Programming Assignment 4
Detailed Instructions

Overview

In this programming assignment (the first increment of our Asteroids game development), you're building the Unity project and adding the player ship to the game.

Step 1: Create the Unity project

For this step, you're creating the Unity project and getting it set up.

1. Create a new 2D Unity project called Asteroids.
2. Create a new scenes folder and save the current scene as scene0 in that folder.
3. Select the Main Camera object and click the Background color picker in the Camera component in the Inspector. Change the color to whatever you want your background color to be in the game.
4. Select Edit > Project Settings > Physics 2D from the top menu bar and set the Y component of Gravity to 0.
5. Save and exit Unity.

When you run your game, the Game view should just be the background color you selected above.

Step 2: Add the ship

For this step, you're adding the player ship to the scene. It won't be moving yet, but you'll see it in the Game view.

1. Create a new sprites folder and add a sprite for the player ship to that folder. I haven't provided a sprite for you, so you'll have to draw or find one yourself. You can certainly use "programmer art" (that's what I'm doing), but since this is an academic assignment you could also use an image of Serenity, the Millenium Falcon, the Enterprise, or some other cool ship if you'd like. Note: To make the screen wrapping easier, we'll be using a Circle Collider 2D for the ship. That means a more "circular" ship will work better. **CAUTION: Make you ship sprite the correct size for your game; don’t scale your game object in Unity.**
2. If you haven't done so already, make sure your sprite is facing to the right. This is important, because if you don't do this your ship will look like it's going in the wrong direction when you apply thrust. This is because an angle of 0 degrees is to the right.
3. Drag the ship sprite from the sprites folder in the Project window onto the Hierarchy window.
4. Rename the resulting game object Ship.

When you run your game, you should see the ship in the center of the window.

Step 3: Drive the ship

For this step, you're applying thrust to drive the ship forward.

1. Select Edit > Project Settings > Input from the top menu bar. Expand Axes (if necessary) and change Size to 19. Rename the bottom axis Thrust, set it to use space for the Positive Button, and delete the entry for the Alt Positive Button.
2. Attach a Rigidbody 2D component to the Ship game object.
3. Create a new scripts folder and add a new C# script named Ship to that folder.
4. Double click the Ship script to open it in your IDE.
5. Add a documentation comment above the line declaring the class.
6. We don't want to have to retrieve the Rigidbody 2D component every time we apply thrust, so declare a field to hold that component and add code to the `Start` method to set that field to the Rigidbody 2D component attached to the ship. Note: Declaring a field is just like declaring a variable. Declare the field below the line that starts `public class Ship`.
7. Add a `Vector2` field called `thrustDirection` and set the field to a new `Vector2` with an `x` component of 1 and a `y` component of 0. We're doing this so we don't have to create a new `Vector2` every time we apply thrust to the ship.
8. Declare a constant in the class called `ThrustForce` and give the constant a reasonable value (you'll probably have to experiment with this to get a reasonable value once you can drive the ship!).
9. For physics-based actions (like applying thrust), we should use the `MonoBehaviour FixedUpdate` method. Add a `FixedUpdate` method with the appropriate return type and parameter list to your `Ship` script (the easiest way to do this is by just copying the example from the Unity Scripting Reference documentation for the method and deleting the code between the `{` and the `}`). Add a documentation comment above the new method.
10. Add code to the `FixedUpdate` method to detect input on the Thrust axis and apply the `ThrustForce` in the `thrustDirection` to the rigidbody (remember, you saved that in a field) if there's input on that axis. Use the `Input GetAxis` method to check for input on the Thrust axis. Your first argument to your call to the `AddForce` method should be `ThrustForce * thrustDirection`. You should NOT be applying an impulse force here, so make sure you use the appropriate `ForceMode2D` value for the second argument.
11. Add the `Ship` script as a component to the Ship game object.

When you run your game, you should be able to drive the ship forward using the space bar. Of course, your ship just drives off the right edge of the screen, never to return!

Step 4: Make the ship wrap

For this step, you're making the moving ship wrap when it leaves the screen instead of disappearing from the game forever.

1. To implement screen wrapping, we'll need to know where the top, bottom, left, and right edges of the screen are (in world coordinates). Luckily, I've written scripts to do this for you! Copy the ScreenUtils.cs and GameInitializer.cs files you extracted from the zip file into the scripts folder for your project.
2. Add the `GameInitializer` script as a component of the Main Camera in your scene.
3. We'll need to use information about the collider for the ship to do the wrapping, but the ship doesn't have a collider yet! Add a Circle Collider 2D component to the Ship game object. Edit the collider so the collider is completely inside the ship sprite.
4. Double click the Ship script to open it in your IDE.
5. It will be more efficient wrapping the ship if we save the radius of the collider attached to the ship instead of retrieving that information every time we need to wrap. Declare a field to store that value and add code to the `Start` method to retrieve the `CircleCollider2D` component and save its radius into your field. You'll probably have to read the `CircleCollider2D` documentation to do this properly.
6. Go to the Unity Scripting Reference and look up the documentation for the `MonoBehaviour OnBecameInvisible` method. Add an `OnBecameInvisible` method with the appropriate return type and parameter list to your Ship script. Add a documentation comment above the new method.
7. Add code to the body of the new method to make the ship wrap around to the other side of the screen when the method is called. Because we know the position of the ship and the radius of the collider, we can figure out where it has gone off the screen and move it to the other side. You should use a sequence of if statements for this, remembering that the ship might have exited the screen at a corner. You should find the `ScreenUtils` properties useful as you check each of the 4 screen sides. As an example of how to move a ship that just left the right side of the screen, negating the `x` value of the ship's position will move it just to the left of the left side of the screen. All the other cases are similar. Remember, you can't change `transform.position.x` directly, so you'll have to save the `transform.position` property in a local `Vector2` variable, change that local variable as necessary, then copy the local variable into `transform.position` at the end of the method.
8. **Caution: Even if the screen wrapping is working properly, so a player could play the built game and screen wrapping would work perfectly, it may not seem to be working in the Unity Editor. The best thing to do is to build the game (like you have to do to submit it) and play the built game; be sure to click the browser page to make the game "listen for" user input when you play it. If, however, you want to just stay in the editor, double click the Main Camera in the Hierarchy window, then use Ctrl + Middle Mouse Wheel to zoom in on the Scene view until the box that shows the edges of the camera view just disappears from view.**

When you run your game, you should still be able to drive your ship to the right. When the ship leaves the screen on the right it should wrap back into the screen from the left. It would be nice to test the other 3 sides of the screen, but we need rotation for that. Speaking of which ...

Step 5: Rotate the ship

For this step, you're rotating the ship.

1. Select Edit > Project Settings > Input from the top menu bar. Expand Axes (if necessary) and change Size to 20. Rename the bottom axis Rotate, set it to use left for the Positive Button and right for the Negative Button, and delete the entry for the Alt Positive Button if there is one.
2. Double click the Ship script to open it in your IDE.
3. Declare a constant in the class called `RotateDegreesPerSecond` and give the constant a reasonable value (you may have to adjust this one you test rotation).
4. Add code to the `Update` method to detect input on the Rotate axis and rotate the `transform` of the ship appropriately. Use the `Input GetAxis` method to check for input on the Rotate axis. The best way to do the rotation is with the following code (assuming you saved the Rotate axis value in a variable called `rotationInput`):

```
// calculate rotation amount and apply rotation
float rotationAmount = rotateDegreesPerSecond * Time.deltaTime;
if (rotationInput < 0) {
    rotationAmount *= -1;
}
transform.Rotate(Vector3.forward, rotationAmount);
```

When you run your game, you should be able to rotate the ship using the left and right arrow keys. Of course, it still only moves to the right!

Step 6: Thrusting in correct direction

For this step, you're making it so the thrust is applied to the ship to move it in the direction its pointing.

1. Now that we can rotate the ship, using the `(1, 0) Vector2` for the `thrustDirection` doesn't work anymore (since the ship no longer only moves to the right). The good news is that the only time we need to change the `thrustDirection` is when we rotate the ship. That means you can add the required code at the end of the body of the if statement you used in the previous step to detect input on the Rotate axis.
2. Double click the Ship script to open it in your IDE.
3. The good news is that the `transform` object exposes an `eulerAngles` field that gives us a `Vector3` containing the current rotation of the ship on each of the axes in degrees. We DEFINITELY don't want to access the `rotation` field of the `transform`, which is something called a Quaternion. Once you have the rotation around the z axis (the only axis rotation we'll have in our 2D game), you can convert the angle to radians using

`Mathf.Deg2Rad` and calculate the appropriate values of the `x` and `y` components of the `thrustDirection` vector using the `Mathf.Cos` and `Sin` methods.

When you run your game, you should be able to rotate and apply thrust to your ship and your ship should behave appropriately as you drive it around.