Michelle Snowball

Learning Journal

25/2/2020

Today I made a quest system in Unity. I needed it to be as general as possible so that it could easily be adapted into any game. I followed the Brackey’s tutorial on making a quest system.

I learned that you can use enumerations to make quest types. These can correspond with other functions in the script that can increment the quest data when a specific type of action has been completed.

March 6th, 2020

Today I decided to do a simpler component to give myself a bit of a break and created a particle effect that can be instantiated in bursts to give the effect of dust appearing when a character walks. I watched a tutorial on making particle effects appear in bursts and learned that instead of using a function on one particular particle effect to make the particle appear, I could instead instantiate a new particle effect every time I want it to run.

This would instantiate a huge amount of gameObjects that wouldn’t be needed after the first burst though. To handle that, I attached a script to the prefab that makes it delete itself after a second, giving it enough time to play the effect before it is deleted.

March 10th, 2020

I used something that I had already made for my AGP for this component. It is an AI path following script that uses a predetermined list of transforms to follow a path. I did not think of it as path following at the time, but this game will involve a lot of AI. The time-following component of it will be vital to my AGP as well. By making this, I learned that modularity is essential in scripting.

Included in the game manager script is a timeState enumeration. This allows for each agent to access the time state and move towards the appropriate goal. I had a paused time state, but it didn’t change enough in the script to warrant having a whole state for it. This also allowed me to resume playing in the same direction when hitting play again.

March 15th, 2020

Today I followed a Brackey’s tutorial to make a homing missile in Unity. I learned quite a bit about Vectors and 3D maths in the process.

To begin with, I had to get the player’s position each frame to know where to move towards. This was accomplished with using GameObject.FindWithTag and using the player tag. Next, we needed to make the missile move a certain amount each frame. This was accomplished by adding to the rigidbody’s velocity in the upwards direction each frame.

Making the missile rotate towards the player was the trickiest part of it all. In the tutorial, it is handled by first getting the vector representing the direction between the player and the missile. After that, the direction is normalized so that it only has a scale of 1 but maintains its direction. After that, we take the cross product between the desired travel direction and the current travel direction (which, in this case, will always be transform.up). Because the cross product will always be in the Z direction, we only need the z-component of the cross product to get the angle direction. By setting the angular velocity to this direction multiplied by the speed, multiplied by -1 (the positive version makes the missile avoid the player), we can make the missile follow the player’s position.

I decided to add a mechanic to the script that accounted for passing by the player to make the missile dodge-able. That just involved finding when the turn angle was the strongest and only setting the angular velocity when the turn needed was under a specific amount. This is why I added the follow strength variable, which is adjustable in the inspector.

April 3rd, 2020

Today I did work on my small game project. I decided to make a small system where you can run around and dodge missiles in 3D space. Because the components I made were in 3D for the most part, decided to take the homingMissile system and make it 3D instead.

Making a homing missile work in 3D proved to be more difficult than expected, mostly because the missile needed to rotate on more axes than just the z. I tried doing it manually, but couldn’t figure out how to do it until I came across the Vector3.Cross function. In 2D I was just doing it in Vector2, so theoretically I could just change everything that was using Vector2 to Vector3 to translate it into 3D space. It worked! I added elements of the dustGenerator and a spawner into the scene to spawn missiles. I also added a function that makes missiles delete themselves when they collide with anything.