Exercise 7: An arcade flight sim

Important

Remember to

- Use the version of Unity that we have standardized for this quarter
- Set the aspect ratio to full HD
- Modify only the two files we tell you to (Target.cs and PlayerControl.cs). If you modify other files, then your peer reviewers may not be able to run your game or even to compile your code.

Overview

In this assignment, we'll build a simple arcade-style flight simulator, similar to StarFox or PilotWings. Arcade style means that while it has some simulation of flight dynamics (lift, drag), it doesn't try to be especially realistic since flying real aircraft is hard. In particular, we'll finesse the issues of steering the plane and let the player directly control the plane's orientation. The flight physics will then control only translational forces that are applied to it (thrust, force, and drag).

Note: we'll be using a controller for this assignment. You may want to remap the controls for your particular controller. The shipped version will work for most controllers, although not necessarily playstation or Switch controllers. It assumes:

- **Horizontal** axis (roll control of the plane)
  X axis of left thumbstick. Right to roll right, left to turn left.
- **Vertical** axis (pitch control of the plane)
  Y axis of the left thumbstick. Push to dive, pull to climb.
- **Thrust** axis
  Y axis of the right thumbstick. Pull back to accelerate.

Getting started

Once again, begin by familiarizing yourself with the code for the following classes:

- **PlayerControl**
  This will implement the player's piloting of the plane.
- **LandingPlatform**
  This is just used to mark the gameobject representing the platform where the player lands at the end of the game
- **Target**
  This implements the spinning transparent capsules that the player flies through to get points.
- **Updraft**
  This an object as representing a wind that raises the plane up when it flies through it.
- **ScoreManager**
  Pretty much the same as in other assignments.

You'll be filling out the code for the PlayerControl and Target.

Steering

For an arcade flight sim, you let the player have direct control over where the plane is pointing; you don't simulate torques due to ailerons or the rudder. Everything is controlled in terms of the rotation of the plane, which is stored in the rotation component of the transform. But as with 2D physics, in 3D physics, we need to manipulate rotation using the rotation field of the plane's `RigidBody` component (stored in `planeRB`) and its `MoveRotation()` method. Unlike 2D physics, however, 3D rotations are represented using quaternions. So you should begin by familiarizing yourself with Unity's `Quaternion` struct.[1]

For our application, it will be more convenient to represent the plane's orientation in terms of yaw, pitch, and roll. The control stick directly controls pitch and roll, and roll determines the rate of change of the yaw.

Start by making a FixedUpdate() routine for the PlayerControl class that updates the pitch and roll fields based the position of the thumbstick. The roll should be based on the "Horizontal" input axis, and should range from -RollRange to +RollRange (RollRange being a field that can be changed in the inspector. And the pitch should be controlled by the "Vertical" input axis, and range from -PitchRange to +PitchRange.

The yaw changes based on the roll. If we're rolled to the left, we turn left. If we're rolled to the right, turn right. In particular, you should implement the control law:

$$\frac{d}{dt}\text{yaw} = \text{roll} \times \text{RotationSpeed}$$

Where RotationSpeed is a constant that can be tuned in the unity inspector.

Having computed yaw, pitch, and roll, you should now use the MoveRotation() method of the RigidBody to change the aircraft to point in the specified direction. The only problem is that MoveRotation() wants a Quaternion as its argument. So you first need to make the quaternion that corresponds to the specified yaw, pitch, and roll, then call MoveRotation() on the result. Unity has a built-in method for making quaternions from yaw, pitch, and roll, called Quaternion.Euler. The slides tell you that Euler angles are different from yaw, pitch and roll, and they are. But YPR is sometimes incorrectly referred to as Euler angles (see the Wikipedia entry for Euler angles if you really care). Suffice it to say that if you call Quaternion.Euler(*pitch*, *yaw*, *roll*), you'll get the quaternion you want.

When you have this working, you'll notice that this is pretty disorienting since the plane rolls and pitches instantly as you move the stick. It's less stomach churning and more realistic if you smooth out the motions. So what you really want to do is compute the roll and pitch that the joystick is specifying and then set the roll and pitch to a weighted average of the current roll/pitch, and the target roll/pitch specified by the joystick. That means that if you move the joystick to some position, the plane will move a little bit toward the target position for each frame, until they eventually converge. You can do that using Mathf.Lerp(), since Lerp's whole point is to compute a weighted average of its two inputs:

[1] In C#, a struct is a record structure like a class, but which is passed by value (e.g. on the stack) rather than by reference (i.e. in the heap). Structs are commonly used for math, since they don't need to be garbage collected.

```
roll = Mathf.Lerp(roll, joystickRoll, weight);
pitch = Mathf.Lerp(pitch, joystickpitch, weight);
```

I used 0.01f as my Lerp weight, but you can choose whatever looks good to you.

Thrust

Of course right now, you can steer the plane around as it falls into the ocean, but that's about all you can do. Let's make it … you know… fly.

Let's start by giving it an engine. Apply a force in the plane's forward direction that's proportional to the value of the "Thrust" axis. Since the axis is a thumbstick, you can read negative numbers from that axis. So don't apply backward thrust when the thumbstick is in the negative direction, treat negative values as zero thrust. The thrust should go from 0 to MaximumThrust (another value that can be tuned in the Unity inspector).

Try it out. You should be able to do some rudimentary flying now.

Aerodynamics

Real planes interact with the air around them in two important ways:

- **Drag**
  Drag is quadratic in the speed at which the air hits the plane, and it slows the plane down. But the amount of drag also depends on the direction from which the air is hitting the plane. When it's coming from the front of the plane, the plane has a relatively small cross-section and so doesn't experience much drag. When it comes from below, it's hitting the wings and body cross-wise rather than end-on and so there's a lot more surface area for it to hit. Hence more drag.

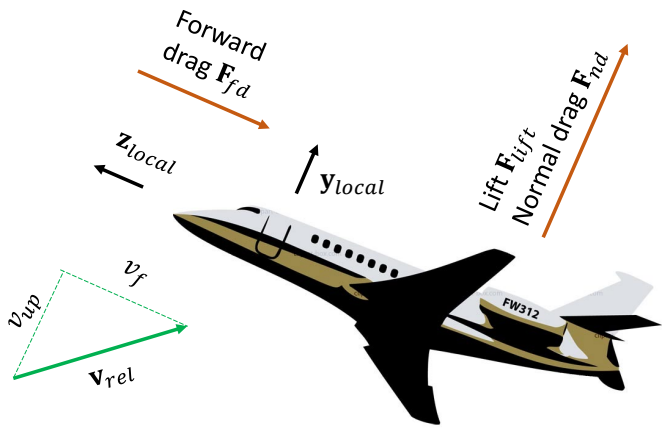
- **Lift**
  This produces an upward force due to the air sliding along the wing. It's also proportional to the square of the speed of the air, but specifically the speed with which it slides *along* the wing.

Let's start out by defining axes. Let's say the axes for the plane's local coordinate system are $\mathbf{x}_{local}$, $\mathbf{y}_{local}$, and $\mathbf{z}_{local}$, i.e. transform.right, transform.up, and transform.forward, respectively. These are the vectors for what the plane thinks of as right, up, and forward, but expressed in global coordinates.

Let $\mathbf{v}_{rel}$ be the relative velocity of the air with respect to the plane, again in global coordinates. For the moment, we'll assume there's no wind, so the this is just the speed of the plane, or rather, negative the velocity of the plane.[2] So we have a picture like this:[3]

[2] That is, if the plane is moving forward, the relative velocity of the air is backward from the plane's reference frame.

[3] Clip art from cliparts.org: http://cliparts.co/clipart/48392



Given this, we can approximate the lift as being a force pointing up (in the direction of $\mathbf{y}_{local}$), and proportional to the square of the velocity of the wind along the wing, i.e. in the plane's forward direction:

$$\mathbf{F}_{lift} = c_{lift} v_f^2 \mathbf{y}_{local}$$

Where $v_f = \mathbf{v}_{rel} \cdot \mathbf{z}_{local}$ is the forward component of the relative velocity (forward in the sense of the direction the plane is pointing). The coefficient c_{lift} is just a fudge factor. We've put a field for it in the PlayerControl object called LiftCoefficient.

Now for drag. To do this properly, we'd compute the actual cross-section of the plane with respect to the wind heading, but we don't need anything nearly that complicated. We'll just approximate the drag as the sum of two forces: one to forward motion and the other to vertical motion (in the plane's coordinate system), each with its own magic coefficient. The forward drag is just:

$$\mathbf{F}_{fd} = \text{sgn}(v_f) c_{fd} v_f^2 \mathbf{z}_{local}$$

This looks intimidating, but it's simple. It says: the drag is in the forward direction; it's proportional to the square of the forward component of the wind velocity; it's proportional to a magic constant c_{fd}, called ForwardDragCoefficient in the code); and it's in the same direction as the wind velocity (that's the use of the sign function at the beginning).

The vertical drag is the same, but in a different direction:

$$\mathbf{F}_{nd} = \text{sgn}(v_{up}) c_{nd} v_{up}^2 \mathbf{y}_{local}$$

Again, where $v_{up} = \mathbf{v}_{rel} \cdot \mathbf{y}_{local}$ is the component of the wind velocity in the up direction and c_{nd} is a magic fudge factor called NormalDragCoefficient in the code. Computing the two components of drag separately means we can have separate drag coefficients, with c_{fd} being very small and c_{nd} being large.

Scoring

Now modify the Target class to detect when the player flies through it and then call the ScoreManager to increase the score of the player by ScoreValue and destroy the target.

Landing and crashing

Now add to PlayerControl a collision handler that notices when the player collides with something and calls OnGameOver(). OnGameOver takes a Boolean argument specifying whether the player successfully landed or crashed. They successfully landed if they collided with the LandingPlatform object and their vertical speed (the z component of their velocity in global coordinates) was less than the MaxLandingSpeed field of the platform they landed on. Otherwise, they crashed.

Updrafts

Finally, you'll notice that there is a transparent cylinder in the level. That represents an updraft of air that will lift your plane if you're just gliding. When you fly into the updraft, the wind will suddenly be blowing upward. That will generate a drag force in the direction of the wind, which will push the plane upward (or in whatever direction the wind is pointing).

Modify your aerodynamics code so that it updates the relative velocity of the air based on whether you're in an updraft. You can check if you're in an updraft by using the Physics.OverlapSphere() operation. You can check specifically for only updrafts by passing along a layer mask for the updrafts. The updrafts are in the layer called "Updrafts" so you can get a layermask for it by calling LayerMask.GetMask(). We could have had you do this by treating the updraft as a trigger collider, but we wanted you to learn to use OverlapSphere.

If you are in an updraft, then the air velocity is the Updraft object's WindVelocity field and so the air's relative velocity (to the player) is this minus the player's velocity.

Turning it in

As always, start by quitting unity, restarting it, and just making sure that everything still runs. Make sure your code compiles without warnings and that you've removed any calls to Debug.Log and your code doesn't throw any exceptions while running.

Now turn in your code. **Since this assignment has some large textures in it, you should only turn in your Target.cs and PlayerControl.cs files (as a single zip file)**. That will keep Canvas from being overloaded.