

# Click the Dots Game: Part 1 – Base Game

## Lesson Goals

- Create a simple click the dots game to help build the foundation for our knowledge of game development.
- We will be making the basic gameplay elements first, then in the next lesson we will be polishing each element to make for more engaging gameplay.
- Emphasize making smaller games when first starting out in game development.
- Emphasize making the base of the game first and then adding polish later.

## Lesson Intro and Game Development Theory

1. For this lesson we are going to be making a simple click the dots type game. The game allows the player to click on various dots that will appear on screen. Each dot clicked will award the player with some amount of points. The player's goal is to get as many points as they can within a set amount of time. Once the game is over the player will have the opportunity to restart the game and play again.
2. As beginner game developers, we want to start off by making **small and simple** games. We do this so that we can create a game that is achievable for our skill level and so that we can start to build a foundation of knowledge for game development. We need to have this foundation so that we can make bigger and better games in the future.
3. The first big lesson that we want to try and learn as game designers is that we need to build out the core elements of our game first and then add on top of it. As game developers we need to ensure the base of our game is **functional** and **fun**.

A good industry example of this is the story behind Mario 64's development. The first year of the game's development was spent fine tuning Mario's movement and abilities. Once the designers felt like Mario was solid, they could then build a game around him. This meant level designers could now make a level that was tailored to how Mario could move and interact with the world.



4. Let's imagine now that the Mario 64 level designers had not waited for Mario's movement to be fine-tuned. This means that they would have made levels that were not made for Mario. A level designer could have spent all their time and energy on a really nice looking level, only for it to be scrapped when Mario was finally polished because the level did not fit how Mario moved.

In fact we may even have a little proof that this did happen during the game's development. A few of Mario 64's beta maps were found in what was known as the Nintendo Giga Leak. The maps themselves were way too huge for Mario, suggesting that the designers didn't quite know how big Mario was going to be at the time.



5. This is why it is important to build the base of the game first and then build upon it. We do not want to waste our time working on something that may not even fit into the game. This is why I want to try and put emphasis on making the foundation of the game first before we add a polish. We need to make sure it is fun and functional before anything else.

## Demo

1. By the end of this lesson, we will have something the looks like:

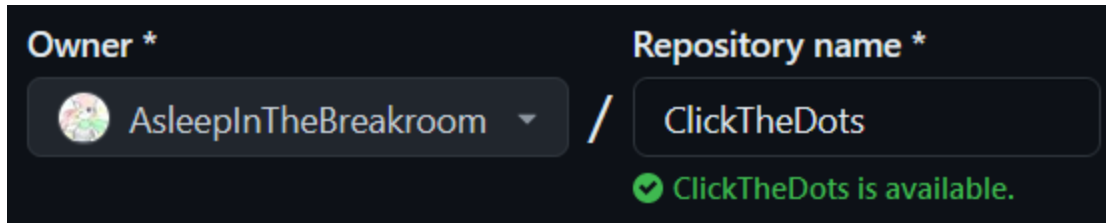


## Creating a GitHub Repository

1. Let's first start our project by creating the repository that will hold all the files for our project.
2. Go to GitHub.com, make sure you are logged in and then click on the green **New** button.



3. First you need to select the **Owner** and **Name** of the Repository.

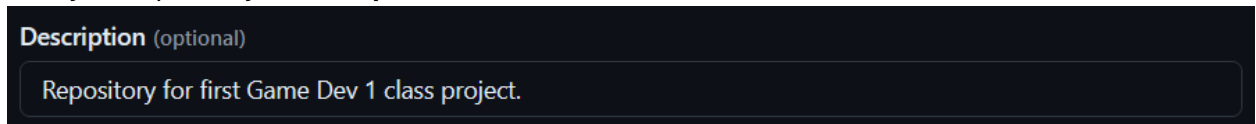


Owner \*  
AsleepInTheBreakroom

Repository name \*  
ClickTheDots

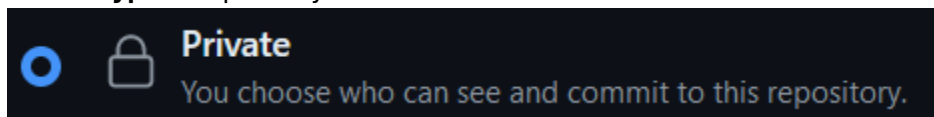
ClickTheDots is available.

4. Give your repository a **Description**.



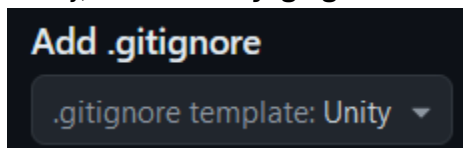
Description (optional)  
Repository for first Game Dev 1 class project.

5. Set the **type** of repository to **Private**.



Private  
You choose who can see and commit to this repository.

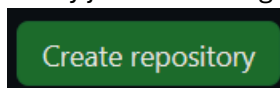
6. Lastly, add the **Unity .gitignore** file.



Add .gitignore  
.gitignore template: Unity

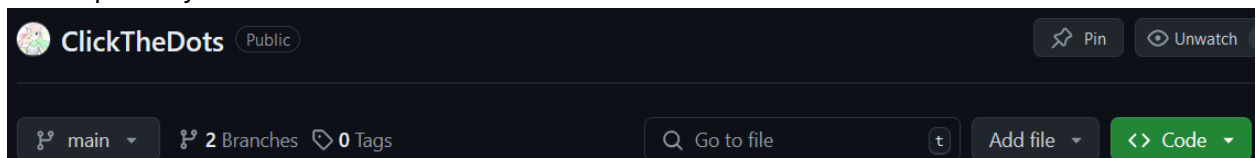
Be sure to have this or you will not be able to upload you project to github.

7. Laslty just Click the green **Create Repository** button.



Create repository

8. Your repository should then be created.



ClickTheDots Public

main 2 Branches 0 Tags

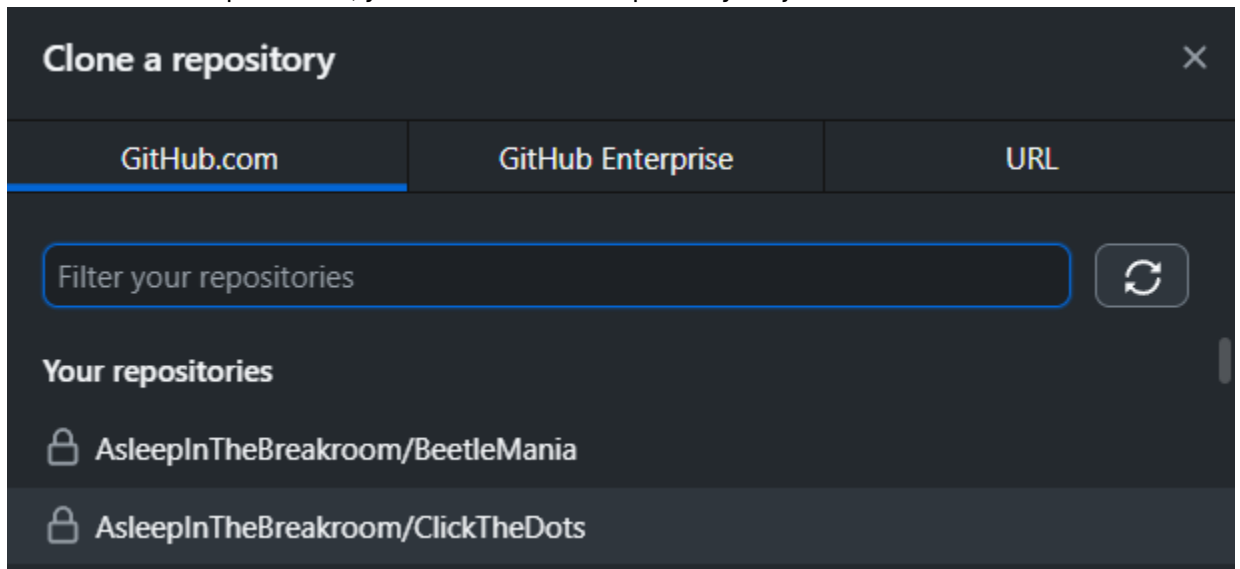
Go to file Add file Code

## Cloning the Repository

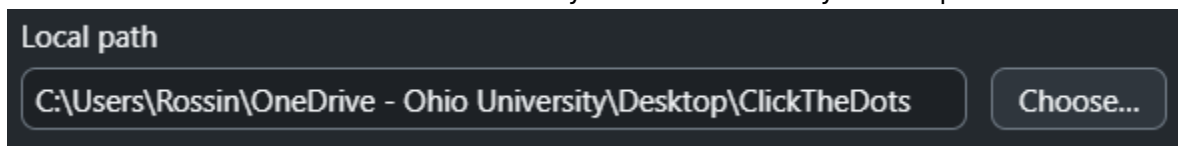
1. We now need to download our project. We will do this using **GitHub Desktop**.

Open up **GitHub Desktop**, sign in and click **File > Clone Repository**.

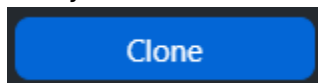
- From the list of repositories, you should see the repository we just created on GitHub.



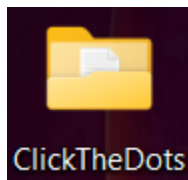
- Click it and then set the **Local Path** to an easy to find location on your computer.



- Lastly click the **Clone** button.

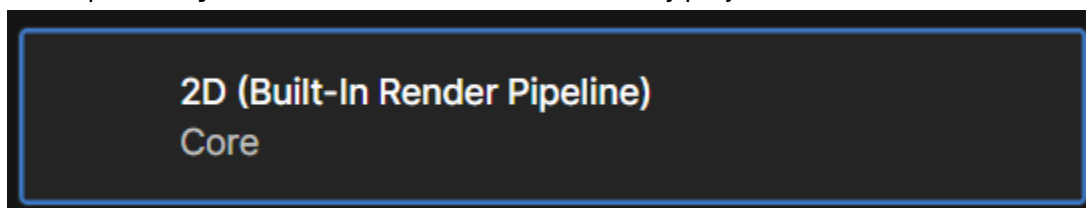


- You should then see the cloned repository on your computer.

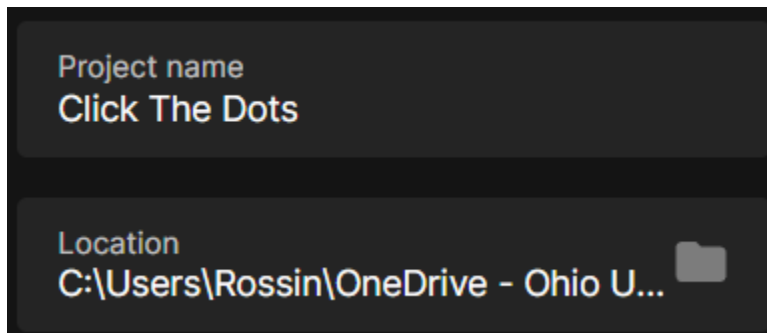


## Unity Project

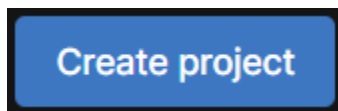
- Next open **Unity Hub** and create a new **2D Core** Unity project.



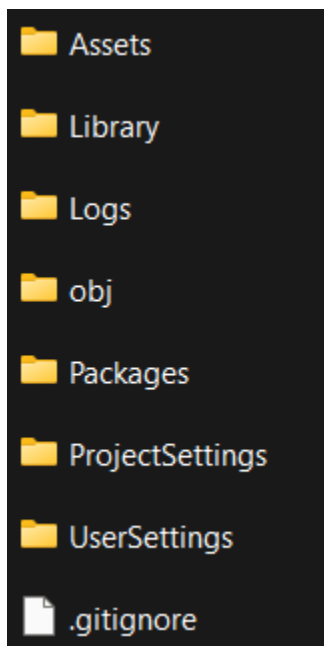
2. Name the project **Click The Dots** and set the location to the **repository we just cloned from github**.



3. Once done, click the **Create Project** button and wait for a few minutes.



4. Once the project is created, we just need to move the .gitignore file into the right place. Open the cloned repository and find the .gitignore file, then move it into the same folder the houses all the files for the project.



**We need to make sure the .gitignore is in this spot for our project. If not, we will not be able to commit the project to GitHub. We will get a large file size error.**

5. The only thing we need to do before we start making our game is adjust the aspect ratio. To do this switch over to the **Game** tab at the top of the game window.



- There should be a box labeled **Free Aspect**,

Free Aspect ▼

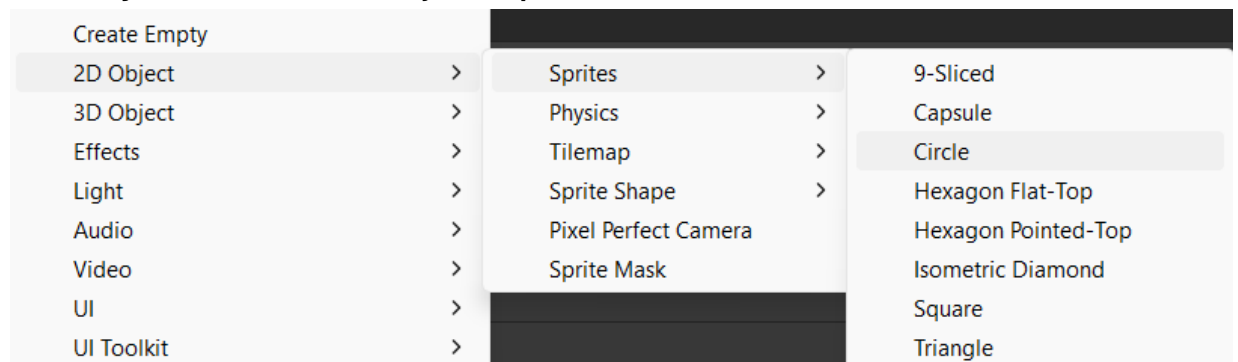
Click it and from the drop down select **16:9 Aspect**.

16:9 Aspect ▼

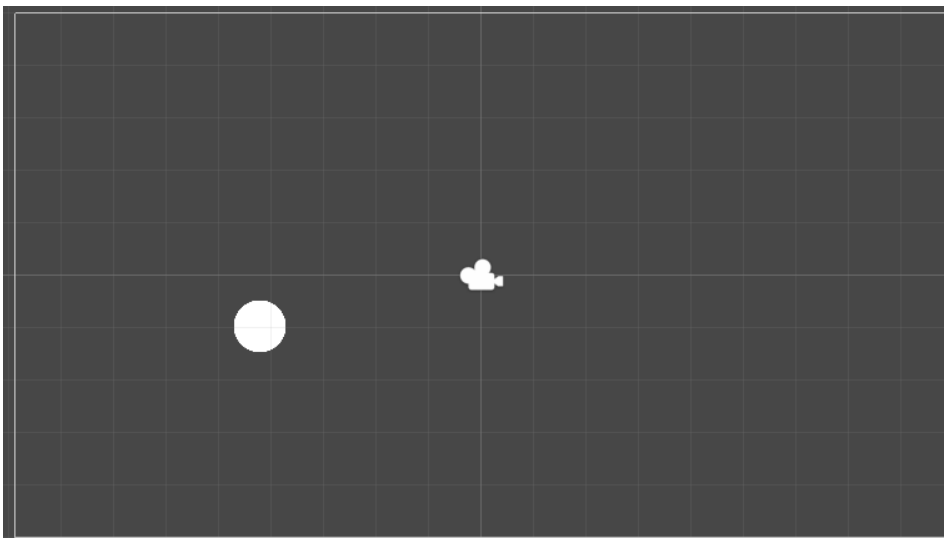
There should now be black bars on the sides of the game window.

## Creating a Dot

- The first element of our game that we are going to make is to allow the player to click a dot on the screen.
- Let's first start off by making our dot object. To do this we can right click within the **hierarchy** and then select **2D Object > Sprites > Circle**.

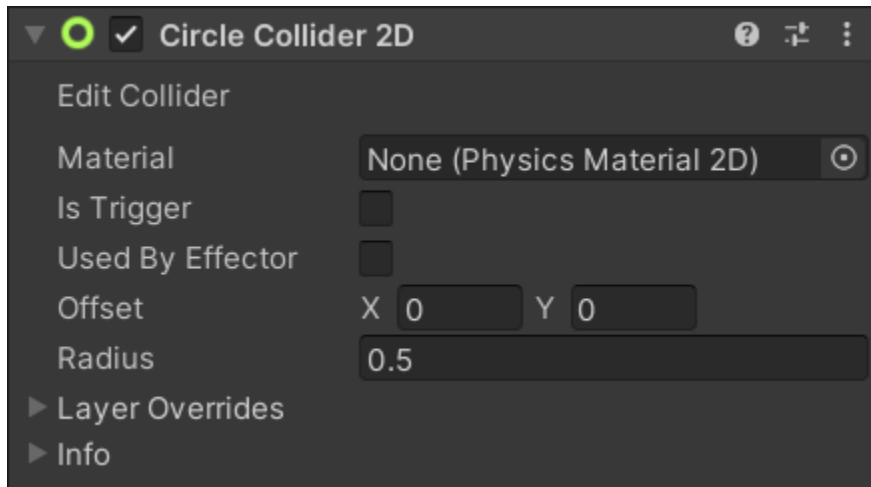


This will create a circle sprite object in our scene. Let's rename this circle to **Dot**.

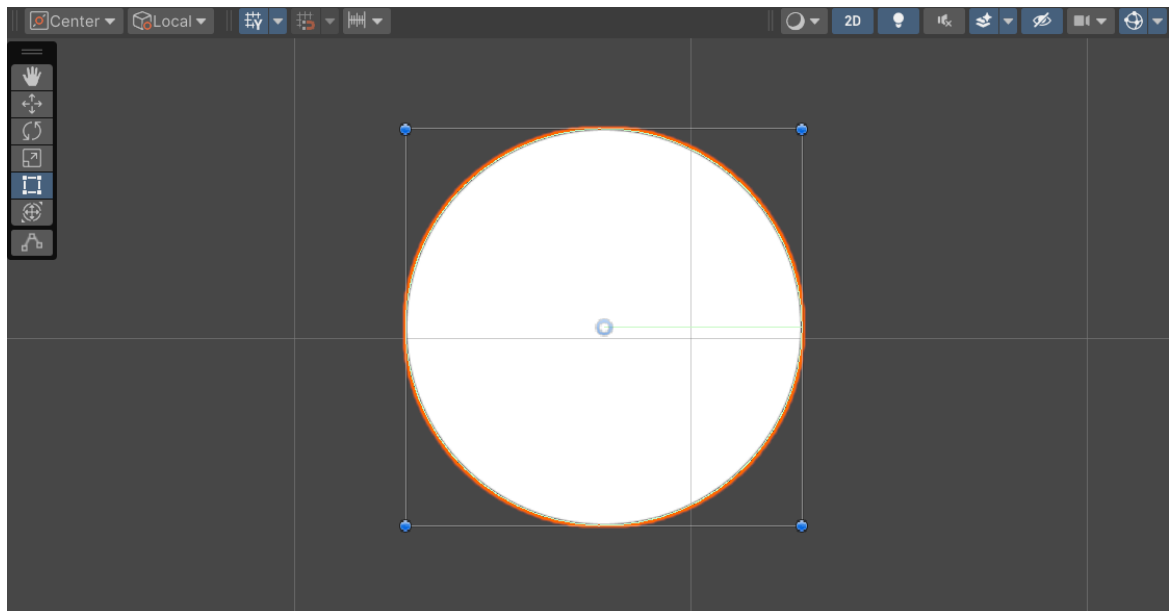


- Now that we have our dot, let's work on making a way for it to be click on by the player. Whenever we want objects in our games to be interactable, we need some way to collide with them. Luckily for us, we can add **colliders** to our objects to handle these types of interactions.

Clicking on the Dot and going to the **Inspector**, find the **Add Component** button and search for **Circle Collider 2D**. Once found add it to the Dot.



It may be hard to see, but our Dot should now have a lime green circle around. This is the collider that we just added.



Colliders allow our game objects to process **Collision** information. This is good for use because we can think of us clicking on the dot as a form of collision.

## Click Interaction

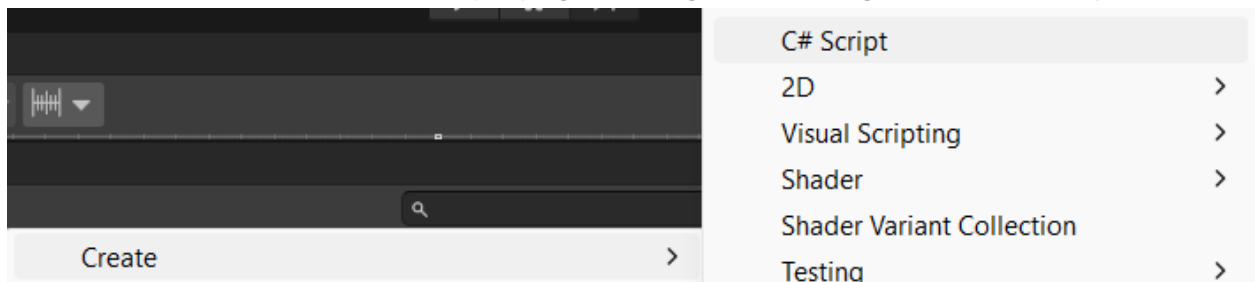


1. Next let's make an object that will handle most of our game's logic, this will include the logic of clicking on our Dot. In the **hierarchy** right click and select **Create Empty**. Rename the Empty to **Game Logic**.

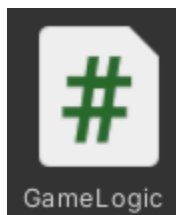
Create Empty

An **Empty** is a game object that only has a transform on it, meaning it has position, rotation and scale. These are usually useful for grouping objects together and moving them around all at once. But for our purposes we are going to use it to run the game's main logic script. So, let's do that!

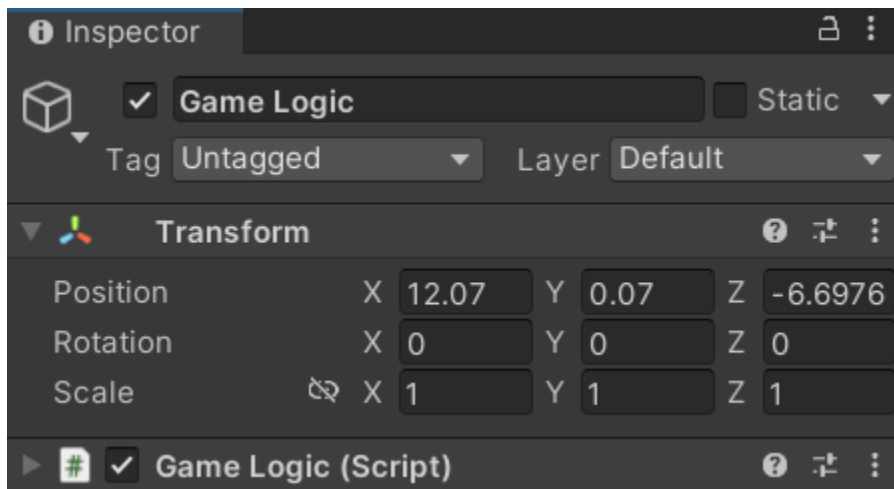
2. In the **Assets** folder create a new script by right clicking and selecting **Create > C# Script**.



Name this script to **GameLogic**.



Once the script has been added to the project. Select the **Game Logic** empty object and then drag the script onto the object's inspector. This will **attach** the script to the object.



3. Double click the script to open it up Visual Studio. Before we code anything, lets first think about what we want our script to do.

Script Goals:

- a. Check to see if the left mouse button was pressed.
  - b. Find a way to collide with the Circle Collider 2D on our Dot.
  - c. Delete the Dot once it was collided with.
4. Alright, so let's first start off by first seeing if the player clicked the left mouse button. In the **Update()** function let's write the code:

```
if(Input.GetMouseButtonDown(0))  
{  
    ...  
}
```

We use the [Input](#) class's [GetMouseButtonDown](#) function to check if one of the mouse buttons was being pushed down. We can supply this function with three numbers, 0, 1 or 2. Zero in our case represents the left mouse button.

5. Next, in order to collide with the Circle Collider 2D, we are going to use what is known as a **Raycast**. A raycast can be thought of as an invisible line that shoots out from a point in space, in a direction, for a set length. If this line collides with anything it will provide information on what was collided with.

In our code let's first get a reference to the camera that is in our game. At the top of the class let's add the variable:

```
public Camera cam;
```

We get a reference to our camera because we will use it to find the point in space that we will emit the raycast from.

6. Then inside the if statement we created from a few steps ago, lets write the code:

```
Vector2 worldPoint = cam.ScreenToWorldPoint(Input.mousePosition);  
RaycastHit2D hit = Physics2D.Raycast(worldPoint, Vector2.zero);
```

Here we create two variables.

**worldPoint** takes the Mouse's position on screen and converts into a point in 3D space in our game. It then stores the result. This part may be a little bit confusing so don't worry about too much.

**Hit** is the information that we receive from doing a Raycast. We use the [Physics2D](#) class's [Raycast](#) function to emit the raycast at the **worldPoint** position in the direction of

## Vector2.zero.

To help us visualize things, we can just think about this as if we are shooting a line forward from the position of our mouse. If that line hits something, we get the information about what was hit.

7. Lastly, we just need to destroy the Dot if hit by the raycast. To do this we can write the code:

```
if (hit.collider != null)
{
    Destroy(hit.collider.gameObject);
}
```

We first check to see if the hit information was able to return a collider. **Null** in computer languages means nothing, so here we are checking to see if the collider was not nothing. And if it was, we Destroy the game object that was attached to the collider. This would be the Dot. **Destroying** an object means to remove from the scene of our game.

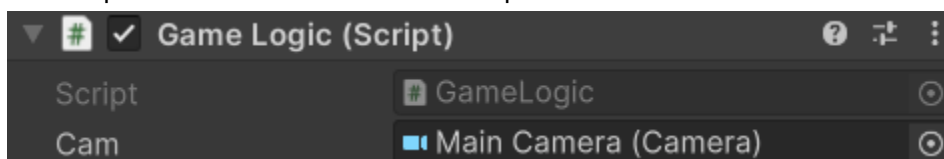
8. Here is what the full update function:

```
// Update is called once per frame
void Update()
{
    /**Clicking a Dot**
    //If the left mouse button is clicked.
    if (Input.GetMouseButtonDown(0))
    {
        //Get the mouse position on the screen and send a raycast into the game world from that position.
        Vector2 worldPoint = cam.ScreenToWorldPoint(Input.mousePosition);
        RaycastHit2D hit = Physics2D.Raycast(worldPoint, Vector2.zero);

        //If something was hit...
        if (hit.collider != null)
        {
            //Destroy the dot.
            Destroy(hit.collider.gameObject);
        }
    }
}
```

I've added some **comments** to help break down what the code is doing.

9. Save the script. Returning to Unity, we just need supply the **cam** variable from our script with the **Main Camera** from our game. We can drag the **Main Camera** from the **hierarchy** and drop it into the **Cam** slot on the script.

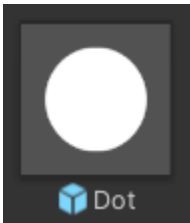


When we run the game, we should now be able to click on the Dot and see it disappear.

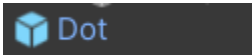
## Dot Prefab

1. Now that we can click on a dot, we need to find some way of making it so they can appear at a random position on screen. We will be using the game logic script to do this.
2. The very first thing we need to do is turn our Dot game object into a **Prefab**. A prefab allows us to store a game object we make in the files of our project.

To do this, we drag the Dot object from the **hierarchy** and drop into the assets folder.



You will notice that the object in the hierarchy has turned blue, this signifies that the object in our scene is a prefab.



3. Now that that Dot object is a prefab, let's **delete** it from our scene. Since the object is a prefab, it still exists within our game for later use.

## Spawning Dots

1. Before we dive into coding, let's first break down what our goals should be for spawning dots.

Goals:

- a. Spawn a dot after a certain amount of time has passed.
- b. The spawned dot will appear at a random position on the screen.

For this task, we will be creating a timer that will run within our script. The timer will be set to a value, then when the timer fully counts down it will spawn a Dot.

2. Let's go ahead and open the **GameLogic** script and add a few variables:

```
//Spawning Dots
public GameObject dot;
public float time_between_spawns = 1f; //In seconds.
private float dot_timer = 0.0f;
```

**Dot** will store a reference to the Dot prefab object we just created. We will use this to instantiate dot game objects into our scene.

**Time\_Between\_Spawns** is the time between each dot spawn.

**Dot\_Timer** will keep track of how much time remains until a new dot can spawn.

3. In the **Start()** function let's write the code:

```
//Set Timer.
dot_timer = 0.5f;
```

Here we are setting the timer to have a time of 0.5 seconds. This will essentially tell the game to wait 0.5 seconds before spawning a dot at the beginning of the game.

4. Next in the **Update()** function, under the code we wrote, lets add:

```
/**Spawning a Dot**
//Make timer count down.
dot_timer -= Time.deltaTime;

//If timer has counted down.
if (dot_timer <= 0)
{
    //Reset Timer.
    dot_timer = time_between_spawns;

    //Spawn a new circle.
    SpawnDot();
}
```

We first make the dot\_timer count down. We do this by subtracting **Time.deltaTime** from the dot\_timer. Time.deltaTime represents the amount of time between each frame of our game.

We then check if the dot\_timer has fully counted down by seeing if it's value is less than or equal to zero.

If it is we reset the dot\_timer by setting to the time\_between\_sapwns value.

Lastly, we call the SpawnDot() function. We get an error because we do not have this

function declared yet. So, let's fix that.

5. Under the Update() function lets declare the SpawnDot() function:

```
void SpawnDot()  
{  
    ...  
}
```

Inside, lets write the code:

```
GameObject new_dot = Instantiate(dot);
```

We first create a variable called **new\_dot**. This variable keeps a reference to the newly instantiated dot object. To **instantiate** means to create an object within a game.

```
int x_pos = Random.Range(0, cam.scaledPixelWidth);  
int y_pos = Random.Range(0, cam.scaledPixelHeight);
```

Next we create two new variables **X\_Pos** and **Y\_Pos**. These will be used to determine where the dot will spawn at. We use [Random](#) class's [Range](#) function to get a select a random number between 0 and the Width or Height of the camera's view.

```
Vector3 spawn_point = new Vector3(x_pos, y_pos, 0);  
spawn_point = cam.ScreenToWorldPoint(spawn_point);  
spawn_point.z = 0;
```

Next we create another variable, this time called **Spawn\_Point**. This variable will be the location the dot will spawn at. We just set the X and Y values of the spawn position to the X\_Pos and Y\_Pos.

Then like we did for the mouse position when clicking the dot, we have to take this position and turn it into a position that is in 3D space using the camera's [ScreenToWorldPoint](#) function.

Lastly, we just set the Z value of the spawn point to zero. All this does it make it appear in the front of our camera.

```
new_dot.transform.position = spawn_point;
```

Then finally we move the new dot to the spawn point's position.

6. In full the function will look like:

```
void SpawnDot()
{
    //Instance the object.
    GameObject new_dot = Instantiate(dot);

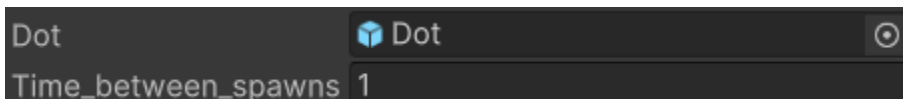
    //Set the spawn point of the circle.
    var x_pos = Random.Range(0, cam.scaledPixelWidth);
    var y_pos = Random.Range(0, cam.scaledPixelHeight);

    //Create the spawn point.
    Vector3 spawn_point = new Vector3(x_pos, y_pos, 0);
    spawn_point = cam.ScreenToWorldPoint(spawn_point);
    spawn_point.z = 0; //Move the dot to be infront of the camera's view.

    //Move dot to the spawn point.
    new_dot.transform.position = spawn_point;
}
```

7. Save the script and return to Unity. We then just need to give our script a reference to the Dot prefab we made. We can drag the dot prefab from our assets folder and drop it into the dot slot on the script.

We can also change the time\_between\_spawn variable if needed.



8. Now if we run the game, we should see dots begin to appear within the bounds of the camera.

## Score

1. Next let's add a score system to our game. In our GameLogic script lets add the variables:

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

**Score** will keep track of how many points the player has.

**Points** is the value that each dot is worth.

- Next, under where our dot is being destroyed, let's write the code:

```
//Add Points  
score += points;
```

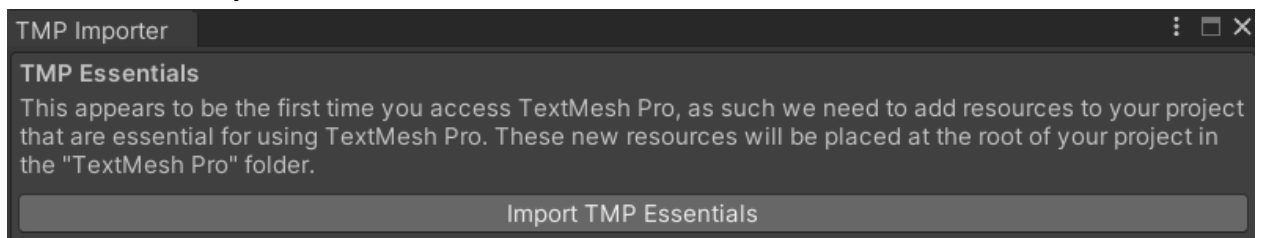
- Save the script and run the game. Each time we click a dot our score value will increase. The problem is, we have no way of knowing this. So, let's find a way to display the score to the player using UI elements.

## Displaying Score

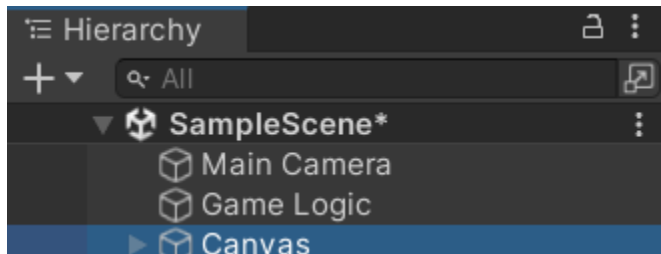
- In order for us to have a UI in our game we need to have what is known as a **Canvas**. A canvas will just be a place where we can keep all of our UI elements together.
- In order to add a canvas to our scene, right click on the hierarchy and select **UI > Text – TextMeshPro**.

Text - TextMeshPro

A pop up box is going to appear because we need to import **TextMeshPro** into our project. **TextMeshPro** allows us to have simple text and other UI elements in our game. Click the button labeled **Import TMP Essentials**.

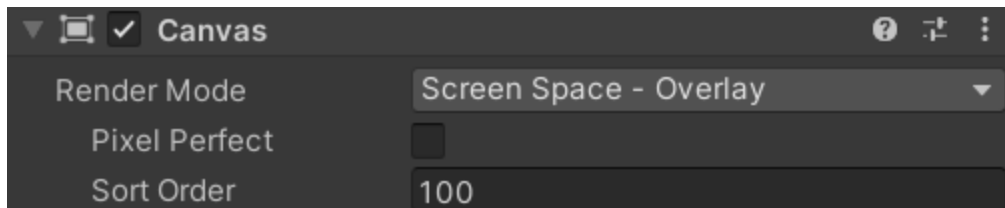


Once imported you should see the Canvas object inside the hierarchy.



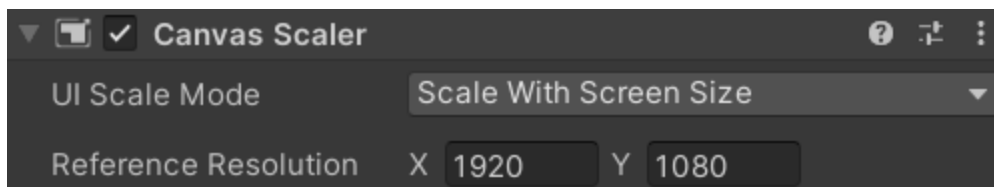
- With the Canvas selected go over to the inspector and find the **Render Mode** options on the **Canvas** component. Set the **Render Mode** to **Screen Space - Overlay**, then set the **Sort Order** to **100**.





Setting the render mode to screen space will make the UI fit within the bounds of our game window. Setting sort order to 100 will make sure that the UI will be drawn on top of our game objects.

Next to ensure that our UI will scale to our window find the **Canvas Scaler** component. Set the **UI Scale Mode** to **Scale With Screen Size**. Then set the **Reference Resolution** to **1920** for the **X** and **1080** for the **Y**.

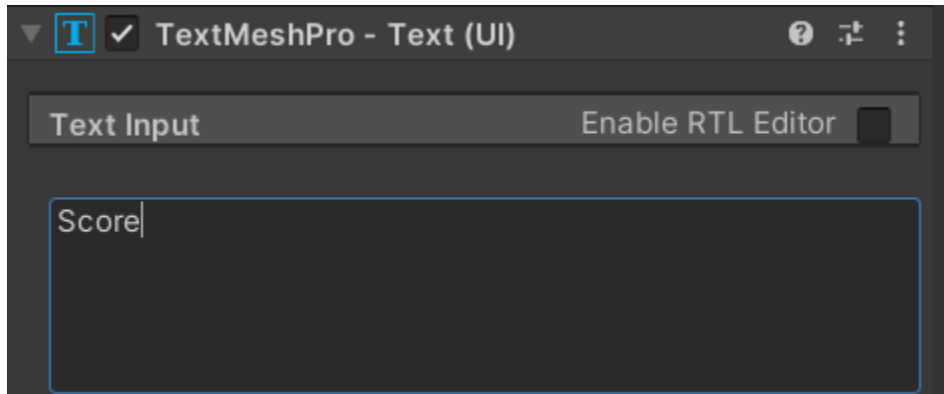


When we scale the size of our game window, we should see that the UI will scale with it.

4. Inside our canvas we should have a **Text (TMP)** object. Rename this object to **Score**.



Click on the score object. In the inspector set the text field to say the word **Score**.



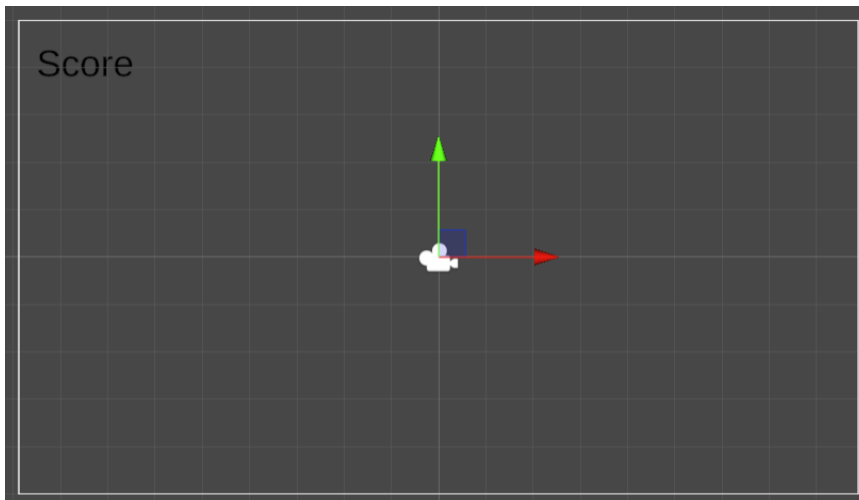
Let's then change the **Font Size** to **40** so the text can better show up on screen.



Let's also change the color of the text by changing the **Vertex Color** property.



Lastly move the score object to the top left of the canvas.



5. Let's hop into the GameLogic script. We first need to add a library to our script that will allow us to use TextMeshPro variables in our script.

```
using TMPro;
```

6. Next let's create a variable that will store a reference to our score text object.

```
//Score Text  
public TMP_Text score_text;
```

7. Then in the Start() function let write the line:

```
//Set score text at start of game.  
score_text.text = "Score: 0";
```

We are accessing the **Text** property of the score\_text object. We then just set that text to say "Score: 0".

8. The lastly in the Update() function, under where we add points, let's write the code:

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

Like before we access the text property of the score\_text object except now we set it to read "Score: " and then whatever our current score is.

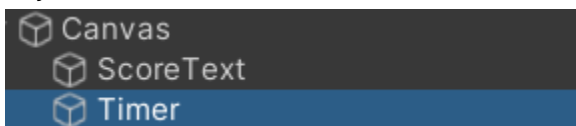
9. Save the script and hook up the Score object to the script.

```
Score_text | T Score (Text Mesh Pro UGUI) |
```

If we now run the game, we can now see the player's score value.

## Game Timer

1. We are able to click dots and see how many points the player has, but our game has no real sense of urgency or much of a point to it since it goes on forever. So, let's add a game timer to communicate to the player that they are trying to click as many dots as they can within a set of time.
2. Like before we are going to add a **Text – TextMeshPro** object to our canvas. Let's call this object **Timer**.



3. Also like before let's set text of the Timer to say **Timer**. We can also position the object to the center of the screen as well as color it black. We can also adjust the alignment of the text if need.



4. Since we are making another timer, we can more or less make the same variables as well as code the same logic that was needed for spawning dot. In the GameLogic script let's add the variables:

```
//Game Timer
private float game_timer = 60; //In seconds.
public TMP_Text game_timer_text;
```

**Game\_Timer** will keep track of how many seconds are left until the game end. It will have a default value of 60 meaning the game only lasts for one minute.

**Game\_Timer\_Text** will store a reference to the timer text object on our canvas.

5. At the **top** of the Update() function lets add the code:

```
/**Game Timer Text**  
game_timer -= Time.deltaTime;  
if(game_timer < 0)  
{  
    game_timer = 0;  
    game_timer_text.text = "Time: 0";  
    return;  
}  
game_timer_text.text = "Time: " + game_timer.ToString();
```

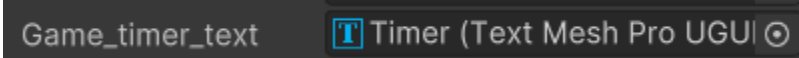
As with the dot\_timer, we subtract Time.deltaTime from the game\_timer to make it count down.

We then check to see if the game timer is less than zero. If it is, we set the game timer to zero and make the game\_timer\_text text property read "Time: 0". Then we write the word

**Return. Return stops any code that is underneath it from running.** In our case this would be clicking dots, spawning dots and updating the game\_timer text. This makes sense since we do not want any of those things to happen when the game is over.

Lastly we just set the timer text to read "Time: " and then whatever the current value of what the game timer is.

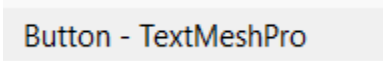
6. Save the script and then hook up the game\_timer\_text object for script.



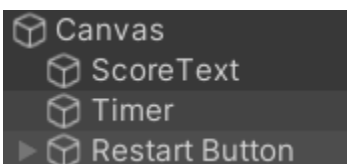
If we run the game, we can see that the timer is counting down. When the timer stops, we can no longer click on any dots nor will any more spawn.

## Restarting

1. Now that our game has an end, we should code a way for the player to restart the game. In order to do this let's add a **Button – TextMeshPro** to the canvas.



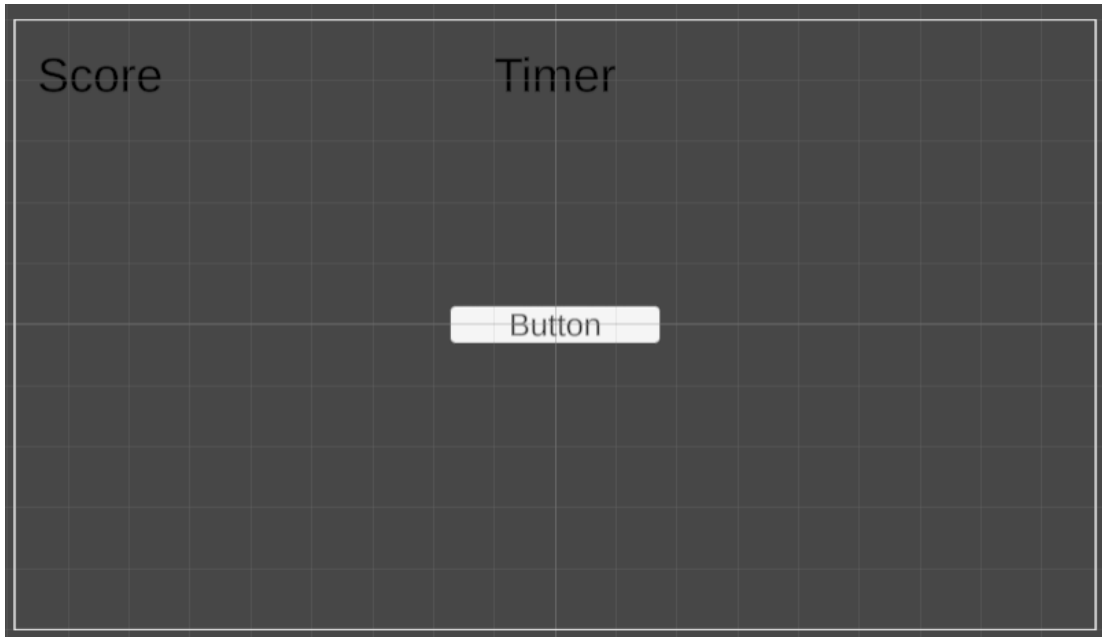
Rename the button to **Restart Button**.



**Note:** I'm not sure when this object gets added to our scene. I think it is when we add the button to the scene but there will now be an event system game object in our hierarchy. This object handles various UI events such as when the player presses a button.



2. Position the button so that it is at the center of the game window.



3. Inside the button game object in the hierarchy, we will find a text object. Set the text of that object to **Restart**. This will change the text that is on the button.



4. In the GameLogic script:

Add the library:

```
using UnityEngine.SceneManagement;
```

This library allows us to manage our scene. Mainly this is used for switching between scene or restart the scene that the player is currently in.

Let's add the variable:

```
//Restarting Game  
public GameObject restart_button;
```

This will just hold a reference to the button object we just created.

In Start():

```
//Disable the restart button at the start of the game.  
restart_button.SetActive(false);
```

We want to disable the button at the start since we want to only restart our game once it is finished.

Where the game timer has counted down add the line:

```
//Enable the restart button.  
restart_button.SetActive(true);
```

Since the game timer has counted down, we want to enable the button so that we can press it.

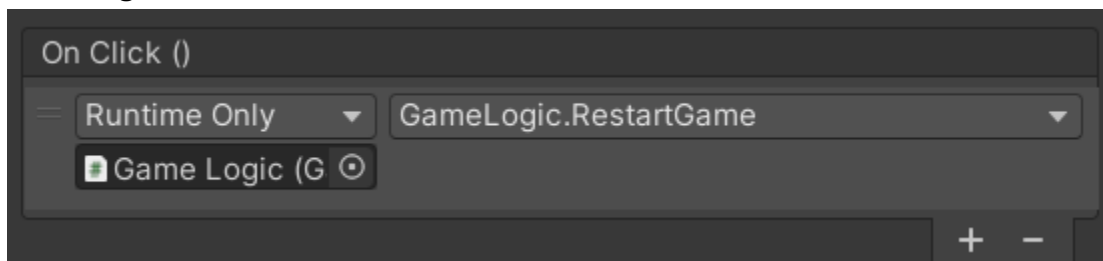
5. Lastly, we need to create a function that will restart our scene.

```
public void RestartGame()  
{  
    //Reloads the current scene, effectively restarting.  
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);  
}
```

6. Save the script and return to Unity. Hook up the restart button to the GameLogic script.



7. Next click on the restart button object. Find the **On Click ()** box and drag in the **Game Logic** empty game object into the **empty** slot. From the **drop down** you should be able to select **GameLogic > Restart Game**.



What this does is essentially tell the button that when it is clicked, it will call and run the RestartGame() function that is in the GameLogic script.

8. With everything set up we should be able to run the game. When we do, as soon as the game timer reaches zero, the restart button will appear. When we click on it, the scene will restart, and we will be able to play again.

## GitHub

1. Push the project to the GitHub repository.