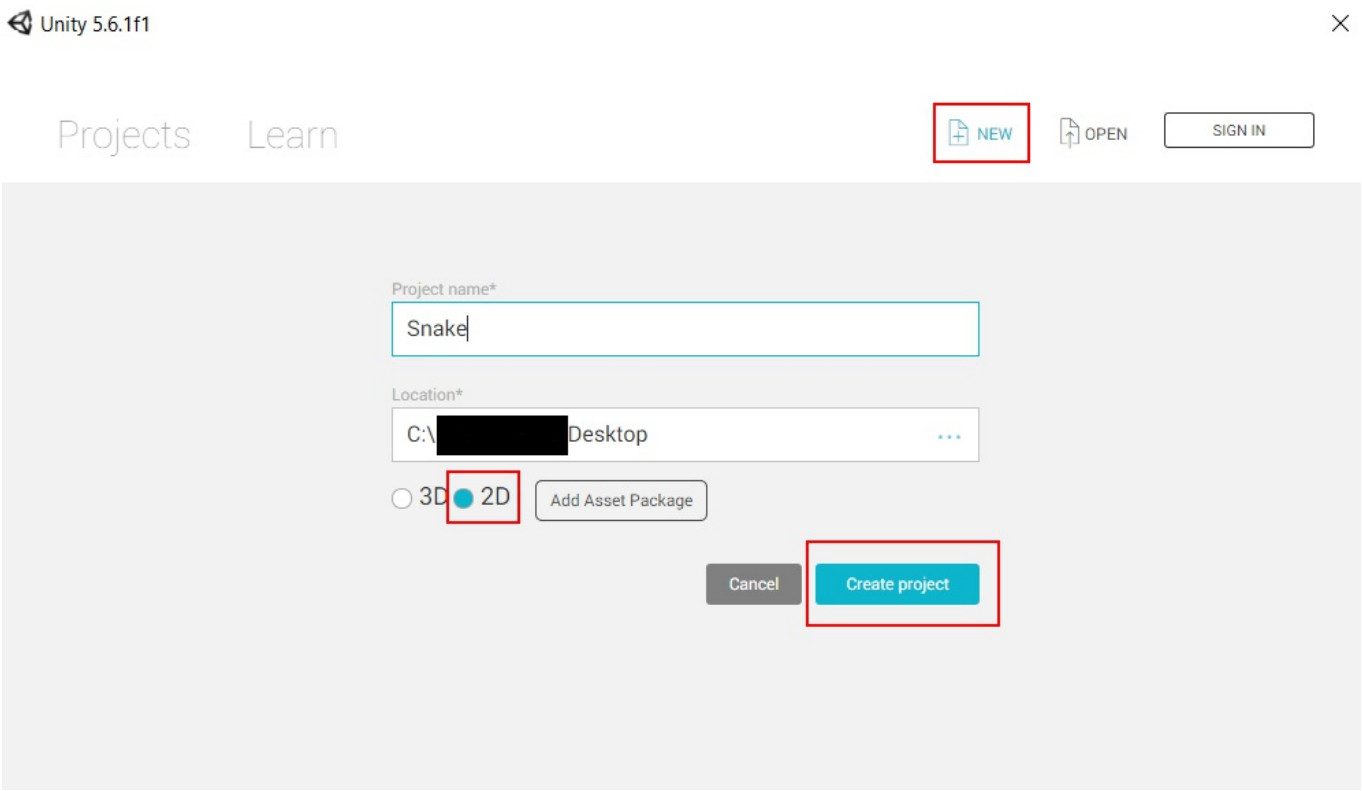
Snake

Starting a new Project

Once Unity is started, select **NEW**. We will name the project **Snake**, save the project on our **Desktop**, select **2D**, and click **Create Project**.



Setting up the Project

In the Project section, create the following four folders inside the Assets folder.

- _Scenes
- Prefabs
- Scripts

- Sprites

Note: The '" for Scenes is only used for ordering the folders and holds no significant meaning.

Importing the Assets

We will be using three images to represent the Objects in the game.

- line_horizontal.png and line_vertical.png would represent the walls
- image pixel.png would be used to represent the snake and food.

You can download the images below:

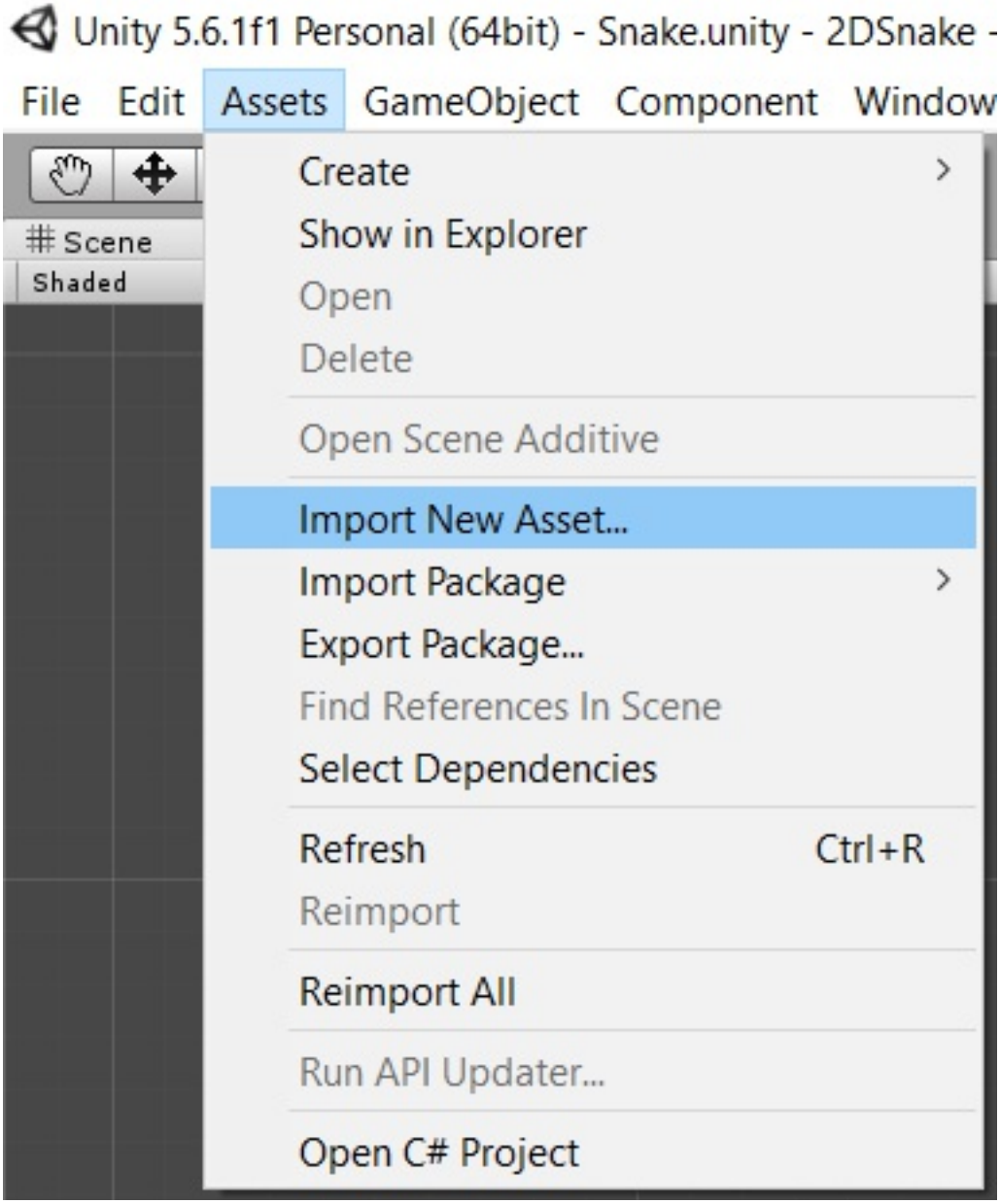
image files

There are several ways to add these images to the project.

- After clicking **Save As...**, save the images in the project's **Assets/Sprites** folder.
- Within Unity, go to the menu bar and navigate to **Assets->Import New Asset...** and select the image to be imported. The imported image should then be moved into the **Assets/Sprites** folder.
- Find the location the images were saved in then click on the image and drag-and-drop the image into the **Assets/Sprites**

folder within the Unity editor.

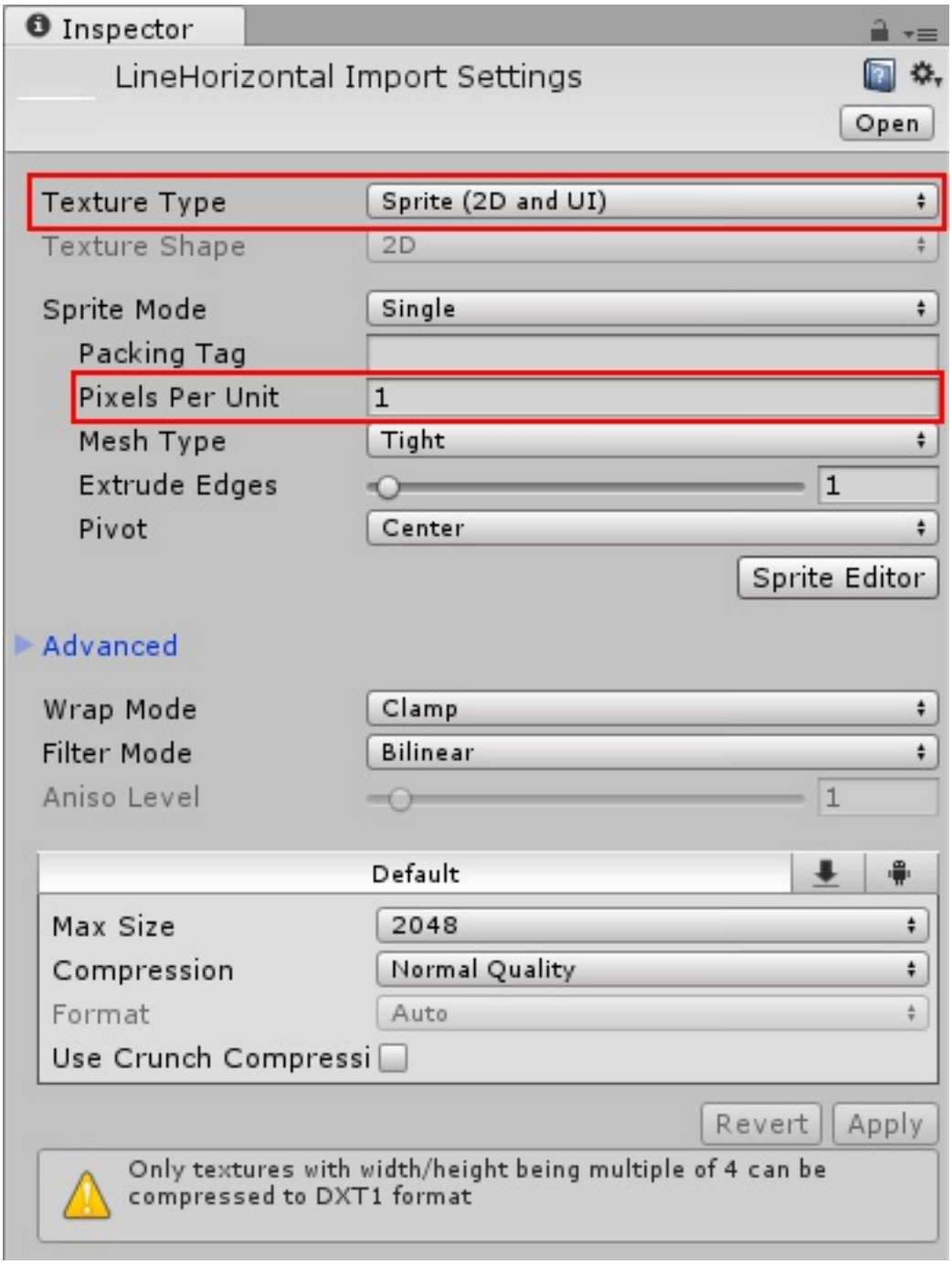
The second method is shown below:



Setting up the Sprites

Once the three images (textures) are added to the **Assets/Sprites** folder, select line_horizontal.png. This should bring up the **Import Settings** in the **Inspector** for the selected texture. If the **Texture Type** is not **Sprite (2D and UI)**, change the **Texture Type** to **Sprite**

(2D and UI) using the drop-down menu. Change **Pixels Per Unit** to **1**. Make sure to hit **Apply** at the bottom to apply these changes. This process should be repeated for all the texture currently in **Assets/Sprites**.

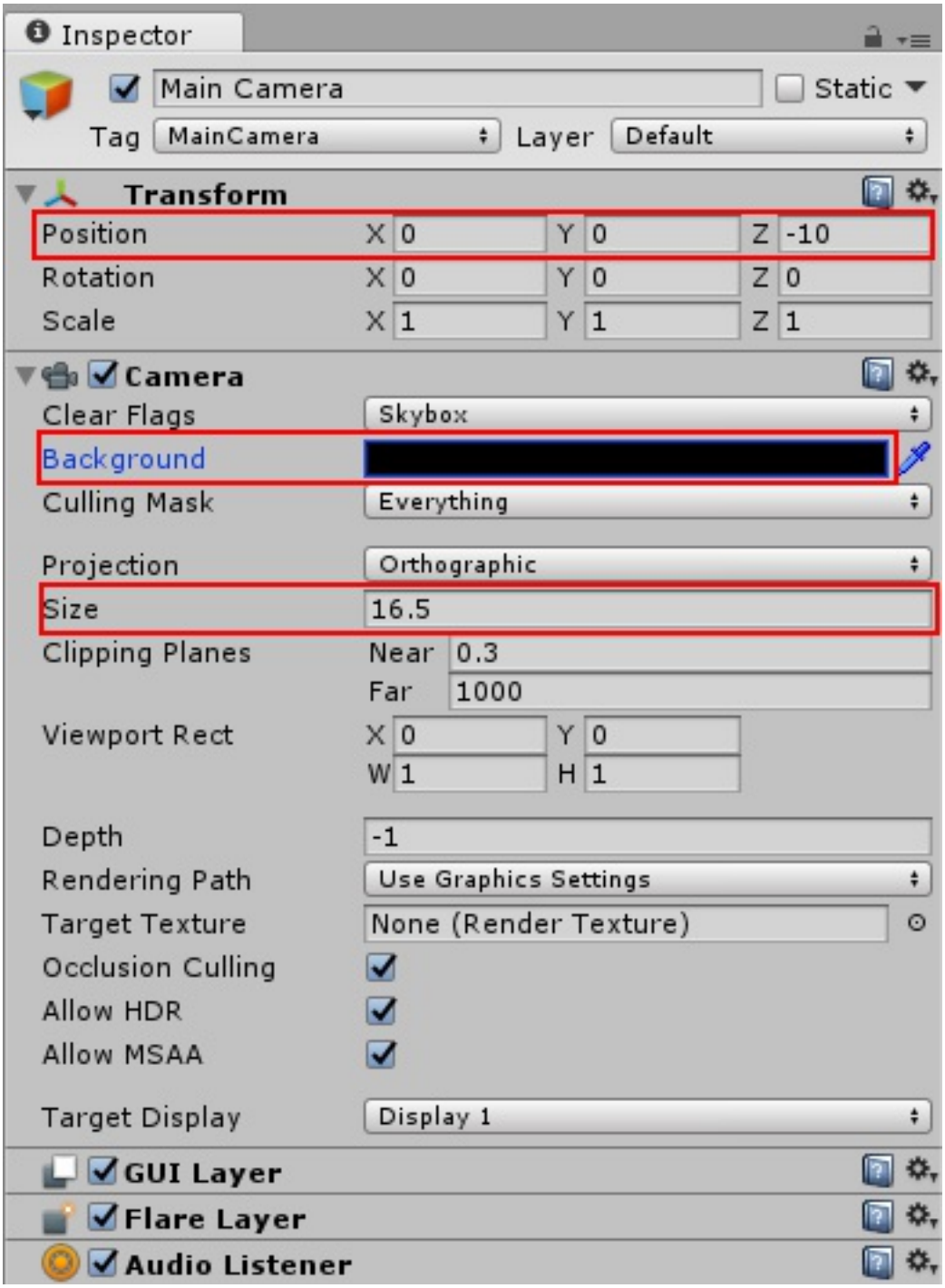


*Note: **Pixels Per Unit** is the ratio between one pixel in the image and one unit in the world. The **Snake** will have a size of 1x1 pixel, which should be 1 unit in the game world, This is why we will be using a*

Pixels Per Unit *value of 1 for all the textures.*

Setting up the Camera

From the **Hiearchy**, select the **Main Camera** to view its properties in the **Inspector**. From here, we will change **Camera**'s the **Background** color to black and adjust the (x,y,z) **Position** to **(0,0,-10)** and the **Size** to **16.5**.



*Note: **Size** is basically the zoom factor of the **Camera**. The **Size** of **16.5** fits for the sizes of our textures.*

Creating the Walls

To create the area for the Snake to traverse in, we will need to drag-and-drop the line_horizontal.png and line_vertical.png to the current

Scene's **Hierachy** or the **Scene** itself. Rename the lines to **WallTop**, **WallBottom**, **WallRight**, and **WallLeft**.

To rename the lines:

- right-click the GameObject in the Hierarchy for the rename option
- change the name of the Gameobject in the Inspector
- select the GameObject in the Hiearchy and then left-click (not double-click)

To position the four walls, we will go to the Transform component in the Hierachy and change the Position. The (x,y) values for each of the walls are shown below:

- **WallTop** (0,16)
- **WallBottom** (0,-16)
- **WallRight** (29,0)
- **WallLeft** (-29,0)

We will now add a **Box Collider 2D** to the walls to make them part of the physics world, otherwise, the Snake would be able to move through the walls. To add a **Box Collider 2D** to each of the walls, select one of the walls and in the **Inspector** click **Add Component**.

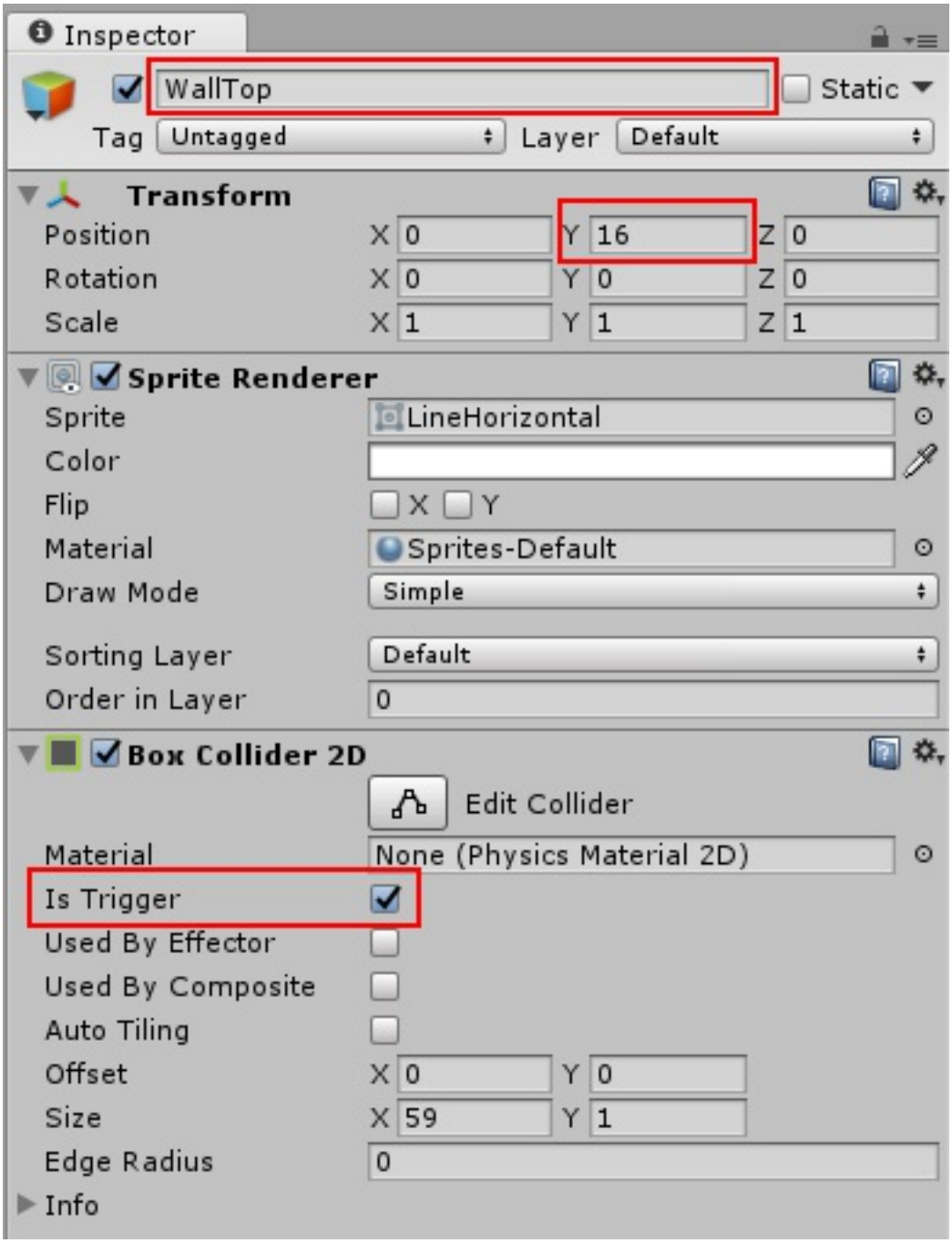
From **Add Component**, there are two ways to add a new component to the **GameObject**:

- Find the appropriate **Component Type** and then the specific **Component**
- Typing in the **Search** bar

Using the first method, **Add Component->Physics 2D->Box Collider 2D**. Once all the walls have a **Box Collider 2D**, enable the **Is Trigger** property for each of the walls' **Box Collider 2D**.

*Note: When a **Box Collider 2D** is added, Unity automatically resizes the collider to match the size of the **GameObject**'s sprite/texture/image. When **Is Trigger** is enabled, events can occur when a collision occurs with another **GameObject**.*

The end product for **WallTop**:



Saving the progress

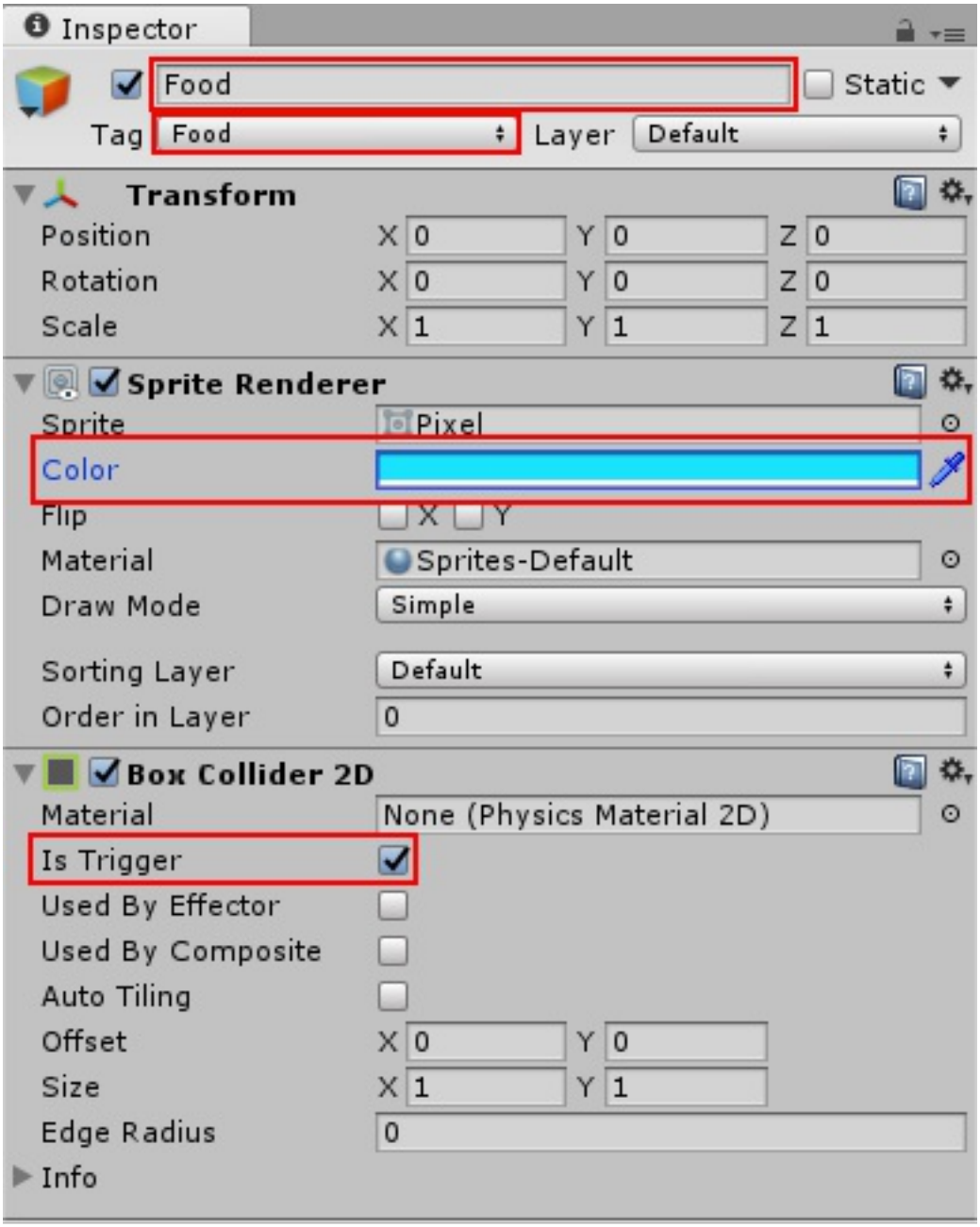
In order to save our current progress, we will go to **File->Save** or **File->Save Scene As...** or **Ctrl+S**. We will name this **Scene** as **Snake**. Be sure to save periodically!

Creating the Food Prefab

We will now create the food for the snake to eat, so it won’t go hangry. Just like how we created the walls, we will move the pixel.png to the the current **Scene**'s **Hierarchy** or the **Scene** itself. The pixel.png will be renamed **Food**, colored in **Cyan**, and given a **Box Collider 2D** with **Is Trigger** enabled.

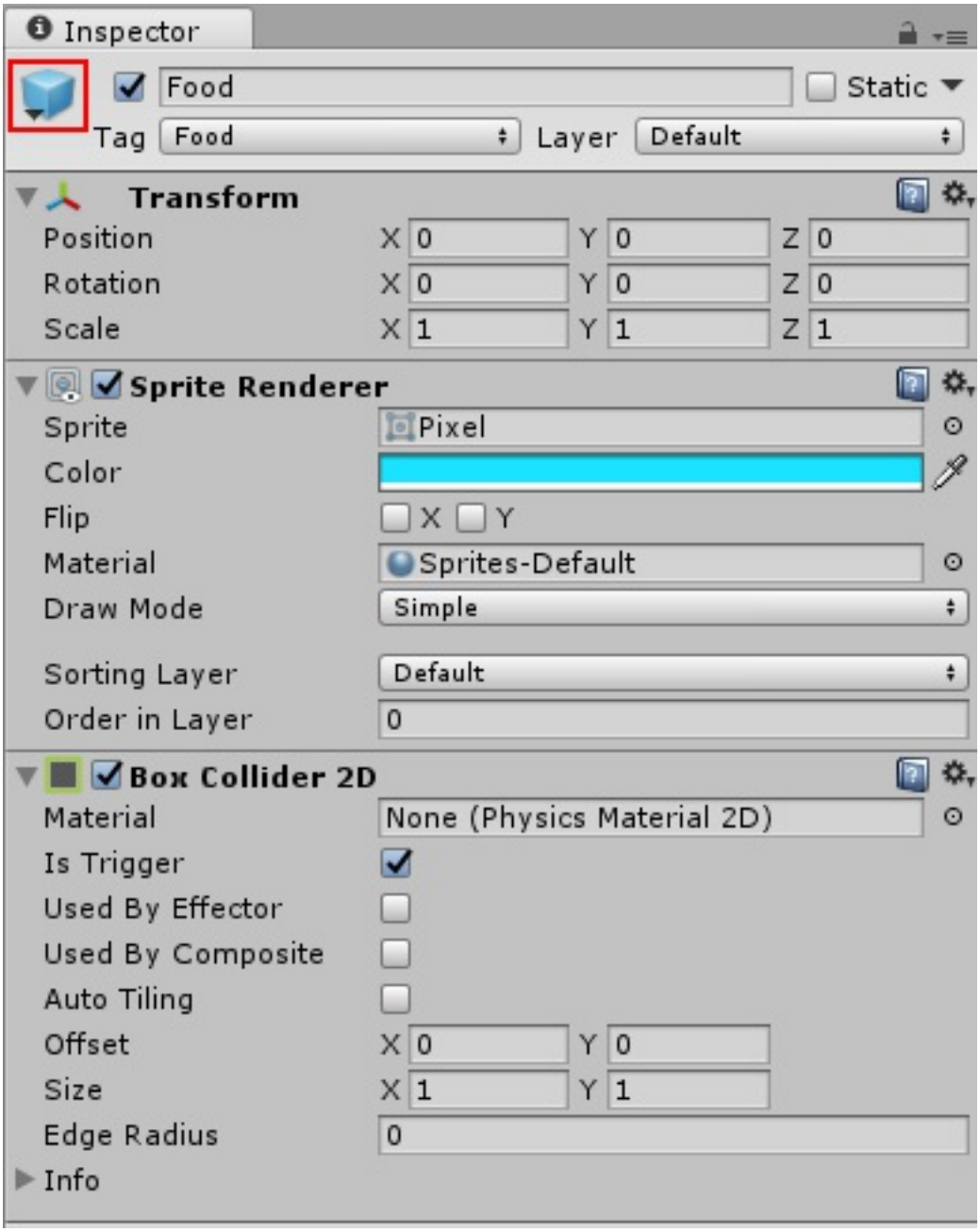
A new property we will be adding to **Food** is giving it a **Food Tag**. Right below the **Food** name, click on the **Untagged** to select **Add Tag...** from the down-down menu. This will change the **Inspector**'s view to **Tags & Layers**. Add a new tag with the name **Food**.

From the **Hierarchy**, select the **Food GameObject** and give it the **Food Tag**. The tag will be useful for identifying which **GameObject**'s can be eaten by the Snake later on.



Currently, one **Food** is in the **Scene**, but we will want to create multiple **Food GameObjects** with the same properties for the Snake to eat. By making the **Food GameObject** as a **Prefab**, we can create multiple instances of the same **Food** with the **Cyan** color, **Box Collider 2D** with **Is Trigger** enabled, and **Food Tag**, but still have the ability to change its properties like **Transform** for randomly spawning **Food**. This can be easily done by simply dragging the **Food GameObject** from the **Scene** into the **Assets/Prefabs** folder in the

Project panel.



*Note: Notice the cube icon next the **Food** name on the left-handside. This icon can be used to differentiate between a **GameObject** and a **GameObject** with an existing Prefab. GameObjects has a RGB cube while GameObjects with an existing Prefab has a Blue cube.*

Spawning Food

With the **Food Prefab** created, we are now ready to dive into some code to spawn some food for our Snake to eat. Since we will need this code to always be running in the current **Scene**, for simplicity, we will add this code to the **Main Camera** since it will always be with the current **Scene**. Looking at the **Main Camera**'s properties with the **Inspector** we will click on **Add Component->New Script** and give it the name **SpawnFood** with **CSharp** as the programming language for the **Script**. **Create and Add it**.

Double-clicking the **Script** will open **SpawnFood** in Unity's built-in MonoDevelop IDE for editing. The `Update()` function will not be used, so we will remove it from the code.

*Note: When creating a **Script**, Unity always creates two functions as*

part of the script. The `Start()` function is typically used for setting up and initialization while the `Update()` function is typically used for moving non-physics-related objects, timers, and receiving input, since the function is only called once every frame.

The **SpawnFood** script needs to know about other **GameObjects**. This includes **Food** to randomly spawn them and the four walls to randomly spawn food within the walls.

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

public class SpawnFood : MonoBehaviour {

    // Food to spawn
    public GameObject food;

    // Walls for spawning area
    public Transform wallTop;
    public Transform wallBottom;
    public Transform wallRight;
    public Transform wallLeft;

    // Use this for initialization
    void Start () {

```

```
    }
```

Note: We only need the Transform component of the four walls for the x and y positions, so we do not necessarily need the whole GameObject. If we did use the GameObject instead of the Transform, the code would be `wallTop.transform.position` *instead of* `wallTop.position`

We will create a `Spawn()` function to create an instance of the **Food Prefab** we created earlier. In order to find a random (x,y) value for the **Food** to spawn at we will use `Random.Range(float min, float max)`. This function will randomly return a value between the min and max values given. The value returned is a float (contains decimal points), so we will use `(int)` to round the result to an integer. This will prevent **Food** from spawning at weird positions like (1.568,2.654). Once we have the random (x,y), we can instantiate a new **Food Prefab** using `Instantiate(GameObject original, Vectore2 position, Quaternion rotation)`.

```
    // Spawning one food
    void Spawn() {
        // random x position between the left wall and ri
ght wall
        int x = (int)Random.Range(wallLeft.position.x, wa
llRight.position.x);
        // random y position between the left wall and ri
ght wall
```

```
        int y = (int)Random.Range(wallTop.position.y, wal
lBottom.position.y);

        // create food instance at (x,y) with a default r
otation (Quaternion,identity = no rotation)
        Instantiate(food, new Vector2(x,y), Quaternion.id
entity);
    }
```

We can now use the `Spawn()` function to periodically spawn Food. `InvokeRepeating(string methodName, float time, float repeatRate)` is a function made just for doing something periodically. After 3 seconds since `Start()` is executed, the `Spawn()` function will be called every 4 seconds.

```
    // Use this for initialization
    void Start () {
        // after 3 seconds, Food is spawned every 4 secon
ds by calling the Spawn() function
        InvokeRepeating ("Spawn", 3, 4);
    }
```

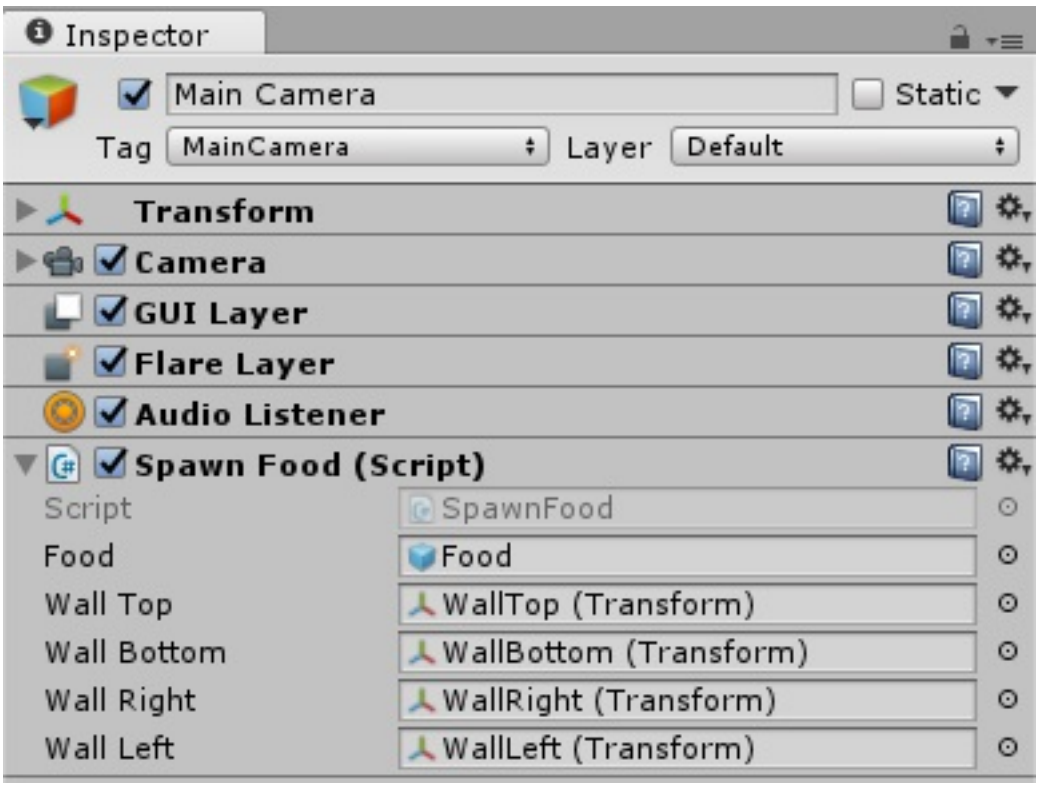
Final **SpawnFood Script**:

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

```
public class SpawnFood : MonoBehaviour {

    // Food to spawn
    public GameObject food;

    // Walls for spawning area
    public Transform wallTop;
    public Transform wallBottom;
    public Transform wallRight;
    public Transform wallLeft;

    // Use this for initialization
    void Start () {
        // after 3 seconds, Food is spawned every 4 secon
ds by calling the Spawn() function
        InvokeRepeating ("Spawn", 3, 4);
    }

    // Spawning one food
    void Spawn() {
        // random x position between the left wall and ri
ght wall
        int x = (int)Random.Range(wallLeft.position.x, wa
llRight.position.x);
        // random y position between the left wall and ri
ght wall
        int y = (int)Random.Range(wallTop.position.y, wal
```

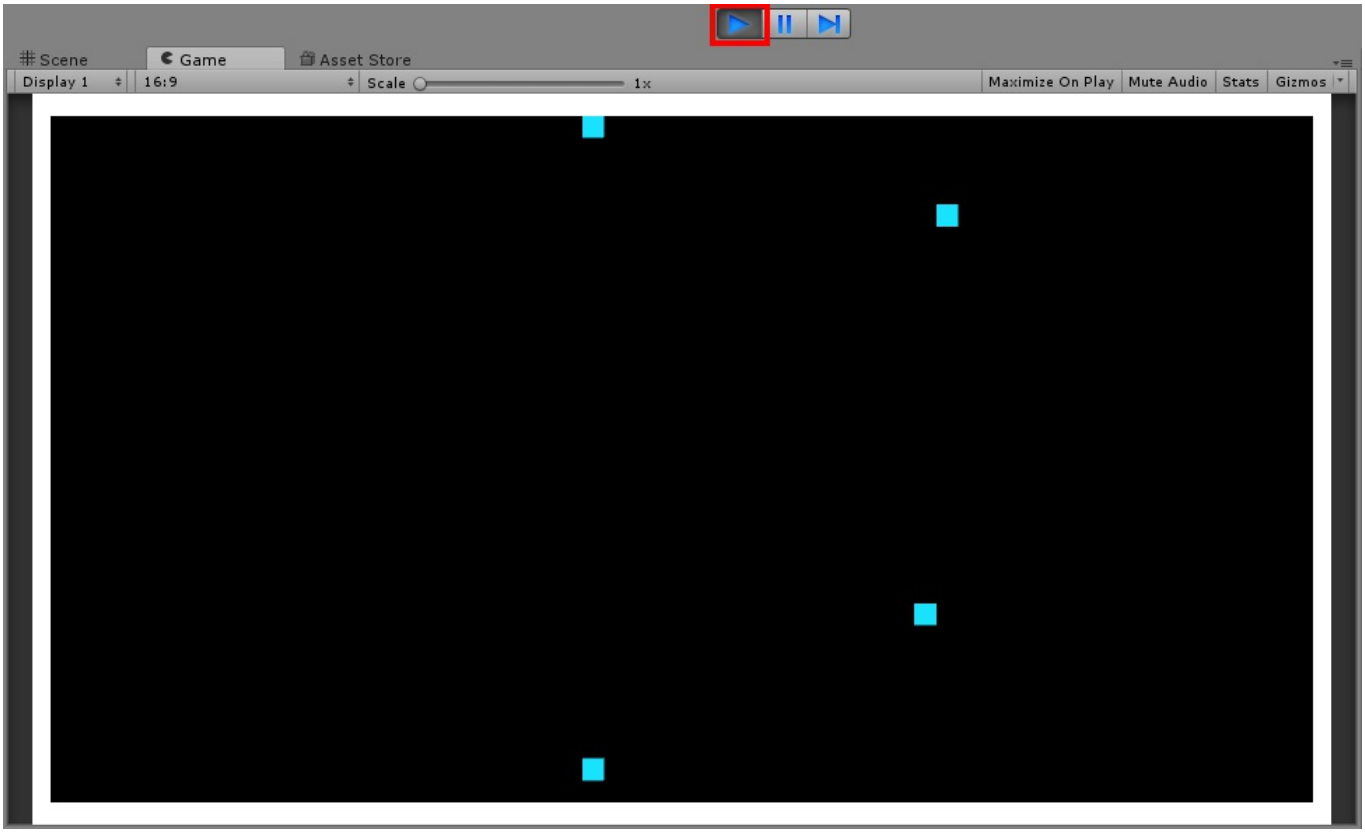
```
lBottom.position.y);

        // create food instance at (x,y) with a default r
otation (Quaternion,identity = no rotation)
        Instantiate(food, new Vector2(x,y), Quaternion.id
entity);
    }
}
```

Save the **Script** with **CTRL+S**. With no errors, looking at the **Main Camera** in the **Inspector** should show the 5 public variables defined in **SpawnFood**. Drag the GameObjects from the Hierarchy to their respective variable.



Now, press the **Play** button, and the **Food** should spawn within the

walls.



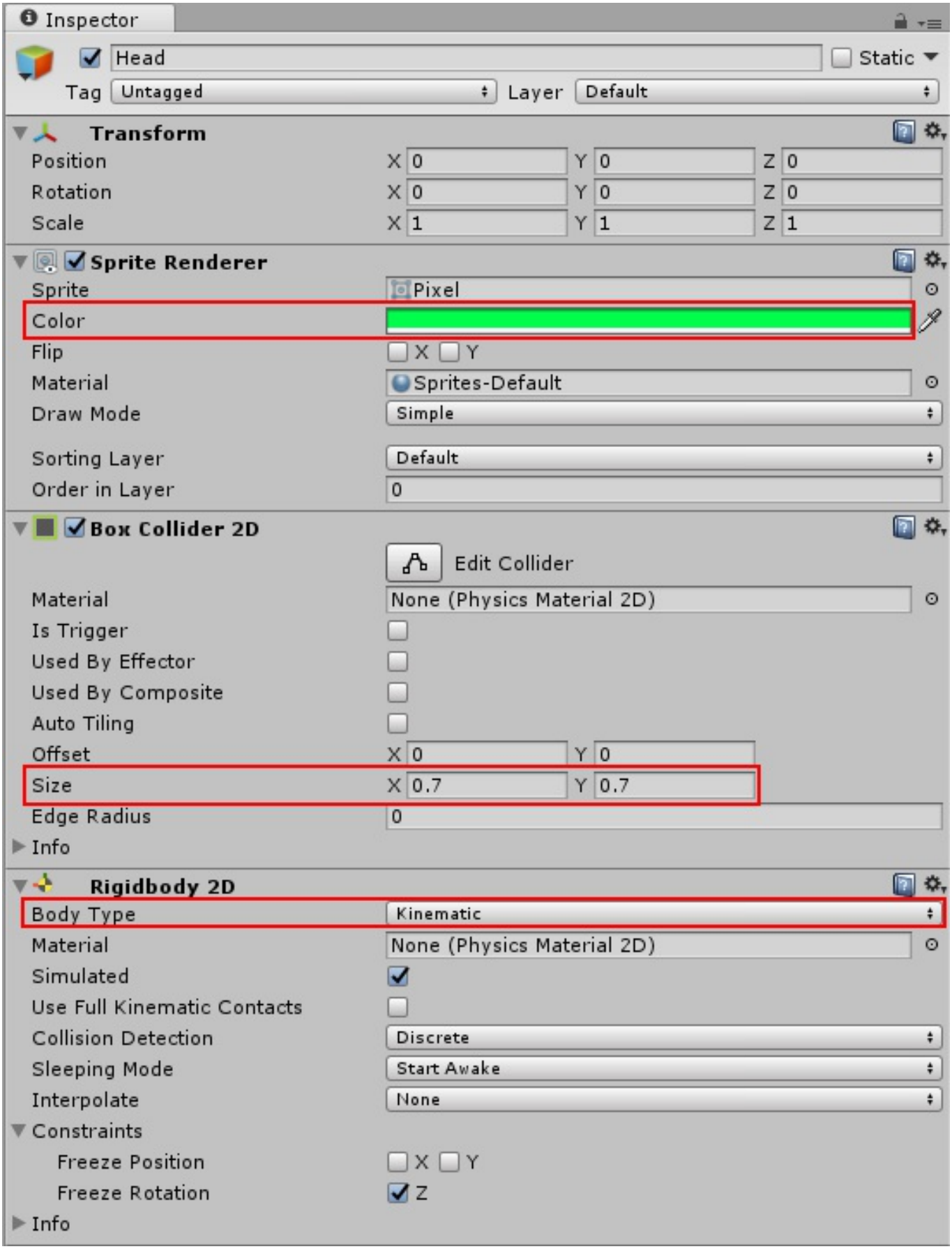
Save!

Creating the Snake’s Head

Like before, drag the pixel.png to the current **Scene**'s **Hierarchy** or the **Scene** itself. The pixel.png will be renamed **Head**, colored in **Green**, and given a **Box Collider 2D** (**Is Trigger** is **disabled**). This time around, we will change the (x,y) Size for the **Box Collider 2D** to **(0.7,0.7)**, so the **Head** will not collide with the growing **Tail** right next to it.

The Snake needs to move around according to the world of physics. This is done by the **Rigidbody 2D**. A **Rigidbody** will take care of many

aspects of physics like gravity, velocity, and movement. To add the **RigidBody 2D** to the Head we will select **Add Component->Physics 2D->RigidBody2D**. The only change we need to make to the **Rigidbody 2D** is to change the **Body Type** to **Kinematic**. This allows the **Head** to be unaffected by the physics of gravity or collisions.

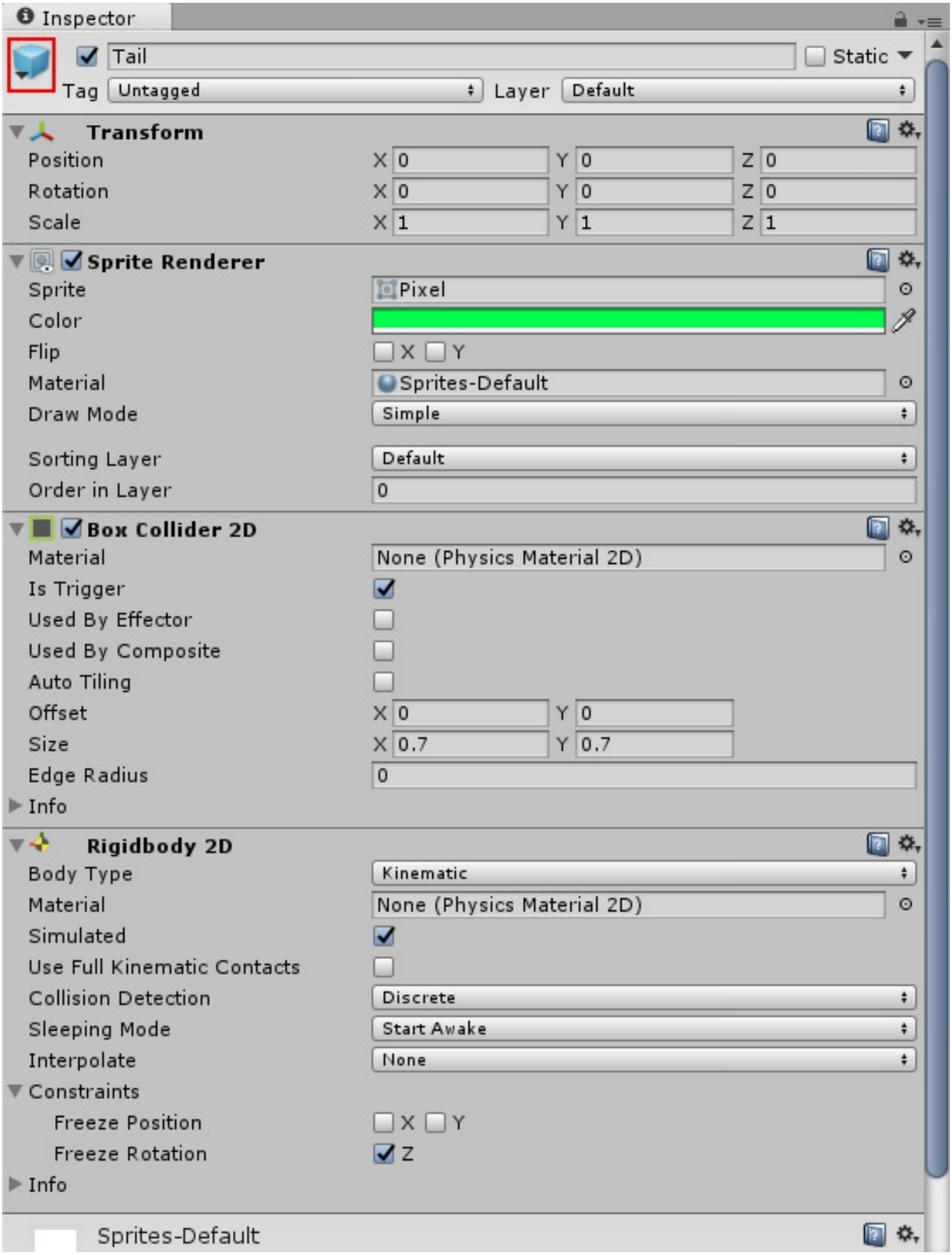


The **Head** and **Tail** of the Snake currently share the same properties. The main difference between the **Head** and **Tail** is the Head acts like

the brain, it does all the thinking, with several **Tail** elements simply follows behind the **Head** or other **Tail**s. This is shown below with the **Head** represented as X and **Tail**s represented as O:

```
0000X
```

Since the current **Head** will share the same properties as the **Tail**, we will duplicate the **Head** by selecting the **Head** in the **Hiearchy** and doing **CTRL+D** or right-clicking the **Head** and selecting **Duplicate**. The newly duplicated **Head** should be named **Head (1)**. We will rename **Head (1)** to **Tail** and drag-and-drop the **Tail** into the **Assets/Prefabs** folder to create a **Prefab** of the **Tail**.



Going back to the **Head**, we will add a new **Script** named **Snake**. We will add `using System.Linq;` to the top of the script for some List

functionality we will use later. In addition, we will add a `private Vector2 direction` variable only used by the current script. This variable will be used for the **Head**'s movement. Once again, we will use the `InvokeRepeating(string methodName, float time, float repeatRate)` in `Start()` to call the `Move()` function which will just change the **Head**'s **Transform** component according to the (x,y) value of the `direction` variable. `Start()` is typically used for initalization/setting up, so we will assign the `direction` variable to `Vector2.right` in order for the Snake to start moving to the right side when the game starts.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;

public class Snake : MonoBehaviour {

    // direction of Snake's movement
    private Vector2 direction;

    // Use this for initialization
    void Start () {
        // Snake starts moving to the right
        direction = Vector2.right;

        // Snake moves every 100ms by calling the Move()
```

```
function
        InvokeRepeating("Move", 0.1f, 0.1f);
    }

    // Update is called once per frame
    void Update () {

    }

    // Move used to move the Snake
    void Move()
    {
        // move Snake's Head into a new direction
        transform.Translate(direction);
    }
```

We will write the **Head**'s movement code in `Update()` since it is typically used for movement as mentioned above. The code is pretty straight forward. It will consist of multiple `if` statements to change the **Head**'s direction based on the Input detected. `Input.GetKey(KeyCode.RightArrow)`, for example, will be `true` if the right arrow key was pressed resulting in the new `direction` to be right so the **Head** will move to the right.

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

```
using System.Linq;

public class Snake : MonoBehaviour {

    // direction of Snake's movement
    private Vector2 direction;

    // Use this for initialization
    void Start () {
        // Snake starts moving to the right
        direction = Vector2.right;

        // Snake moves every 100ms by calling the Move() function
        InvokeRepeating("Move", 0.1f, 0.1f);
    }

    // Update is called once per frame
    void Update () {
        // Snake direction changes based on pressed key and NO TURNING BACK
        if (Input.GetKey(KeyCode.RightArrow))
        {
            direction = Vector2.right;
        }
        else if (Input.GetKey(KeyCode.LeftArrow))
        {
            direction = Vector2.left;
```

```
        }
        else if (Input.GetKey(KeyCode.UpArrow))
        {
            direction = Vector2.up;
        }
        else if (Input.GetKey(KeyCode.DownArrow))
        {
            direction = Vector2.down;
        }
    }

    // Move used to move the Snake
    void Move()
    {
        // move Snake's Head into a new direction
        transform.Translate(direction);
    }
```

If we save the current script and **Play** without any errors, we should see the Snake move it ay direction we want by using the arrow keys.

Snake’s Tail Movement

Snake’s Tail Movement

Before proceeding with implementing the code for the Snake’s **Tail**, there is a potential problem that may cause more work, time, and

problems.

Let’s take a look at a Snake with the **Head** and 4 **Tail** elements being represented below:

`OOOOX`

How would this Snake move if the **Head** goes to the right? The answer would obviously be:

```
OOOOX          \\ Step 0: initial Snake
OOOO X         \\ Step 1: Snake Head moves right one
OOO OX         \\ Step 2: Snake Tail moves right one
OO OOX         \\ Step 3: Snake Tail moves right one
O OOOX         \\ Step 4: Snake Tail moves right one
 OOOOX         \\ Step 5: Snake Tail moves right one
```

As you can see, the **Head** moves once and then *every* **Tail** element moves onces as well.
The next question: Should you implement the Snake’s Tail movement this way?

From how the question was worded, you can probably guess the answer would be: No
Why not? Snake is a fairly simple game, so if a player manages to have a Snake with 1000 **Tail** elements, *every* **Tail** element would need to move once. This may not seem like a huge problem since today’s

computers are pretty powerful, but implementing the **Tail**'s behavior this way would use up time and resources which can easily be remedied.

The provided solution may not be the *best* solution but it is definitely a better solution. The same example will be used.

```
OOOOX           \\ Step 0: initial Snake
OOOO X          \\ Step 1: Snake Head moves right one
^---^           \\         Movement to gap
 OOOOX          \\ Step 2: Snake Tail moves to the gap
```

This approach basically takes the last **Tail** element and moves it to the gap created when the **Head** moves. This solution is clearly less computationally intensive and saves time (less steps). With this solution in mind, we will begin implementing the Snake's **Tail** movement.

Completing the Snake's Movement

We will be using the `List` data structure to implement the solution. The `List` data structure is pretty self-explanatory. It holds a list of elements that can be manipulated using the `using System.Linq;` we added to the top of the **Snake Script** earlier.

Now, we will add one `public` variable and one `private` variable for the solution. `public GameObject tailPrefab` for the **Tail Prefab**

since it will be used in a very similar manner as the **Food Prefab**. `private List<Transform> tailList` for the **List** of the **Tail**s' **Transform**s. In `Start()`, we will instantiate the `tailList` by doing `tailList = new List<Transform>();`.

```
    // Snake's Tail Prefab for eating Food
    public GameObject tailPrefab;

    // keeping track of the tail
    private List<Transform> tailList = new List<Transform
>();

    // Use this for initialization
    void Start () {
        // initastiating the List
        tailList = new List<Transform>();

        // Snake starts moving to the right
        direction = Vector2.right;

        // Snake moves every 100ms by calling the Move()
function
        InvokeRepeating("Move", 0.1f, 0.1f);
    }
```

In order to implement the solution, we first need to save the Head's current position before the `transform.Translate(direction)` moves

since this would be the position of the gap that is created. The `if` statement is for checking if the Snake has a Tail because the solution would not work if the Snake does not have at least one Tail (no Tail to swap to the gap). The rest of the code does exactly what the solution does.

```
    // Move used to move the Snake
    void Move()
    {
        // save current position of the Snake's Head
        Vector2 v = transform.position;

        // move Snake's Head one space based on the new d
irection (creates gap)
        transform.Translate(direction);

        if (tailList.Count > 0)
        {
            // move last Tail element to where the Head w
as (fills gap)
            tailList.Last ().position = v;

            // add Tail to the front of the list
            tailList.Insert (0, tailList.Last ());

            // remove Tail from the back of the list
            tailList.RemoveAt (tailList.Count - 1);
        }
```

Feeding the Snake

In order to grow the Snake's **Tail**, we will be using the `OnTriggerEnter2D()` function which executes the function's code when a collision information is recieved by using the enabled **Is Trigger** property of the **Box Collider 2D**. We will also be extending the solution we came up with; when the Snake eats **Food**, the **Tail** will get added to the gap rather than the end of the Snake. Before going to the `OnTriggerEnter2D()` function, we will need to add a `private bool ate`, a true and false flag to check if the Snake ate **Food**.

```
    // flag for checking if the Snake ate Food
    private bool ate = false;

    // Use this for initialization
    void Start () {
        // Snake starts with being hungry
        ate = false;

        // initalize the List
        tailList = new List<Transform>();

        // Snake starts moving to the right
        direction = Vector2.right;

        // Snake moves every 100ms by calling the Move()
```

```
function
        InvokeRepeating("Move", 0.1f, 0.1f);
    }
```

The `OnTriggerEnter2D()` function is a special function like `Start()` and `Update()` since Unity automatically calls them when the game is running. The argument `Collider2D other` is the other **GameObject** with **Is Trigger** enabled. Along with using the **Food Tag** from earlier, we can check if the other **GameObject** has the **Food Tag**, this will set the `ate` flag to true and destroy the **Food Prefab**, otherwise, the **Head** would be destroyed for colliding with anything else.

```
    // Trigger event when another object collides with th
is object
    void OnTriggerEnter2D(Collider2D other)
    {
        // checks to see if collided object has the Food
tag
        if (other.CompareTag("Food"))
        {
            // set ate flag to true
            ate = true;
            // destroy the Food
            Destroy (other.gameObject);
        }
        // Snake is detroyed if collided with any other o
bject
```

```
        else
        {
            Destroy (gameObject);
        }
    }
```

Now, we will utilize the `ate` flag in the `Move()` function where we are already modifying the Snake's **Tail**. If the `ate` flag is true, we just need to instantiate the Tail much like the instantiating the Food earlier. In addition, we will need to add this Tail to the list and reset the `ate` flag.

```
    // Move used to move the Snake
    void Move()
    {
        // save current position of the Snake's Head
        Vector2 v = transform.position;

        // move Snake's Head one space based on the new d
irection (gap)
        transform.Translate(direction);

        // check if the ate flag is true
        if (ate) {
            // loads the Tail Prefab to be placed where t
he Snake's Head was (fills gap)
            GameObject gameObject = (GameObject)Instantia
```

```
te (tailPrefab, v, Quaternion.identity);

            // adds the loaded Tail Prefab to the Tail li
st
            tailList.Insert (0, gameObject.transform);

            // Snake finished eating setting ate flag bac
k to false
            ate = false;
        }
        // check if the Snake have a Tail
        else if (tailList.Count > 0)
        {
            // move last Tail element to where the Head w
as
            tailList.Last ().position = v;

            // add Tail to the front of the list
            tailList.Insert (0, tailList.Last ());

            // remove Tail from the back of the list
            tailList.RemoveAt (tailList.Count - 1);
        }
    }
```

Final **Snake Script**:

Note: The Script below contains additional code not detailed in the

README. This code is commented out for that reason.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;

public class Snake : MonoBehaviour {

    // direction of Snake's movement
    private Vector2 direction;

    // Snake's Tail Prefab for eating Food
    public GameObject tailPrefab;

    // keeping track of the tail
    private List<Transform> tailList = new List<Transform
>();

    // flag for checking if the Snake ate Food
    private bool ate = false;

    // Use this for initialization
    void Start () {
        // Snake starts with being hungry
        ate = false;

        // initalize the List
```

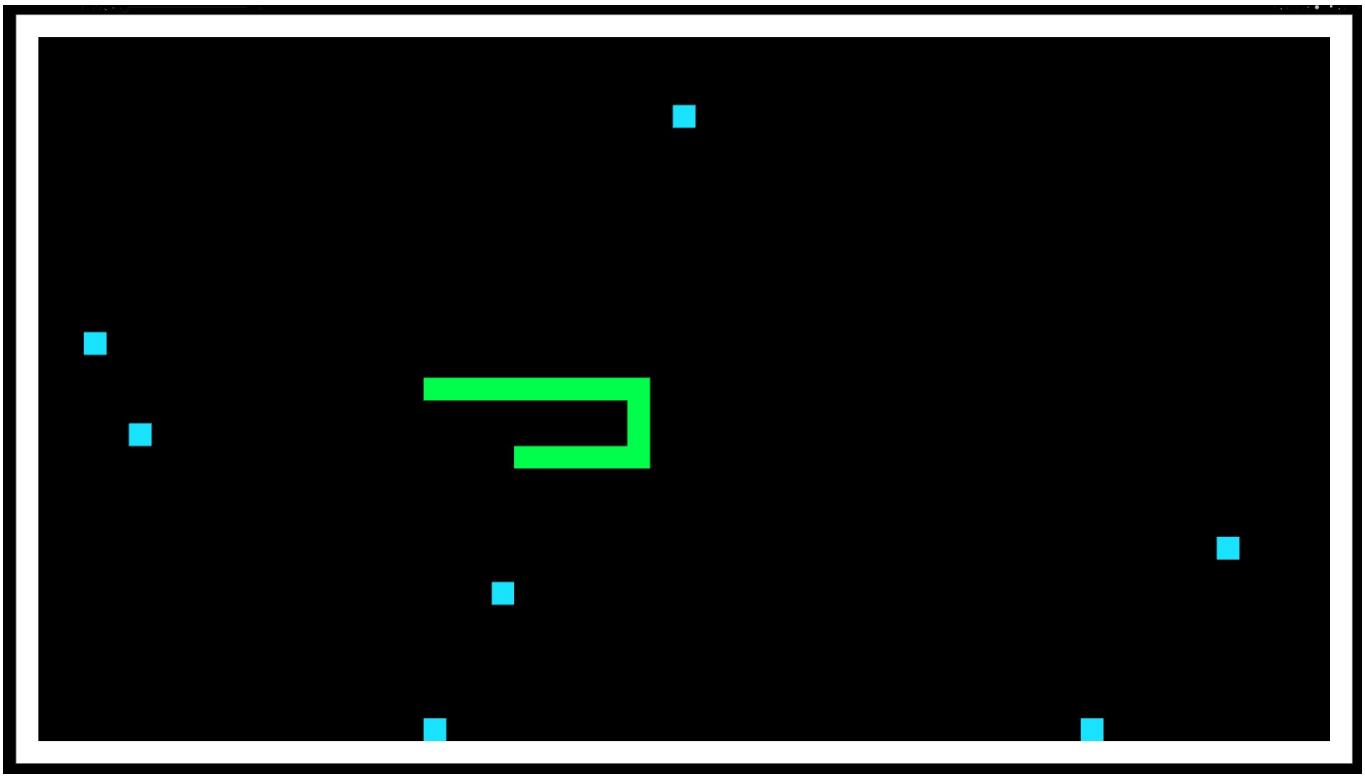
```
        tailList = new List<Transform>();

        // Snake starts moving to the right
        direction = Vector2.right;

        // Snake moves every 100ms by calling the Move() function
        InvokeRepeating("Move", 0.1f, 0.1f);
    }

    // Update is called once per frame
    void Update () {
        // Snake direction changes based on pressed key and NO TURNING BACK
        if (Input.GetKey(KeyCode.RightArrow) && (direction != Vector2.left))
        {
            direction = Vector2.right;
        }
        else if (Input.GetKey(KeyCode.LeftArrow) && (direction != Vector2.right))
        {
            direction = Vector2.left;
        }
        else if (Input.GetKey(KeyCode.UpArrow) && (direction != Vector2.down))
        {
            direction = Vector2.up;
```

```
        }
        else if (Input.GetKey(KeyCode.DownArrow) && (dire
ction != Vector2.up))
        {
            direction = Vector2.down;
        }
    }

    // Move used to move the Snake
    void Move()
    {
        // save current position of the Snake's Head
        Vector2 v = transform.position;

        // move Snake's Head one space based on the new d
irection (gap)
        transform.Translate(direction);

        // check if the ate flag is true
        if (ate) {
            // loads the Tail Prefab to be placed where t
he Snake's Head was (fills gap)
            GameObject gameObject = (GameObject)Instantia
te (tailPrefab, v, Quaternion.identity);

            // adds the loaded Tail Prefab to the Tail li
st
            tailList.Insert (0, gameObject.transform);
```

```
            // Snake finished eating setting ate flag bac
k to false
            ate = false;
        }
        // check if the Snake have a Tail
        else if (tailList.Count > 0)
        {
            // move last Tail element to where the Head w
as
            tailList.Last ().position = v;

            // add Tail to the front of the list
            tailList.Insert (0, tailList.Last ());

            // remove Tail from the back of the list
            tailList.RemoveAt (tailList.Count - 1);
        }
    }

    // Trigger event when another object collides with th
is object
    void OnTriggerEnter2D(Collider2D other)
    {
        // checks to see if collided object has the Food
tag
        if (other.CompareTag("Food"))
        {
```

```
            // set ate flag to true
            ate = true;
            // destroy the Food
            Destroy (other.gameObject);
        }
        // Snake is detroyed if collided with any other o
bject
        else
        {
            Destroy (gameObject);
        }
    }
}
```

PLAY SNAKE!



CTRL+S

Building the Game

Simply go to **File->Build Settings**, selected **PC** as the **Platform**, add the **Snake Scene** to **Scenes In Build** and click **Build** or **Build And Run**