Homework 3

Making an Endless Runner

In this assignment we'll be making an endless runner. We'll focus in particular on 2D physics and making useful menus.

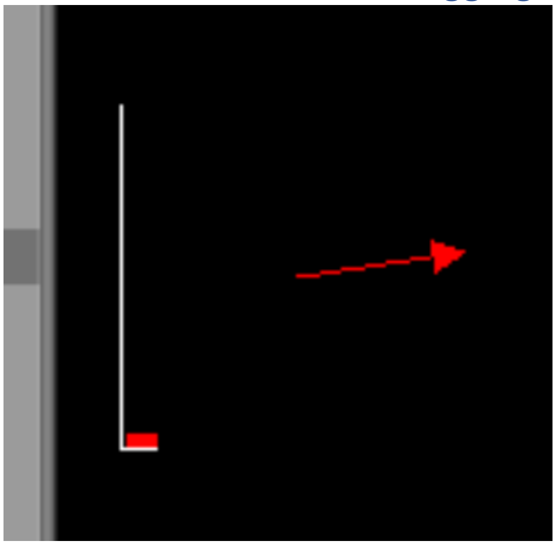
Part 1: A Main Menu

First things first, we should make something "pretty" for the player to see.

In MainMenu.cs, fill in the constructor for the MainMenu class.

- There is a "Main Menu" prefab in the **Resources/Menus** folder
- Add listeners to each of the buttons in the Main Menu after you've Instantiated it
  - Clicking the "Start" button should call *Game.Ctx.StartGame*
  - Clicking the "Quit" button should call *Game.Ctx.QuitGame*
- Look in the abstract class *Menu* (in UIManager.cs) for guidance on what the MainMenu class needs

Part 2: Some Useful Debugging Tools



Next, we'll add some handy debugging tools for the next part (implementing physics). One of the trickiest parts of working on games is that, since they tend to be real-time, breakpoints and other traditional debugging methods don't always work out (try going frame-by-frame through your Player's movements and you'll see how frustrating it can be to figure out what's going on).

We're going to add in two features: a direction indicator and a speed indicator. Just for fun, we're going to implement them using GL, Unity's OpenGL subset. OpenGL (Open Graphics Library) has been around for just about forever, and knowing how to use a few of its features will serve you well in your games career. Plus, it's like coding in C: the suffering makes you stronger.

- Fill in the body of **SimplePhysics.DrawArrow**(Vector3) to draw the Vector passed to it as an arrow. It should start at position ArrowStart. It should consist of a line, followed by a triangle for the arrowhead.
  - We've placed a Material you can use for drawing in the field _mat.
  - This will be called from the OnGUI() method, which doesn't necessarily set the transform matrix in a predictable manner. So use GL.LoadPixelMatrix() to set the transform to let you draw directly in screen coordinates, but remember to use GL.PushMatrix and GL.PopMatrix to save the original transform, so you don't confuse Unity.
  - Remember that you can get a vector perpendicular to (x, y) by using the vector (-y, x).

- Fill in the body of **SimplePhysics.DrawMagnitude**(*float*) to draw a quad to indicate how fast the player is moving
    - You don't *need* the white lines at the border of your quad, they just look cool
    - The exact starting/ending positions of your debugging display aren't important, so long as they're legible and distinguishable from one another

Part 3: (Mediocre) Physics

When you click "Start", the player pops up, but nothing happens right now. Because we're edgy, we've decided to roll our own physics engine, and nothing will happen until we fill in the blanks.

The three things that we care about for our version of physics are:

1. Position, and how it changes over time
2. Velocity, and how it changes over time
3. Collisions, i.e., whether or not we've hit anything

There is (obviously) a lot more to actual physics, but these are enough for our purposes.

Position and Velocity

The player's position can be estimated with the following equations:

$$x(t + \Delta t) \; = \; x(t) + \frac{dx}{dt}(t)\Delta t$$

$$y(t + \Delta t) = \; y(t) + \frac{dy}{dt}(t)\Delta t$$

Aside: If you took calculus or linear algebra, you may recognize these as the equations used in *Euler's Method*. This is called an *Euler integrator*.

The player's velocity can be estimated similarly:

$$\frac{dx}{dt}(t + \Delta t) = \frac{dx}{dt}(t) + \frac{d^2x}{dt}(t)\Delta t$$

$$\frac{dy}{dt}(t + \Delta t) = \frac{dy}{dt}(t) + \frac{d^2y}{dt}(t)\Delta t$$

The layman's version of these equations might look like:

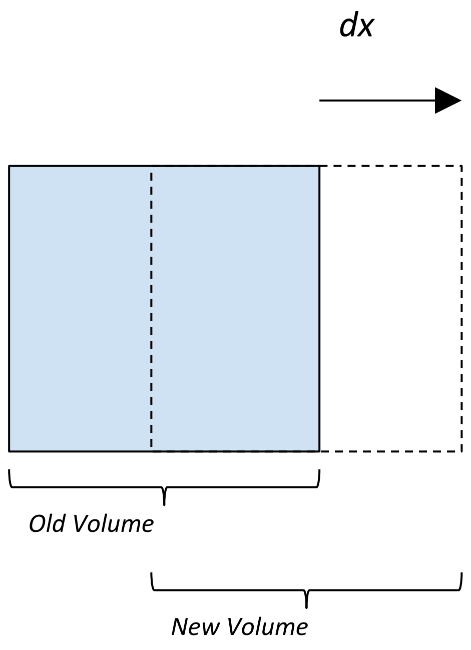
$$newvalue \; = \; oldvalue \; + valuechangerate \; * \; timeincrement$$

Fill in SimplePhysics.FixedUpdate to adjust _*velocity* and _rb.position appropriately. Now, when you start up the game, the player should fall down (accelerating faster and faster).

Handling Collisions

Okay, so our player moves around, but they fall straight through our platform! This isn't what we want. Let's try collision handling.

First, a warning: actual collision handling is ***very*** complicated. We'll cover it in later lectures and you'll see that the method we're about to use is *way* too simple to be useful. In fact, you'll probably see some of the faults in this method as soon as you hit "Start." But, this assignment is about getting a sense of what's involved in physics simulation, not about matching Unity's physics engine in its complexity.

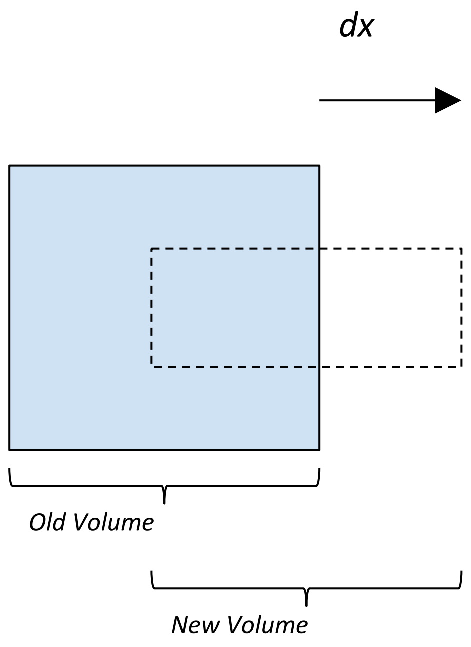


Effectively, if we know what *dx* is, we can predict whether or not we'll have a collision by *projecting* our collision boundary ahead by that much distance and seeing if there is anything in that volume. Unity has a handy function for just this: *Physics2D.BoxCast.* See the doc pages here.

- In *ProcessCollision*, use *BoxCast* to determine whether or not the player will hit something in the next physics update
  - Do this for both directions (right and down), changing the passed arguments appropriately
- If there is a collision below (in other words, if the player landed on something), remember to call *CollisionDown.Invoke* to notify anyone who might be listening to it
  - We didn't really talk about **event**s in class, but here's one in the wild

Pay particularly close attention to two of *BoxCast*'s arguments:

- For the *size*, don't pass the size of the Player's BoxCollider2D. If you do that, the physics engine will think that it's hit something when it hasn't (told you it was bad…); instead, pass a reduced dimension, like this:



- This will still catch collisions to the right and below, but you won't get false positives from clipping into the platforms
- Secondly, *BoxCast* has an argument *layerMask*, which determines which objects the cast needs to check and which it can ignore; we've provided an appropriate **LayerMask** for you to use

One last thing: remember to call *ProcessCollision()* in-between updating `_velocity` and moving the RigidBody's `position`. If you don't, the Player will continue moving at its old `_velocity` for an extra `FixedUpdate`, which isn't desirable.

When it's all said and done, your player should move around, collide with your platforms, and be able to jump. In fact, you've got most of a game on your hands.

Part 4: A Pause Menu

This game is pretty fast-paced, so it would be nice to be able to pause it to take a breather. There exists a *PauseMenu* class, you just need to fill in the gaps.

- Fill in *PauseMenu*'s constructor, similarly to how you handled the *MainMenu*
- Don't forget to initialize the buttons
  - The "Resume" button should just call the *TimeManager*'s *Unpause* function
  - The "Main Menu" button should **hide** the *PauseMenu* and call the *ReturnToMenu* function from *Game.cs*

Part 5: Submission

As always, we only need your "Assets" and "ProjectSettings" folders. Zip those up, and upload to Canvas.