Lecture 8

The Fourth Wall: User Interface in Unity

98-127: Game Creation for People Who Want to Make Games (S19)

Written by N Carter Williams

Instructors:
Adrian Biagioli (abiagiol@andrew.cmu.edu)
Carter Williams (ncwillia@andrew.cmu.edu)

1 The Unity Canvas

The **Canvas** is a component in Unity which acts unlike any other component. In combination with the **Event System**, the canvas allows for you to easily make UI elements like **Buttons** and **Input Fields**. In this Lecture we are going to learn about how to use the **Canvas** and how to make your UI look great in **Unity**.

1. Placing Images in the Canvas

To place an image in the **Canvas**, navigate to GameObject > UI > Image. This will automatically create a **Canvas** if you didn't already have one. As a child, it will also add a **GameObject** with a **Image** component. The **Image** component is a script which renders a sprite on the screen, much like a **Sprite Renderer**. The primary difference between them is that an **Image** is used for UI elements in the **Canvas**, while **Sprites** are used in world space.

One aspect of that is how **Sprite Renderers** and **Images** sort overlapping sprites. Outside of the **Canvas** you usually use *sorting layers* or the z position of the object to determine which images will render on top of one another. Each sprite is assigned to a layer, which has a priority, and the layers with a lower priority render first (behind the other layers). Within a **Canvas**, instead of using *sorting layers*, you must use the *Hierarchy* to determine which sprites appear on top. Elements at the top of the *Hierarchy* render first and will be covered by elements further down. This also includes the children of a sprite. All of an objects children will render over that object. This means that you may need to use empty **GameObjects** to serve as organizational placeholders so that you have fine control over render order.

2. Canvas settings

Looking back at the **Canvas** object, let's try to figure out what's going on here. In the **Canvas** script itself, we have a choice of *Render Mode*. The options are *Screen Space - Camera, Screen Space - Overlay, and World Space*. If you want your UI to exist in the game like a health bar or street sign, you need to use the *World Space* option. Otherwise, you can choose between *Screen Space - Camera*, which renders the UI at

a specified distance from the camera and resizes along with the camera or *Screen Space - Overlay*, which always appears in front non-UI elements and only resizes with the screen size. A new addition to these options is the *Pixel Perfect* option which will make Unity keep your images pixel aligned. This is a must for pixel art games.

Moving down the **Canvas** we see the **Canvas Scaler** component. Again we get three choices. This time its *Constant Pixel Size, Scale With Screen Size, and Constant Physical Size*. If you want the size (not just the position) of every UI element to scale to meet different screen sizes, you can select *Scale With Screen Size*. This will scale elements with the selected ratio of width and height of the screen. To see this in action, you can put your **Game** window in *free aspect* and play around with resizing it and watching your UI elements. The constant size options determine the size of elements either by the amount of pixels in the image or by the physical space the take up on the screen. In general, *Constant Physical Size* will maintain the look that you want on screens with varying DPI. *Constant Pixel Size* is the default possibly because not all monitors report their DPI correctly, so if you wish to change this you will have to remember to do so at the start of your project.

Regardless of whether you want to the size all of your elements scale with screen size, **Unity** has a system for scaling the position UI elements: *anchors*. Each **Rect Transform** has four *anchor points*, which will determine the how that element moves as the canvas changes size. You can click and drag on each triangle in the *Scene Window* to move the anchors around. When the screen resizes, each corner of the **Rect Transform** maintains a constant distance from its corresponding *anchor*, while the *anchors* themselves maintain a relative position to the parent's **Rect Transform**. The general use case is to put all *anchor points* on the edge or in the corner of the screen that you want the element to stay by. If the anchors are not all in the same position, the object will stretch with the canvas. One note to make here: the anchors are attached to the parent's *Rect Transform* and not the **Canvas** itself, so they cannot be placed outside of the parent's *Rect Transform* and will scale whenever parent scales. This is useful for creating windows inside the screen, but can be confusing if you aren't anticipating it. If you want something to be anchored to the **Canvas** itself, place it at the top level of the **Canvas**.

1.0.1 Import Settings

Before we're done with Images, Unity has a few features we ought to cover in the *Import Settings*. To see a sprites *Import Settings*, find the sprite in the *Project Window* and select it. Now look in the *Inspector Window* to see the same *Import Settings* that we looked at in Lecture 04.

The first feature we will consider is the ability to *nine slice* images with borders and corners that shouldn't scale. If you select the Sprite Editor button, a window will pop up with the sprite you selected. You may notice a few green squares at the edges of the sprite. If you drag these into the frame, the image will be divided up into 9 sections. These sections or *slices* represent areas that will be scaled differently as the image scales. The center will stretch or tile normally depending, the top and bottom stretch or tile horizontally, and the left and right edges tile vertically. The corners on the other hand will not tile or stretch at all. This is really useful for buttons with decorative borders that should remain the same size. If you want corners of various sizes, its a good idea to have couple versions of the same sprite with different pixels per unit. To use sprites sliced this way, go to the **Image** component and select *sliced* as the *Image Type*.

The other *Image Type* that you may find useful is the *Tiled* mode. If you want an image to have a certain texture and you are unsure of how large it will be or what dimensions it will have, you can use a texture that continuously wraps around the sides. To use a texture like this, select the it Tiled Image Type and it will automatically repeat when scaled instead of stretching.

2 The EventSystem

Unity's default system for handling interaction between player input and UI elements is called the **Event System**. In order to use *Buttons*, *Input Fields*, or *Scroll Views* you need to have a component in the world called *EventSystem*. This system is the interface between *StandardInputModule* and the UI system. The settings here will help you configure an automatic selection system, where players can select different buttons on screen by press the directional arrows or moving the joystick. These components are also essential for using mouse controlled UI.

Any elements that you want to be selectable must be under a *Canvas* with a *Graphic Raycaster* component. [1] Unity handles clicks by shooting a *ray* out from the camera through the camera and toward the world. If the ray collides with a *Raycast Target* on the way it stops and sends that event to the *EventSystem*. *Raycast Targets* can be any of Unity's default UI elements or custom elements that you make. They can even be in *World Space* like a button on the wall. Only one *Raycast Target* can be selected by a click. If multiple overlap, the target which is renders last will be selected. If you hide UI elements by setting their alpha to 0, those UI objects can still block *Raycasts* from hitting objects behind them. To prevent this, you can disable *Raycast Target* from most UI elements, or you can make a *Canvas Group* which will allow you to let *Raycasts* pass through all a UI element and all of its children. [2]

3 Scripting with the Camera

Anything that you want to show the players must pass through a *camera*. If your game uses a mouse you are going to want know how to convert between coordinates in the world and coordinates on screen. You can get the mouses position by calling *Input.mousePosition* . If you want to convert this position from screen space to world space you can call *Camera.ScreenToWorldPoint* on the output. If you are not using a mouse, disable the mouse cursor by setting *Cursor.visible* to false. Please do this if you do not need to need the mouse. It takes one line of code and will make your game instantly more immersive by getting rid of the little cursor that just sits in the middle of the screen.

Another quick thing you can do to make your 2D games more professional is to change the *Background* on the *Camera*. The default color is this kinda steely blue, but you can set it to a color that matches your aesthetic better even if you don't make custom background sprites.

If you didn't start your project in 2D or vice versa, you can always change the camera settings to 2D by setting the *Clear Flags* to solid color and the *Prospective* to orthographic. An othrographic camera will

[1] These three necessary components will come by default when you use the Create menu to create a UI element, but you will need to add them individually if you just attach a *Canvas* component to a *GameObject*.

[2] This is also a great component to hide large groups of UI components without just disabling them.

render objects at the same size regardless of how far away they are.

3.1 Layout Groups

On of the challenges of creating a user interface is working with an arbitrary number of elements. You can set the spacing on each element manually because they will appear at runtime. Instead you can rely on Unities *Layout Groups* to space and scale your objects for you. All the UI elements that are children of a *Layout Group* will have their *Rect Transforms* controlled to match the settings you choose. You can use this to create a grid of evenly spaced objects with the *Grid Layout Group* or a vertical column of objects with the *Vertical Layout Group*. If you want a child of a *Layout Group* to override some of the settings defined by it's group you can attach a *Layout Element* component to it and it you can enter your desired settings there. Be wary that nested *Layout Groups* will conflict in undocumented ways. Also be aware that the *Layout Groups* will redraw the whole group when any object is added or removed.