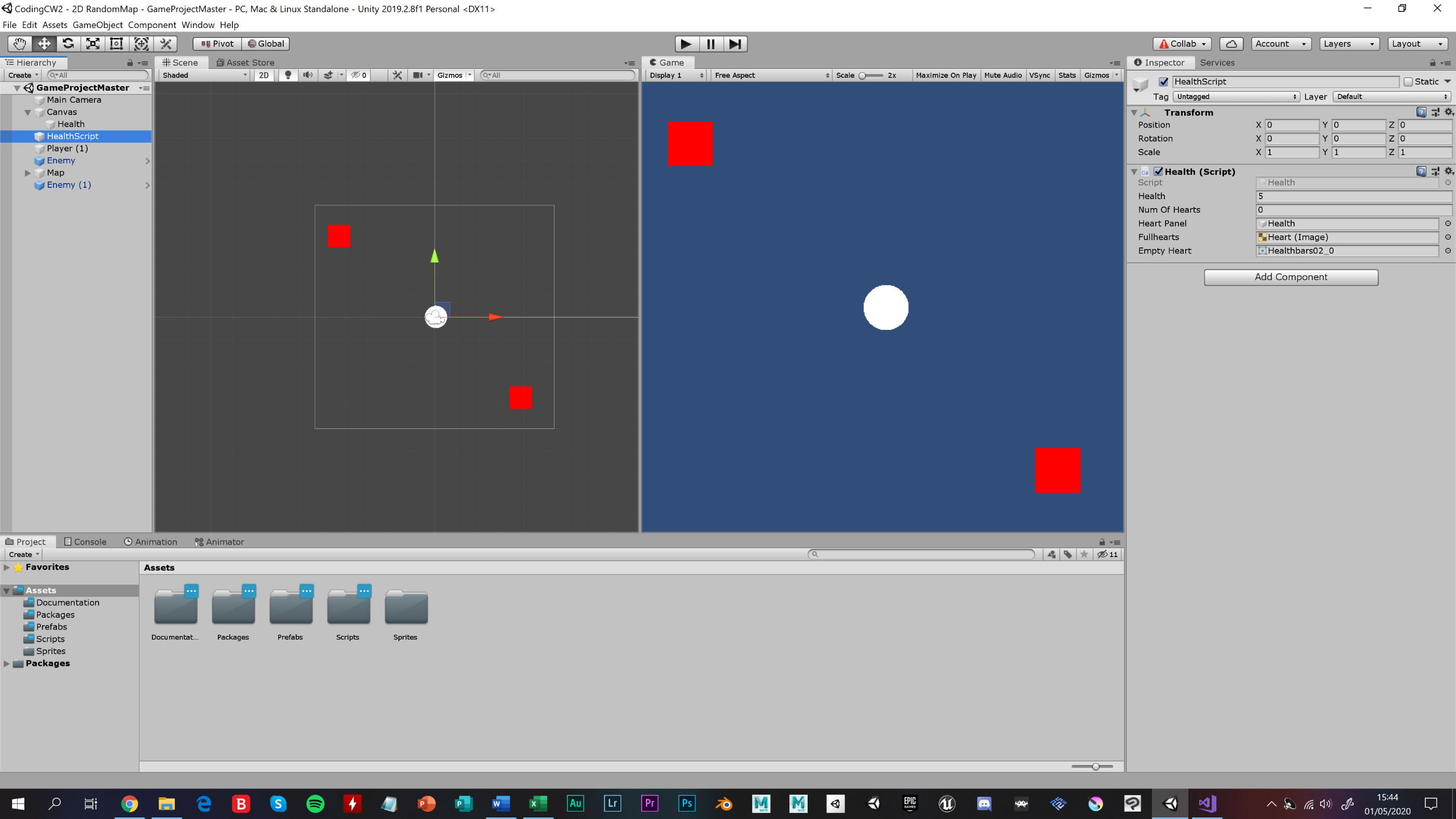
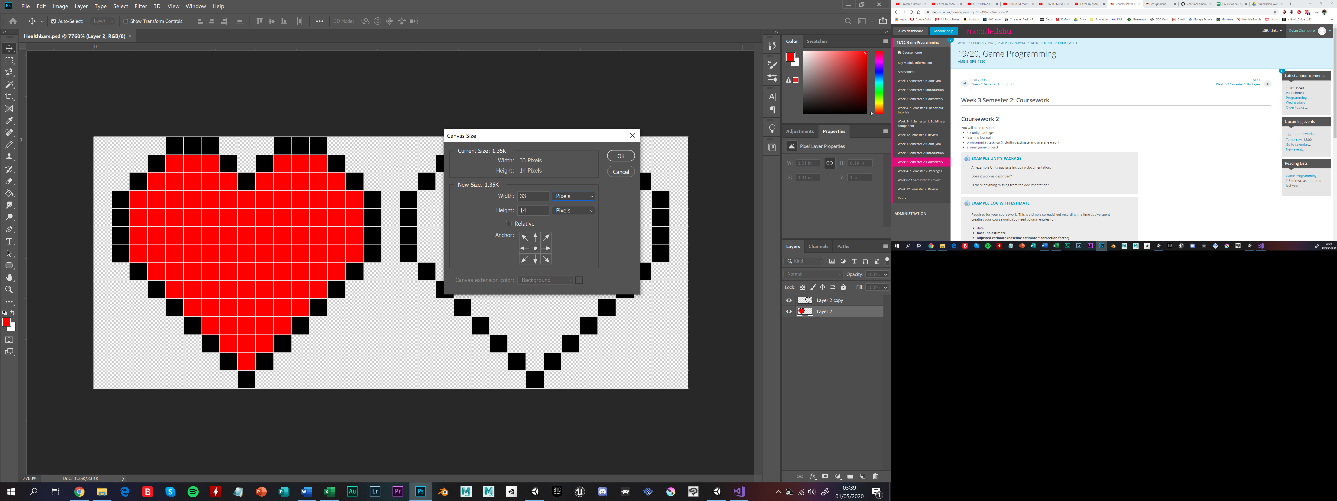
The third package I made was the Random map generator. This behaviour uses the spawn points function for the level generator by turning it into a prefab object and using the gizmos icons and spreading it around the scene and each time the game load, it generates the sprites randomly and varies differently. To add a function for the map to generate randomly and makes the scene different.

This test demonstrates just the [`*LevelGeneration*`] behaviour. It contains multiple gizmos icons spread on the scene and has a variety of sprites to fill in the scene. Each time the scene runs, the positioning and shapes changes and the player must navigate with these spawn points while facing the enemy AI. I have not gotten errors when making this and it was the fasting package I made so far in this project.

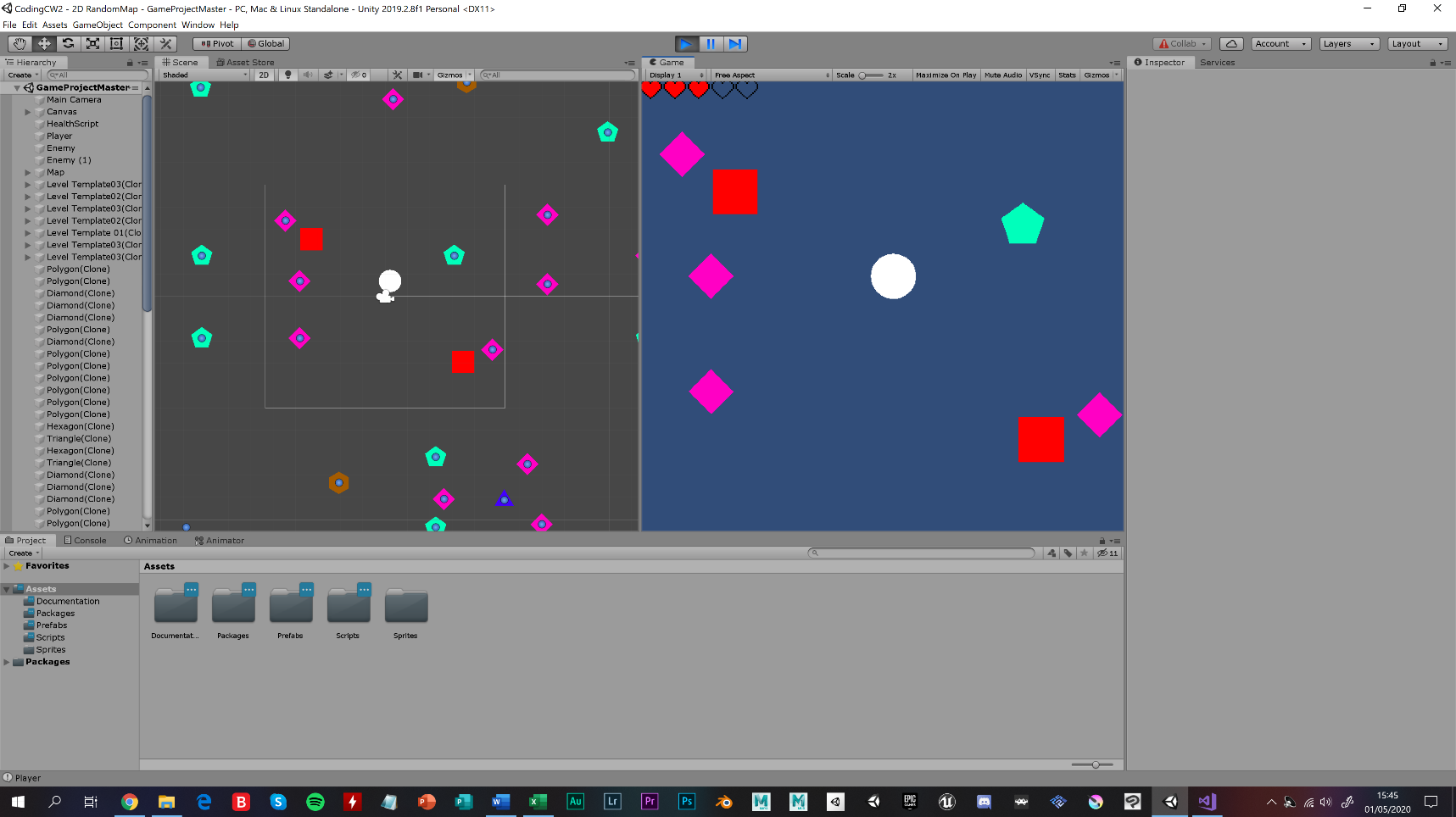
**Health Bar System:**

The last package I made was the health bar system. This was the hardest package to make due to changing the UI design for the heart that I made on Photoshop. This behaviour displays the number of health bars on the canvas and using multiple images to set up the UI with the [`**Health**`] behaviour. It can alter the amount **'numOfHearts'** and let the player game object have certain 'health' in the scene.

To place the health bar behaviour onto the [`**PlayerController**`] in this package, I had to canvas for the hearts and had a separate game object for the *health script* that calculates the health in-game and how many empty hearts that can be added in the scene. This test demonstrates with the **Destroy(GameObject.FindObjectOfType<PlayerController>().gameObject);** that links with the [`**PlayerController**`] behaviour. It contains the health bar sprites position as the player's health in the scene. It sets up a certain amount of hearts active and the empty hearts is representing the total amount of health that the player has in the scene.



**The small game project:**

After making four different packages, logging in the information for the programming task log and making the user guide in the unity files. Once all packages are combining into one small game project [the image below] where the white circle has five heart lives and has to survive from different projectiles from two enemies and once the lives are gone, the game ends and the functions stop with **if (player == null) {return;}**. This small project has not had too many problems with the errors, the spawn points change each time the scene is run and mixes up the flow of the game.