UI with Unity

Rishav, Omang

March 5, 2025

Contents

---

1. Introduction

Welcome to the fourth studio! For this assignment, we'll be building a Heads-Up Display using Unity's UI system. We'll be using the prototype third person character controller built in the last assignment as a base game, and building 3 UI components on top of that:

- A better coin counter visual
- An in-world canvas showing the controls

- A settings menu that pauses the game, and lets you change the player's speed

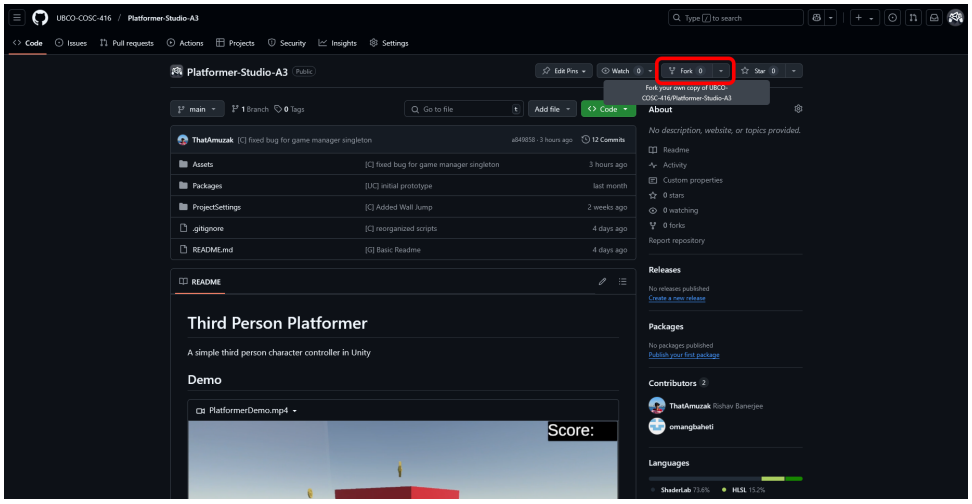
You can check out the demo video for the above features, and check out the GitHub repository for the project.
You are expected to have finished the Roll-A-Ball, Bowling with Physics and Third Person Platformer guides before starting this guide. The goal will be the same as the first two guides, that is you will be replicating the functionality showcased in this guide, and submit a GitHub repo link with a demo video. Rubric details can be found on the course assignment page. **For this guide, you won't need to set up a new project. Instead, you'll be working off of the Third Person Character controller implementation that we provide.**

> **Note**
>
> It's worth noting Unity has a new UI system called the UI toolkit. As with the new Input System, it's better to learn that if you intend on working on bigger projects. However, the scope of this course is to get everyone capable of prototyping basic games with Unity, and so we'll be using Unity's older and more popular UI system.

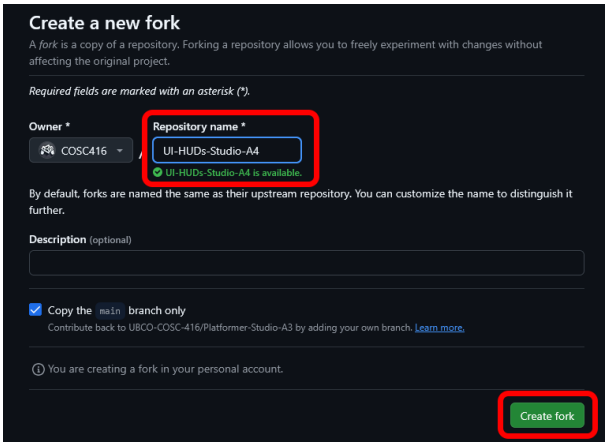
2. Project setup

2.1. Forking the Base Repo

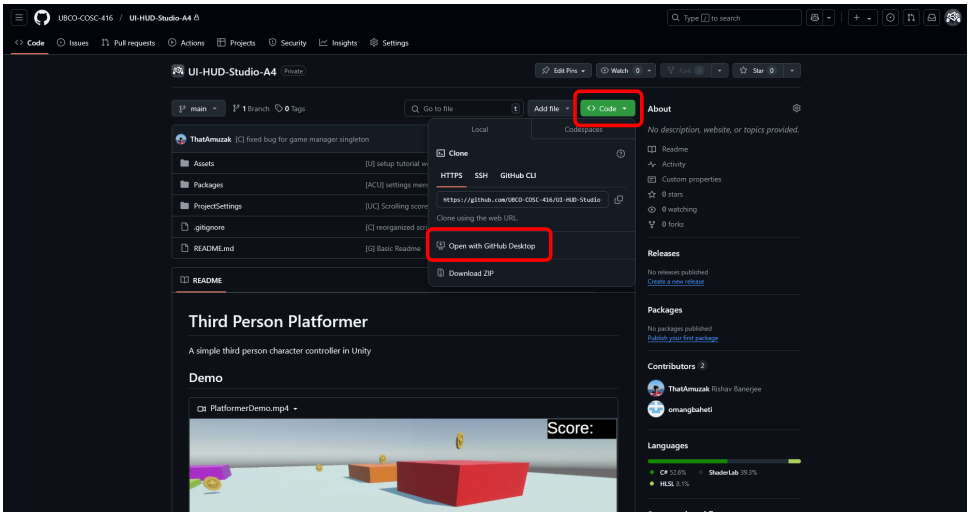
To get started, we'll be making a fork of the base repository from the UBCO COSC 416 GitHub organization. This is so that you can start from the same set of features as the guide and avoid confusion. Go to the Platformer Studio A3 project, and click on the fork option on the top right.



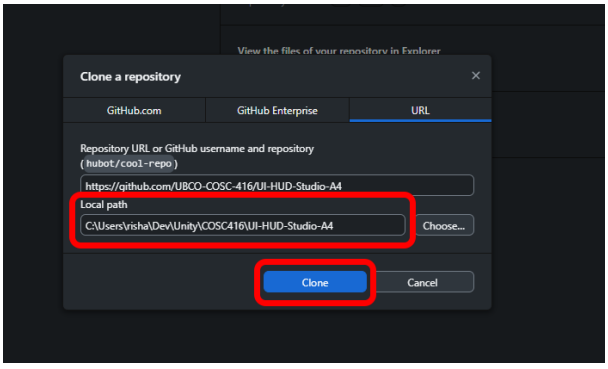
Next, name your fork something appropriate, and create the fork on your GitHub account.



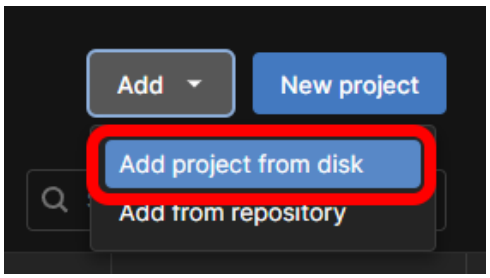
Once the fork has been created, you can get it on your machine, by clicking on `Clone > Open with GitHub Desktop`.

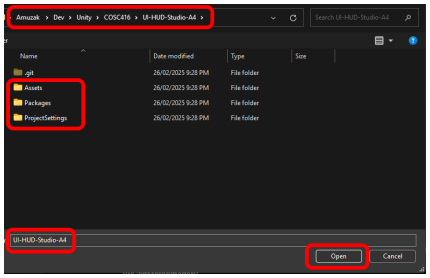


And then place the project somewhere appropriate on your system.



Finally, to open the project, you can open it with the **Unity Hub**, by clicking on `Add > Add Project from Disk` And make sure you select the right path for your project.





Unity will take some time to open the project, since all the Unity files will take some time to generate.

2.2. Utility Scripts

This section provides a clear explanation of the key functionalities of some scripts found in the "Utils" folder when you clone the repository. **There's no need to write any code here**, as these scripts are already included in the project. Instead, **this serves as a reference guide** to help you understand their purpose and functionality.

2.2.1 Singleton

The `SingletonMonoBehavior<T>` script defines a generic singleton class for the Unity MonoBehaviour components, ensuring only one instance of a given type exists in a scene. It uses a static variable instance to store the singleton reference, accessible via the Instance property. The `Awake()` method enforces the singleton pattern by checking if an instance already exists. If it does, it destroys the duplicate; then, it assigns itself as the instance.

This approach prevents multiple instances of essential objects like GameManager or AudioManager, provides global access without needing FindObjectOfType<T>(), and helps maintain a structured and efficient game architecture. To use it, a class like GameManager can inherit from SingletonMonoBehavior<GameManager>, ensuring GameManager.Instance is always available without duplication.

How to Use This Script?

Simply replace the `MonoBehaviour` inheritance with `SingletonMonoBehavior<T>` and replace T with your class name. For example, if you wanted a GameManager to be a Singleton, you would do:

```
public class GameManager : SingletonMonoBehavior<GameManager>
{
    protected override void Awake()
    {
        // to ensure the base class Singleton logic is executed
        base.Awake();
    }
}
```

Now, any script can access `GameManager.Instance`, ensuring only one GameManager exists in the scene. This also means no SerializeField references are required.

2.2.2 Reset Transforms

Reset Transforms is a quick hotkey to automatically set the position and rotation of a selected GameObject to Vector3.zero;

How to Use This Script?

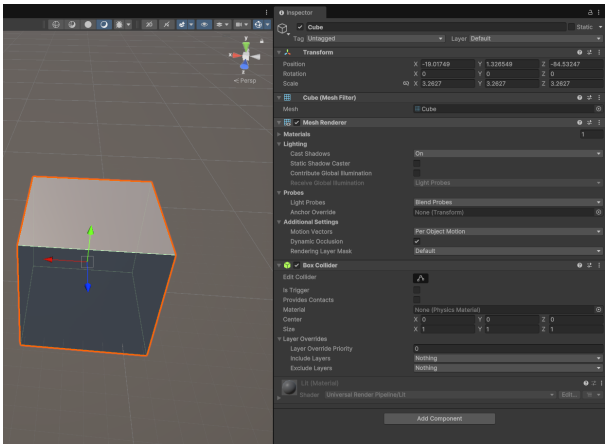
When you press Shift + R, your current selected GameObject resets its position and rotation to the world center.

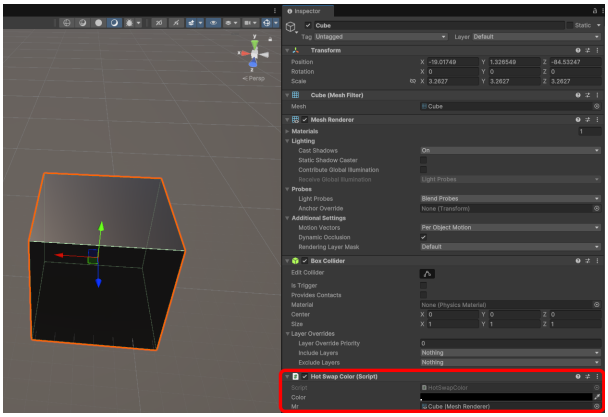
2.2.3 Hot Swap Color

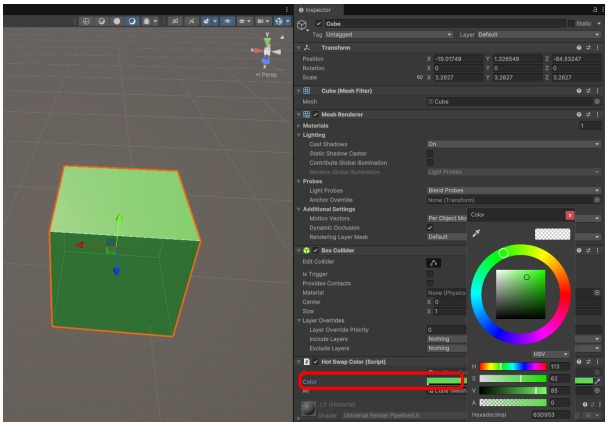
Hot Swap Color is a script that sets the color of a material without requiring a new separate material for that color. It uses Material Property Blocks to implement this, and edits the default material's properties only for that object. The colors are applied using the OnValidate function, which is called when changes in the editor occur. That way you do not need to go into play mode to see the color.

How to Use This Script?

Simply attach the `HotSwapColor` component to any GameObject with a mesh renderer component. It will automatically get the Mesh Renderer of that object if present. You can then change the color of the object.



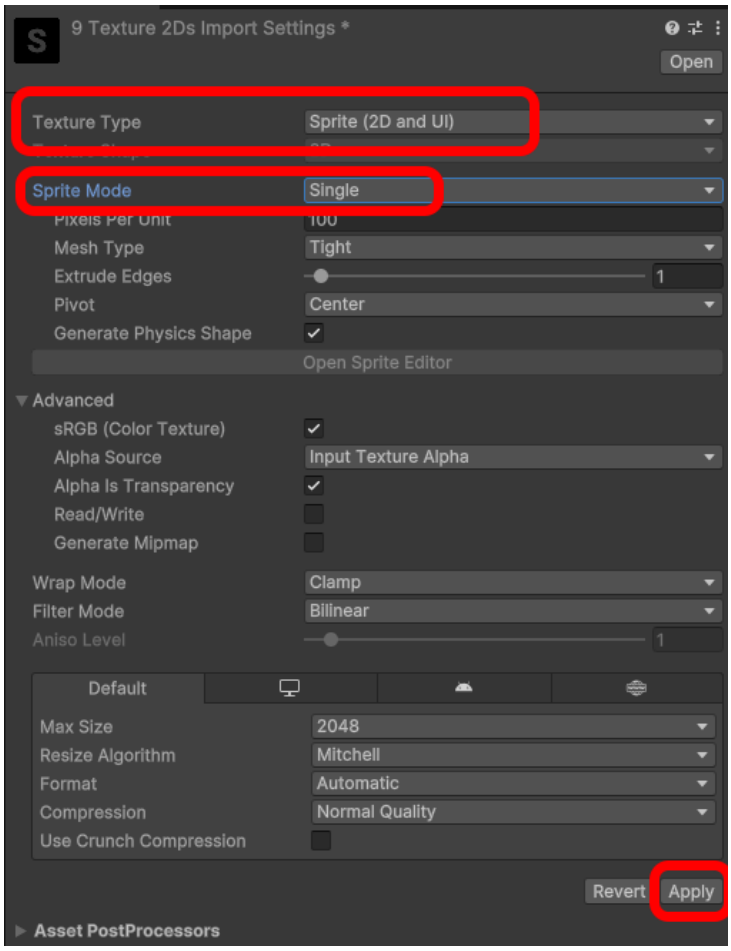




2.3. Importing UI Assets

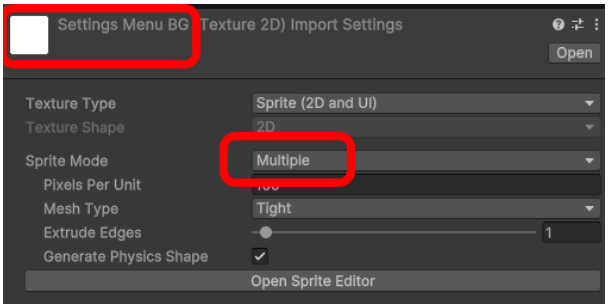
This guide will require some art assets. You can download them here. Download and unzip the contents of the folder, and place the files in your project in a folder called "Art" or "Resources/Art".

These assets will need to be configured for UI usage. Select **all the assets** in your project window (you can shift-click all of them at once). In the inspector, set the following settings for **all of these art assets**.



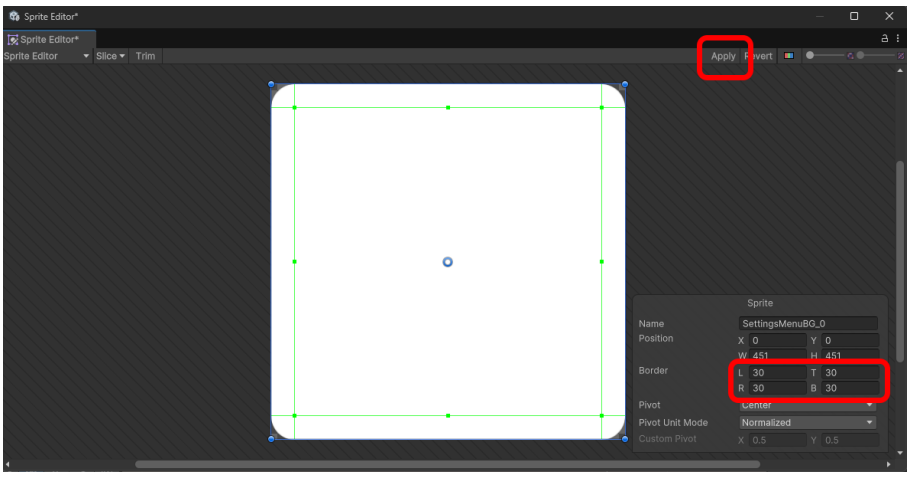
This allows you to utilize all the assets as art sprites.

We also want to modify the "SettingsMenuBG.png" file to treat it as a 9-tile sprite. This is because we want the rounded corners to stay proportional as we stretch and squash the image around. To do this, only select the "SettingsMenuBG.png" file, and change that asset's sprite mode from single to multiple. Apply, and then click on the sprite editor.



> **Note**
>
> Unity might ask you to install the Sprite Editor package if it isn't installed already. If shown over where the screenshot shows "Open Sprite Editor", then click on it to install the sprite editor package.

Once you've applied the settings and opened the Sprite Editor, set the border values to 30 for Top, Bottom, Left and Right.



This tells Unity that the borders exist beyond those limits from the borders of the image (as visible from the green lines). Once done, click on apply. Now all our assets are ready to be used for UI.

Tip

Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Imported UI Assets" would suffice.
If you see the git history on GitHub Desktop (or the commit history on GitHub), you'll see a commit notation that we follow: [U] for Unity editor changes, [C] for code changes, [A] for asset changes and [G] for git changes. You can continue following that notation if you wish, but it isn't mandatory

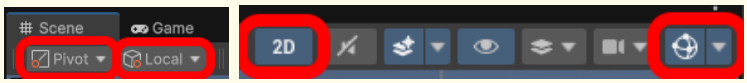
3. Improving Coin Counter

3.1. Visuals Upgrade

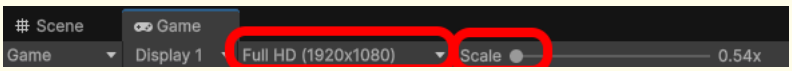
We're going to start by improving the visuals for the coin counter. Currently, it simply updates the number next to the "Score:" string. We would rather have the new number update the old one by sliding up into place.

Note

Remember to set the Scene view transforms to Pivot and Local, and Enable 2D mode and Gizmos. 2D more is especially useful when setting up UIs.

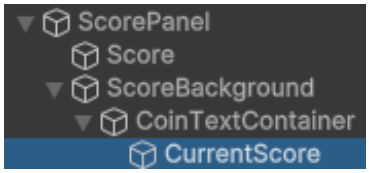


Also don't forget to set your game view resolution to 1080p and minimize the scale slider.

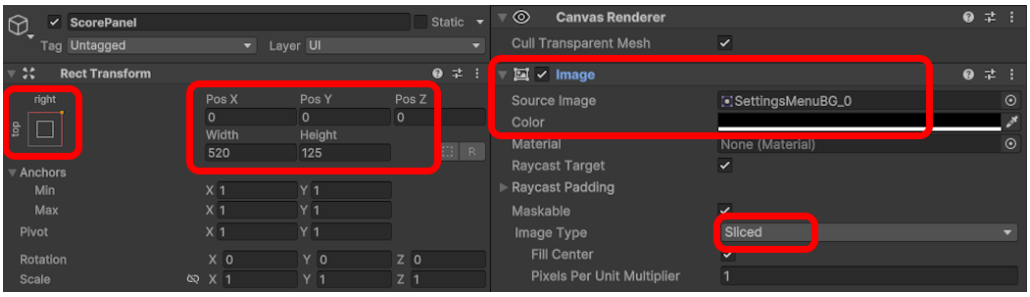


Currently, our Score UI looks as follows. It's fine, but a little bland.

Let's first separate the word "Score:" from the actual score text. We'll rename the panel to "ScorePanel" for clarity, then create `UI > Image` called "ScoreBackground". Next, create an empty child for this ScoreBackground called CoinTextContainer. Finally, create a `UI > Text - TextMeshPro` child called CurrentScore.

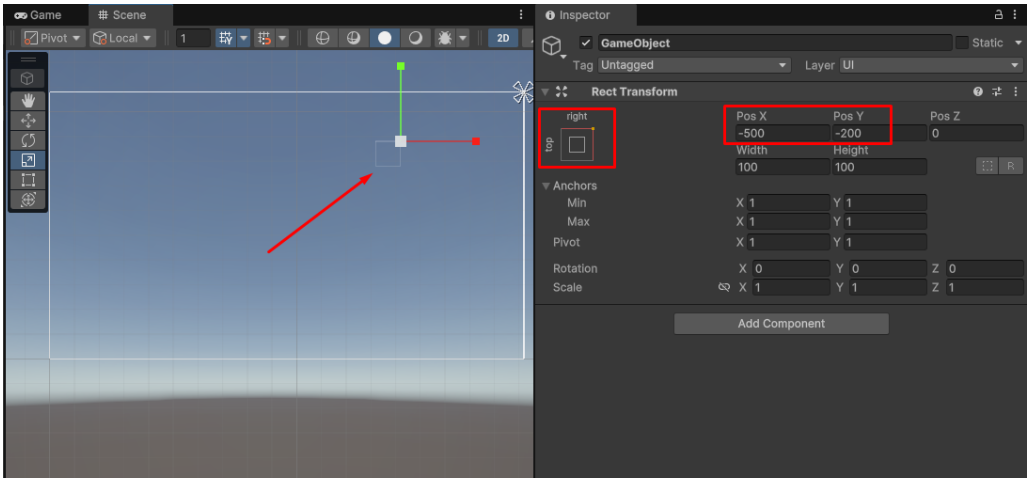


The Score Panel will have the following component settings.

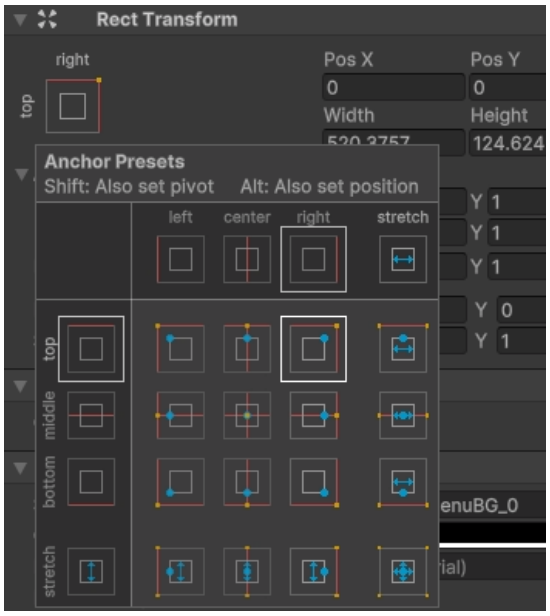


In Unity, UI components utilize RectTransforms instead of standard Transforms. RectTransforms are specialized versions of Transforms designed specifically for UI layout and positioning. Unlike regular Transforms, where Position is defined by simple X, Y, and Z coordinates, RectTransforms introduce a more flexible system. One key feature is the ability to set an anchor point for UI elements, determining how they respond to changes in their parent container. These anchors can be set to one of nine predefined positions: Top Left, Top Center, Top Right, Middle Left, Middle Center, Middle Right, Bottom Left, Bottom Center, and Bottom Right.

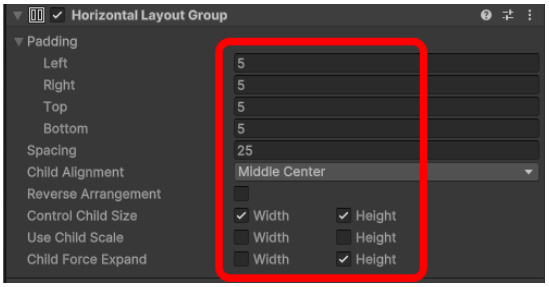
The Pos X, Y and Z values are relative to the selected anchor. For example, if we have selected the top right anchor on the transform, and we set the position as (-500,-200,0), it will position the UI element -500 units from the top right anchor in the X direction and -200 units in the Y direction.



In order to set the anchor to top right, click on the box icon in the Rect Transform. This lets you anchor an object to the extents of the parent (which is our entire canvas right now). You can also set the position and the pivot by holding down shift and alt on Windows or option on Mac.



You can then set the position x and y to 0 (meaning it should be in the top right corner), and then set the width and height as required. A value of 520 for the width and 125 for the height looks good.
Set the image to the SettingsMenuBG.png sprite that we configured, and make sure to set the image type to sliced to utilize the 9-tile setup we did earlier.
Next, add a "HorizontalLayout" component to the ScorePanel.



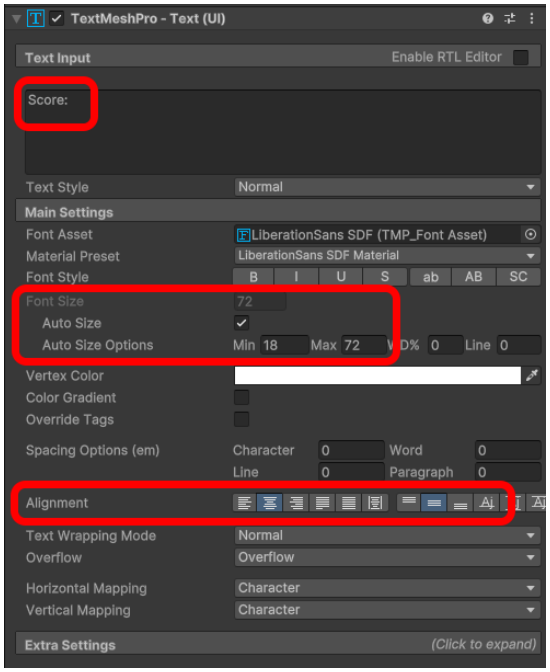
Horizontal and Vertical Layout groups allow you to define layout rules for all children underneath a given object. The settings work as follows:

- Left Right Top Bottom paddings decide the space around all the children items
- Spacing decides the space between each of the child elements
- Child Alignment decides the anchor for the children to be organized by
- Control Child Size dictates if the layout group controls the size of the children object or not. You can toggle control of the width and height separately
- Use Child Scale considers the scale of the children elements. This is rarely ever used.
- Child Force Expand decides if the children elements should be expanded to fill the space or not. For a horizontal layout group, where items will be laid out horizontally, it makes sense to expand and maximize all heights. Similarly, for a vertical layout group, enabling only the Width for Child Force Expand would make most sense.

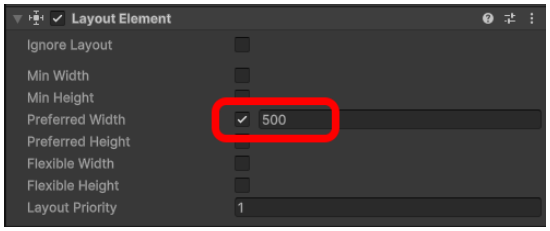
Tip

If you're familiar with front-end development, think of Unity's Vertical Layout Group like a `flex-direction: column` container in CSS, stacking elements top to bottom with automatic spacing. Likewise, the Horizontal Layout Group is like `flex-direction: row`, aligning elements side by side. Unity handles positioning and spacing dynamically, just like flexbox does with justify-content and align-items

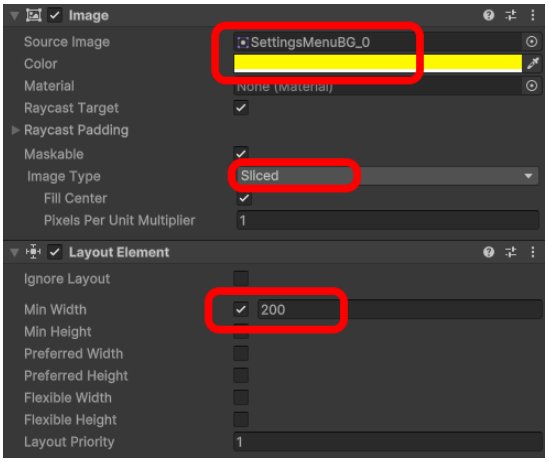
Now, for the Score text, first let's set the TextMeshPro settings



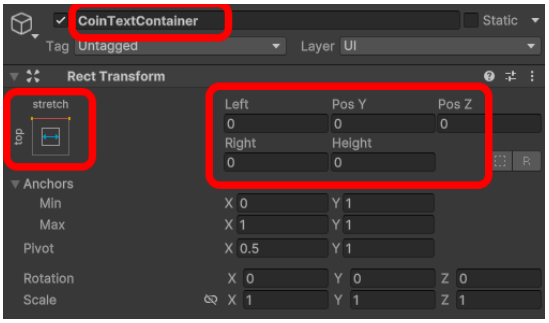
By enabling Auto Size, it will maximize the size of the UI element. We also set the alignment to be vertically and horizontally centered.
Next we'll add a layout element to the Score element, and set the preferred width to 500.



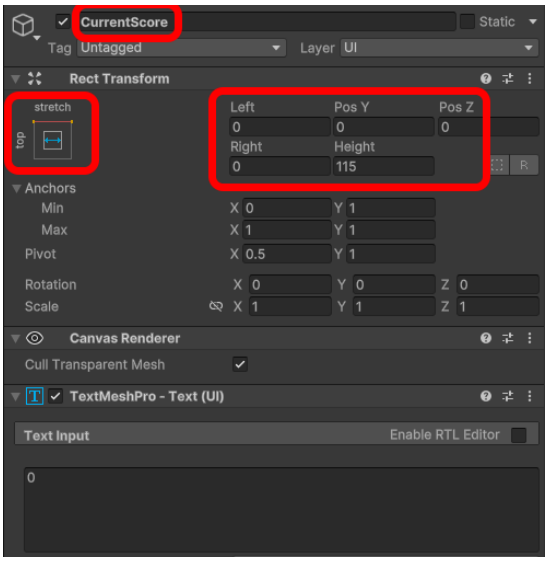
This tells the horizontal layout group that, if possible, set the width of this element to as much as possible till 500 units.
Finally, for the ScoreBackground GameObject, set the image properties similar to the ScorePanel background, but change the color to something else (say Yellow), and add a layout element that set the Min Width to 200. This tells the horizontal layout group that this element must have 200 units of horizontal width, and everything else should accommodate for this. Consequently, the score text will become a bit less than 500 units to properly fit.



Next, set the CoinTextContainer to anchor as Top Stretch (instead of top right) and zero out all the positional data. We're using "Top Stretch" instead of "Stretch Stretch" (where it expands fully to the extents of the parent) for this container, since we want an effect of the next score value coming from underneath, which means we will be moving this container later on. Making it fully stretched out would not make sense for this functionality.



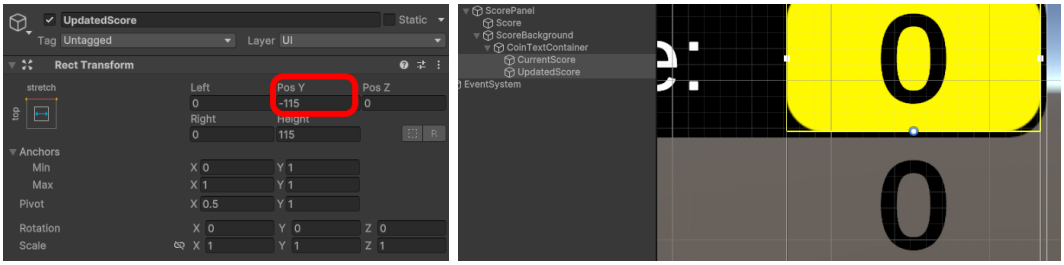
Finally, set the Current Score text to top stretch as well, and extend the height to 115 (since the score panel height was 125, and had 5 and 5 padding for the top and bottom). Also set the text mesh pro settings the same as the Score Text.



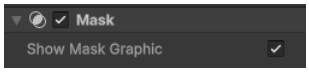
The score panel now looks much better!

3.2. Masks

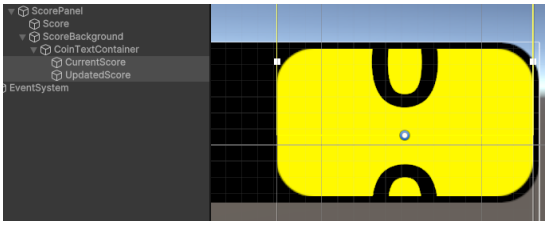
Next, we would like to have an updated number slide in for the score. Make a copy of the "Current Score" GameObject, call it "UpdatedScore" and move the Pos Y down to -115.



Next, add a "Mask" component to the ScoreBackground Gameobject.



You'll see that the UpdatedScore text will now disappear. This is because the mask acts such that any UI elements not present within the bounds of the GameObject will automatically disappear. You can see this behavior if you pull the CoinTextContainer object slightly up (set the y position of the CoinTextContainer to 50 for instance).



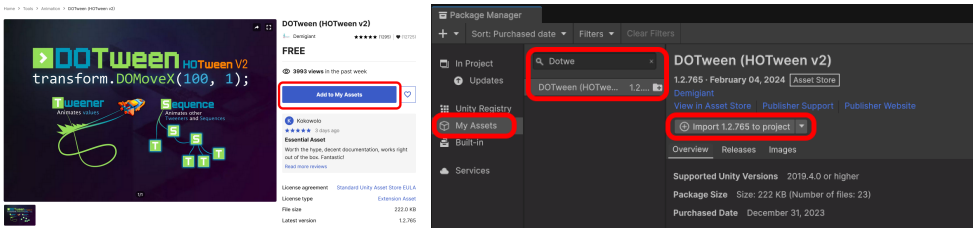
Set the position back to 0. Now we can build the sliding feature.

3.3. Animating the New Score

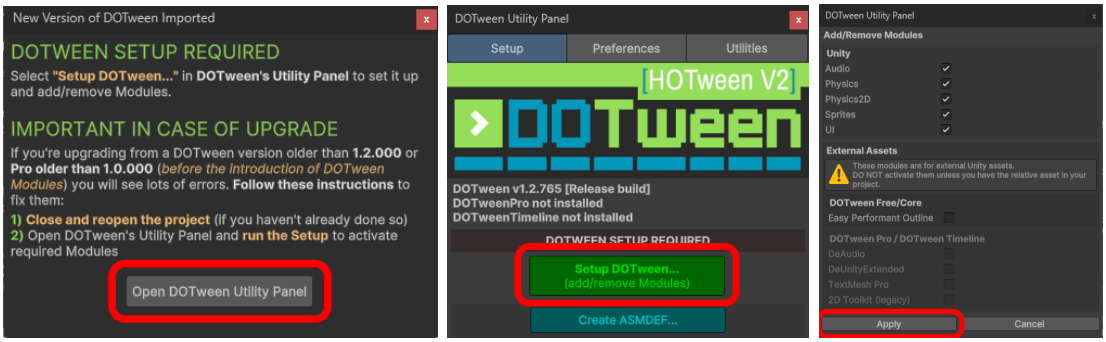
3.3.1 Installing DoTween

The word "Tween" comes from in-between frames for animation. Essentially you have one group of animators draw the "Key-Frames" in an animation, and another group of animators fills in the frames in the middle with some form of interpolation. We'll be implementing a sliding up animation using a tweening library, that allows us to define

the "Key-Frames" or the important positions for our animation, and the library will automatically implement the in-between positions.
To get started, go to the Unity Asset Store, and get the free DoTween package. Once added to your assets, import it into your project.



Once imported, configure it by following the prompts.



Now you have DoTween installed! This will allow us to easily create simple but powerful animations.

3.3.2 Implementing Sliding Animation

Create a new script called `CoinCounterUI`. We'll take in 4 references: the current coin counter (text mesh pro), the one that needs to be updated, the transform reference for the coin text container and the float duration of the animation. We'll also define two float variables: The initial position of the container, and the amount that needs to be moved.

```
using System.Collections;
using DG.Tweening;
using TMPro;
using UnityEngine;

public class CoinCounterUI : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI current;
    [SerializeField] private TextMeshProUGUI toUpdate;
    [SerializeField] private Transform coinTextContainer;
    [SerializeField] private float duration;

    private float containerInitPosition;
    private float moveAmount;
}
```

Next, we'll initialize in the Start Function to set both the text mesh pro texts to 0, and extract the initial container position and the amount to move from the height of the CurrentScore Rect component;

```
using System.Collections;
using DG.Tweening;
using TMPro;
using UnityEngine;

public class CoinCounterUI : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI current;
    [SerializeField] private TextMeshProUGUI toUpdate;
    [SerializeField] private Transform coinTextContainer;
    [SerializeField] private float duration;

    private float containerInitPosition;
    private float moveAmount;

    private void Start()
    {
        Canvas.ForceUpdateCanvases();
        current.SetText("0");
        toUpdate.SetText("0");
        containerInitPosition = coinTextContainer.localPosition.y;
        moveAmount = current.rectTransform.rect.height;
    }
}
```

UI rendering and positioning works asynchronously to the main game engine. This can cause incorrect calculations if we fetch some data from a UI component, and it's not been fully updated beforehand. `Canvas.ForceUpdateCanvases()` forces Unity to update the UI setup, so that we can ensure correct data is fetched, such as the local position, and height in this case.[1]
Next, we'll create a public function to create the animation using DoTween. This will set the `toUpdate` text mesh pro text to the updated score, and then use dotween to create an animation. We'll use the local move y to move the container to the height amount from the current position over the provided duration.

```
public class CoinCounterUI : MonoBehaviour
{
    .
    .
    .

    public void UpdateScore(int score)
    {
```

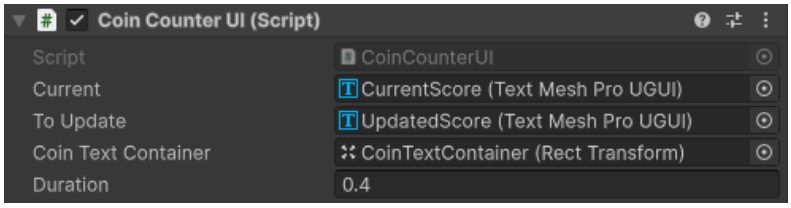
[1]Special thanks to Ronit and Lucas for pointing this issue out!

```
        // set the score to the masked text UI
        toUpdate.SetText($"{score}");
        // trigger the local move animation
        coinTextContainer.DOLocalMoveY(containerInitPosition + moveAmount,
            duration);
    }
}
```

Finally, update the `GameManager` script with the `CoinCounterUI` class to update the score.

```
using UnityEngine;

public class GameManager : SingletonMonoBehavior<GameManager>
{
    [SerializeField] private int score;
    [SerializeField] private CoinCounterUI coinCounter;

    protected override void Awake()
    {
        base.Awake();
        Cursor.visible = false;
        Cursor.lockState = CursorLockMode.Locked;
    }

    public void IncreaseScore()
    {
        score++;
        coinCounter.UpdateScore(score);
    }
}
```

Attach the `CoinCounterUI` script to the canvas, and make all the relevant references with the CurrentScore and UpdatedScore text mesh pro GameObject, and the Transform of the CoinTextContainer. Set the duration to something short like 0.4, and make sure to attach the reference to the `GameManager` script (attached to the GameManager GameObject) for the `CoinCounterUI`



Press play and move the character over a coin to see the effect.

If everything has been configured correctly, you should see the coin counter slide up from 0 to 1!

But there's an issue. If you try to collect more coins, then it goes back to updating in a static manner. This is because the current position of the text container is the target position, since we haven't reset it after the slide animation. We would want to perform the position reset right after the sliding animation is complete. To achieve this, we can use a "Coroutine", and wait for the duration of the animation before resetting the position. Finally, we can update the current text right before resetting the position to make it seem like nothing happened for the user.

```
public class CoinCounterUI : MonoBehaviour
{
    .
    .
    .

    public void UpdateScore(int score)
    {
        toUpdate.SetText($"{score}");
        coinTextContainer.DOLocalMoveY(containerInitPosition + moveAmount,
            duration);
        // this is how you start a coroutine
        StartCoroutine(ResetCoinContainer(score));
    }

    private IEnumerator ResetCoinContainer(int score)
    {
        // this tells the editor to wait for a given period of time
        yield return new WaitForSeconds(duration);
        // we use duration since that's the same time as the animation
        current.SetText($"{score}"); // update the original score
        Vector3 localPosition = coinTextContainer.localPosition;
        coinTextContainer.localPosition = new Vector3(localPosition.x,
            containerInitPosition, localPosition.z);
        // then reset the y-localPosition of the coinTextContainer
    }
}
```

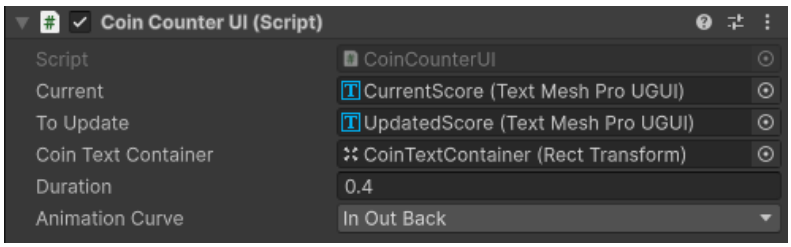
Press play and collect 3-4 coins.
If everything has been configured properly, then you should have a nicely animated sliding coin UI!

Hint

Coroutines are powerful functions that can execute code over time. You can script logic to execute over frames, over a fixed duration, or some other criteria. For now, you can consider them a great and easy way to execute any piece of logic after a given duration of time, using `yield return new WaitForSeconds(time);`
It's worth noting that Coroutines are not multithreaded. You do not have to worry about thread safety when using Coroutines.

DoTween allows us to set different animation curves easily. Right now it's using the default animation curve of `Ease.Linear`, which is fine, but we can try out some other easing curves. You can check out this easings cheat sheet for a quick reference on different easing functions and what they feel like.

```
1  public class CoinCounterUI : MonoBehaviour
2  {
3      [SerializeField] private TextMeshProUGUI current;
4      [SerializeField] private TextMeshProUGUI toUpdate;
5      [SerializeField] private Transform coinTextContainer;
6      [SerializeField] private float duration;
7      [SerializeField] private Ease animationCurve;
8
9      .
10     .
11     .
12
13     public void UpdateScore(int score)
14     {
15         toUpdate.SetText($"{score}");
16         // adding .SetEase(animationCurve) lets us select different
17         // animation curves to the dotween animation
18         coinTextContainer.DOLocalMoveY(containerInitPosition + moveAmount,
19             duration).SetEase(animationCurve);
20         StartCoroutine(ResetCoinContainer(score));
21     }
22
23     .
24     .
25     .
26 }
```



Try out different animation curves and see what you like!

Tip

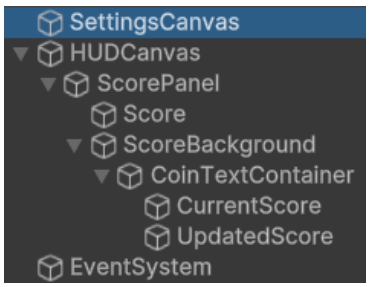
Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Implemented animated coin counter" would suffice.

4. Settings Menu

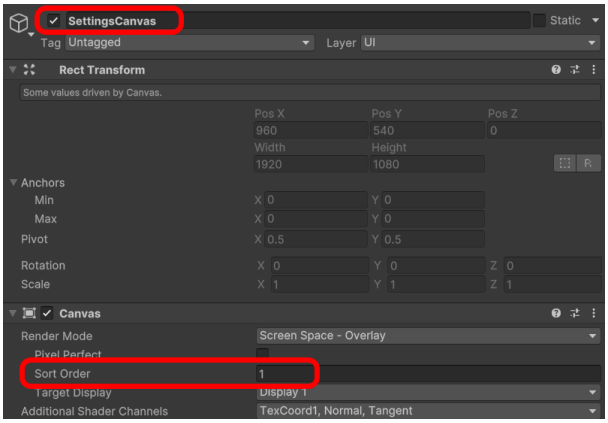
Next, we'll create a simple settings menu. This will allow the user to pause the game, and provide options to resume, change the player speed, or exit the game.

4.1. Canvas Setup

To keep things organized, rename the original canvas to "HUDCanvas", and create a new canvas called "SettingsCanvas".



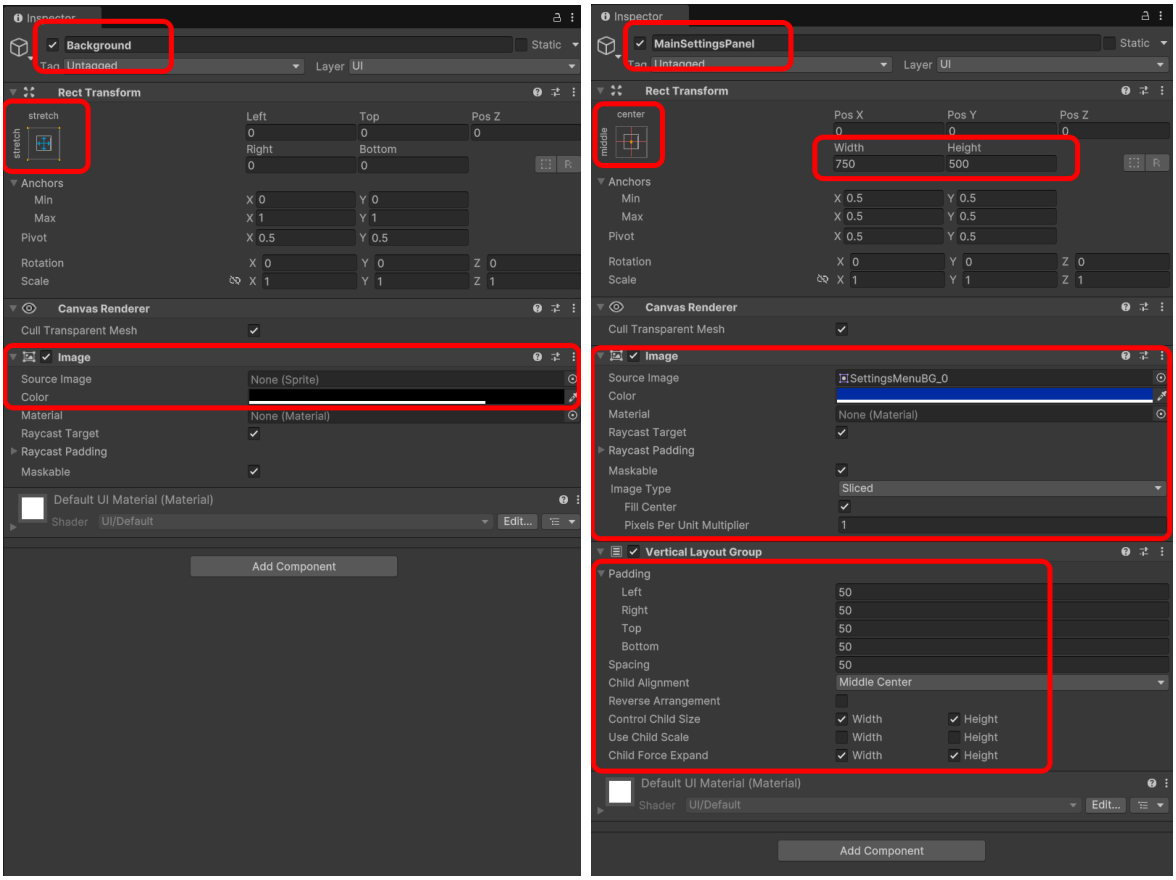
We'll also want to make sure we want to render the SettingsCanvas over the HUD. We can do so with the canvas sort order.



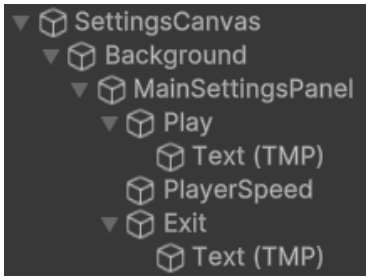
Since the default sort order is 0, setting the SettingsCanvas to sort order 1 will cause it to be placed in front of the HUD.

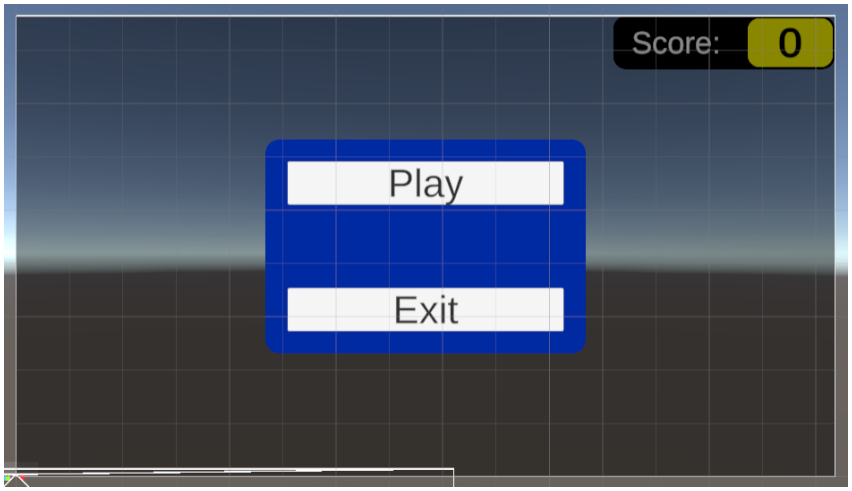
Next, create a child panel, which will be the translucent background. Set the source image of this panel to None. Expand the anchor to be "Stretch Stretch" (remember to hold shift + alt or option to set the position and pivot as well). Set the color to be black, and the alpha value to 75. Create another panel under that called MainSettingsPanel, make sure the anchor is center middle, the height to be 500 and the width to be 750, the image as the SettingsMenuBG image, and the color blue. Finally, for this MainSettingsPanel,

attach a vertical layout group with 50 padding on all sides and spacing, and enable the control child size and child force expand width and height. This will automatically force all elements under the item to be equally sized based on its shape as discussed earlier.



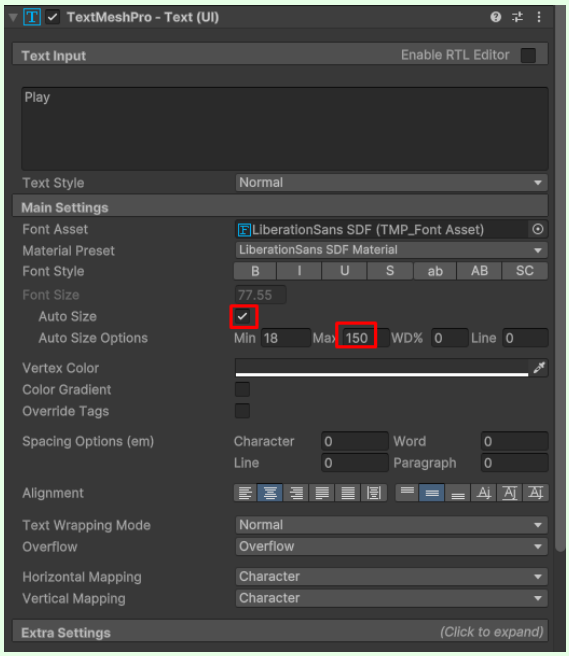
Next, create 2 buttons as children to the MainSettingsPanel. Change the text to the top one as Play, and the bottom one as Exit. You can change the text by editing the text mesh pro component that's automatically created as a child of the button GameObject. You can also rename them in the inspector. Then create an empty item called PlayerSpeed and place it between Play and Exit in the hierarchy.



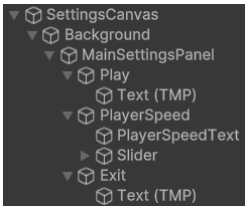


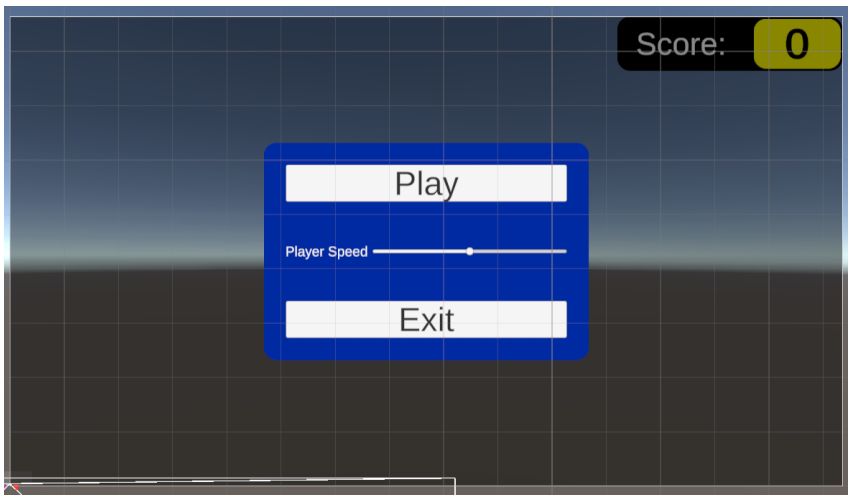
Tip

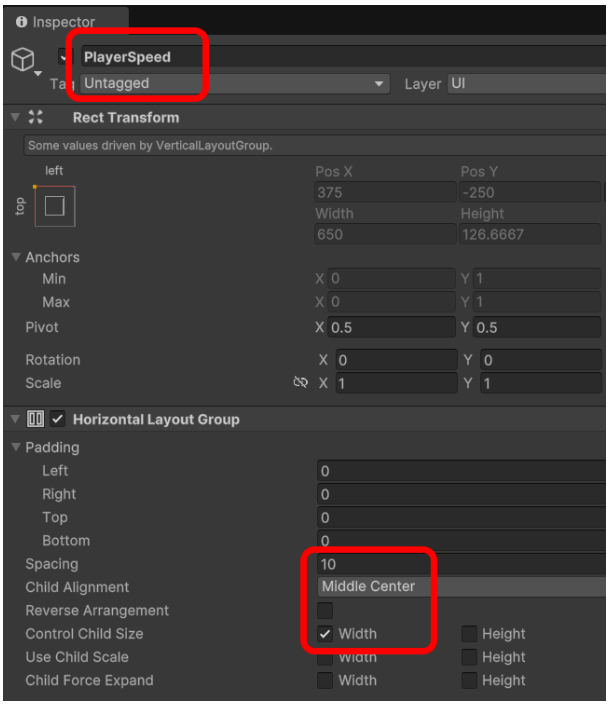
If your text size looks small, remember to enable auto-size and increase the max size till the text looks big enough!

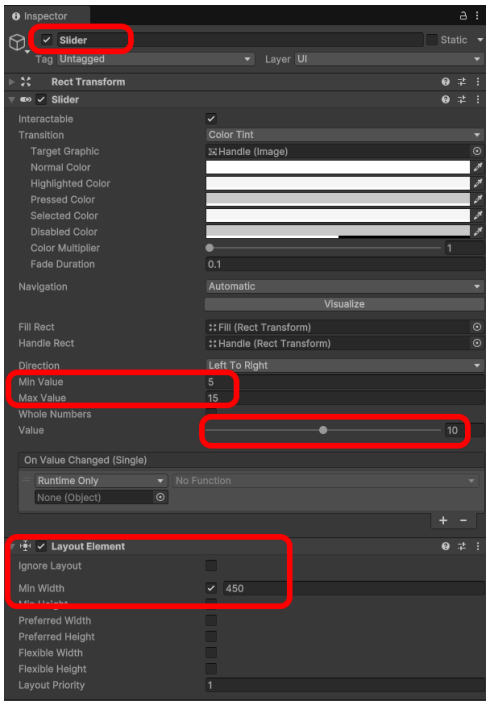


Next, under the PlayerSpeed, create a new text mesh pro item, and a `UI > Slider` GameObject. Change the text to say "Player Speed". On the Slider, change the Min Value to 5 and the Max Value to 15. This will set the range of speed for our player. Feel free to try more extreme ranges! Set the current value to 10. Now put a horizontal layout group on the PlayerSpeed object. Add just a spacing of 10, align it by middle center, and only enable the control child size width. This is because the sprites for the slider handle will get distorted if the height is also set. To have equal heights, more tuning will be required for the slider subcomponents, which is out of the scope of this guide. Finally, add a layout element to the slider GameObject, and set a min width of 450.









4.2. Implementing Logic

So we have this settings menu setup, but there's no way to actually enable and disable it, and the buttons don't do anything as of now.
Let's start with modifying the input manager to open the settings menu with the P key.

Note

Making a classic in-game settings menu has been covered in the lectures by Dr. Jalilvand and uploaded to the course YouTube channel in this playlist. Make sure to review the in-class material (especially week 6 both sessions 1 and 2) which covers more aspects on this topic as you go through this section.

```
public class InputManager : MonoBehaviour
{
    public UnityEvent<Vector2> OnMove = new();
    public UnityEvent OnJump = new();
    public UnityEvent OnDash = new();
    public UnityEvent OnSettingsMenu = new();

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.P))
        {
            OnSettingsMenu?.Invoke();
        }

        .
        .
        .
    }
}
```

Next, let's modify the GameManager to add the ability to open and close the settings menu.

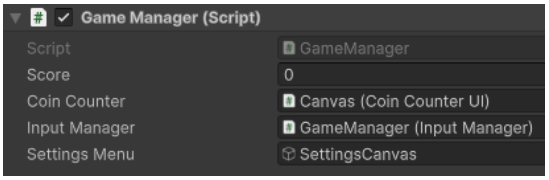
Hint

You could maintain a separate settings script, but the functionality is general enough for now to just be a part of the game manager. If more settings were to be added, a refactor could be considered.

```
using UnityEngine;

public class GameManager : SingletonMonoBehavior<GameManager>
{
    [SerializeField] private int score;
    [SerializeField] private CoinCounterUI coinCounter;
    [SerializeField] private InputManager inputManager;
    [SerializeField] private GameObject settingsMenu;

    private bool isSettingsMenuActive;

    // this creates a public getter for the bool
    // this way the variable is read only
```

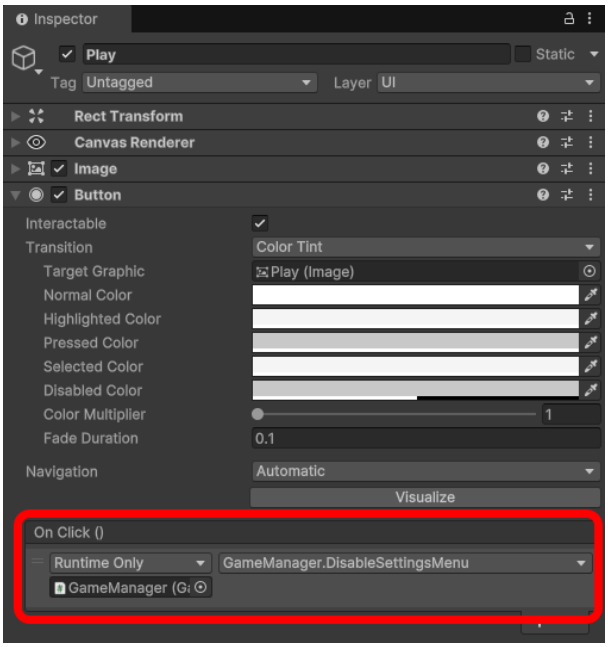
```
14      // without making it public
15      public bool IsSettingsMenuActive => isSettingsMenuActive;
16
17      protected override void Awake()
18      {
19          base.Awake();
20          Cursor.visible = false;
21          Cursor.lockState = CursorLockMode.Locked;
22          inputManager.OnSettingsMenu.AddListener(ToggleSettingsMenu);
23          // the game starts with the settings menu disabled
24          DisableSettingsMenu();
25      }
26
27      .
28      .
29      .
30
31      private void ToggleSettingsMenu()
32      {
33          if (isSettingsMenuActive) DisableSettingsMenu();
34          else EnableSettingsMenu();
35      }
36
37      private void EnableSettingsMenu()
38      {
39          Time.timeScale = 0f;
40          settingsMenu.SetActive(true);
41          Cursor.lockState = CursorLockMode.None;
42          Cursor.visible = true;
43          isSettingsMenuActive = true;
44      }
45
46      private void DisableSettingsMenu()
47      {
48          Time.timeScale = 1f;
49          settingsMenu.SetActive(false);
50          Cursor.lockState = CursorLockMode.Locked;
51          Cursor.visible = false;
52          isSettingsMenuActive = false;
53      }
54  }
```

Add the references for the game manager. The settings menu can directly reference the entire SettingsCanvas.



Press play to try it out.
This is good, but we want the buttons to also work as intended. We can implement the play button by simply making the `DisableSettingsMenu` function public, and then connect the button to that.

```
using UnityEngine;

public class GameManager : SingletonMonoBehavior<GameManager>
{
    .
    .
    .

    public void DisableSettingsMenu()
    {
        Time.timeScale = 1f;
        settingsMenu.SetActive(false);
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
        isSettingsMenuActive = false;
    }
}
```



Now the play button works as expected. To implement the quit button, we can write a QuitGame method in our Game Manager. Since we're using this functionality in editor,

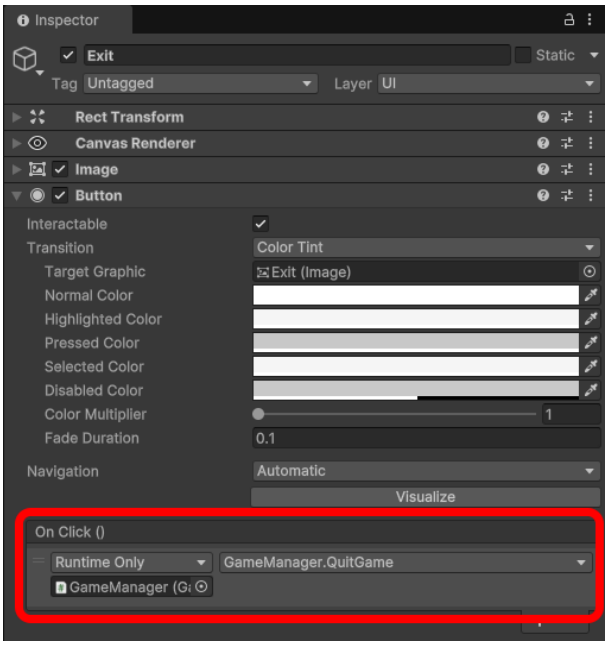
we can make use of "Preprocessor Directives" to exit play mode when we are using the unity editor, and exit the game if we run the game normally.

```
using UnityEngine;

public class GameManager : SingletonMonoBehavior<GameManager>
{
    .
    .
    .

    public void QuitGame()
    {
#if UNITY_EDITOR
        EditorApplication.isPlaying = false;
#else
        Application.Quit();
#endif
    }
}
```

Hint

Preprocessor Directives are a powerful tool for conditional compiling. It effectively tells the compiler to look at specific lines of code based on certain environment variables. In this case, the environment variable `UNITY_EDITOR` is true since you are playing this code in the editor. When the game is compiled and built, that variable will be false, and `Application.Quit()` will be executed instead. You can check out Unity's list of variables for preprocessor directives here.

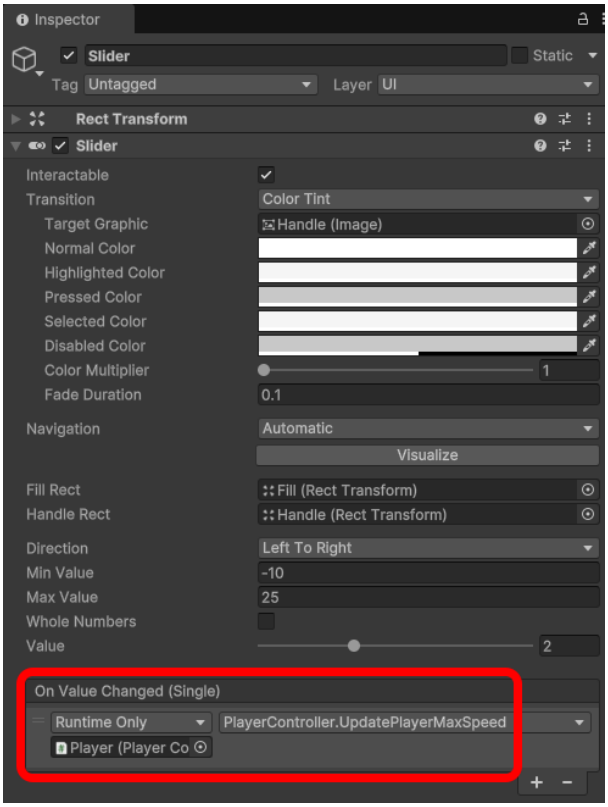
Connect this function to the Exit button.



Finally, let's modify the PlayerController to update the player max speed.

```
public class PlayerController : MonoBehaviour
{
    .
    .
    .

    public void UpdatePlayerMaxSpeed(float speed)
    {
        maxSpeed = speed;
    }
}
```

And add the reference to the slider component.



Press play to try it out.

If you've implemented everything correctly, your settings menu buttons should work as shown in the demo video!
There's one last change we can make. Currently, if the player is jumping and moving in a direction, and the game is paused midair, the player can press on the opposite side input to apply forces while the game is paused. We don't want the player's input to be read while the game is paused. Hence, we can update the `InputHander` script as follows.

```
public class InputManager : MonoBehaviour
{
    .
    .
```

```
5       .
6
7       private void Update()
8       {
9           if (Input.GetKeyDown(KeyCode.P))
10          {
11              OnSettingsMenu?.Invoke();
12          }
13
14          // we can access the game manager through the singleton instance
15          // and then access the public read-only bool
16          // which reflects the state of the game
17          if (GameManager.Instance.IsSettingsMenuActive) return;
18
19          .
20          .
21          .
22      }
23  }
```

Now the player's inputs will be discarded if the game is paused, except to pause or play the game!

Tip

Now would be a good time to make a commit. Something to the effect of "Implemented Settings Menu" should suffice.

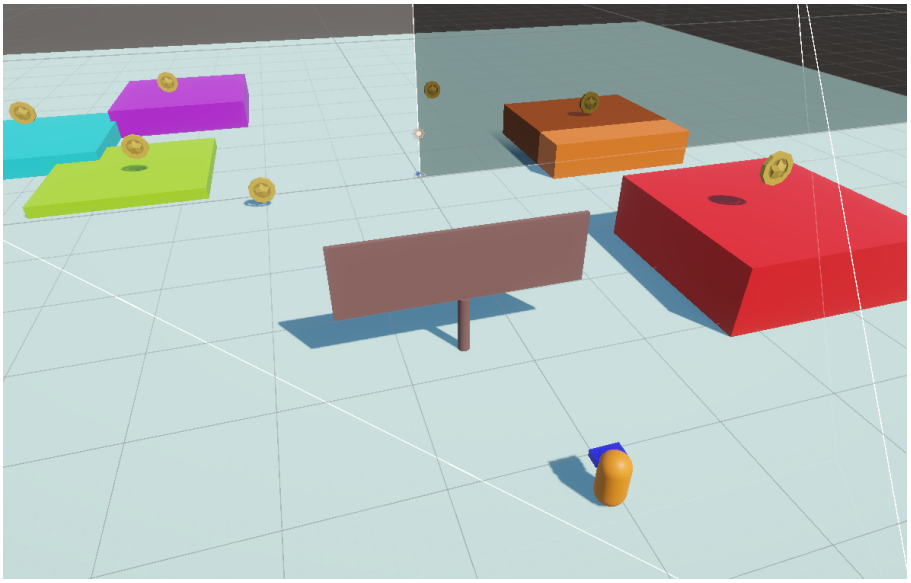
5. In-World Canvas

Unity Canvases can exist either on the screen, or as a panel in the real world. Let's create a tutorial board to convey to the player all the controls of the game.

Hint

You might have to shift back to 3D view for this section

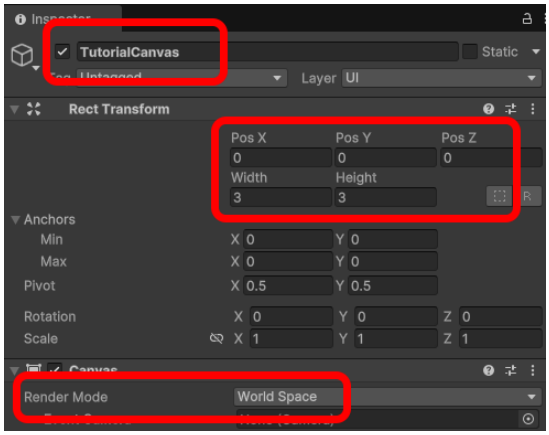
Start with making a Tutorial Board made of a cube and a cylinder and place it in front of the player. Modify it as you'd like.

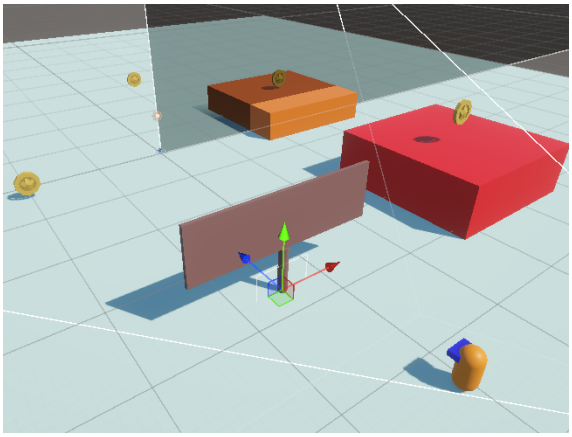


Hint

Feel free to use the `HotSwapColor` script to change the color of the board if you'd like!

Next, create a new canvas called TutorialCanvas. Under the Render component, set the Render Mode to World Space. Now, any UI elements you made under this canvas will not have any anchoring requirements, and behave the same as a regular game object. Set the position of the canvas to x = 0, y = 0, and z = 0, and reduce the width and the height of the object to something like 3 and 3. Now you should be able to see it in the Unity environment.

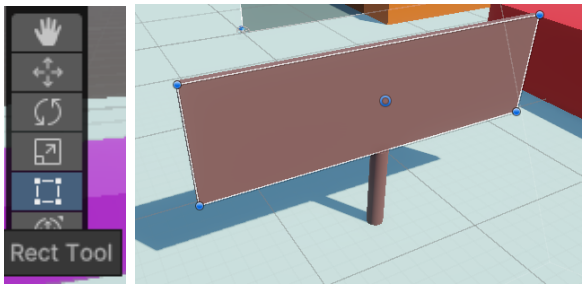




Hint

This is the third canvas we're adding. Remember to keep your UI GameObjects organized and separated from your physics and logic GameObjects!

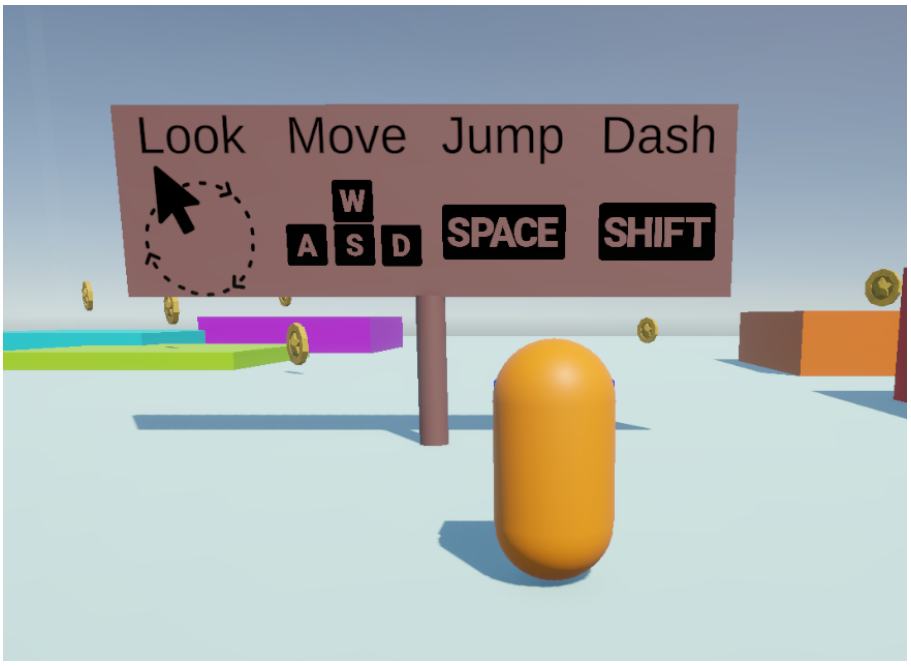
You can now position it on the board using the transform, and control the width and the height to properly stretch it across your tutorial board. You can also use the rect tool to properly stretch the canvas across the relevant area. **Remember to use the rect tool and not the scale tool when working with canvases!** The scale tool will distort items within it, while the rect tool only defines the area or bounds of the UI canvas.



Hint

Make sure the canvas is not inside the cube, else you won't be able to see the UI items on it!

Now you can place UI elements like texts, images, etc. on it as normal! Use what you have learned from the guide so far, and implement a tutorial board with the provided assets from earlier as follows. You can use a combination of horizontal and vertical layout groups, or simply place the icons and texts manually. Both are fine as long as the final result looks appropriate.



Tip

Once you have implemented the board, you can make a git commit. Something to the effect of "Created Tutorial Board" should suffice.

6. Conclusion

And that's it for this guide! This should give you a good overview of how to use the different UI elements, and connect them with custom C# logic. Unity UI can get pretty unwieldy in the inspector, so make sure to properly organize all your GameObject and contain different parts of the interface in empty containers. Same as with the other guides, please make sure to provide a readme with a video demo showcasing all this functionality.