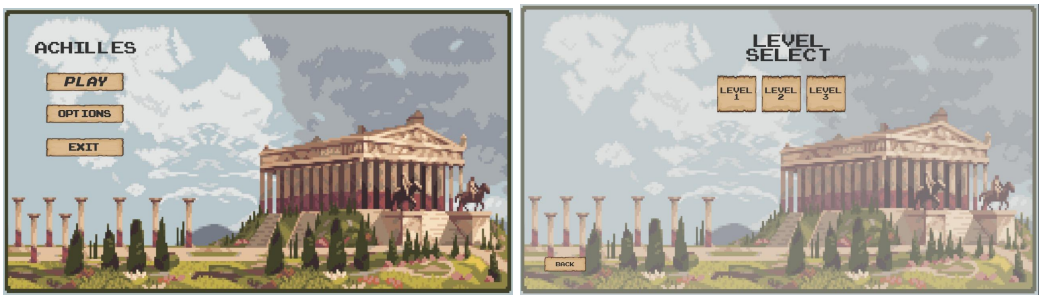
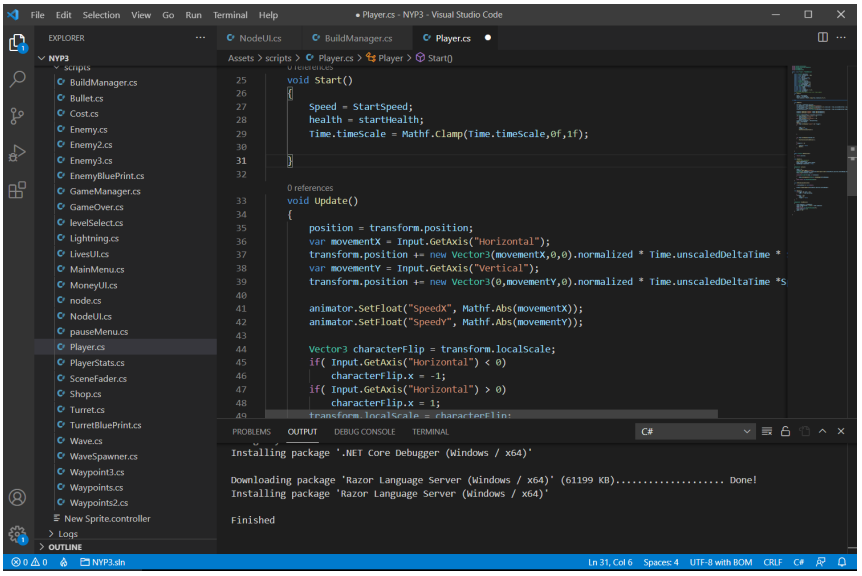
OUR GOAL IN CREATING THIS PROJECT

This project was prepared for the Object-Oriented Programming class of Trakya University under the tutorship of Emir Öztürk. The project consists of a 2D top-down tower defense like game containing some 2D platformer elements, made in the Unity game engine developed by Unity Technologies. The foremost objective of this project, outside of completing and successfully delivering a project for the class was to gain more experience in both C# and the Unity game engine development environment. Therefore broaden our horizon on OOP fundamentals and create new opportunities for the future. We wanted to differentiate our project by bringing two different genres together and switching in between them occasionally to keep the experience new and different in each scene.



TECHNOLOGIES USED IN THE MAKING

The whole project is written in C# using the Unity game engine and its scripting ability. Each GameObject that appears in the game, outside of the design elements, has its own script to receive commands from the user or to act in an automated way.



THE PLANNING PROCESS

After the general direction and topic of the project was decided the majority of our time was spent researching and learning. Although we had experience in C# We were all new to Unity scripting and because the built-in methods and objects were specific to Unity we spent time

learning these methods and objects until we felt confident enough to start writing our scripts and keep learning as we progressed further.

Each member of the project was assigned a different task. This could range from UI design, game design, and animation to creating a wave spawner, special abilities that trigger with certain key inputs, or coming up with a "simple AI" for enemy behavior. Team meetings were held weekly to discuss progress and what features should be added next.
According to the results of these meetings, new features were created and implemented or previous ones were improved for better performance.

THE PROJECTS FEATURES & THEIR SOURCE

Prefabs are widely used in Unity. They can be thought of like objects since they contain multiple components in them(sprite renderers, colliders, and scripts), can be duplicated, yet every duplicate can be different from its predecessor. In short, they make your life easier than creating a new GameObject each time.

Tower bases placed in specific locations on the map contain sprite renderers and colliders that allow us to draw on the scene and register a mouse Input, making a tower prefab appear on the center of the defined tower base using the onClick() method.

Our player GameObject can be controlled with keyboard inputs with the Input.getKey() method. The same method is used to trigger the special abilities the player has.

Towers detect enemies with the tag "enemy" and calculate the distance of each enemy on the scene relative to the position of the tower. If the distance is less than the defined range of the tower a bullet prefab is instantiated and starts to travel to the defined enemy's position which is being updated every frame. After the bullet prefab arrives at its destination the TakeDamage() method for that enemy is called and the bullet destroys itself.

Enemy prefabs follow many previously placed waypoints until they reach the last one to be destroyed outside of the scene. If they encounter the player on their way and the distance to the player is less than the previously defined chase range, the prefab starts to move towards the player to "chase" it. If the range falls under the defined attack range the player starts to take damage. After the player leaves the enemy prefab’s defined chase range, it will continue to follow its waypoints until the last one.

The WaveSpawner object consists of three different scripts in communication with each other. The EnemyBluePrint file contains each prefab to be spawned, the number of that prefab to be spawned and the rate it should be spawned in. The Wave file contains a List that stores variables of the type EnemyBluePrint. Finally, the WaveSpawner file takes that list and iterates over each element, and instantiates them according to the data provided in the EnemyBluePrint. Because the EnemyBluePrint and Wave files can be edited over the inspector it gives us full control of how many waves there will be and how many different enemies each wave will have.