CS 376 Exercise 3: Alien Dodgeball

Important notes

- Remember to only use the official class version of Unity: 6000.2.6f1.
- When running this assignment, be sure to set the Aspect Ratio pulldown in the editor to "Full HD". The dropdown is at the top of the Game/Scene window in the editor, right to the left of the Scale slider.
- This assignment will be peer reviewed, but you will only turn in your .cs files. So while you are free to change the rest of the game if you want, don't make any changes that your .cs files will depend on, because your peer reviewers will be running the .cs files in an unmodified version of the rest of the game. So don't change the names or tags of game objects, or rename classes or public fields or things like that. If you follow the instructions below, you'll be safe.

Overview

In this assignment, you'll use the skills you learned in the last assignment to implement the gameplay logic for a simple arcade-style shooting game. The game is a cross between Asteroids, dodgeball, and sumo. You and the enemies pursue and shoot each other. However, instead of the missiles blowing you up, they knock you around. You and your enemies score points by knocking each other off the screen. When an enemy falls off the screen, it respawns in a random position. Moreover, a new enemy spawns every 20 seconds, so you eventually get overwhelmed.

The game is asymmetric in that the enemies' missiles have much more momentum and so push you around more. Your missiles have less momentum, but you can shoot a comically large number of them.

Reading the code

As always, you should start by reading the source code in the assignment (ignore the TextMesh Pro folder, which is provided by Unity itself).

Making your player move

Start by implementing the `Start()` and `Manoeuver()` methods of the `Player` component. This will involve applying forces to the `RigidBody2D` component of the player GameObject using the `AddForce()` method, and doing so every time physics updates. Since you don't want to be calling `GetComponent<RigidBody2D>()` 50 times a second to get the rigid body component, make a field to store the rigid body in, and then add a Start() method to initialize it.

`Manoeuver()` is called by the FixedUpdate() method, which Unity calls every time the physics system updates (50 times a second). So each time physics updates, `Manoeuver` will get called to compute a new force for that update. It should read the "Horizontal" and "Vertical" virtual axes using the `Input.GetAxis()` method. From that, you should make a `new Vector2()` that points in the direction the player's joystick is pointing. Then scale it by `EnginePower`, and apply that as a force to the `RigidBody2D` component.

We've mapped the Horizontal and Vertical virtual axes to the "X" and "Y" controller axes, but you may need to adjust the mappings in the Input Manager (under the Project Settings dialog in Unity) for your controller. Simply go into the Horizontal and/or Vertical axes and change joystick axis to whichever one you want.

Make sure you can move your character around the screen before moving to the next part.

Making your character aim

Now change the `Manoeuver()` routine to also set the rigid body's `angularVelocity` field (a scalar in degrees per second) to the value of the "Rotate" input axis, multiplied by the `RotateSpeed` field. Again, you may need to adjust what joystick axis is mapped to the Rotate axis.

Check that you can pilot your ship around the screen. You can tune the responsiveness of your ship by changing `RotateSpeed` and `EnginePower` in the Unity editor if you like.

Spawning enemies

The game has a game object called `EnemySpawner`. It contains a `Spawner` component, which periodically creates ("spawns") an enemy in a random location on the screen.

Go to `Spawner.cs` and fill in the `Update()` method to instantiate an enemy at a random point every `SpawnInterval` seconds. We've provided you with a method, `SpawnUtilities.RandomFreePoint()`, that finds a random point on the screen that doesn't have any other objects within the specified radius. The class already has a public field called Prefab, that will have already been filled in with the prefab for the enemy. So you just have to call `Instantiate()` on it.

How do you get it to spawn every 10 seconds? Remember that `Time.time` tells you how many seconds the game has been running for. So keep a field in the object that tracks when the next spawn should happen (it can start at zero). Whenever `Time.time` is greater than that field, spawn an enemy and then push the time in the field forward by `SpawnInterval` seconds.

Check that the enemies are spawning every 20 seconds before moving on. We've filled in some simple code to move the enemy around, so it should roughly follow you.

Shooting

Now fill in the FireOrb() and MaybeFire() methods of Player.cs.

FireOrb() should create a new PlayerOrb prefab in front of the player's ship. We've stored the prefab in the OrbPrefab field for you. You need to instantiate it so that it appears in front of the ship. Since the sprite for the player points to the right in the original image, you can use `transform.right`, a unit vector pointing in the direction of the local X axis, to give you a vector in the direction the player is pointing. Spawn the orb one unit in front of the player's position.

Then, grab the RigidBody2D of the orb you just instantiated, and set its `.linearVelocity` field to `OrbVelocity` times `transform.right` to make it shoot out in front of the player. You can then tune the speed of the orbs by changing the `OrbVelocity` field in the Unity editor if you like.

Now modify MaybeFire() so **every other time it is called**, it calls FireOrb() if the button corresponding to the "Fire" input axis is pressed. Again, you may again need to remap it to your liking.

MaybeFire() is also called by FixedUpdate, so it's also called 50 times a second. So holding the fire button down should shoot 25 orbs a second (because it only fires every other call). That's a lot, but the player's orbs are so weak, it's not too much. Test it out by working out your aggressions on those evil aliens.

Getting rid of unwanted orbs

Once an orb goes off screen, it's never coming back. So it's a waste of resources to keep doing physics updates on it.

- Go to Orb.cs and fill in `OnBecameInvisible()`, which is called when the orb goes off screen, so that it destroys the orb's game object (it's not enough to just destroy the Orb component). Unity helpfully provides a `Destroy()` method.
- Now fill in `OnCollisionEnter2D()` so that the orb also destroys itself if it hits something other than another orb (if it hits another orb, neither should be destroyed). The collider of the gameobject that hit the orb will be in the field `collision.collider`. The collider is a component of the gameobject. How could you use the collider to figure out whether the overall gameobject is another orb, or something else?

Getting shot at

Now go to Enemy.cs and fill in `Update()` to call `Fire()` every `CoolDownTime` seconds, using the same technique you used in spawning. Then fill in Fire() to shoot an OrbPrefab in the direction `HeadingToPlayer`, at speed `OrbVelocity`. But also set the rigidbody's `.mass` field to `OrbMass`. Again, this lets you tune how hard the orbs hit in the Unity Editor by changing their mass and velocity. (Note: you don't need to tune anything for this assignment. You should just feel free to do so if you like.)

Now test out the game to make sure the enemies are making life difficult for you.

Note: if you don't do this right, later enemies will fire multiple shots when they're first spawned. How do you prevent that?

Keeping Score

We've already included a component (Respawner) that moves the player and enemies back on screen if they move off screen. But we also want those events to score points.

Go to ScoreKeeper.cs and fill in `ScorePointsInternal()` so that it increments the score by the specified number of points, and then updates the `.text` field of `scoreDisplay`.

Now add `OnBecameInvisible()` methods to Player.cs and Enemy.cs that call `ScoreKeeper.ScorePoints()`. The score should go up by 1 when an enemy goes off screen, down by one when the player does.

Note: if you can't find the score on the screen, you probably don't have the aspect ratio set to Full HD.

Making sure your code doesn't have issues

Now you want to make sure you code doesn't have any errors in it. First, let's make sure it compiles without any warnings. In Rider, choose Build>Rebuild Solution from the menu and make sure the error list at the bottom of the window doesn't have any errors or compiler warnings.

Now go to the Unity window and find the Unity "Console". You'll find it in the Console tab in the bottom pane of the window. This is where exceptions get printed if your code throws and exception. You can also display messages here manually using Unity's Debug.Log() method. However, the final code you turn in for your project should not call Debug.Log() or otherwise print any messages in the console window.

Run your project. Let it run for a minute or so, pressing buttons and moving the joysticks around, just to make sure no errors happen and you don't have any Debug.Log() calls left.

Turning it in

You now have a working game. The only thing that's missing is sound, but you've done enough for this week. We'll deal with sound later.

To turn your project in,

- Exit out of Unity and Rider, just to make sure that everything got saved.
- Make a folder with copies of the .cs files from the Assets folder (again, ignore the TextMeshPro folder). **DO NOT SUBMIT THE ENTIRE PROJECT**
- Make a zip file of the folder with your .cs files, and upload it to Canvas.

You're done!