Technical Report – Game Data Project

Samuel R. Mortimer

Part 1:

Researching the LevelManager script resulted in breaking down the code into its key “phases” of activity. First, the script runs a kind of “clean up” to remove any already present items strewn around the scene. From that point, it fills each of the variables we’ve set up using information parsed from the TMX file Tiled creates. Finally, once it’s obtained the needed data, it populates the scene with sprites and objects based on what it’s obtained.

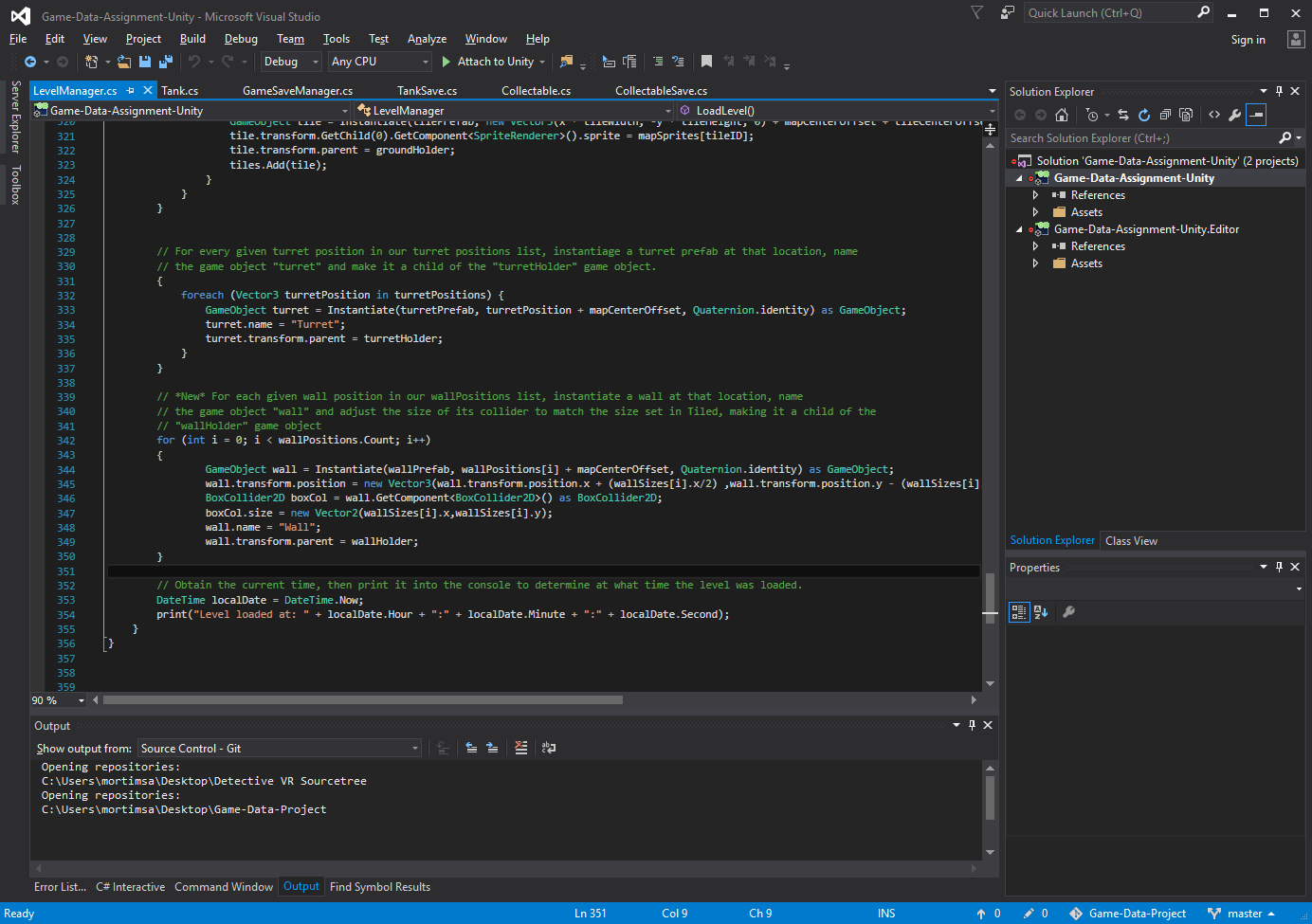
As for step’s 2 and 3, I decided to add proper collisions to the map. Drawing objects that would stop the player’s movement if they passed over any of them.



The level would now be a task to get from point A to B. Getting into the compound while avoiding turret fire. Parsing the new object data ended up being fairly straight forward. Initially, it runs much like the turrets, obtaining a needed position for each to spawn. However, each of the walls needs a second variable, a reference to its size. Given that’s the case, rather than using a “foreach” method that the turrets used to spawn, a “for” loop was determined to be more fitting.

Reflective Statement for Part 1:

When each wall (a basic empty object with a 2D collider) was placed, they were each put at the top-left corner of where they should be. So adjusting them became a two-step process. First, the instantiate position was shifted by half of the X and Y value of each wall’s size, and then, the wall’s colliders were re-sized to match the X and Y values. This resulted in accurately placed walls, with only a little modification of the Tank to make it function fully (adding physics2D and collider, and adding functionality where player can hold right mouse to reverse and get away from walls). These ended up being fairly simple fixes to the problems however.

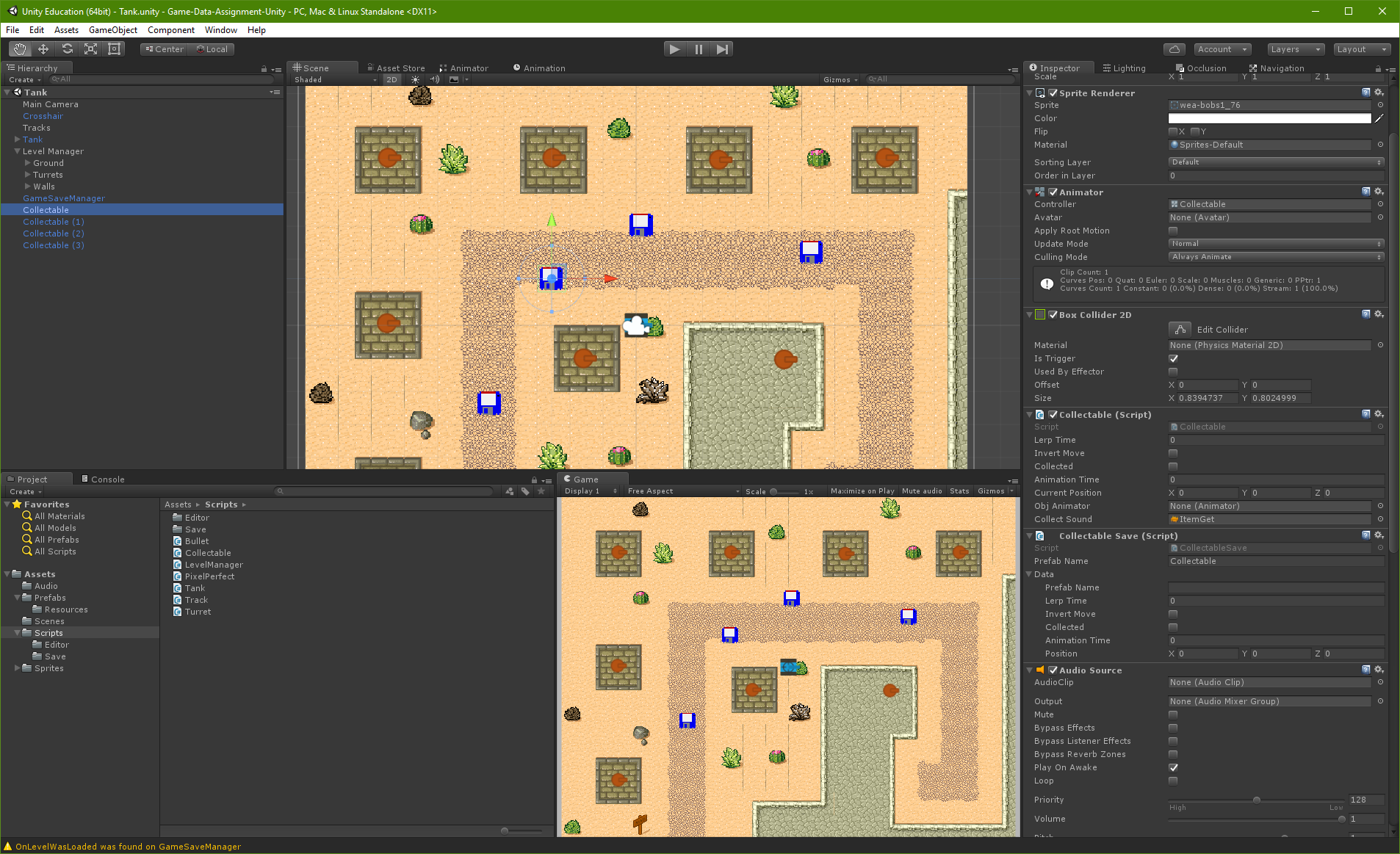


Part 2:

For commenting, it’s been detailed that in GameSaveManager.cs, a list of sevable objects is made, determined by any object that makes use of the base “Save.cs” script. It handles destroying old objects and calling the instantiate methods (through deserialize) of each after, along with making sure that the load function only triggers by a player-induced level load.

TankSave.cs uses two different kinds of base classes, first of all Save.cs and BaseData.cs. These apply to all savable objects, with the functions and classes therein being modified to be more specific to each item (eg. Each item has prefab name, serialize, and deserialize, but tanksave in particular has Tank scripts and its own modified Data script).

As for Step 3, I decided to include a “collectable” item. This item would spin in place, raise up and down, play a sound when collected, and vanish after doing so. To do this, 5 variables needed to be passed to the data structure to be put into the save file: lerpTime, invertMove, collected, animationTime, and position.



lerpTime: from what point between 0 and 1 the item is in its movement.

invertMove: whether the item is moving up or down.

Collected: has the player obtained this item already?

animationTime: what point in the animation the item currently is.

Position: Where the item is located.

Reflective Statement for Part 2:

The most difficult part was deciding what variables needed to be passed (whether to pass the animator itself or some portion of it, etc). Once that had been decided on, the only other hiccup involved the items all loading into the same spot every time. This ended up being fixed by moving the “start position” and “end position” of the up/down movement to start rather than awake.

