Exercise 3: endless runner

In this assignment, you'll be implementing an endless runner: a platformer that generates an endless series of random platforms for you to jump across. It also slowly accelerates, so it becomes more challenging as you play.

Out of the box, the game is configured for the space bar to jump and the left shift key to do a small jump. There's no run key because the character always runs, except when they're jumping or falling.

The game defines the following custom components:

- **Runner**
  Implements motion and control of the player character.
- **PlatformManager**
  This randomly places platforms for the player to jump between.
- **Score**
  Displays the score, as usual. You don't need to do anything with this.
- **CameraControl**
  Moves the camera to follow the player, with a little bit of low-pass filtering to smooth out the motions.

We've set up the basic level file. For this assignment, you'll be implementing most of the methods for the components, save for the Score component.

The Runner component

This implements:

- Position and velocity update for the player: running, jumping, and gravity.
- Player control of jumping
- Notification of other components when the player lands on a platform or falls into the void

Start by reading the code for the Runner.cs file and familiarize yourself with the fields of the object and what methods there are. **We will refuse to answer Piazza questions** that you could have answered by first reading the source file.

You should fill in the following methods:

- **Update**
  This should:
  - Set the vertical velocity to **JumpSpeed** if the player is on a platform and they press the **"Jump"** button (i.e. if **Input.GetButtonDown**("Jump") is true), and to JumpSpeed/2 if they press the **"Short Jump"** button (but are on a platform. Otherwise, the vertical velocity should accelerate downward at a rate of **Gravity** units per second
  - Accelerate rightward at a rate of **RunningAccelerationRate** units per second if the player is touching a platform.

    - Update the player's **position** based on their **velocity**.
    - If the player's Y coordinate is less than **BottomOfTheWorld**, it should:
        - Call **FellIntoTheVoid** to tell the rest of the game that the player has lost. Note: it should only do this once; it shouldn't keep doing this on successive frames.
        - If the player falls below 2*BottomOfTheWorld, it should deactivate the player object completely. If can do this by calling the **SetActive** method to deactivate the player GameObject, e.g. run gameObject.SetActive(false). This will make the player disappear from the screen and stop updating.
- **OnTriggerEnter2D**
  This will get called when the player lands on a platform, so it should:
    - Update **isTouchingPlatform**
    - Call **LandedOnPlatform** to tell the rest of the game
    - Call **AlignToPlatform** with the position of the platform. This will fix up the player's position to be exactly on top of the platform.
    - Set the player's Y velocity to 0.
- **OnTriggerExit2D**
  This should update isTouchingPlatform.

The PlatformManager component

The platform manager makes sure there are always new platforms for the player to jump on as she moves. Each platform has a random width, and is placed at a random offset from the previous platform.

We don't know how far the player will get, so we don't know how many platforms to create. So we place platforms on demand, checking each frame to see if the player is getting close to the end of the platforms that have been placed, and if so, placing a new one.

Moreover, we don't want to have hundreds of platforms created, which would take up memory and slow down the collision detection system. We could destroy the platforms as the player passes them and they move off screen. However, creating and destroying GameObjects is expensive. It wouldn't really be a problem in a game like this, but industry practice would be to use a **pool** of platforms: instead of destroying a platform when it moves off screen, we deactivate it[1], and return it to the pool.[2] When we need a new platform, we just take one from the pool, reactivate it and reposition it. So while it looks to the player like there's a limitless series of platforms, it's actually the same half dozen or dozen platforms being endlessly recycled.

Again, you should **start by reading the code** to familiarize yourself with the fields and methods. Where does it store the pool of free platforms? How does it track what platforms are in use?

Now start filling in the methods:

---

[1] That is, we call its SetActive(false) method. This causes Unity to stop updating and drawing it and causes collision detection to ignore it. An inactive object has almost no impact on game performance.

[2] So a pool is like a special-purpose heap in the EECS-213 sense of memory management (not in the EECS-214 sense of a priority queue). In fact, console games often have extremely elaborate memory management, with things like special heaps and pools that get erased every frame, in order to try to minimize the time spent on object creation and memory management. See the Gregory text, chapter XXX for more discussion of pooling

- **Update**
  This does nothing if the player is dead. If they aren't dead, it:
  - Calls **RecycleOldestPlatform** if the øldest platform is more than **RecycleDistance** behind the player. Note that the platform manager is inside the player gameobject, so its position is the player's position.
  - Calls **SpawnNewPlatform** if the next spawn point is less than **SpawnAtDistance** ahead of the player.
- **RecycleOldestPlatform**
  This method should take no arguments and should:
  - Move the oldest platform from the **platformsInUse** queue to the **unusedPlatforms** queue
  - Deactivate it (the oldest platform) by calling **SetActive(false)** on it. This will prevent it from displaying, participating in collisions, or otherwise doing anything.
- **MovePlatformToSpawnPoint**
  This method should take a platform as an argument, and:
  - Set its position to **nextSpawnPoint**
  - Set its width to a random number in the range [**MinPlatformWidth**, **MaxPlatformWidth**].
    - Use the method **Random.Range** to get a random number in a range.
    - An easy way to set the width of the object is to set the **localScale** property of its transform. This is a Vector3 whose components are the object's scale along the X, Y, and Z axes, respectively. We've helpfully initialized all the platforms to have colliders and sprites that are 1×1, so if you set their local scale to $(width, 1, 1)$, they will magically become $width \times 1$ platforms rather than 1×1 platforms.
  - We've just created a platform at nextSpawnPoint, so we need to move nextSpawnPoint forward to the location of the platform we'll make after this one. So adjust (add to) its X coordinate by a random value in the range [**MinXSpacing**, **MaxXSpacing**], and adjust its Y coordinate by a value in the range [**MinYSpace**, **MaxYSpacing**].

We've taken care of writing **SpawnNewPlatform**, but you should look at the code to understand how it works. We've also written some other methods that you should familiarize yourself with.

The CameraControl component

This component's only job is to move the camera toward the player. We could do that by just setting the position of the camera to the position of the player, but that has two issues:

- The player would always be in the exact center of the screen. We'd like to be able to shift the player to be **offset** from the center so that we can see more to the right of the player (where she's running to) and less to the left (where she's already been). So we don't move the camera to the player's position, but instead to the player's position plus the offset.

- If we snapped the camera's position directly to the player's position (plus offset), it would make the screen look jerky. So we'll **smooth** the motion of the camera by moving the camera a fixed

percentage of the way toward the player on each frame,[3] rather than moving it all the way. If the player suddenly moves a lot, the camera will move a lot, but not all the way. Then on the next frame, it will move a little less, and on the next frame a little less than that. That makes for a much more pleasing motion.

Again, **start by reading the code** to familiarize yourself with the fields of the component. Now implement the following method:

- **Update**
  Should set the camera's position to a weighted average of the camera's current position and the target position for the camera. Note that:
  - The CameraControl component is inside the camera object, so it's position is the camera's position. (i.e. you can get the camera's position just by saying **transform.position** because the camera and CameraController have the same transform).
  - The target position for the camera should be the Target's position (i.e. the player), plus the **offsetFromTarget**
  - You should use the Vector3.**Lerp** method to compute the weighted average ("lerp" is graphics-speak for "**L**inear Int**erp**olation," i.e. weighted averaging. Set the interpolant[4] to Smoothing*Time.deltaTime.

Turning it in

When you have a working game, turn in your Runner.cs, PlatformManager.cs, and CameraControl.cs files as a single zip file. You're done!

[3] This is also called "low pass filtering."
[4] The weighting of the average; see the Lerp documentation.