Click the Dots Game: Part 2 – Polish

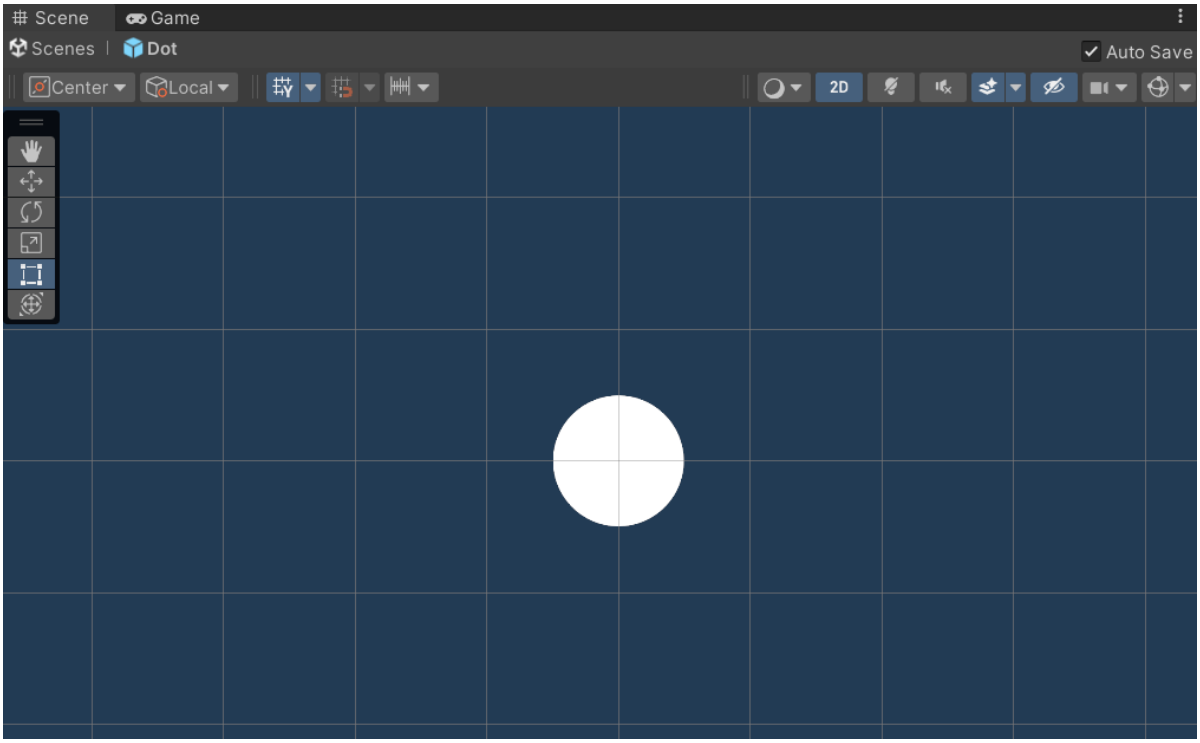
Lesson Goals

- The goal of this lesson is to add polish to our Click the Dots game.
- We will learn ways to add simple polish to our game.
- Learn how to export the games we make.

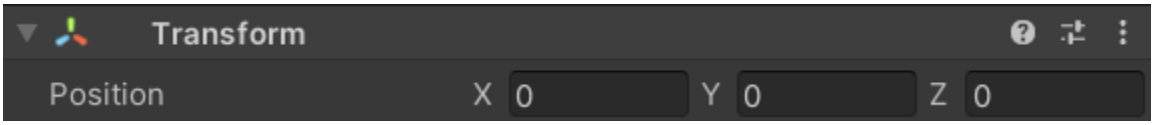
Dot Class Script

1. To start off our lesson, let's create a new Script called **Dot**. This script will hold information about the dot such as its point value, color, scale and a few other things. We will also be using this script to make the dot have a simple grow and shrink animation.

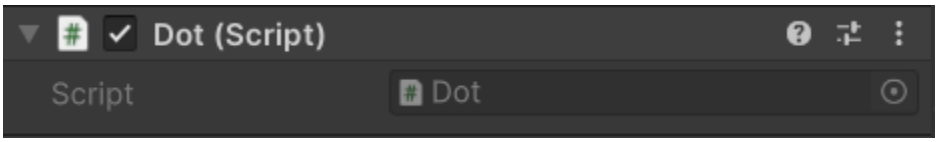
2. Once the script is created, double click the Dot prefab. This will open up the prefab scene for the Dot.



3. While in a prefab scene, any changes we make will update every instance of the prefab. Firstly, let's make sure the to Dot is in the center of the screen. We will zero out it's postion on the transform component.



4. Next let's attach the Dot script to the Dot object. Make sure the Dot script appears in the inspector.



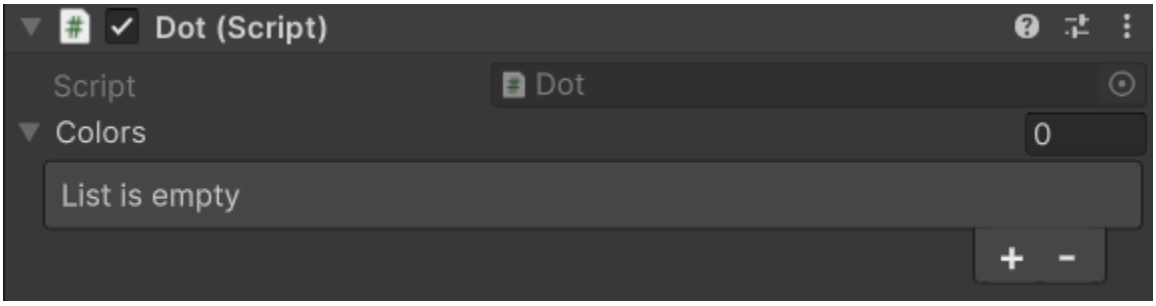
Colors

1. First let's allow the Dot to have a random color when it is first spawned in. To do this, in the Dot script, lets add the variables:

```
//Colors
public Color[] colors;
private SpriteRenderer sprite_renderer;
```

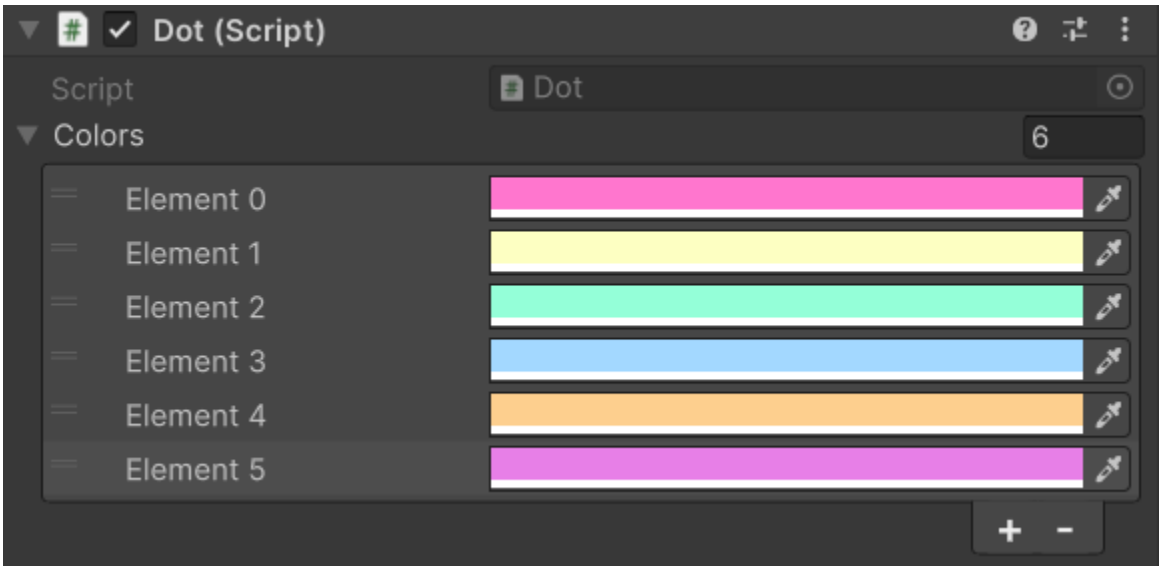
We create two variable.
Colors is an array of colors. These will be all the colors that the dots are able to be.
Sprite_Renderer will be used to store a reference to the sprite renderer component on the Dot object.

2. If we save our script and return to Unity, the Dot script component on the Dot should now have a color array as a variable.



If we set the size of the color array to 6, we will have a list of colors that we will be able to have our dot choose from.

If we click on one of the black bars, a color picker will appear. We can use this to set the colors for our dots. Be sure that the **Alpha** value of each color is set to **1**. **Alpha** controls the transparency of a color, 1 is solid and 0 is transparent.



Here are all the hex codes for each of the colors above:
FF76CE FDFFC2 94FFD8 A3D8FF FDCF8E E77FE7

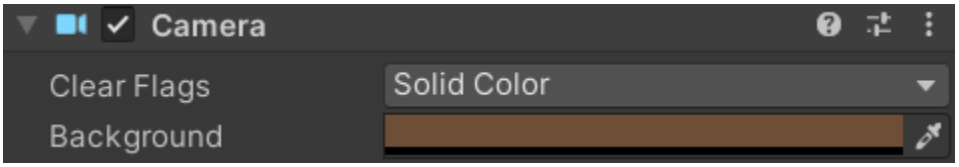
3. Back in the script, at the top of the start function, let's write the code:

```
//Set Sprite Color
sprite_renderer = GetComponent<SpriteRenderer>();
sprite_renderer.color = colors[Random.Range(0, colors.Length - 1)];
```

We set the sprite_renderer variable to the SpriteRenderer component that is on the dot.
We then access its **Color** property on the renderer and set it to one of the colors that was in

the array.
Remember that we can access elements of an array by putting a number in square brackets at the end of the array name. Here we get a number that is between zero and the length of the color array – 1, this is essentially zero through five, so six in total.

4. Save the script and return to Unity. The last thing we need to change is the background color of our game. To do this, find the camera and set it background color to a color of your choosing.

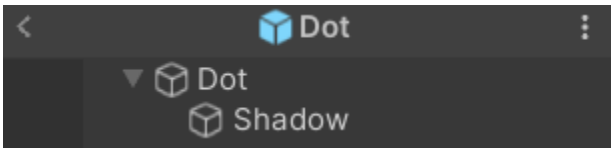


Here is the Hex code for color in the image above.
6F4E37

5. If we run the game now, we should see that each dot will have a random color assigned to it as well as the background is a different color.

Drop Shadow

1. One other thing we can do to make our dots stand out a little bit more is to give them a drop shadow. To do this go into the Dot prefab scene and then add a **2D Object > Sprites > Circle** as a child of the Dot object. Name this new sprite **Shadow**.



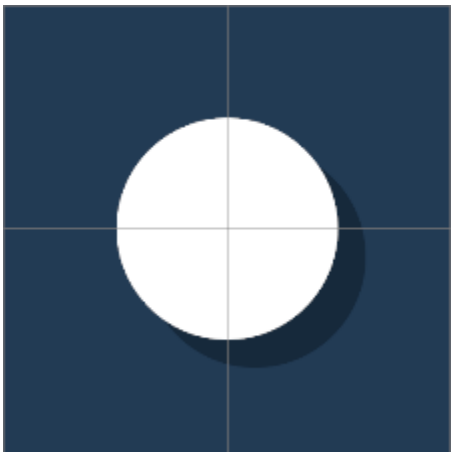
2. Set the shadow's position value to:

3. Set Color value to black and make it have an **alpha** value of **0.5**.

4. Lastly, set order in layer to:

This will make the shadow appear behind the dot.

5. The dot should now have a drop shadow.



Different Sizes and Score

1. Next let's make the dots that spawn have different sizes. We can then assign different point values to the dots based on their size.

2. In the Dot script let's and a few variables:

```
//Sizes
private int size = 0;


//Point Value
[HideInInspector]
public int point_value;
```

Size will store a value between 0 and 3. 0 means the dot will be small, 1 will mean the dot is normal size and anything greater than 2 will mean the dot will be large.
Point_Value will keep track of how many points a dot is worth. We use the **[HideInInspector]** line to make the public variable not be visible in the inspector. We want the variable to be public so other scripts can access it, but we do not want to edit this variable in the inspector.

3. Then in the Update() function let's write the code:

```
//Set Size
size = Random.Range(0, 3);
```

Here we set the size variable to be a value between 0 and 3. This includes decimal numbers.

4. Underneath let's write the code:

```
//Set value and scale based on size.
switch (size)
{
    case 0: //Small
        transform.localScale *= 0.7f;
        point_value = 20;
        break;

    case 1: //Medium
        transform.localScale *= 1.4f;
        point_value = 10;
        break;

    case 2: //Large
    default:
        point_value = 5;
        transform.localScale *= 2.1f;
        break;
}
```

We use a switch statement to determine the value of the dot based on its size variable.
If the size is 0, the scale is set to 0.7 and its point value will be 20.
If the size is 1, the scale is set to 1.4 and its point value will be 10.
If the size is 2 or some other number, the scale is set to 2.1 and its point value will be 5.
You can change any of these values to whatever you think is best.

5. Lastly, now that the dots have their own point values, we need to make sure that the GameLogic script will add the correct amount of points when the dot is clicked. In the GameLogic script let's remove the variable for points.

```
//Score
private int score = 0;
//private int points = 10;
```

I've commented out the line that we need to delete.

Next, find where we are adding points to the score and change the line to:

```
//Add Points
score += hit.collider.gameObject.GetComponent<Dot>().point_value;
```

Here we get the Dot script component that is on the dot that we clicked on and then access its point value. We then just take that value and add it to the score.

6. If we save the scripts and run the game, the dots should now have different size. If a dot is clicked on, it should have a different point value based on the size of the dot.

Animations

1. Next let's add some simple animations. The dot will grow in size when it first spawns in. After a short amount of time the dot will shrink and then disappear.

   To do this, lets first add a few variables to the Dot script:

```
//Scale
private Vector3 scale = Vector3.zero;

//Life Time
private float time_before_shrink = 0.0f;
```

   Scale will be used to store the size of the dot.
   Time_Before_Shrink will be the duration of time the dot will spend on screen before it starts to shrink.

2. It the Start() function, lets add a few lines of code to our switch statement cases.

```
case 0: //Small
    scale = transform.localScale * 0.7f;
    point_value = 50;
    time_before_shrink = 1f;
    break;

case 1: //Medium
    scale = transform.localScale * 1.4f;
    point_value = 20;
    time_before_shrink = 1.5f;
    break;

case 2: //Large
default:
    scale = transform.localScale * 2.1f;
    point_value = 5;
    time_before_shrink = 2f;
    break;
```

Instead of setting the scale of the dot at the start we are going to store how big it will be in the scale variable.
Then for each case we will set the time_before_shrink variable.

3. Then underneath the switch statement we write the line:

```
//Set scale to zero;
transform.localScale = Vector3.zero;
```

This just sets the scale of the dot to zero. We want its scale to be zero at the start so it can grow in size.

4. Then in the Update() function let's write the code:

```
//Grow/Shrink based on scale.
transform.localScale = Vector3.MoveTowards(transform.localScale, scale, 3.5f * Time.deltaTime);
```

First, every frame we are going to be setting the scale of the Dot. We use the **Vector3** class's **MoveTowards** function to make the Dot grow or shrink depending on the dot's scale.

Underneath we write the code:

```
//Shrinking
time_before_shrink -= Time.deltaTime;
if(time_before_shrink <= 0)
{
    scale = Vector3.zero;
}
```

Like we have been doing with timers, we subtract Time.deltaTime from the time_before_shrink variable. Once the time_before_shrink is zero or below we set the scale of the dot to zero.

Lastly, underneath we write the code:

```
//Destroy when fully shrunk.
if (transform.localScale == Vector3.zero)
{
    Destroy(this.gameObject);
}
```

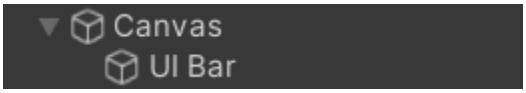
Once the scale has reached zero, we just destroy the game object.

5. If we save the scripts and run the game, we should see the Dots grow and shrink in size.

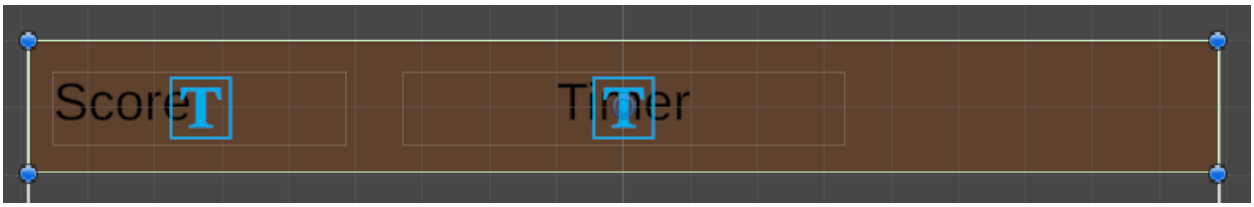
UI

1. Let's next work on getting our UI to look a little bit nicer. To do this let's add bar that the UI can sit on top of.

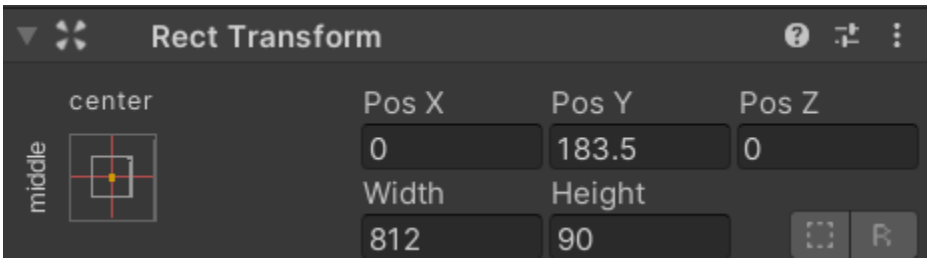
   First in our Canvas lets add a **UI > UI Image**. Name this image **UI Bar** and make it the **first** child of the Canvas. We make it the first child of the canvas so the other UI elements will be drawn on top of it.

2. Move the UI Bar to the upper left-hand corner and scale it to fit across the screen. Change color to be slightly darker than the background.

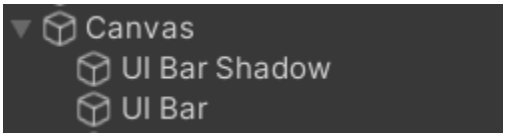


Here are the exact valuse of the UI Bar.



3. Let's also take the time to extend textboxes a little so longer text can fit inside them.

4. Let's also add a drop shadow to the bar by using a similar process to the one that we used to add drop shadow to the Dots.



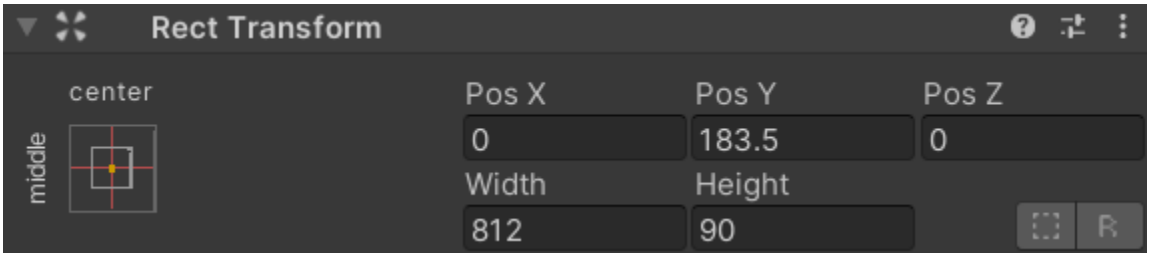
5. Next, we can change the text color as well as make the text bold to make the text pop out a bit.

6. Now that we have a UI bar that covers the screen, we need to make sure dots cannot spawn underneath it. To do this let's go to the **SpawnDot** function in the **GameLogic** script.

   Find where we are setting the X and Y position of the dot and change the code for the Y position to be:

```
int y_pos = Random.Range(0, cam.scaledPixelHeight - 90);
```

   The **90** is the height value of the UI bar. You may need to have this be a different value depending on the size of your UI bar.



7. Lastly let's make it so the Timer's text reads out a whole number value. To do this we can find where we are setting the timer's text and set its value to:

```
game_timer_text.text = "Time: " + Mathf.Floor(game_timer).ToString();
```

   All we do is just floor the timer's value. This will round the timer value down so it will read as a whole number.

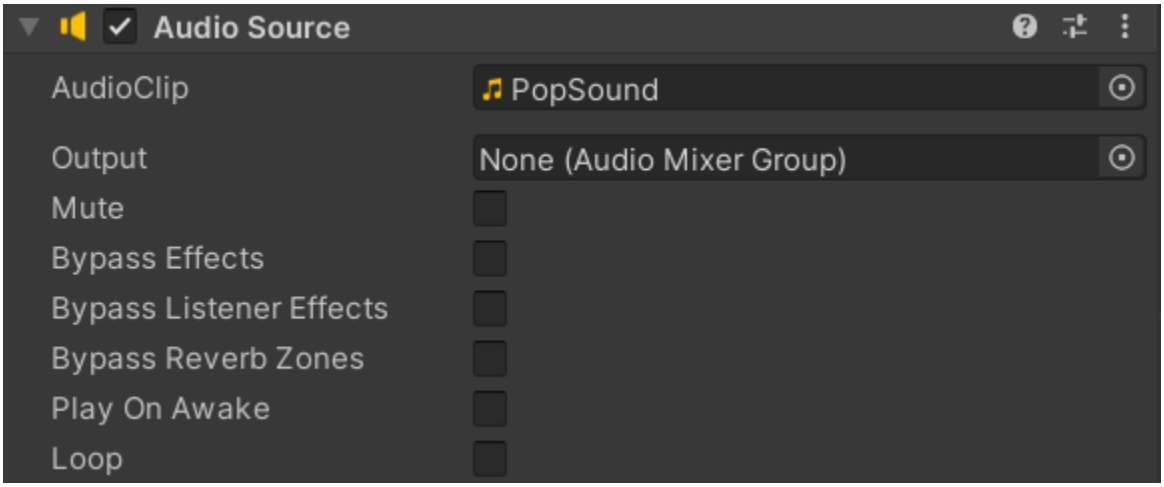
8. Save the script and run the game to see the changes.

Playing Audio

1. To add some feedback to the player's action, we can add sound effects. We are going to make the dots play a pop sound when they are clicked.

2. First, we need to download the **PopSound** file to use for this project. Once downloaded, put it in the assets folder.

3. Next, in the Dot prefab scene, click on the dot and add an **AudioSource** component. Set the **AudioClip** to the **PopSound**. The **disable** the **PlayOnAwake** option.



4. In the Dot script add the variable:

```
//Audio
public AudioSource audio_source;
```

Audio_Source will store a reference to the AudioSource component that is on the Dot.

5. In the Start() function write the code:

```
//Get audio source component.
audio_source = GetComponent<AudioSource>();
```

Here we just set the audio_source to the component on the Dot.

6. Let's switch over to the GameLogic script. Find the code that is checking to see if we click on the Dot. Inside the if statement for the collider, lets add the code:

```
//Play pop sound.
hit.collider.gameObject.GetComponent<Dot>().audio_source.Play();
```

Here we are accessing the audio_source variable of the Dot object and telling it to play the AudioClip we assigned.

7. If we save the script and run the game. When we click on a Dot **no sound will play**. This is because when we go to play the sound, we are also deleting the object responsible for playing the sound. To fix this we are going to change how are dot is being destroyed when we click on it.

8. Back in the code, under where we added the code for playing the sound, lets write the code:

```
//Hide the object, cannot play audio after an object is deleted.
hit.collider.gameObject.GetComponent<SpriteRenderer>().enabled = false;
hit.collider.gameObject.transform.GetChild(0).GetComponent<SpriteRenderer>().enabled = false;

//Destroy the dot, when the sound is finsihed playing.
Destroy(hit.collider.gameObject, hit.collider.gameObject.GetComponent<Dot>().audio_source.clip.length);
```

The first two lines disable the sprite renderer for the Dot and its shadow.
We then add on to the Destroy line by supply a time. This is the audio_source.clip.length part. Once this time has passed the dot will be destroyed.

To put everything in perspective, we make the dot invisible by disabling its renderers and then cause it to be destroyed when the pop sound effect has finished playing.

9. Lastly, we are just going to change the If statement that is containing all of this code. Let's make it say:

```
//If something was hit...
if (hit.collider != null && !hit.collider.gameObject.GetComponent<Dot>().audio_source.isPlaying)
{
```

Here we are using the And operator to check for another condition. This condition is if the audio source on the Dot is currently playing.

This may seem a little bit random but this prevents the player from clicking on a dot multiple times to greatly increase their score. Remember the dot is not destroyed instantly but rather turns invisible. The player could still click on the dot even though it appears to not be there.

10. Here is what the full if statement will look like:

```
//If something was hit...
if (hit.collider != null && !hit.collider.gameObject.GetComponent<Dot>().audio_source.isPlaying)
{
    //Play pop sound.
    hit.collider.gameObject.GetComponent<Dot>().audio_source.Play();

    //Hide the object, cannot play audio after an object is deleted.
    hit.collider.gameObject.GetComponent<SpriteRenderer>().enabled = false;
    hit.collider.gameObject.transform.GetChild(0).GetComponent<SpriteRenderer>().enabled = false;

    //Destroy the dot, when the sound is finsihed playing.
    Destroy(hit.collider.gameObject, hit.collider.gameObject.GetComponent<Dot>().audio_source.clip.length);

    //Add Points
    score += hit.collider.gameObject.GetComponent<Dot>().point_value;

    //Update score text.
    score_text.text = "Score: " + score.ToString();
}
```

11. Saving the script and playing the game now, we should be able to hear a popping sound when the dots are clicked.

Modulating Audio

1. Last thing we should do is make the dots have a different popping sound depending on what size they are. We can do this easily by modulating the pitch of the audio source.

2. In our switch statement for the Dot script, lets add a few lines of code:

```
//Set dot properties based on scale.
switch (size)
{
    case 0: //Small
        scale = transform.localScale * 0.7f;
        point_value = 50;
        time_before_shrink = 1f;
        audio_source.pitch = Random.Range(1.3f, 1.5f);
        break;

    case 1: //Medium
        scale = transform.localScale * 1.4f;
        point_value = 20;
        time_before_shrink = 1.5f;
        audio_source.pitch = Random.Range(0.9f, 1.1f);
        break;

    case 2: //Large
    default:
        scale = transform.localScale * 2.1f;
        point_value = 5;
        time_before_shrink = 2f;
        audio_source.pitch = Random.Range(0.5f, 0.7f);
        break;
}
```

   The pitch will be higher for smaller dots and lower for big dots.

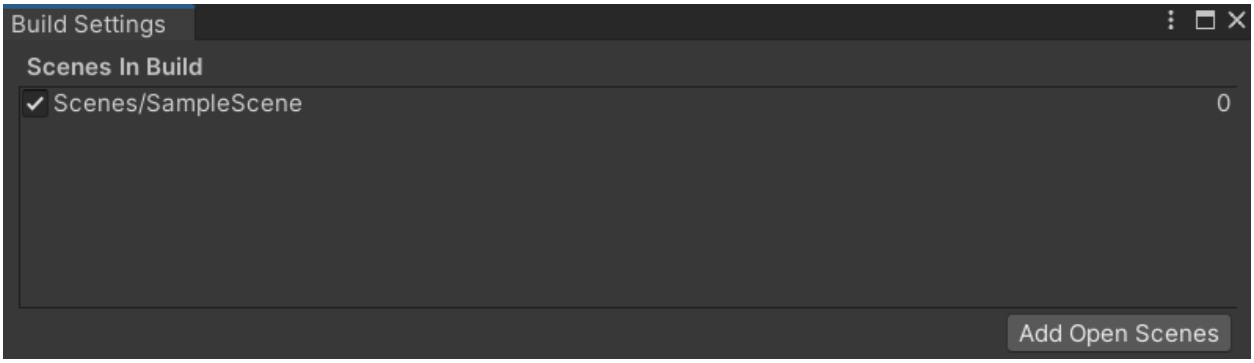
3. If we save the script and run the game, each dot should now have a unique sound.

Exporting

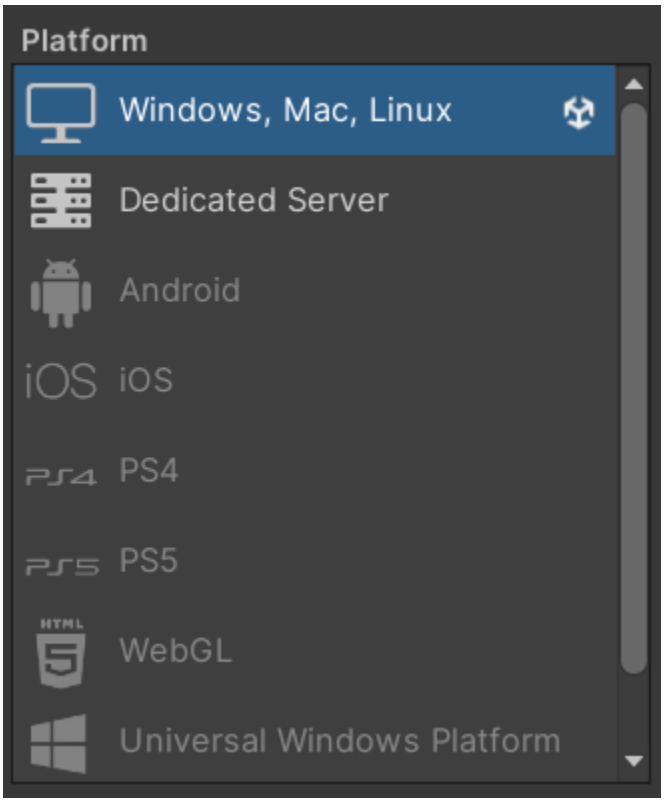
1. The last thing we are going to do is create a build of our game.

2. To do this go to **File** > **Build Settings**. Once you do you should be greeted with the Build Settings window.

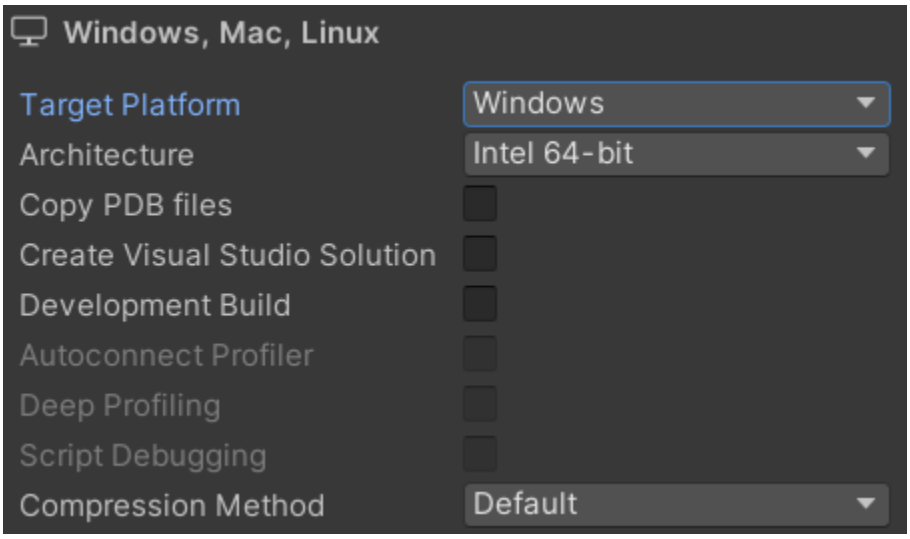
3. At the top you will see a box. You can drag and drop scenes from the project into this box. The scenes that are dropped into the box will be the ones that are compiled for the executable.



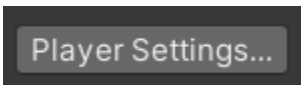
4. On the left you can see what you want your target platform to be.



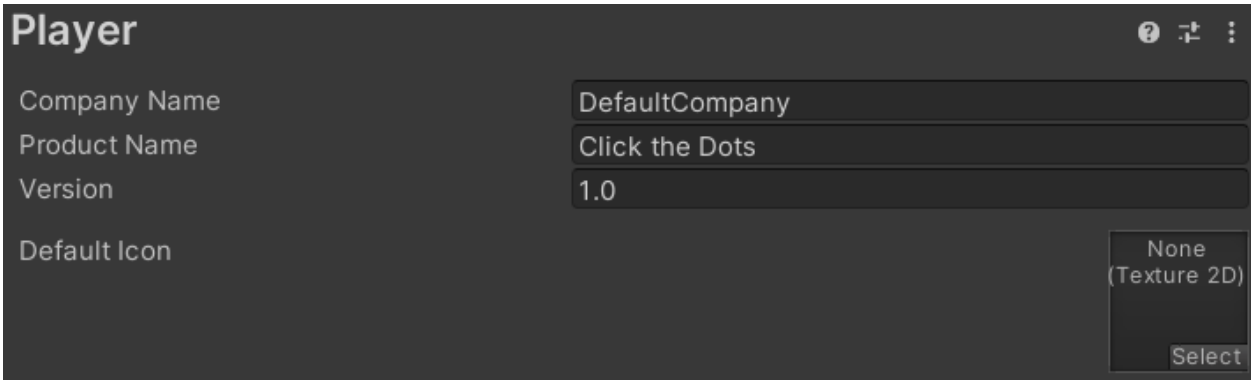
You can also fine tune the platform settings with the options to the right of this box.



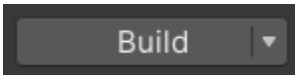
5. At the lower left-hand corner, you can find the **Player Settings** button.



Clicking it will open a new window. This window will allow you to change some settings about the build such as the company name or the icon for the executable.

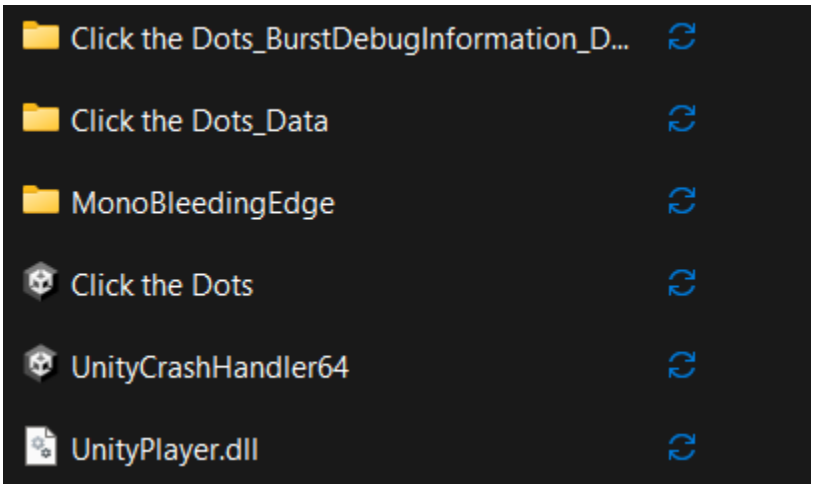


6. Back in the build settings, to actually build out our game, we just need to click the **Build** button.



This will prompt you will a window asking where you would like to build the game to. Once you find the location, just click on **Select Folder** and then the game will start to build. Once the build is complete you should be able to see all of the file for the build in the location you

chose.



Improvements

1. The elements that we added to this game are only a few examples of what we could do to polish this game. For instance, if we wanted to polish this game even further, we could add elements such as a title screen, custom text, music, custom sprites for the dots, particle effects, maybe even put cute little faces on the dots and probably a lot more.

2. Making game is designing the systems for the game first and then adding polish.

GitHub

1. Push the project to the GitHub repository.