CS 376 Exercise 2:
Making some debug widgets

Important

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.

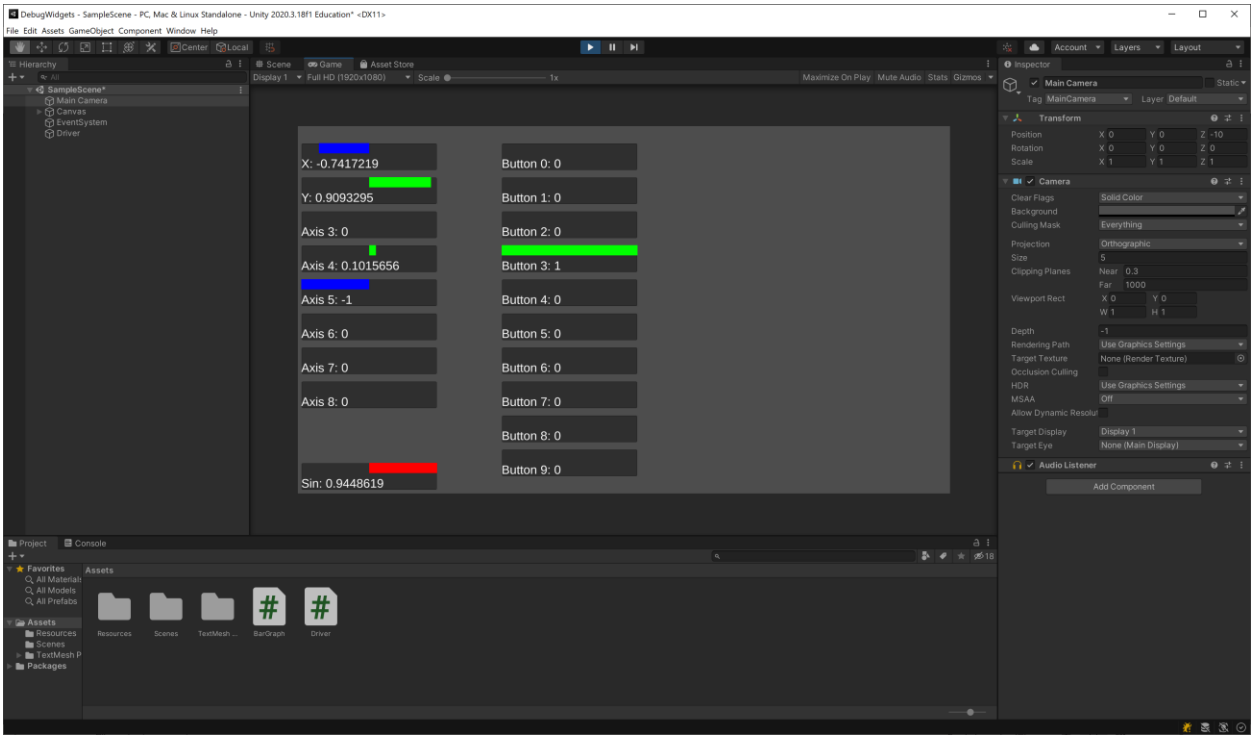
We will use the Full HD setting for all assignments in the class.

Overview

In this assignment, you'll learn to use Unity's systems for

- Controlling the positioning of objects on the screen (Transforms and RectTransforms)
- Displaying user-interface components
- Interfacing to game controllers
- Creating GameObjects at run-time

In particular, you'll implement an object you can put on the screen that displays a bar that moves as some input changes. You'll then use that to monitor the status of all the buttons and joysticks on your game controller. When it's working, the screen will look something like this:



From a user's point of view, you're building a system where you can call:

BarGraph.Find(*name*, *position*, *min*, *max*)

And it will find the bar graph object named *name*. If there isn't one, it will create one at *position*, give it the name *name*, and configure it so that it displays numeric values between *min* and *max*.

You can then take that object and call its **SetReading**(*value*) method, which will change it's appearance on screen to show the number *value*. Positive values will show up as green bars, negative ones as blue bars, and values outside the min/max range specified in the **Find**() call will show up as red bars. In addition, it will display the specific value as text below the bar.

The Unity UI System

Unity is now on it's fourth user interface system, however the latest one is still in beta and has been for a few years now. So we're going to use the third UI system. That system provides built-in components for UI elements you might want to appear on screen:

- Images
- Text boxes
- Text input boxes
- Buttons
- Check boxes
- *Etc*.

Each of these is implemented as a component and placed in its own game object. The reason for putting it in its own game object is so that it has its own transform, allowing you to position it independently of the other UI elements. In this assignment, we'll use the text box and image components.

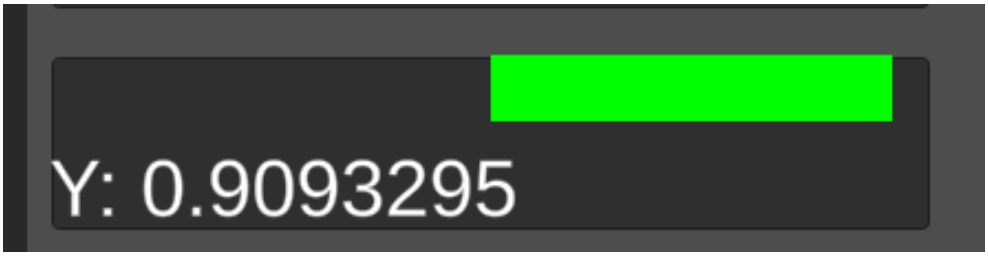
There are a couple of special things about UI components:

- They have to be placed as children of the magic "**Canvas**" game object that Unity makes for you automatically
- They have **RectTransforms**, rather than regular Transforms. RectTransform is a subclass of Transform that has some extra information to keep track of the UI element's width and height in addition to its position. It also has information about how it should grow or shrink if you run the game on different size screens. None of that will matter for this assignment, save for a little bit about width.

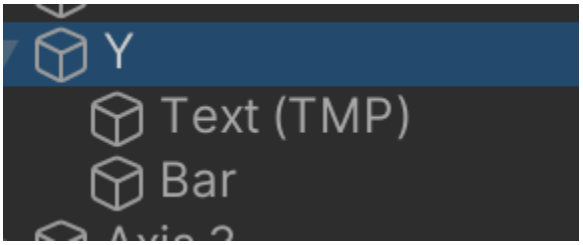
Bar graph widgets

We'll call each little bar graph on the screen a "bar graph widget" because we have to call it something, and "widget" is a common term for a user-interface element that appears on screen.

Each bar graph widget:



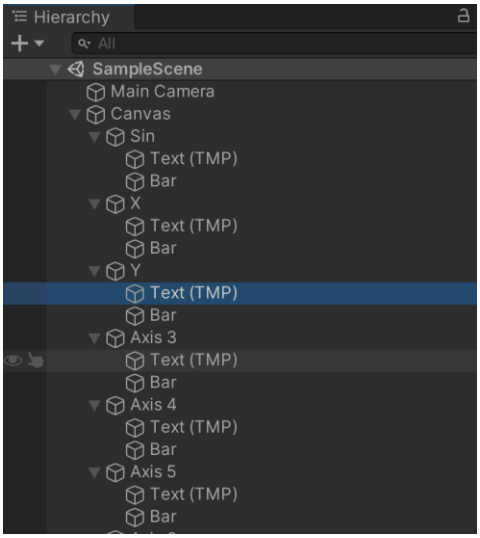
is implemented as a tree of GameObjects:



Here, Y is the parent GameObject, and it has two children inside of it, one to display the text and one to display the bar itself. These GameObjects have the following components:

- The parent object (called **Y** here, because it's displaying the Y joystick) has:
  - A **RectTransform** (again, this is a kind of Transform)
  - A **CanvasRenderer**, which we can ignore
  - An **Image**, which is essentially like a SpriteRenderer, but that works as a User Interface element (why Unity didn't just let you use SpriteRenderers is a little complicated, just go with the flow for now). The Image here is just a background image that goes behind the text and bar graph. So it's not really doing anything. It's there largely because Unity always makes one for a UI element.
  - A **BarGraph** component. These are defined in BarGraph.cs, and are responsible for controlling and updating the components of the child GameObjects (Text and Bar).
- **Text (TMP)** has:
  - A **RectTransform** and **CanvasRenderer**, like the parent
  - A **TextMeshPro UI Text** component, which draws text on the screen, specifically whatever text is stored in its `.text` property. We'll make it display different text by changing its `.text` propery. The code for doing that will go in BarGraph.cs.
- **Bar** has:
  - A **RectTransform** and **CanvasRenderer**, like the parent
  - An **Image**, which is the literal image of the bar. We'll make the bar grow and shrink by changing its scale in its RectTransform to stretch and squash it horizontally. We'll also change the bar's color by changing the Image component's `.color` property. Once again, the code for controlling this will go in BarGraph.cs.

This basic tree for the bar graph widget is defined in a prefab (found in the Resources subfolder of the Assets folder). Each time Find needs to make a new widget, it instantiates this prefab. So that we end up with a bunch of copies of it:



Notice that the parent GameObjects for each widget has the name of the widget as it should appear on the screen, but then each has children with the same names (Text (TMP) and Bar).

The code you're going to write will be the class for the BarGraph component, which is found in BarGraph.cs:

- BarGraph.**Start**()
  Initializes the BarGraph component.
- BarGraph.**Find**(*name*, *position*, *min*, *max*)
  This is a static method. It should keep a table (in a static field) of the widgets its already made and return the widget if it's already been created. If it hasn't already been created, it should create it and add it to the table. In either case, it should return the BarGraph component from inside the widget, not the GameObject, since the BarGraph component is what the user will use to change the reading displayed in the widget, using:
- BarGraph.**SetReading**(*value*)
  This should change the bar to be the right width and color for the specified *value*, and update the text too. To change the width of the bar, it will change the "scale" of the bar in the bar's RectTransform. So it's probably time for us to talk about…

How Transforms work

The Transform of a GameObject, including the RectTransform of UI a element, determines where it's drawn on screen. In particular, it determines its:

- Position
  Where on screen is the (0,0) of its local coordinate system?
- Rotation
  How is its local coordinate system rotated relative to the world coordinates?
- Scale
  How is the object sized relative to world coordiantes? Scale has both X and Y components so that you can stretch and shrink things along each axis independently. We'll take advantage of that to stretch the bar horizontally without changing it vertically.

The Transform class has properties you can get and set for the first two of these: `.position`, `.rotation` properties. Scale is a little more complicated, and we'll get to it in a moment.

Transform chains

Recall that every GameObject has its own transform and that our widgets are each made out of three different GameObjects. That would seem to be inconvenient: in order to move the widget around, we would need to change the positions of all three GameObjects.

Fortunately, the position of each game object is really stored in terms of the position of its parent. So the real position that's stored in the transform is its `.localPosition`: its position *relative to* the position of its parent. If we ignore rotation and scale, then the position of a child object is just its parent's `position` plus the child object's `localPosition`.

Rotation works the same way; the rotation of a child is the rotation of the parent combined with its own `localRotation`. Because rotation is weird in 3D, the rotations are represented with exotic complex numbers called quaternions. Fortunately, we can ignore those for the moment.

Translation and rotation interact with one another. So the real rules for how the local position, rotation, and scale interact with the parent's are a little complicated and basically require we use matrix math. We'll talk about that in lecture shortly. But for this assignment we're only going to have to worry about `localPosition` and `localScale`. In particular, we'll adjust the `localPosition` and `localScale` of the **Bar** game objects.

Instantiating new GameObjects

When you create a new widget, you're going to need to create all three of its game objects, along with their components. We've made a prefab for you with a widget inside it. We've put the prefab inside of a folder called **Resources**. Folders named Resources have a special meaning in Unity projects. If something is in a Resources folder, you can load it at run time by name using the **Resources.Load** method. So that will let you get the prefab. Having gotten the prefab, you can then make an instance of it in the current level using the **Instantiate** method:

Instantiate(Object *prefab*, Vector3 *position*, Quaternion *rotation*, Transform *parentTransform*)

When you call this, it makes a new copy of the prefab as a child of whatever GameObject *parentTransform* is inside of.[1] It places that game object at the specified position in world coordinates, and applies the specified *rotation*. We won't be rotating anything so just use the value **Quaternion.identity** for the rotation; that means no rotation. Instantiate will return the GameObject it just created.

Annoying note: Unity requires all UI components to we inside of a game object with a Canvas component. There's already such a component in the level. You just need to use it as the parent for your instantiated prefabs. The code tells you how to do this.

[1] Remember we said that Unity stores its parent/child pointers in the Transform components rather than the GameObjects themselves, which is why every GameObject has to have a Transform, even if it isn't intended to be visible on screen.

Game controllers

Different game controllers have different collections of buttons, joysticks, analog triggers, and so on. While the USB and Bluetooth standards provide way of interfacing game controllers to host computers, they don't specify any particular layout of buttons and joysticks or even any way of identifying the layout. Instead, buttons are numbered: button 0, button 1, *etc*.

Different models of controllers place those buttons in different locations, and one basically has to determine by trial and error which button number the "X" button corresponds to on an Xbox controller versus a Nintendo controller.

Analog controls are similarly numbered and are referred to as "axes": axis 0, axis 1, etc. A joystick will control two axes: one horizontal and one vertical. Often, if the horizontal axis for a joystick is axis n, then the vertical axis will either be axis $n + 1$ or $n - 1$. But for Sony controllers, it can be $n \pm 2$. Generally, the "main" joystick will be axes 0 and 1, identified within Unity as "X" and "Y", while other axes are just "joystick axis 3", "joystick axis 4", etc. But you can't be sure. Analog triggers are each a single axis. But again, no way of predicting what the axis numbers will be.

So you generally have to determine by trial and error which axes correspond to which joysticks, and which button numbers correspond to which buttons. There are a number of tools to help you do this, but you're making your own for this assignment. The game code puts up bar graph widgets for different axis and button numbers, and then lets you see what happens when you move things around on your controller.

The Unity controller API

Because different controllers map functionality to different joystick and button numbers, it's a bad idea to hard-code such numbers into your game logic. Instead, Unity lets you specify a set of abstract controls, called "virtual axes" in their documentation, that are named with strings. You can then ask for the current value of a given virtual axis by calling the **Input.GetAxis** method, which takes the name of the axis (a string) as its argument and returns a float between -1 and 1:

- For buttons, they will read 0 when not pressed and 1 when they are pressed
- For analog triggers, they will give you a number between 0 and 1, with 0 meaning it's not pulled, 1 meaning it's pulled all the way, and 0.5 meaning it's pulled halfway. For Xbox-style controllers both analog triggers report on the same abstract axis, with one trigger giving you positive values when pulled and the other negative values when pulled (you can't pull both at once).
- For joysticks, you get 0 when the joystick is in its neutral position, 1 when it is pushed all the way in one direction, and -1 when pushed all the way in the opposite direction. Note, however that the rest position of a joystick usually doesn't correspond exactly to its zero position, so you will get a value slightly different from zero. For this reason, it's common to include a "dead zone" when reading a joy stick, where any number near zero is treated as zero. However, we've disabled that for this particular assignment.
- Directional Pads, or "D-Pads", are buttons arranged in a cross shape that act like a non-analog joystick. They generally behave like two axes, as if they were joysticks that don't have any values other than -1, 0, and +1.

In order to use GetAxis, you must map a virtual axis to a particular hardware axis or button. That's done in the Unity Input Manager, which you can find under the Project Settings dialog. However, for this assignment, we've already set it up for you.

What to do

Open the project in Unity. Remember to open the level file from the Scenes folder. If you open the project and the only game object is the camera, then you forgot to open the level file. Go to the Scenes folder in the Project window and double-click the level file.

Now open BarGraph.cs, e.g. by doing Assets>Open C# Project or double-clicking on BarGraph.cs in the Unity Editor's Project window. Fill in the code in code for the following methods. Each method is partially filled in and then has some comments reading "TODO: …". Do what the comments say:

- **Prefab**
  This is a property with a get method. It loads the prefab for the widgets and returns it.
- **Find**(*name*, *position*, *min*, *max*)
  This should keep a table of the widgets its already made and return the widget if it's already been created, otherwise create it and add it to the table. Once you've written this, you can try running the game. The widgets should appear on the screen, even if they aren't updating yet. However, remember to set the Aspect Ratio to "Full HD".
- **Start**()
  This should initialize the BarGraph component. Once you've written this, you can run the game again, and the widgets should all at least have the right names displayed.
- **SetReading**(*value*)
  This should change the bar to be the right width and color for *value*, and updates the text too. To change the width of the bar, it will use SetWidthPercent to change the "scale" of the bar.
- **SetWidthPercent**(float percent, Color c)
  This should changes the width of the bar to the specified percentage of its maximum width. It should also set its color to the specified color. You change the width by changing the x component of the bar's transform's localScale. If percent is zero, the x scale is zero. If it's 1, it's the same as the y scale. If it's -1, it's negative the y scale. Finally, if `signedDisplay` is true, then the bar is moved into the middle of the widget and so it only has half as much space, so decrease the x scale accordingly. Once you've written this, everything should start working.

When you finish, you should check that you have the following behavior. When you call SetReading(*value*),

- The text should always show the *value* passed in, even if it's outside the min and max range for the bargraph.
- The color should be red if the *value* is outside the min/max range. Otherwise, it should be green for positive values and blue for negative ones.
- Note that if *value* is zero, the bar will be invisible, so the color is irrelevant.
- If min < 0, then the bar should be drawn from the center of the widget to the right for positive values, or from the center to the left for negative values
- If min >=0 then the bar should be drawn from the left side of the widget to the right

- The bar should never go outside the bounds of the widget. So if min = -1 and max = 1, but you call SetReading(700), then it should draw a red bar from the center to the right side of the widget, but not extend past the right side of the widge. And and the text should still say 700.

Learning the mapping of your game controller

Now you can use this project to determine the numbers of the axes and buttons for the different controls on your controller. Get to know them, since you'll need them for the next assignment!

Note that we've included an extra widget at the bottom of the screen called "Sin" that's just moving up and down and going out of range so you can test your code even if you don't have your controller in hand yet.

Turning it in

When you're done, upload your BarGraph.cs file.