Silver Belt Ninja Guide
Activity 16: Food Frenzy
Part 1

Activity 16

Food Frenzy Part 1

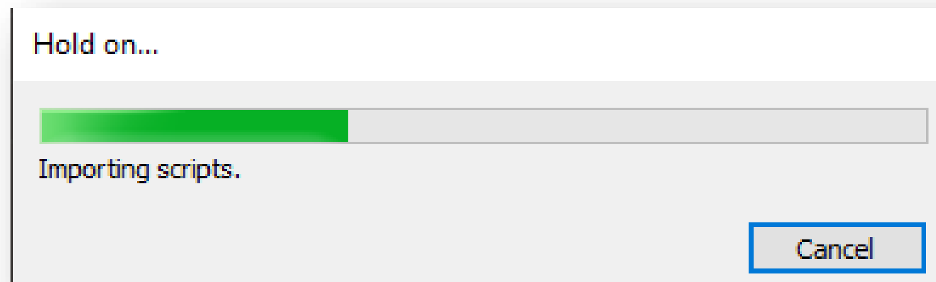
Well done! You've made it to the last game in Silver Belt. You've learned some pretty complicated game building skills, and you're finally on the last challenge to earn your belt.

Have you ever heard of a "Match-3" game? The objective is to match three or more of the objects in the game in a row. Doing so will remove the matched objects and give you points based on how many objects were matched. It's a simple type of game, but our focus is going to be on the interface that gives players information about the game state and what they can do.

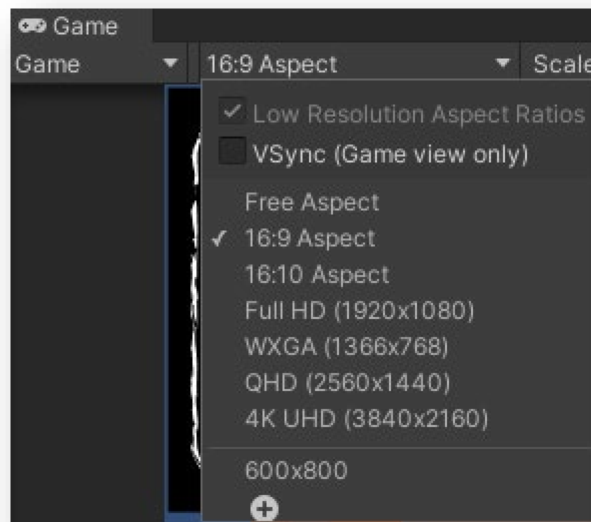
Objectives: Build an interface that guides the players through a game with multiple levels, game over conditions and states, and even animated interface elements.

1 Start a new Unity Project and name it *YOUR INITIALS – Food Frenzy*. Select **2D core**.

2 We've created a starter pack to give you a head start! To use it, import the **Activity 16 – FoodFrenzy Part 1.unitypackage** file by going to **Assets > Import Package > Custom Package**, clicking **All**, and then **Import**.

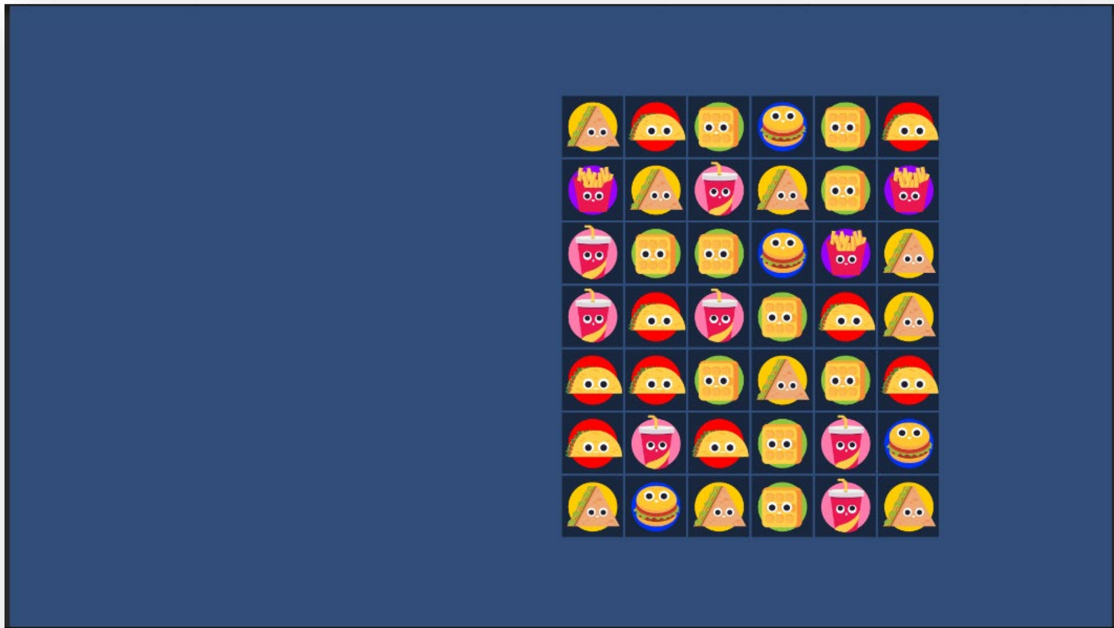


Set the aspect ratio to 16:9.



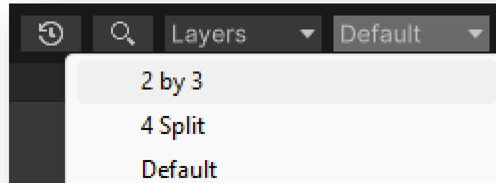
3 Select the **LevelOne** scene. **Play** this scene.

Play Instructions: Make a match by dragging icons from one cell in the grid to another. You can only drag an icon if the new position makes a row or column of three.

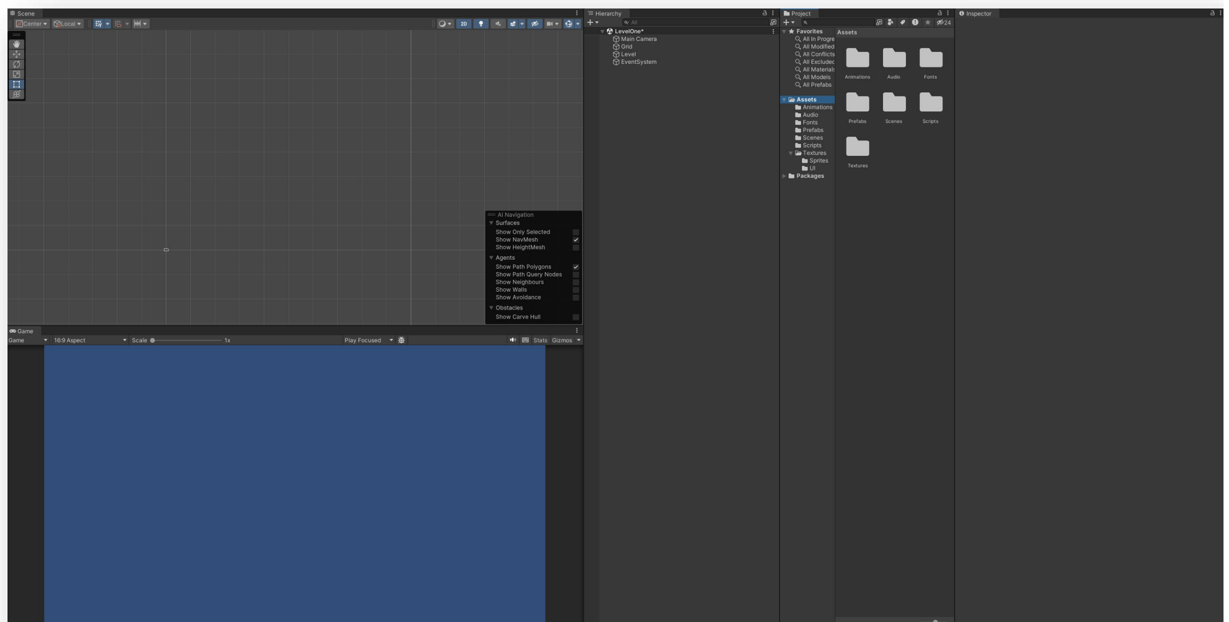


Most of the game is complete. All that's left to do is the User Interface (UI) where we update the player's score and keep track of the goals for the game.

4 Because this project will involve working a lot with UI, a different window layout will help us during this project. In the top right corner of Unity, switch the view from **Default** to **2 by 3**.

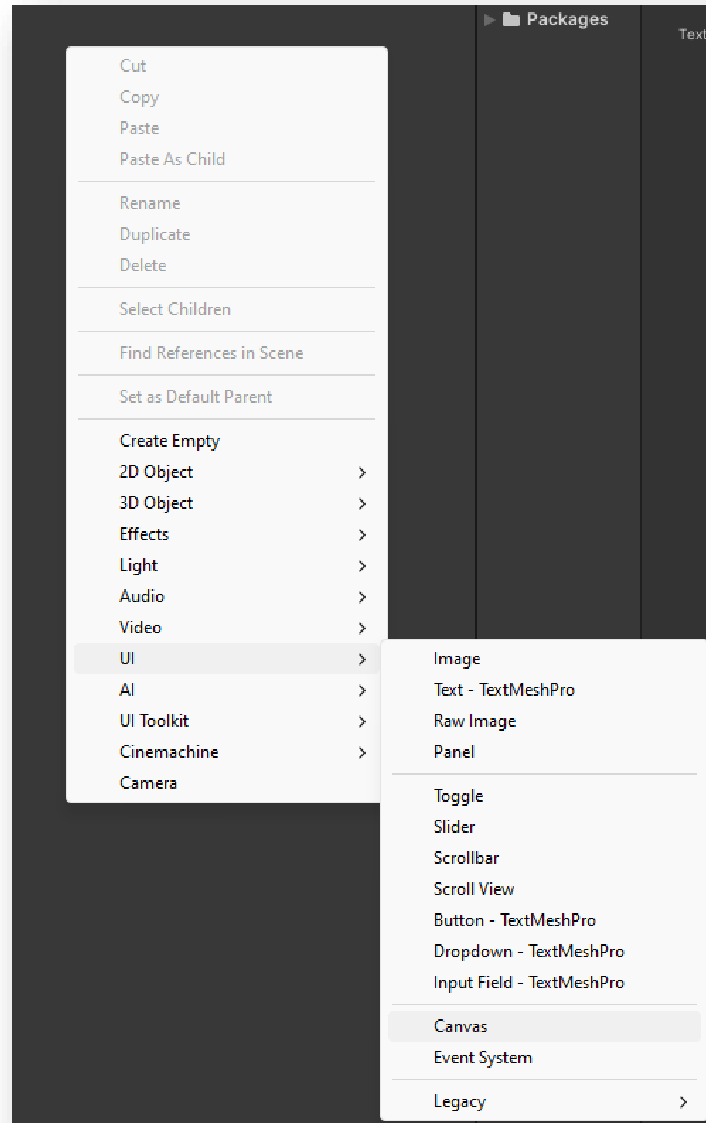


The view should now look like the below image. This makes it much easier to see our UI changes!



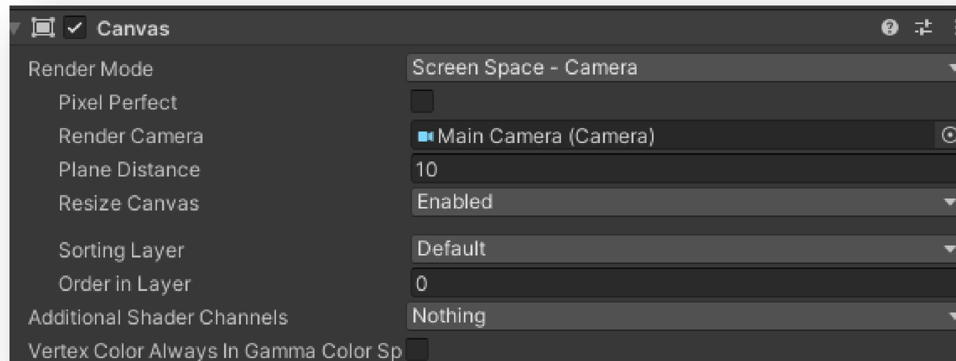
CODE NINJAS®

5 The **User Interface (UI)** uses a **GameObject** called the **Canvas**. Add a new **GameObject** by right clicking in a blank space on the hierarchy, then select the **UI** submenu, then **Canvas** from there.

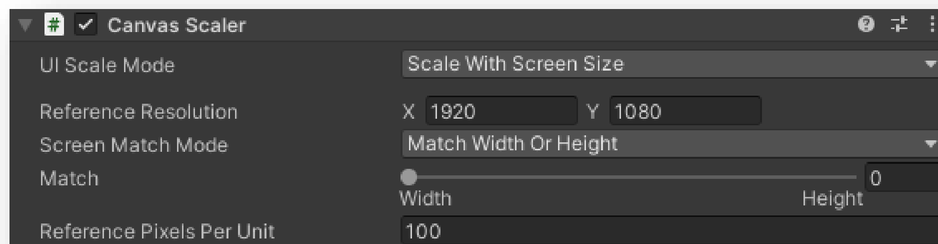


6

With the **Canvas** selected, find the **Canvas Scaler** component in the **Inspector**. Change the **Render Mode** to **Screen Space – Camera**. Add the **Main Camera** as the **Render Camera**, and set **Plane Distance** to 10.

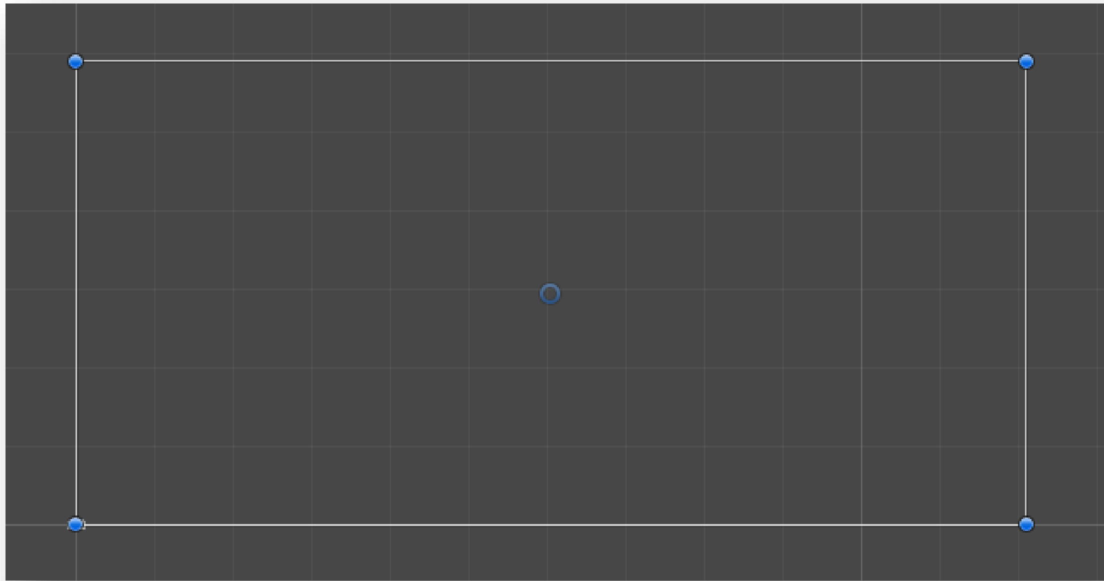


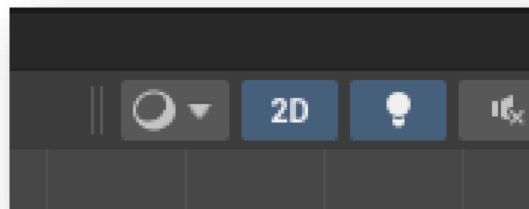
Set the **UI Scale Mode** to "**Scale With Screen Size"**. The **Reference Resolution** should be 1920 x 1080. This makes sure that the UI looks similar, even on different screen sizes!



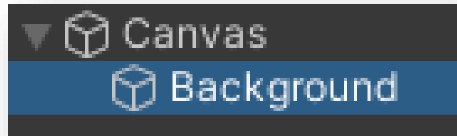
7 Double click on the **Canvas** in the **Hierarchy.** In the **Scene** window, the **Canvas** is much larger than the game itself. This is normal.

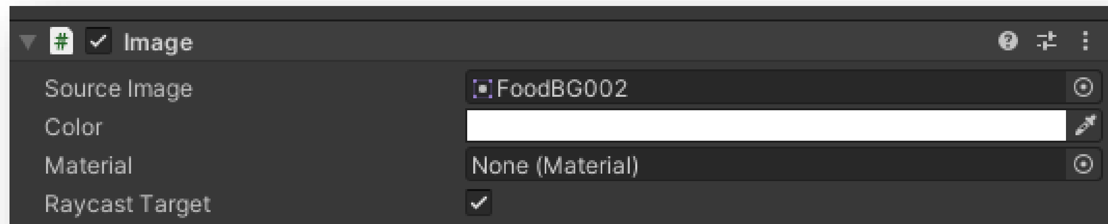
Remember, you can only edit in the **Scene** view and not the **Game** view. Since we are building a 2D game, don't forget to **change the view to 2D** when you're dragging and editing parts in the **Scene**.

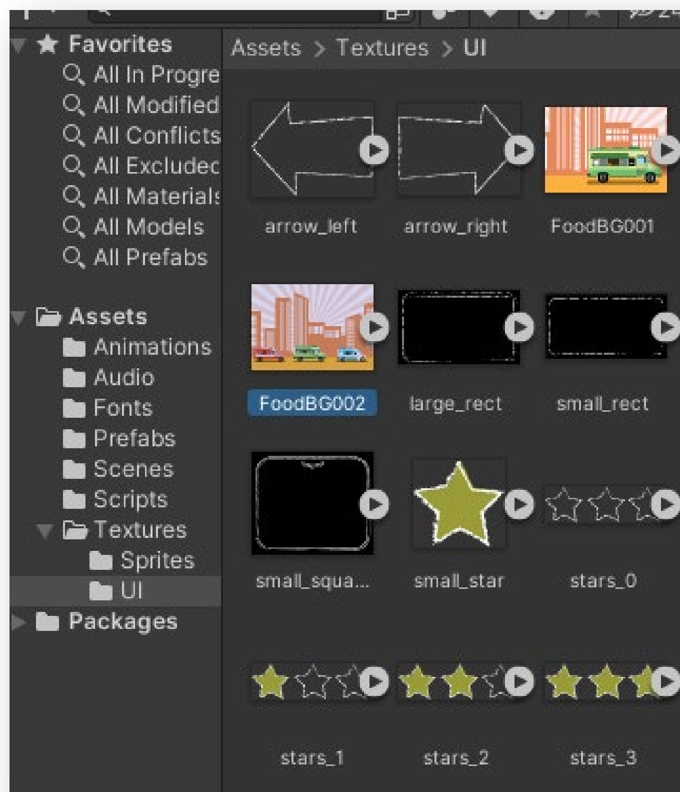




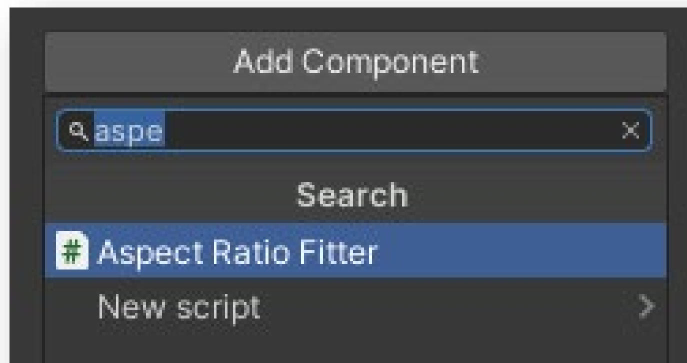
8 We'll start by adding a background image for the game. Add a **UI > Image** object and call it "**Background**". With the **Background** object selected in the **Hierarchy**, change the **Source Image** in the **Image** component to the **FoodBG0002** sprite available in the **Assets > Textures > UI** folder.



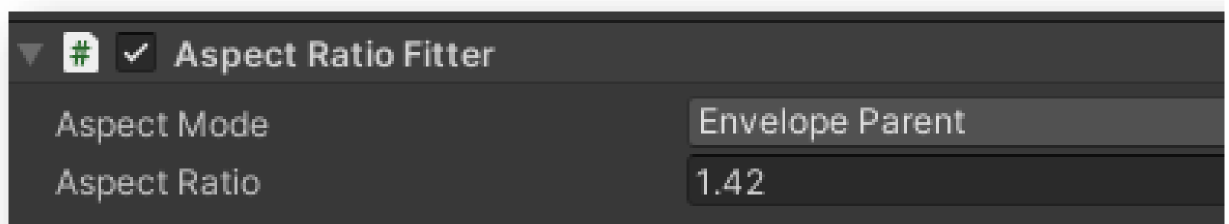




9 The background doesn't fill the whole screen! To make sure that the background fits, add an **Aspect Ratio Filler** component.

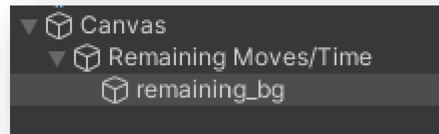


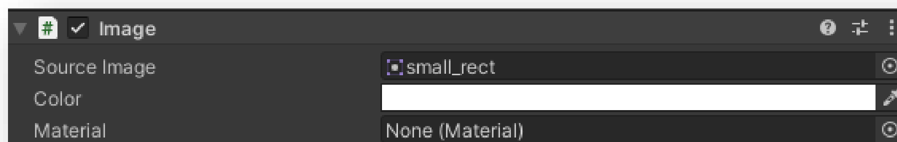
Set the mode to **Envelope Parent** and the ratio to **1.42**. The **Aspect Ratio Filler** component will automatically size the image to fill the screen!



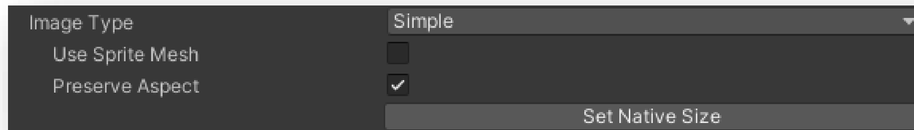
10 Next, we'll put our interface beside the game. There will be three parts – the **remaining moves**, the **target score** (or goal), and the **current score** along with 0 to 3 stars depending on how well the player has done.

Add a **UI > Image** object and call it "**Remaining Moves/Time**". With the **remaining_bg** object selected in the **Hierarchy**, change the **Source Image** in the **Image** component to the **small_rect** sprite available in the **Assets > Textures > UI** folder.

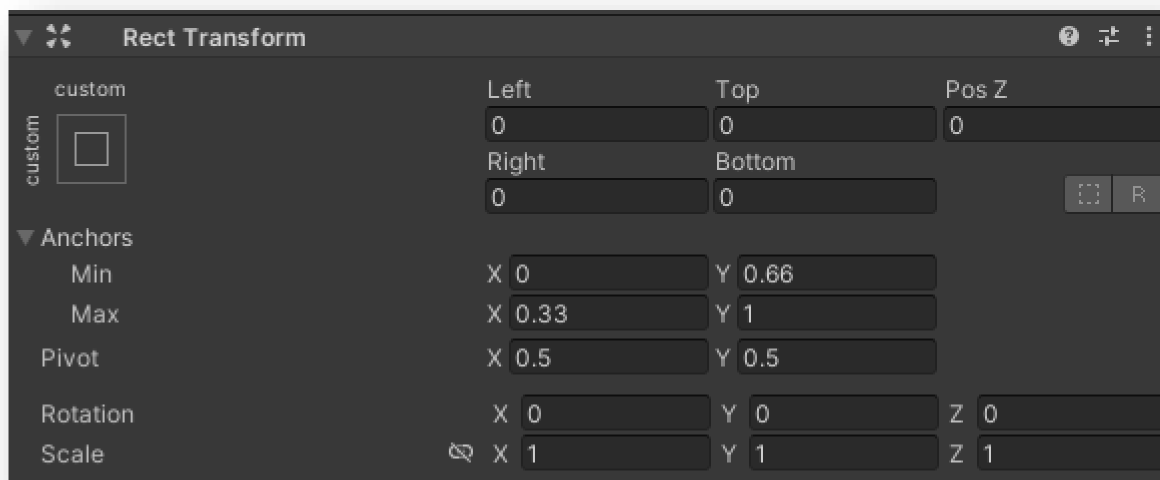




11 In the **Image** component, click the **Set Native Size** button to ensure that the image has the proper height to width ratio. Select the checkbox **Preserve Aspect** to lock in this ratio.



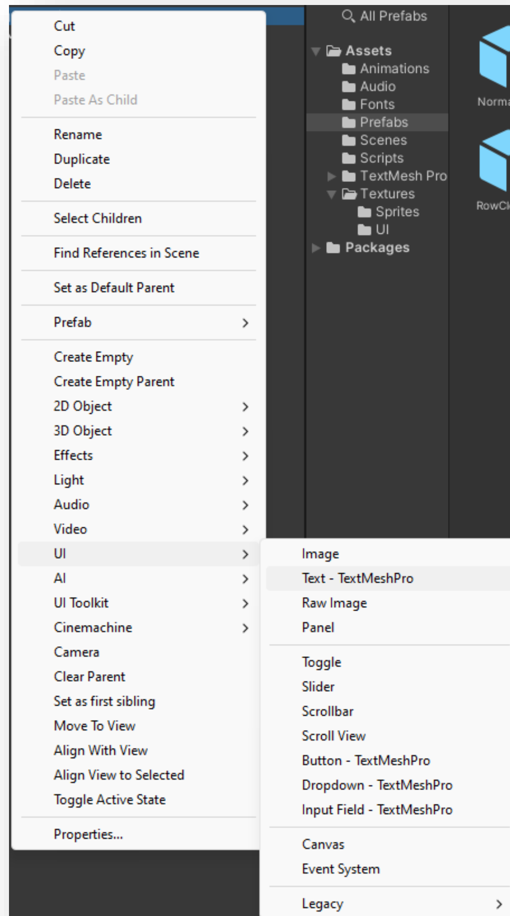
To position the background, set the transform values to match the below values.

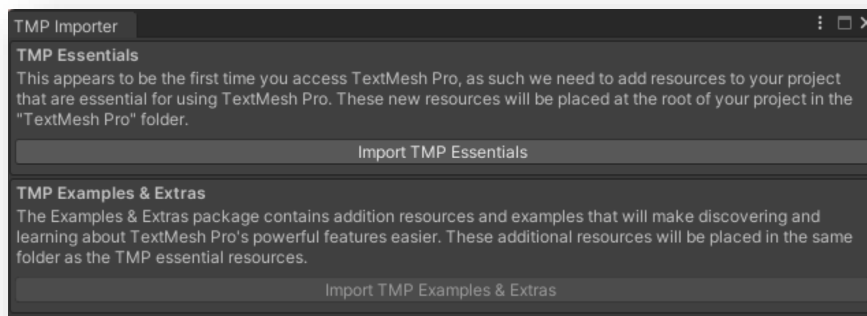


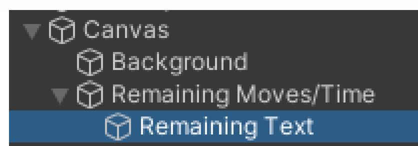
The most important piece is the anchor position. These values makes sure that the box will take up a maximum of 1/3 the width (minimum of 0, max 0.33) and maximum 1/3 the heigh (minimum position of 0.66, maximum 1, which is a difference of 1/3).

12 Play the game to make sure that the UI element you just added looks like it does below. Try scaling the screen to different sizes to see the effect of the anchor settings!

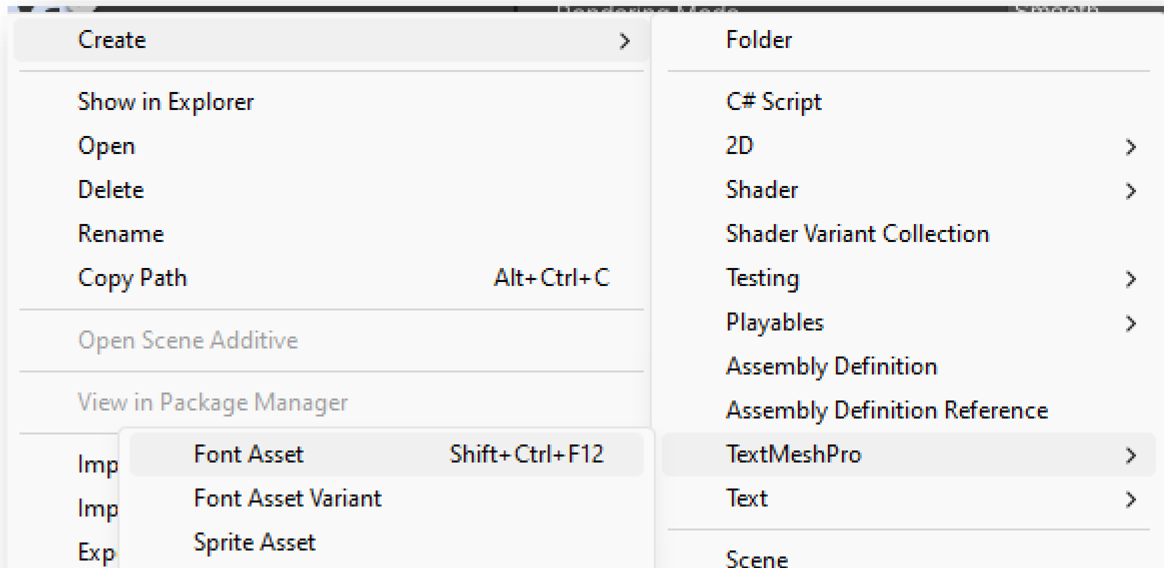
13 The next step is to add text. Add a **UI > Text** object to the **Remaining Moves/Time** object. To do this, right-click on **Remaining Moves/Time**, and navigate to **UI > Text - TextMeshPro**. Change the name to "**Remaining Text**".



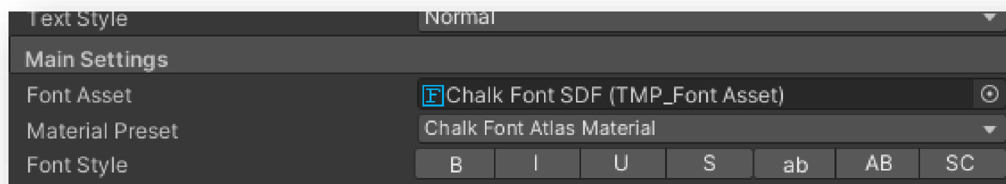




14 The default **font** of Arial is fine, but let's spice up the game with a custom font! Navigate to the Fonts folder and find the **Chalk Font**. This is just a regular font file that was imported into Unity. Right click on the font and navigate to **Create > TextMeshPro > FontAsset** to convert this font for **TextMeshPro**.



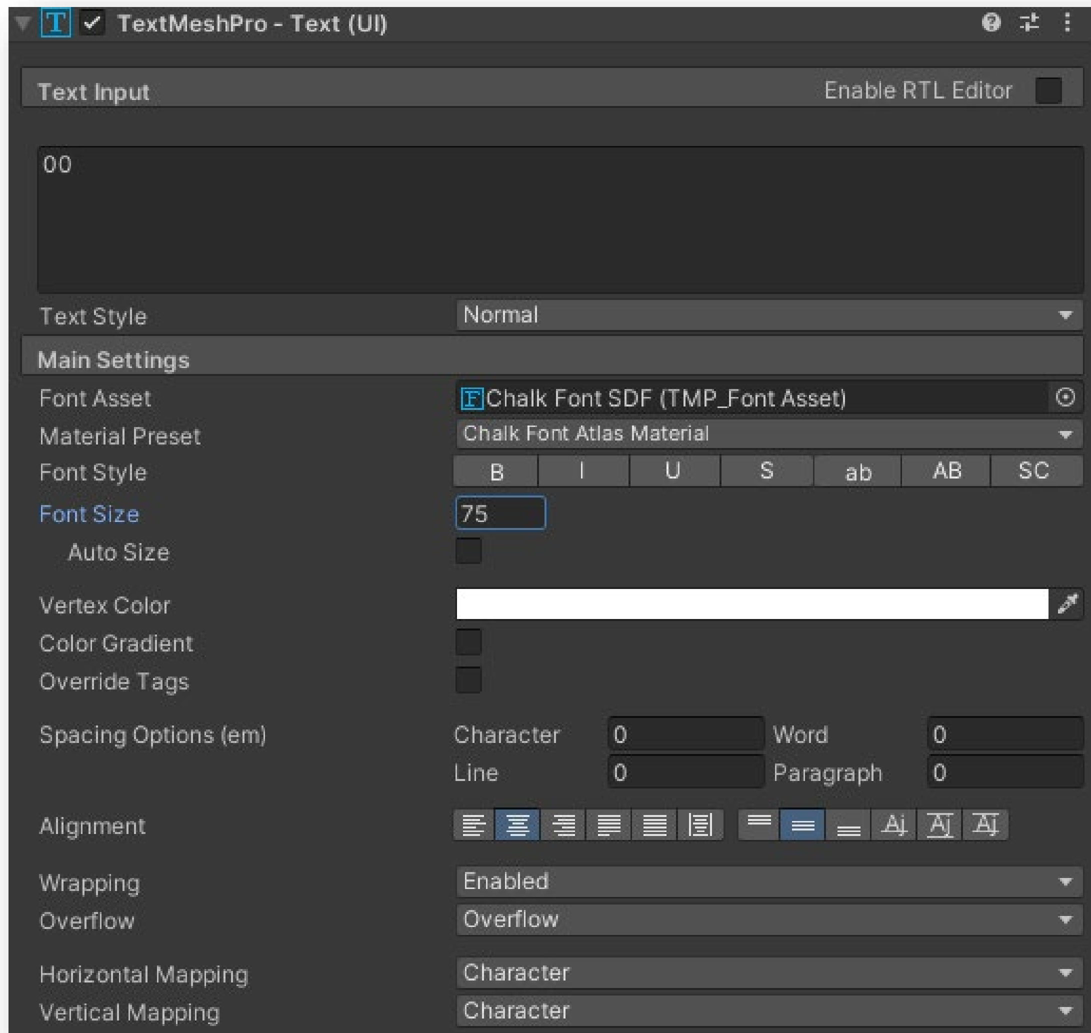
Reselect the Remaining Text object. In the Font Asset setting, use the new Chalk Font asset you created.



Change the **font color** to white and change the **text** to "00" or any other 2-digit number. We'll be changing the text in code, so we just want to get an idea of how much space we'll need.

Adjust the **size** as necessary and change the **Paragraph > Alignment** to center.

You can see in our example that we've changed the font to 75. You can customize it however you want by playing around with the text components. The object itself might need to have its size adjusted to account for the size of the font. Below is an example of the settings.

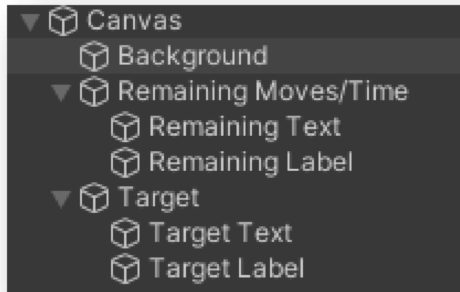


15 **Duplicate** this text object and rename it "**Remaining Label**". This will be a label for the number that we'll be putting in this object.

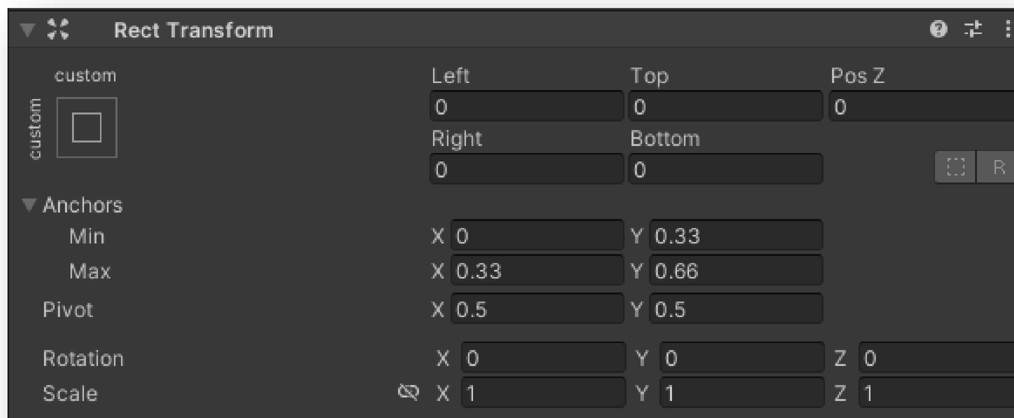
Change the **text** to "**Remaining Moves**" and decrease its size so that it fits beneath the larger number.

As you can see in the image below, we've roughly centered both text objects.

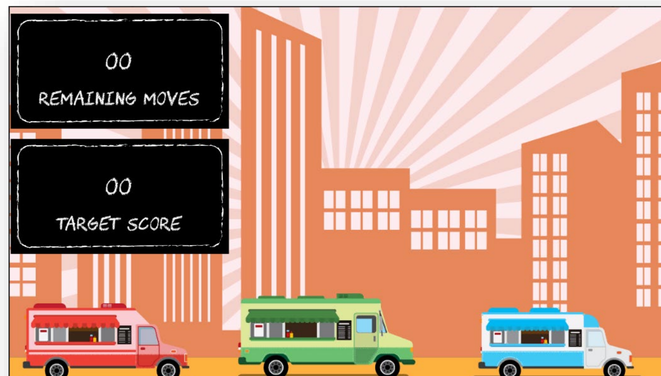
16 Make sure that you have the original **Remaining Moves/Time** object selected, then make a **duplicate** of it. Change the name of this new object to "**Target**" and rename the image and text objects inside it to "**Target Text**" and "**Target Label**".



Change the transform of the Target object to take up the next third of the screen height, using the settings below.

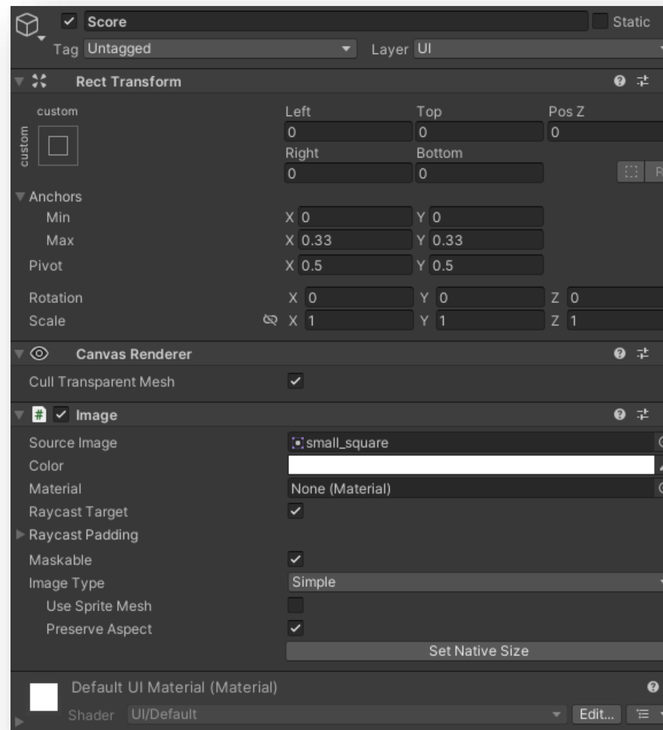


Select **Target Label** and change the **Text** component so that the **text** is now "**Target Score**".



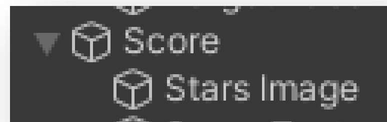
17 The next step is to have an object for the score. This will be different from the first two. Create a new **image object** in the **Canvas** and change the name to "**Score**".

In the **Inspector**, change the **Source Image** to "**small square**". Adjust this so that it fits underneath the first two UI objects.



18 Add another **UI > Image** to the **Score** object. Name it "**Stars Image**" and change the **Source Image** to **stars_0**. Adjust the **stars0** object and the background object until they look how you want them to.

Here's a possible way to set it up:



Stars Image
Static
Tag Untagged
Layer UI
Rect Transform
center
middle
Pos X 0
Pos Y 25
Pos Z 0
Width 350
Height 135.4545
Anchors
Min X 0.5 Y 0.5
Max X 0.5 Y 0.5
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Canvas Renderer
Cull Transparent Mesh
Image
Source Image stars_0
Color
Material None (Material)
Raycast Target
Raycast Padding
Maskable
Image Type Simple
Use Sprite Mesh
Preserve Aspect
Set Native Size
Default UI Material (Material)
Shader UI/Default
Edit...

19 The score will appear beneath the stars. Inside the **Score** object, add a **UI > Text - TextMeshPro** object. Call it "**Score Text**".

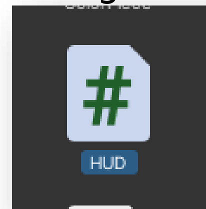
Change the **color** to white, **font asset** to **Chalk**, and change the **text** to "00000" or a similar 5-digit number. Adjust the **font size and position** of the text object so that it is centered under the stars. Make sure that the edges of the text box fit inside the remaining space in the background. If you do not see your text, it might be because you forgot to adjust the **width** and **height**. Go ahead and change the values until you can see your text.

20 What happens if the player's Score is a number that's too big to fit in the space?

Unity has a fix for that. Inside the **TextMeshPro - Text** component in the **Inspector**, there is a check box for **Auto Size**. Make sure it is checked. Set the minimum size to the smallest size you think will fit, and the maximum size you think will fit. Play around with these numbers and the **score_text** positioning and height/width until you like how it looks. When playing the game, Unity will automatically adjust the font size between the min and the max so that the score always fits the available area.

Make sure that these three elements fit within the boundaries for the Canvas. If necessary, use the shift key to select all three of the **Canvas elements** and adjust them so that they fit within the **canvas**.

21 Inside the **Scripts** folder, create a **new C# script** and name it "**HUD**". "HUD" is short for **Heads Up Display**, meaning the information presented is part of the game window.



22 Double-click the **HUD** script to open it. We'll need to add a **directive**, so Unity knows how to handle the UI. At the top of the script, add:

```
using UnityEngine.UI;
using TMPro;
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
```

23 Inside the **class** we'll set up the **variables** that we need. We will need variables for the **Level** and **object**. There will also be a **GameOver** object, but we'll come back to this script later to add that.

```
public Level level;
```

```
public Level level;
```

24 Then we need variables for all the text objects in our Canvas:

```
public Text remainingText;
public Text remainingLabel;
public Text targetText;
public Text targetLabel;
public Text scoreText;
```

```
public TMP_Text remainingText;
public TMP_Text remainingLabel;
public TMP_Text targetText;
public TMP_Text targetLabel;
public TMP_Text scoreText;
public Image starsImage;
```

25 We'll need a variable to store the image object for the stars. We also need an **array** variable to store the four different star images in the **Score** object:

```
public Image starsImage;
public Sprite[] stars;
```

Remember that the **brackets** [] let the program know that this is an **array**.

```
public Image starsImage;
public Sprite[] stars;
```

26 Finally, we'll need 2 **private variables** to identify which star image we want to show, and for when the game is over.

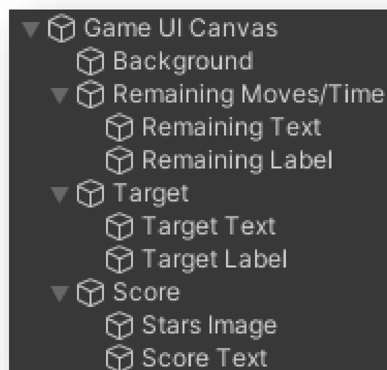
```
private int starIndex;
private bool isGameOver;
```

```
6   public class HUD : MonoBehaviour
7   {
8       public Level level;
9
10      public Text remainingText;
11      public Text remainingSubtext;
12      public Text targetText;
13      public Text targetSubtext;
14      public Text scoreText;
15
16      public Image[] stars;
17
18      private int starIndex;
19      private bool isGameOver;
20  }
```

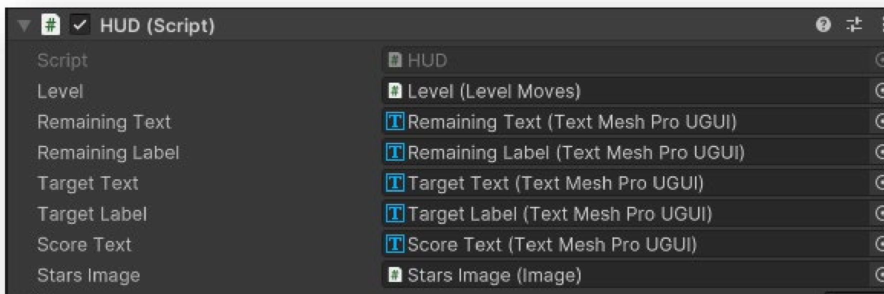
27 Finally, since there is only one HUD in the game, this is another good opportunity for us to use a **singleton**. Remember, that a **singleton** creates a static instance of the script that can easily be referenced from other scripts. Create the singleton variable using the code below.

```
public static HUD instance;
```

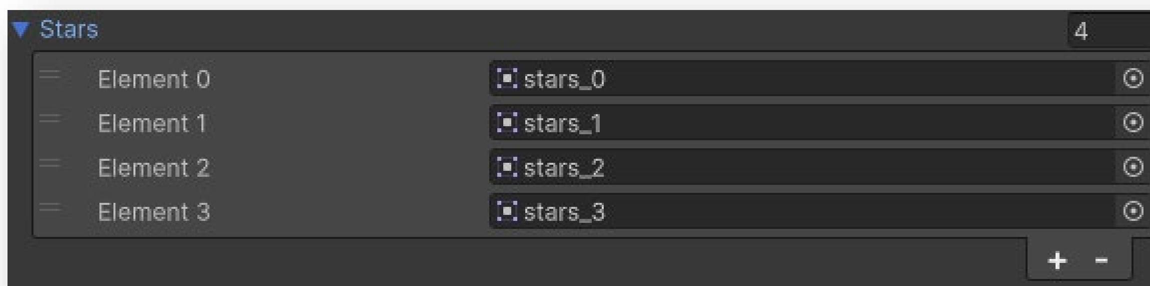
28 **Save** the script and go back to Unity. Add the **HUD** script to the **Canvas** object. Rename the **Canvas** object to "**Game UI Canvas**".



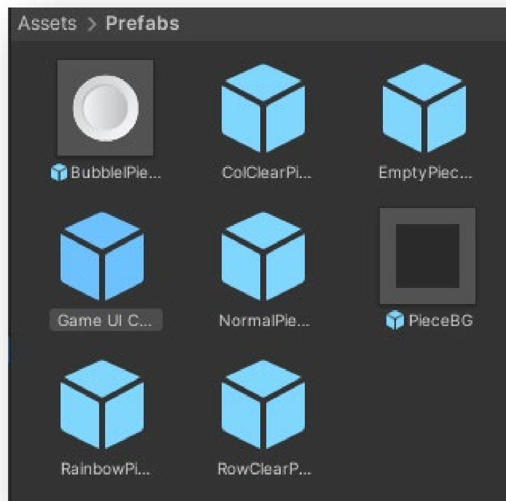
29 In the **Inspector** for the **Game UI Canvas**, link all the variable objects:



30 Click the **arrow** next to **Stars** in the **HUD (Script)** component in the **Inspector**. Change the number to 4 to make the **Elements** appear. Link the **stars0 through stars3** images from the **Textures > UI** folder in the Asset window.

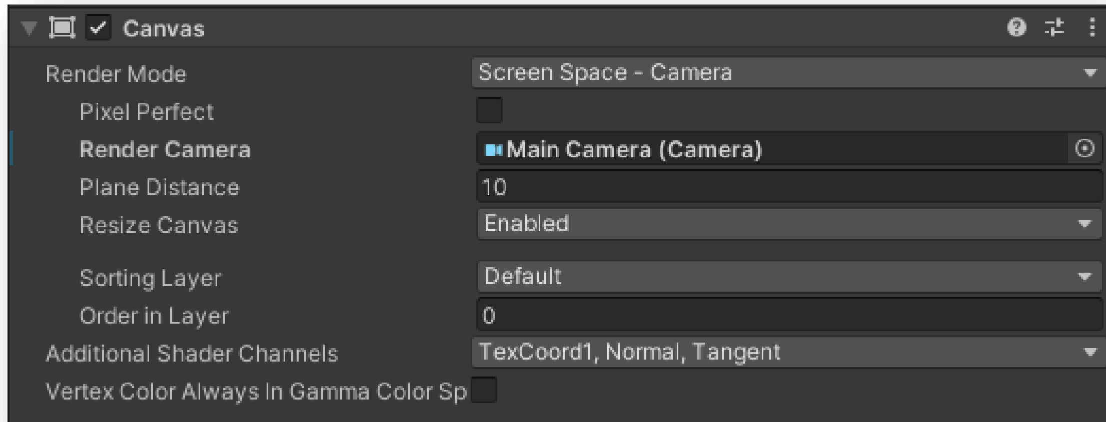


31 We'll want to use the UI for all the levels in this game. Drag **Game UI Canvas** into the **Prefabs** asset folder. **Save** this scene.



Open the **LevelTwo** scene and add the **Game UI Canvas** prefab to the **Hierarchy**. Repeat this for **LevelThree**.

Make sure to set the **Main Camera** as the **Render Camera** in the canvas, and that the children of each group are ordered correctly, otherwise you might end up in a situation where the background is covering up the text.



If you're prompted to save while doing this, go ahead and **save**.

32 The canvas requires an **Event System.** If you don't see an **EventSystem** in the **Hierarchy** like in the images in the last step, add **UI > EventSystem**. Make sure everything has been **saved**.

33 Open the **HUD** script. We'll use the **Awake** function to initialize the singleton and stars. Start by checking if the singleton exists, and if not, assign the current script.

Since we're updating the stars more than once in our game, we'll use a function for it. Add this to the **Start** function:

```
if (!instance)
{
    instance = this;
    UpdateStars();
}
```

```
void Awake()
{
    if (!instance)
    {
        instance = this;
        UpdateStars();
    }
}
```

34 Next, let's write that function. We'll create a function that sets the stars image to the image matching the index. *Outside* of both the **Start** and **Update** functions, write a new function: `public void UpdateStars()`

Inside that, set the sprite of the image to the index of the stars array.

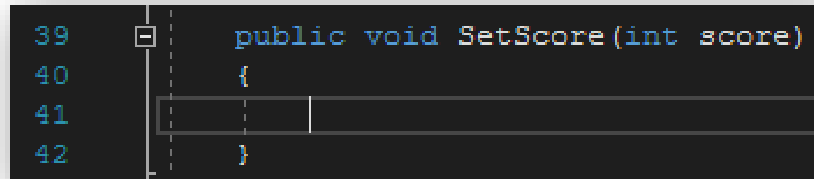
```
starsImage.sprite = stars[starIndex];
```

```
public void UpdateStars()
{
    starsImage.sprite = stars[starIndex];
}
```

35 Save and do a playtest. When the game begins, this loop will make only the **star0** image visible. Experiment by changing the **starIndex** variable to a different number before calling **UpdateStars()**.

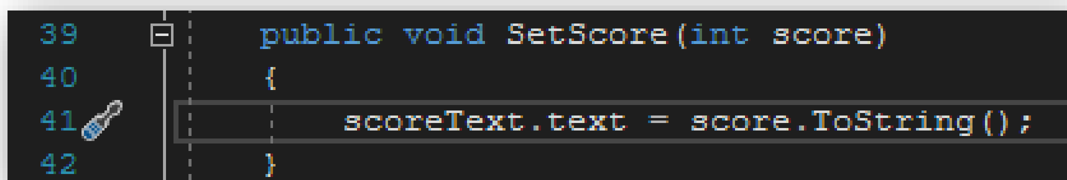
36 The next step is to create a new function for SetScore:

```
public void SetScore(int score) { }
```



37 This function takes the parameter sent to it, **score**, and uses that integer in the rest of the function. First, we'll apply the score to the **scoreText** object:

```
scoreText.text = score.ToString();
```



`ToString` takes the **integer** and turns it into **text** for the **Text** object.

38 Then we need to compare the score to the minimum number defined in the Level object and display the right star:

```
if (score >= level.score3Star)
{
    starIndex = 3;
}
else if (score >= level.score2Star)
{
    starIndex = 2;
}
else if (score >= level.score1Star)
{
    starIndex = 1;
}

UpdateStars();
```

```
if (score >= level.score3Star)
{
    starIndex = 3;
}
else if (score >= level.score2Star)
{
    starIndex = 2;
}
else if (score >= level.score1Star)
{
    starIndex = 1;
}

UpdateStars();
```

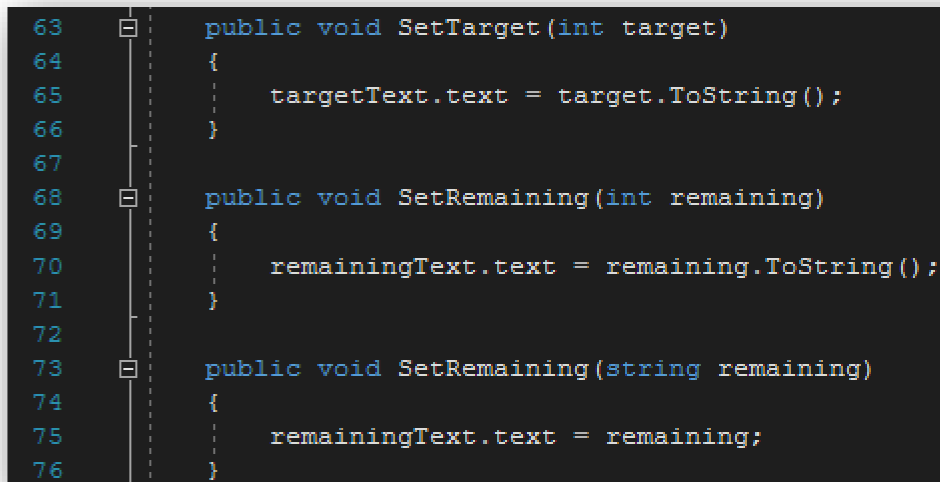
The `UpdateStars()` function is called after it has been decided which **star** image is **visible**, and that value is set to the **starIndex** variable.

39 Then we need **functions** to set the information for the remaining **moves, time**, and **target**. We'll need three functions:

```
public void SetTarget(int target)
{
    targetText.text = target.ToString();
}

public void SetRemaining(int remaining)
{
     remainingText.text = remaining.ToString();
}

public void SetRemaining(string remaining)
{
     remainingText.text = remaining;
}
```



We might be sending an **integer** or a **string** to **SetRemaining**. By setting up the **parameters** for either possibility, we are ready for either situation. Remember, the **integer** needs to be converted to a **string**; the **text** object can display a **string** without having to convert it.

40

The last thing to do is modify the **subtext** objects, depending on which of the three levels is being played. Inside a `SetLevelType` function, we'll be taking the `Level.LevelType type` parameter, and applying a **switch/case** statement to adjust **text** objects accordingly.

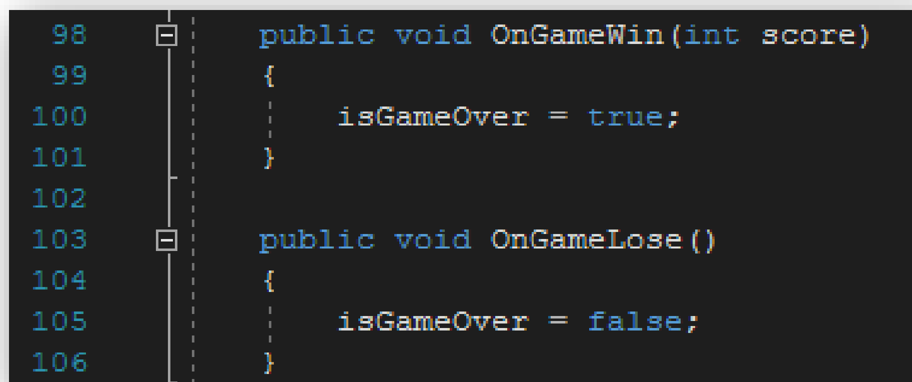
```
public void SetLevelType(Level.LevelType type)
{
     switch (type)
     {
         case Level.LevelType.MOVES:
               remainingLabel.text = "moves remaining";
               targetLabel.text = "target score";
               break;
         case Level.LevelType.OBSTACLE:
               remainingLabel.text = "moves remaining";
               targetLabel.text = "dishes remaining";
               break;
         case Level.LevelType.TIMER:
               remainingLabel.text = "time remaining";
               targetLabel.text = "target score";
               break;
     }
}
```


```
public void SetLevelType(Level.LevelType type)
{
    switch (type)
    {
        case Level.LevelType.MOVES:
            remainingLabel.text = "moves remaining";
            targetLabel.text = "target score";
            break;
        case Level.LevelType.OBSTACLE:
            remainingLabel.text = "moves remaining";
            targetLabel.text = "dishes remaining";
            break;
        case Level.LevelType.TIMER:
            remainingLabel.text = "time remaining";
            targetLabel.text = "target score";
            break;
    }
}
```

We're using a **switch** statement to take the parameter of the **level type** and setting the appropriate **subtext** for the **target** and **remaining** objects.
Each level can only be *one* of these types. The way the game is set up is that Level One contains the amount of moves and the target score. Level Two contains dishes as obstacles, so it shows moves remaining and dishes remaining. Level Three has a timer, hence it shows time remaining and target score.

41 Finally, we need to add functions for `OnGameWin` and `OnGameLose`. In both, we will simply change the **isGameOver** variable to true:

```
public void OnGameWin(int score)
{
     isGameOver = true;
}
public void OnGameLose()
{
     isGameOver = true;
}
```


```
98   public void OnGameWin(int score)
99   {
100      isGameOver = true;
101  }
102
103  public void OnGameLose()
104  {
105      isGameOver = false;
106  }
```

Currently, these functions don't do much since there's nothing calling them.

42 The **Level** object takes care of the game functions such as **score** and so on. In the `Start()` function, add this:

```
HUD.instance.SetScore(currentScore);
```

```
private void Start()
{
    HUD.instance.SetScore(currentScore);
}
```

Notice that this first finds the static HUD instance, then sends the **currentScore** as a parameter to the **SetScore** function in the HUD. The **SetScore** function can then update the score.

43 For the **GameWin** and **GameLose** results, we'll take the value of **didWin** and check to see if the grid has finished filling. If so, then we'll call **OnGameWin** and **OnGameLose** in the **HUD** script. Add this to `WaitForGridFill()`:

```
if (didWin && !grid.IsFilling)
{
    HUD.instance.OnGameWin(currentScore);
}
else
{
     HUD.instance.OnGameLose();
}
```

```
protected virtual IEnumerator WaitForGridFill()
{
    while (grid.IsFilling)
    {
        yield return 0;
    }

    if (didWin && !grid.IsFilling)
    {
        HUD.instance.OnGameWin(currentScore);
    }
    else
    {
        HUD.instance.OnGameLose();
    }
}
```

The first `if` statement checks if the **game has been won** and if the game is **done filling up the grid**. When **both** of these conditions are **true**, we call the `OnGameWin()` function. If those conditions are **not** met, we call the `OnGameLose()` function.

44 We'll also need to update the score when a piece is cleared. Find `OnPieceCleared` in the **Level** script and add this:

```
HUD.instance.SetScore(currentScore);
```

```
public virtual void OnPieceCleared(GamePiece piece)
{
    //Update Score
    currentScore += piece.score;
    HUD.instance.SetScore(currentScore);
}
```

45 We also need to update the **HUD** based on which level is being played. Let's start with **LevelMoves**. Open the **LevelMoves** script (in the **Assets > Scripts** folder) and add these lines to the `Start()` function:

```
HUD.instance.SetLevelType(type);
HUD.instance.SetScore(currentScore);
HUD.instance.SetTarget(targetScore);
HUD.instance.SetRemaining(numMoves);
```

```
void Start()
{
    type = LevelType.MOVES;
    HUD.instance.SetLevelType(type);
    HUD.instance.SetScore(currentScore);
    HUD.instance.SetTarget(targetScore);
    HUD.instance.SetRemaining(numMoves);
}
```

46 Each time the player makes a move, we need to update the remaining moves on the **HUD**. Find the `OnMove()` function and add this beneath `movesUsed++`:

```
HUD.instance.SetRemaining(numMoves - movesUsed);
```

```
movesUsed++;

HUD.instance.SetRemaining(numMoves - movesUsed);

if(numMoves - movesUsed == 0)
```

We need to update the number of moves remaining. So, every time we make a move, subtract the number of `movesUsed` from `numMoves`.

47 We will do the same for **LevelObstacles** and **LevelTimer**. Open the **LevelObstacles** script and add this to the `Start()` function:

```
HUD.instance.SetLevelType(type);
HUD.instance.SetScore(currentScore);
HUD.instance.SetTarget(numObstaclesLeft);
HUD.instance.SetRemaining(numMoves);
```

```
HUD.instance.SetLevelType(type);
HUD.instance.SetScore(currentScore);
HUD.instance.SetTarget(numObstaclesLeft);
HUD.instance.SetRemaining(numMoves);
```

48 Once again, each time the player makes a move, we need to update the remaining moves on the **HUD**. Find the `OnMove` function and add this beneath `movesUsed++`:

```
HUD.instance.SetRemaining(numMoves - movesUsed);
```

```
movesUsed++;

HUD.instance.SetRemaining(numMoves - movesUsed);

if (numMoves - movesUsed == 0 && numObstaclesLeft > 0)
```

This code is exactly the same as the code added to **LevelMoves**, don't get them confused with each other!

49 Once we make a move and there is a match, we also need to update the hud score. In the **OnPieceCleared** function add the following:

```
numObstaclesLeft--;

HUD.instance.SetTarget(numObstaclesLeft);

if (numObstaclesLeft == 0)
```

This way our final score will match in the HUD we created.

50 Then open the **LevelTimer** script. Since we're using time, things will be slightly different:

```
HUD.instance.SetLevelType(type);
HUD.instance.SetScore(currentScore);
HUD.instance.SetTarget(targetScore);
HUD.instance.SetRemaining(TimerString());
```

```
HUD.instance.SetLevelType(type);
HUD.instance.SetScore(currentScore);
HUD.instance.SetTarget(targetScore);
HUD.instance.SetRemaining(TimerString());
```

51 In the `Update()` function, add this after `timer += Time.deltaTime;`:

```
HUD.instance.SetRemaining(TimerString());
```

```
timer += Time.deltaTime;
HUD.instance.SetRemaining(TimerString());

if (timeInSeconds - timer <= 0)
```

52 Now **load** the **LevelOne** scene and **play** the game. Is everything showing up in the **HUD**? If not, you might need to adjust the remaining and target text boxes. Be sure to hit **apply** to update the prefab.

Test the levels and tinker with the **Game UI Canvas** elements!

1:27
TIME REMAINING
50000
TARGET SCORE
0