Homework 2:
Controller input and Saved Games

In this assignment, you'll add features to the skeleton of an Asteroids clone. In Asteroids, you control a ship that flies through space and tries not to hit things. You can avoid hitting things by steering around them, which is hard because of the physics is frictionless, so stopping is hard. Or you can also shoot things, which is dangerous because if you miss, the bullets wrap around the screen and can come back and hit you.

We've given you the code for interacting with most of the parts of Unity that we haven't talked about yet in class. In particular, the linkage to the physics system (moving the ship, and detecting collisions). You'll fill in the code for checking user input on the controller and calling the appropriate functions in response. You'll also write the code to save and load the game state, so you can save your place. Although you won't go all the way down to reading and writing files, this will at least give you a sense of some of what's happening during serialization and deserialization.

Sensing controller input

One of the issues with handling controller input is that the set of available buttons and joysticks varies from platform to platform (e.g. Mac or Windows). Moreover, even when you use the same controller on two different platforms, the different operating systems might number the buttons differently.

For that reason, the Unity input system just assigns abstract names, called **axes**, to the different inputs available on the controller. The developer can then map these names to physical buttons using the **input manager** in the Unity editor by choosing Edit>Project Settings>Input. You can then use the Unity API call **Input.GetAxis**(*axisName*) to tell you the state of the axis. You pass it the name of the axis, and it returns a number between -1 and 1. For analog controls, such as joysticks, that will vary continuously; for digital buttons that are either on or off, 1 means pressed, and -1 means not pressed.

Unfortunately, the input manager doesn't allow you to specify different button numbers for an axis on different platforms. So for our code, we've included a Platform class that lets you ask which platform you're on (Windows or Mac) and a set of methods for returning the correct axis name for the Save and Fire buttons for your specific platform.

What platform am I running on?

The first thing you have to do is to figure out whether you're running on Mac or Windows and fill in the Platform.GetPlatform() method accordingly. It should return either PlatformType.Windows or PlatformType.Mac. How do you do that? Time for some googling!

Handling controller input

Now fill in the HandleInput() method of the Player class to check the controller axes and take the appropriate actions. Here are the axes you need to track:

- "Horizontal"
  This should control turning. The Player class has a method, Turn(*speed*) that takes a speed between -1 and 1.
- "Vertical"
  This should control forward thrust. Again, the Player class has a method, Thrust(*intensity*) that takes an thrust intensity between -1 and 1.
- _fireaxis
  Note this is not "_fireaxis"; it's the string variable that holds the actual string name of the the fire button axis for whatever platform you're running on. This is filled in by the Start() method of the Player object. Whichever platform you're on, this should correspond to the right analog trigger of your controller. When the trigger is pulled, the ship should fire. You can fire by calling Fire(). Note, however, that nothing much will happen yet when you fire, because you haven't implemented firing.

Firing

Bullets are coordinated by the BulletManager class. The Fire() method ultimately calls the **BulletManager.ForceSpawn**() method. It passes ForceSpawn, the position, rotation, velocity, and self-destruct time of the bullet (so the bullets don't fly around forever). Your job is actually make the bullet by calling Object.Instantiate. The prefab you need to instantiate is in the field **_bullet**. You can pass instantiate the position and rotation of the bullet and it will set them up for you. To keep the object hierarchy from getting too cluttered, make the bullet be a child of the Bullets game object, which is helpfully stored for you in the **_holder** field.

Unfortunately, Object.Instantiate doesn't give you a way of telling the newly created bullet what its velocity or self-destruct time should be. So to do that, you need to reach inside the new bullet, and call the **Initialize()** method of its **Bullet** component.

Run the game and make sure that pulling the trigger fires bullets properly.

Notes:

- The rotation argument is specified by an exotic mathematical object called a **Quaternion**. Don't worry about this for the moment. We will talk about Quaternions soon enough.
- **Make sure that you understand** why the Initialize() method exists. For example, why wouldn't one just let its Start() method set the velocity and self-destruct time?

Spawning asteroids

You still don't have anything to shoot at. Fill in the **ForceSpawn**() method of the **AsteroidManager** class. It's pretty much the same idea as firing bullets, only the arguments to the methods are slightly different.

Collisions!

Okay, now there are asteroids and bullets, but not much happens when they hit one another. Let's fix that. Inside the Asteroid class you'll find an empty **OnCollisionEnter2D**() method. Its argument is a **Collision2D** object, which contains information about what the asteroid hit and how it hit. All you care about is its **gameObject** field, which tells you what object the asteroid hit. Add code to

OnCollisionEnter2D() so that it calls HitPlayer() when the asteroid collided with the player, and HitBullet() when it collided with a bullet.

This requires looking inside the GameObject to determine whether it's a bullet, the player, or something else. How do you do that?

Saved games

Although it's not a usual thing for an Asteroids game, we're going to implement a save/load feature. We'll use the XmlSerializer class that's built into C#'s standard libraries. The XmlSerializer takes an arbitrary object and writes all of its public fields to a text file in XML format, recursively serializing any objects contained within the original object. This prevents you from having to open the text file and write the data yourself, but you still get a somewhat human-readable output that you can check.

Saving

When saving the game, we need to save:

- The player object's state, obtained from the Player object
- The states of all the asteroids, obtained from the AsteroidsManager
- The states of all the bullets, obtained form the BulletManager
- The player's score, obtained from the ScoreManager

The good news is that XmlSerializer can just take care of writing data out. The bad news is that it may try to write out more data than we really want it to. For example, it might try to write out irrelevant internal Unity information. So what we're going to make a new object for each of the saved objects (Player, BulletManager, etc.) that contains only the data we want to save. Then we'll save that object.

The logic for that is implemented in the OnSave method of each of the above classes. When you hit the save button, the system will call OnSave on each of the saved objects (Player, AsteroidManager, BulletManager, ScoreManager). The OnSave methods will make new objects (PlayerData, AsteroidsData, BulletsData, ScoreData), copy just the data to be saved into them, and then return those object. The serializer will then save those objects.

So start by finding the OnSave methods for each of those classes, and write code to make the saved data object (PlayerData, BulletsData, etc.), copy the data into it, and return it.

Here are some helpful hints:

- For many of the objects, you'll want to save position, velocity, rotation, and/or angular velocity (rate of rotation). These are all available inside the gameobject's **RigidBody2D** component.
- You can get all the Bullet objects by calling **FindObjectsOfType**(typeof(Bullet)); to get all the asteroids, just change Bullet to Asteroid.

Loading a saved game

When you load a saved game, you have to make the current game look like the saved version. To do that, you'll need to fill in the **OnLoad**() methods of the same classes you wrote OnSave() for:

- For the Player and Score manager, the relevant gameobjects already exist, so all that OnLoad needs to do is to reset the same fields that OnSave had saved.

- For the BulletManager and AsteroidManager, it will need to actually create the bullets and asteroids, by calling ForceSpawn with the appropriate arguments.
- However, you'll also need to destroy any existing bullets and asteroids!
- You can set the rotation of the RigidBody2D of an object by calling MoveRotation.
- You can make a magic Quaternion object for a given rotation by calling Quaternion.Euler(0, 0, *rotation*).
- **Important:** the angularVelocity field of the RigidBody2D object has a bug. When you ask a RigidBody2D what its angular velocity is, it tells you in degrees per second. But when you set it, it assumes you're giving the answer in radians per second. So when you're setting the angular velocity to the saved value from the save file, multiply it by **Mathf.Deg2Rad**. Welcome to software development in the real world ☹