Assignment Project Structure and Instructions

The assignment project is a simple example of how a slightly more advanced game project would look in the real world. Some of the main parts have been simplified and the approach to overall architecture has been limited to the scope of the course.

Game Design Specifications

This is a very simple concept of game where a player will control a tank which is meant to traverse a map, overcome challenges and defeat enemies!

Player

The player will control the tank from a third person perspective using WSAD for keyboard and/or other controller configurations. The tank can move back and forward, and the choice of camera behavior is up for iteration. Make sure to add a Cinemachine camera!

The tank turret is separately controlled by the mouse, both rotation and elevation.

All the player mechanics tuning variables should be alterable from an in-game debug menu.

Once the player reaches the objective a small menu should show the UI game over with the specifications bellow

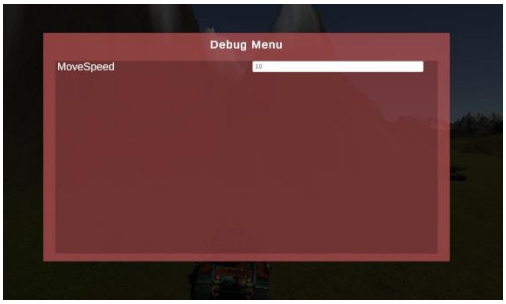
AI Behavior

The map will contain a series of different tank enemies (min 2, different, max. 5, different) that will try to chase and destroy the player tank, in attempt to stop the player from reaching the objective.

For the scope of this assignment, shooting is not necessary, but player detection is. Enemy tanks will spawn in certain areas of the map, when the player hits specific triggers, and will default enter a patrol state, moving around on preset Route. Once the player is in range, the tanks will attempt to follow the player. Once outside the detection range, the tanks will go back to their previous route.

UI Debug Menu (*Mandatory for grading*)

The assignment will need to feature a debug menu where a designer can tune live the values of the player behavior. The debug menu should and it is properly implemented in code, but ONLY available in the debug version of the game and not the release one.



Game UI (*Optional, but considered for VG grading***)**

The game should feature two main UI elements:

1. A Main Menu with Start/Quit buttons UI which will reset the game state. The UI should be created, handled and instantiated based on the current architecture. See the Debug Menu code model.
2. A small UI panel at the end of level containing the text "*Congratulations, you have finished the level!"* with a Restart button should appear once the level end is reached. On clicking Quit, the application should quit in a build (it will not work in the Editor). When hitting Restart, the game state should reset and game either restart either show the main menu
    - One way to handle game restart is to add in the *IEntityHandler* a ResetGameState() function. Then in each of the corresponding classes, make sure you reset the state for the player (position, health, etc.) and the enemies state to be able to play again
    - If you have other mechanics that need to be reset to an initial state, make sure to add a similar functionality for doing that as well

Project Template Information

The project provides a series of preexisting conditions. That includes a project setup with separate scenes for entities and the actual level. The project features a working environment using additive loaded scenes. The code for automatically loading the scenes in the game build is already present. I order to work with the scenes, make sure when you load the project to go into the top ***Unity Menu/Mechadroids/Load Scenes.***

In the Project windows, the /Asset project folder is organized in several folders:

- **Editor** – contains script that can only be executed in the Unity Editor and are never packed in a build. It contains the above script to load the scenes from the menu
- **External** – ignore this, it contains the assets for the tank, projectile, terrain, etc.
- **Input** – All the assets and files necessary for the Input handling
- **Scenes**
- **Scripts** – scripts are sorted here based on what are they used for
- **Settings** – contains the scriptable objects settings assets and the folder /Resources from where we can load asset dynamically from code
- **Etc**.**(**not important**)**

The main scene for booting the game is *Boot.unity* which should always the scene at index 0 and the one loaded by default. This contains a script that will load the Level scene and the Entities scene.

The Entities scene contains an *Entrypoint.cs* class that should initialize most of the game systems. It will load the necessary assets from disk, create the instances for various systems and initialize them.

All the static data, like prefabs to be instantiated, settings for players, enemies, route paths etc. should be contained in scriptable objects (referred here as settings files) which can either be loaded dynamically from disk, either provided as references to the classes that need them using constructor injection. Generally loading settings from disk should be kept to a minimum. Ideally no more than a handful of settings should be loaded, and those settings should contain references to other needed settings.

Both the character behavior and the AI behavior should be designed programmatically using the State Pattern as shown in the course.

Scene Monobehaviours should not contain active behavior with some exceptions. Most of the behavior should be in the handler and state classes. The Entrypoint's *Awake*(), *Start*() and *Update*() control the overall tick of the game. Each of the systems will be instantiated and *Initialized*() here and updated with their respective *Tick*() functions (Tick, FixedTick(), LateTick() etc.,).

The project provides many of the prefabs needed with associated Monobehaviours (if needed) that act as scene data providers for the behavior classes. More prefabs can be created if necessary:

- AI – prefabs for the enemy tanks. They contain everything, from meshes to shaders, animators etc. The correct use of animators and animations is not really graded. They should work by default. Only one tank is configured as an AI with the proper reference script pointing to the enemy settings. You will need to configure the others to be functional.
- Essentials contain backup prefabs for Entrypoint and Camera setup
- Player contains the player's tank prefab and the hit marker
- UI contains the necessary prefabs for building the UI
- The prefabs for the Main Menu UI and end game UI are not provided.

Several C# classes are provided to help the students with the setup. That includes interfaces, templates for the state patterns, comments to guide and some gameplay functions for character movement and AI behavior. The student will need to "**complete the puzzle**" and assemble the functionality from preexisting code and whatever new code is needed to make things work.

Assignment Objectives

The main objective assignment is to make the prototype functional with the following specifications:

1. Read the above design specifications in detail to make sure are properly understood
2. Initialization of system architecture needs to be respected for any present features and followed for future features. This is very important!
3. The Settings assets need to be created and populated for the game to work.* (see bottom of the page instructions)
4. The Player functionality needs to respect the State pattern. The model is already provided. Student will need to Add/Configure the existing states to be functional.
5. The Enemies functionality needs to respect the State pattern. Same as the above, the model is already provided, it just needs to be completed. Some functionality for detecting and following the player is provided but it will need to be added to the correct state, like the PatrolState, AttackState etc.
6. There should be at least two enemies that correctly detect and follow the player when in range, according to the above design specifications. Students can add up to 5 enemies with different paths and behaviors. There are prefabs provided and it will count for VG
7. The game should have instantiation triggers for enemies. That means that enemies will spawn on the map ONLY when the player hits a certain triggers. As for behavior and challenges, nothing complicated is necessary. Shooting, destruction etc. are not necessary and they will not be graded but they will be noticed and appreciated.
8. There should be triggers for detecting the player reaching the end of the level and the optional associated UI according to the specifications
9. Create a custom tool for the Route Scriptable Object that, besides the route data, will show a button that once clicked will display a set of unity Gizmos on the map that will TOGGLE the route information for the route specified.

*Important!

There is a set of Settings, aka Scriptable Object classes, that are necessary for the game functionality. However, the assets themselves are not created. The student will need to make sure all the necessary assets are created or else the game will not function. The preexisting code will hint at their necessity.

These assets scripts are *AISettings, EnemySettings, Route, PlayerSettings, PlayerPrefabs, UIPrefabs* and they will need to be created (from the *right click* - Create Asset *Menu/Create/Mechadroids*), populated and adjusted according to the functionality required.

The code will tell you where and how are needed. For example lines 26-28 in Entrypoint will let you know that the PlayerPrefabs asset needs to be somewhere in a resource folder. That happens to be in *Assets/Settings/Resources* :

playerPrefabs = Resources.Load<PlayerPrefabs>("PlayerPrefabs");

Additionally, The *PlayerEntityHandler* and *EnemyEntityHandler* have commented code as methods that you should use in the respective states. Using this code is optional and you can adjusted it to your needs. It is there only to help you with the enemy and player behavior.

You may add as much features as you wish but the most important thing is for the game to be functional both in editor AND as a build. You can also use any packages external or internal you wish. Use of AI tools is not recommended but not really prohibited.

Delivery

Deliver the project as a .zip file (location will be announced in course). Each zip file should contain the following files and folders:

1. Assets
2. Packages
3. Project Settings
4. .editorconfig
5. *.sln* project file

You do not need to deliver a build, but the game should be playable as a build in case I decide to make one.

DO NOT INCLUDE ANY OTHER FOLDERS and files like Library, Log, .csproject etc. because they are NOT necessary and will only make the zip files massive for no reason!!