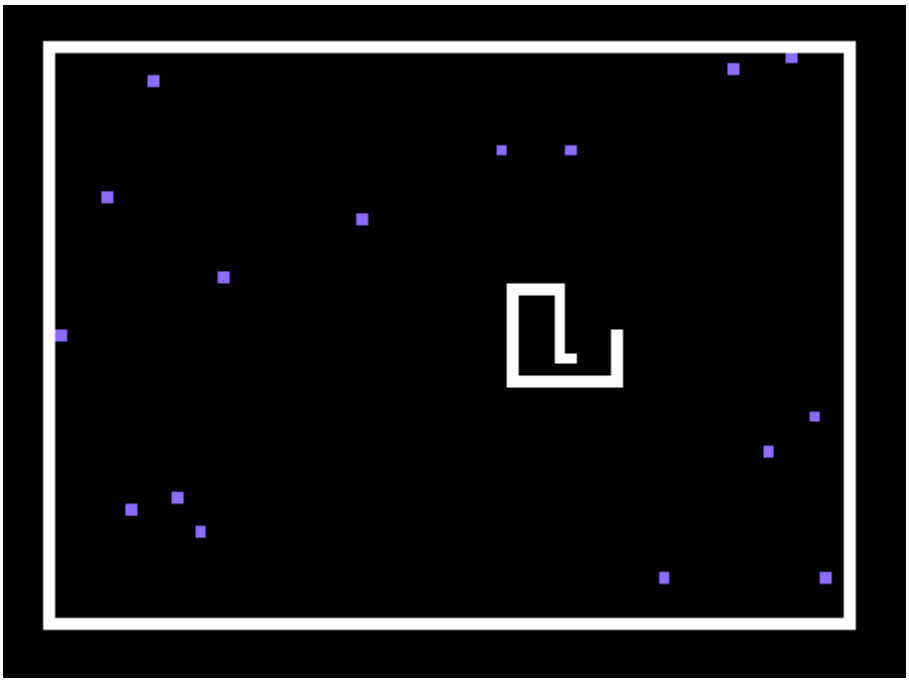
Unity 2D Snake Tutorial



Foreword

This Tutorial will explain how to make a simple 2D Snake Game in Unity. Snake is an arcade game that was created back in the 1970's. And like most arcade games it's still a whole lot of fun and easy to develop, which makes it a great game for a Tutorial.

Requirements

Knowledge

Our Tutorial does not require any special skills except some knowledge about the Unity basics like GameObjects and Transforms. Even if you don't know those concepts yet, the Tutorial should still be doable.

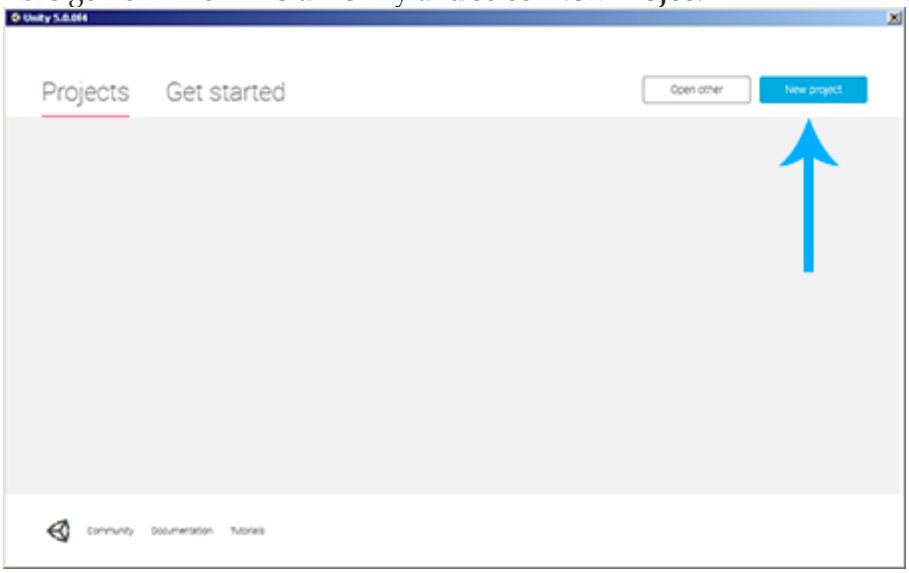
Feel free to read our easier Unity Tutorials like Unity 2D Pong Game to get used to this amazing game engine.

Unity Version

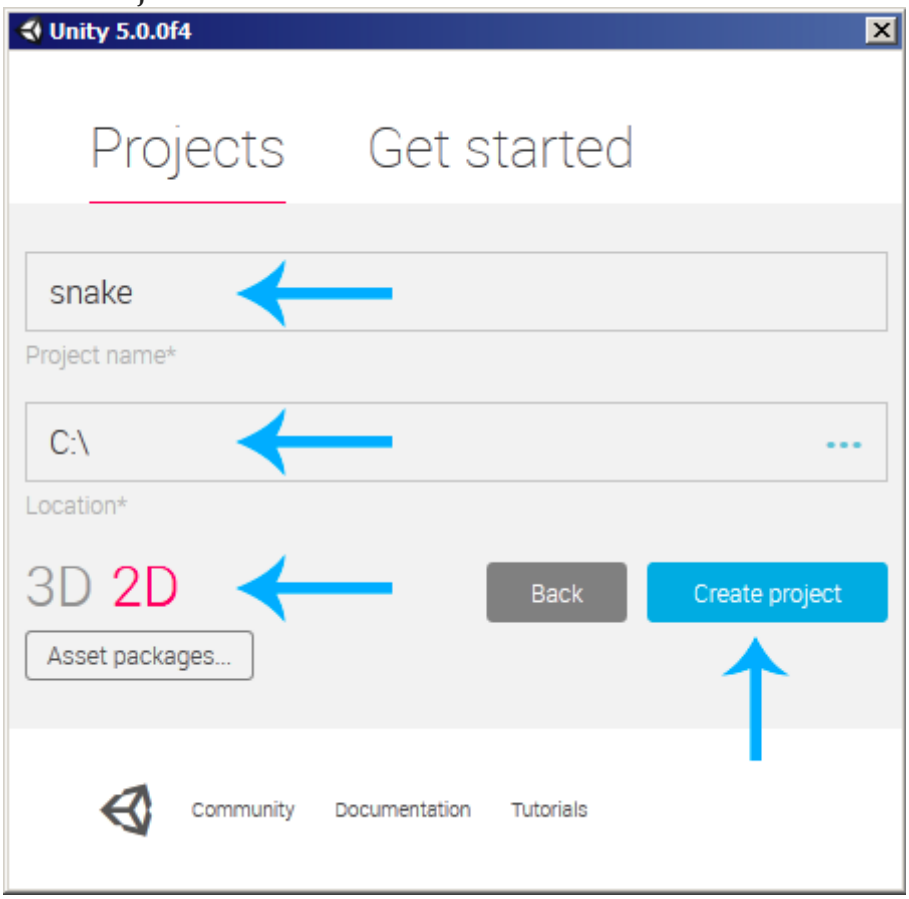
Our Snake Tutorial will use **Unity 5.0.0f4**. Newer versions should work fine as well, older versions may or may not work. The free version of Unity 5 now comes with all the engine features, which makes it the recommended version.

Project Setup

Let's get to it. We will start Unity and select **New Project**:

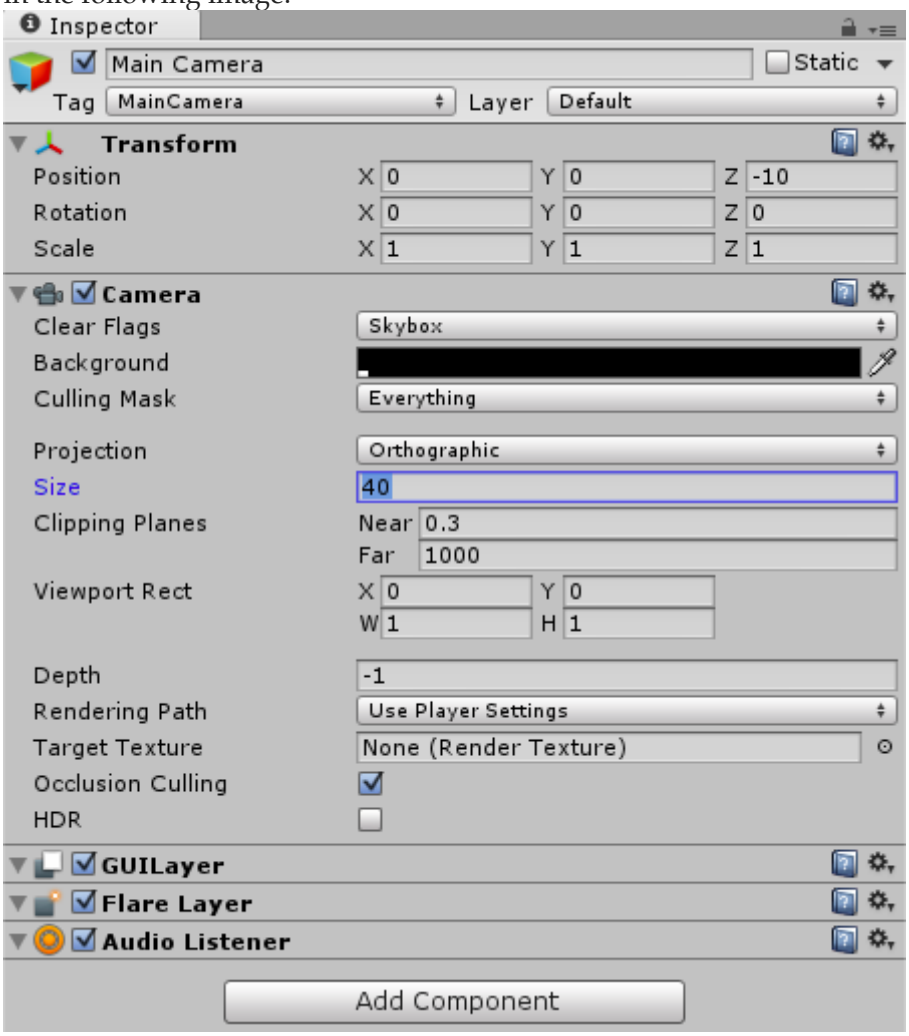


We will name it **snake**, select any location like **C:**, select **2D** and click **Create Project**:



If we select the **Main Camera** in the **Hierarchy** then we can set the Background Color to black, adjust the **Size** and the **Position** like shown

in the following image:



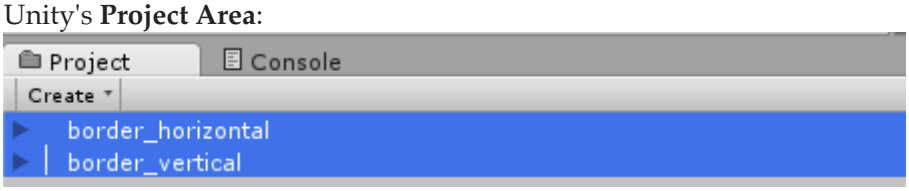
*Note: **Size** is pretty much the zoom factor.*

Adding Borders

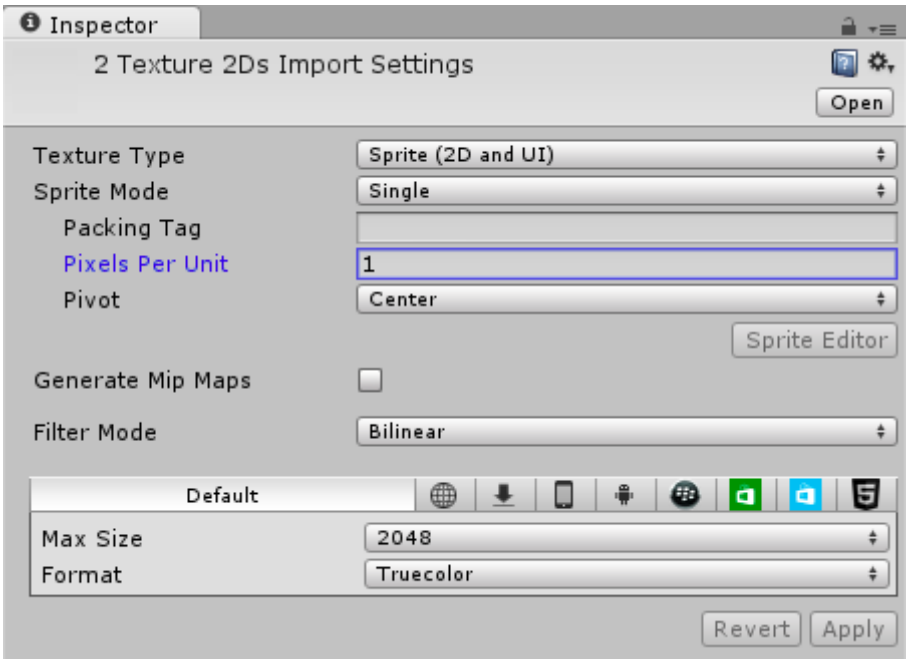
We will use one horizontal and one vertical white line image for our borders:

- border_horizontal.png
- border_vertical.png
  *Note: right click each link, select **Save As...** and save the images in the project's **Assets** folder.*

Once we have them in our Project's Assets folder, we can select them in Unity's **Project Area**:

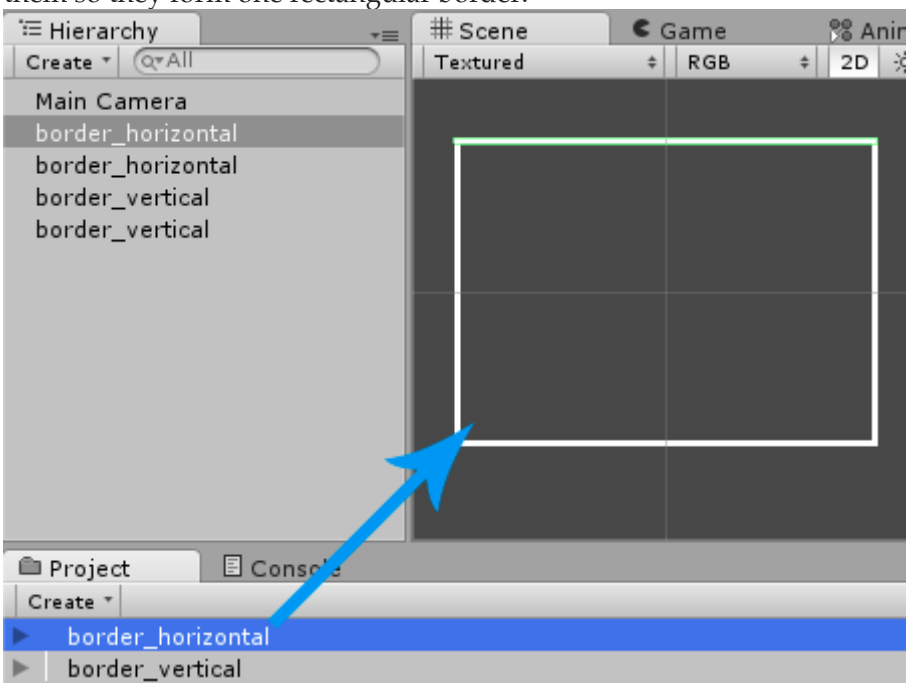


Afterwards we can change their Import Settings in the **Inspector** to make them appear in the right size with the right looks:



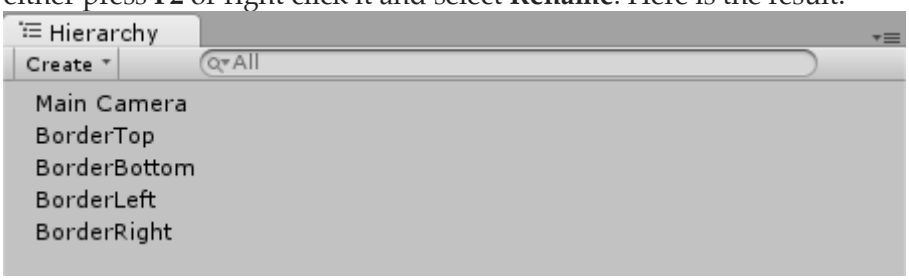
*Note: Pixels Per Unit is the ratio between one pixel in the image and one unit in the world. The Snake will have a size of 1x1 pixel, which should be 1 unit in the game world. This is why we will use a **Pixels Per Unit** value of **1** for all our textures.*

Now we can drag each border image into the scene **twice** and position them so they form one rectangular border:

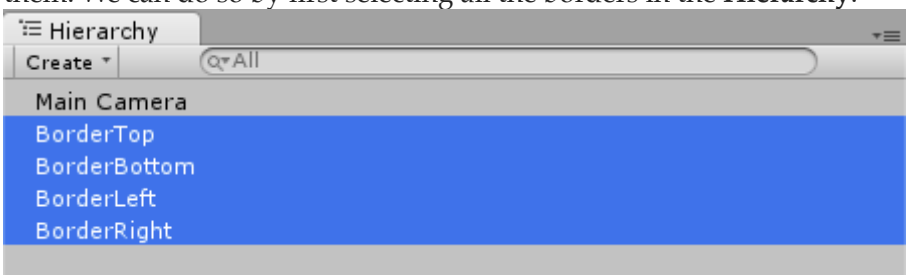


Note: the horizontal border image is used for the top and bottom borders. The vertical border image is used for the left and right borders.

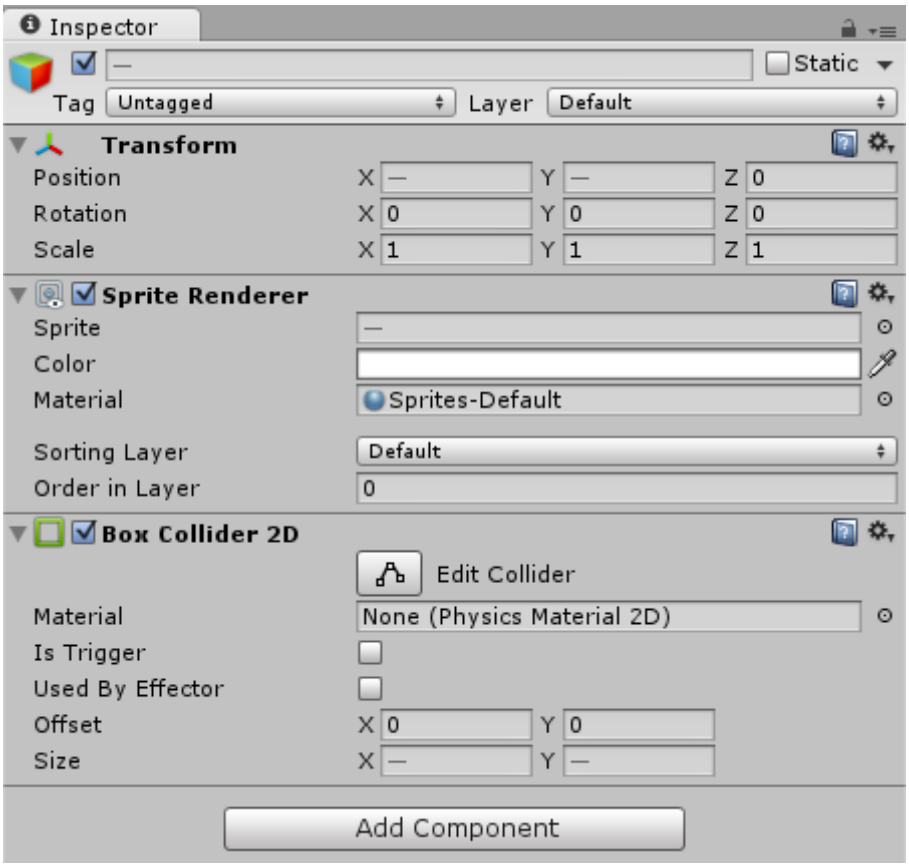
Let's rename the borders to **BorderTop**, **BorderBottom**, **BorderLeft** and **BorderRight**. We will select one after another in the **Hierarchy** and then either press **F2** or right click it and select **Rename**. Here is the result:



Right now the borders are just images in the game. They may look like borders, but they are not part of the physical world just yet. If we want the Snake to collide with the borders, then we have to add **Colliders** to them. We can do so by first selecting all the borders in the **Hierarchy**:



Right now the borders are just images, they aren't really borders. The snake could walk right through them because they are not part of the physics world yet. Let's take a look at the **Inspector** and select **Add Component->Physics 2D->Box Collider 2D**. And since we have all borders selected right now, this will add a **Box Collider 2D** to each of them:



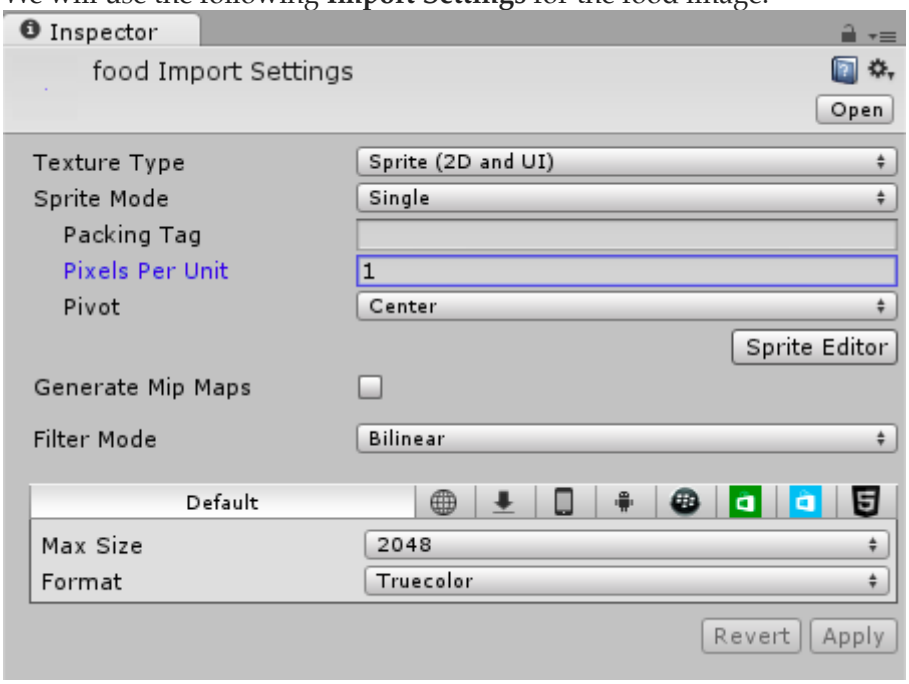
And that's all there is to it. We just created the borders for our game without writing a single line of code, thanks to this powerful game engine.

Creating the Food Prefab

We don't want our snake to get hungry, so let's randomly spawn some food in the game. As usual we will start with a food image, in our case a colored pixel:

- food.png
  *Note: right click on the link, select **Save As...** and save it in the project's **Assets** folder.*

We will use the following **Import Settings** for the food image:



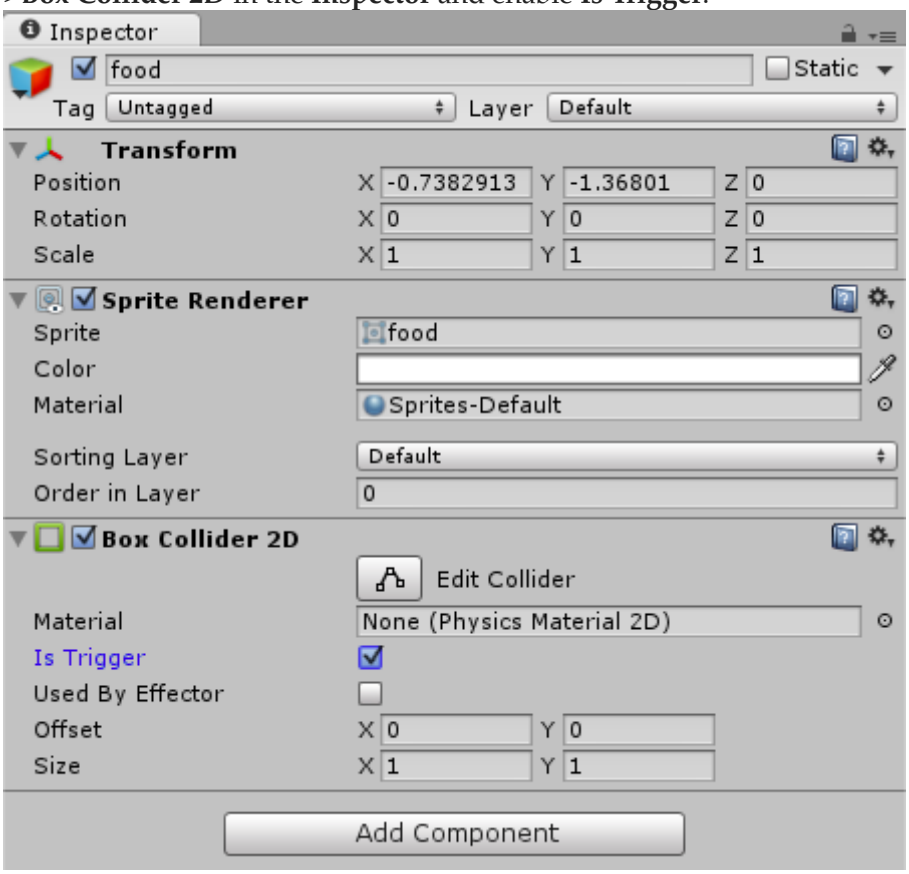
Alright, let's drag the food image into the Scene to create a **GameObject**:

The Snake should receive some kind of information whenever it collides with food. This means that the food has to be part of the physics world, which can be done with a Collider.

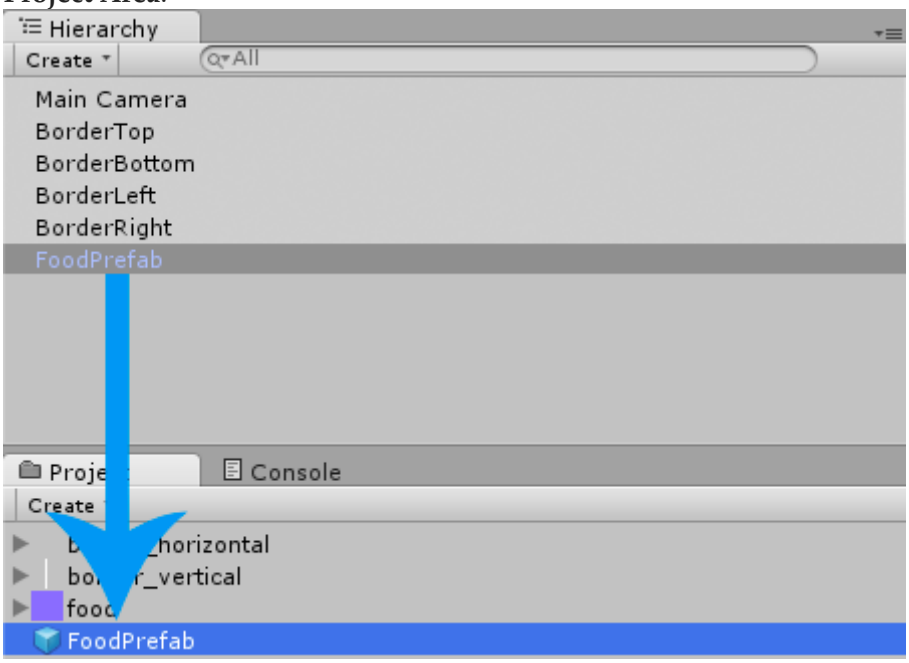
A GameObject without a Collider is just a visual thing, its not part of the physics world. A GameObject with a Collider is part of the physics world, just like a wall. It will cause other things to collide with it, and it will trigger the **OnCollisionEnter2D** event. A GameObject with a Collider that has **Is Trigger** checked will **not** cause other things to collide with it, but it will still trigger the **OnTriggerEnter2D** event.

The Snake should get notified when it walks through food, but it's not supposed to collide with it like if the food was a wall. So let's select the food in the **Hierarchy** and then choose **Add Component->Physics 2D-**

>**Box Collider 2D** in the **Inspector** and enable **Is Trigger**:



Okay, now we don't want the food to be in the Scene from the beginning. Instead we want a Prefab so that we can **Instantiate** it whenever we need it. In order to create a **Prefab**, all we have to do is rename the food to **FoodPrefab** and then drag it from the **Scene** into the **Project Area**:

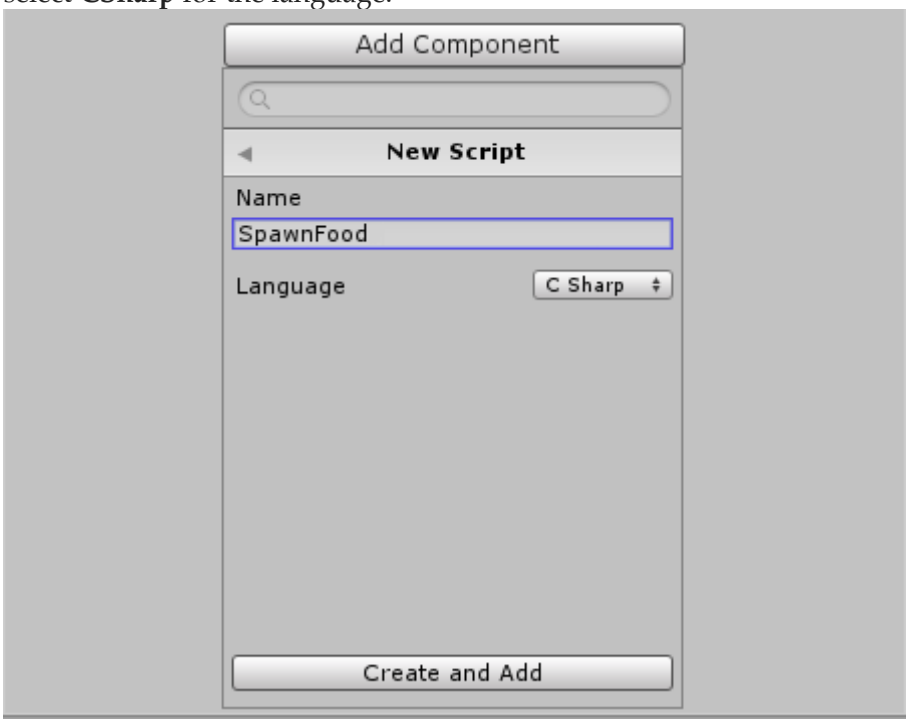


Now we can delete the **FoodPrefab** GameObject from the **Hierarchy**, because we don't want it to be in the game world just yet.

Spawning Food

Let's spawn new food at some random position every few seconds. This kind of behavior can be implemented with a Script, so let's create a **SpawnFood** Script. The Script should be in the Scene all the time, so we will keep it simple and add it to the **Main Camera** *(because it will be in the*

Scene all the time, too). Let's select the **Main Camera** in the **Hierarchy** and then click on **Add Component->New Script**, name it **SpawnFood** and select **CSharp** for the language:



Afterwards we will double click it in the **Project Area** to open it:

```
using UnityEngine;
using System.Collections;

public class SpawnFood : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

We won't need the **Update** function, so let's remove it:

```
using UnityEngine;
using System.Collections;

public class SpawnFood : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }
}
```

Our Script needs to know where the food Prefab is. We will add a public variable of type **GameObject**:

```
using UnityEngine;
using System.Collections;

public class SpawnFood : MonoBehaviour {
    // Food Prefab
    public GameObject foodPrefab;

    // Use this for initialization
```

```
    void Start () {

    }
}
```

The food should be spawned within the borders, not outside. So we will also need a variable for each of the borders so that our Script knows their positions:

```
using UnityEngine;
using System.Collections;

public class SpawnFood : MonoBehaviour {
    // Food Prefab
    public GameObject foodPrefab;

    // Borders
    public Transform borderTop;
    public Transform borderBottom;
    public Transform borderLeft;
    public Transform borderRight;

    // Use this for initialization
    void Start () {

    }
}
```

*Note: they are already of type **Transform** so we don't have to write **borderTop.transform.position** all the time. Instead we will be able to access their positions like **borderTop.position**.*

Let's create the **Spawn** function that spawns one piece of food within the borders. At first we will choose the **x** position somewhere randomly between the left and right border. Then we will choose the **y** position randomly between the top and bottom border. Afterwards we will instantiate the food Prefab at that position:

```
// Spawn one piece of food
void Spawn() {
    // x position between left & right border
    int x = (int)Random.Range(borderLeft.position.x,
                              borderRight.position.x);

    // y position between top & bottom border
    int y = (int)Random.Range(borderBottom.position.y,
                              borderTop.position.y);

    // Instantiate the food at (x, y)
    Instantiate(foodPrefab,
                new Vector2(x, y),
                Quaternion.identity); // default rotation
}
```

*Note: **x** and **y** are rounded via **(int)** to make sure that the food is always spawned at a position like **(1, 2)** but never at something like **(1.234, 2.74565)**.*

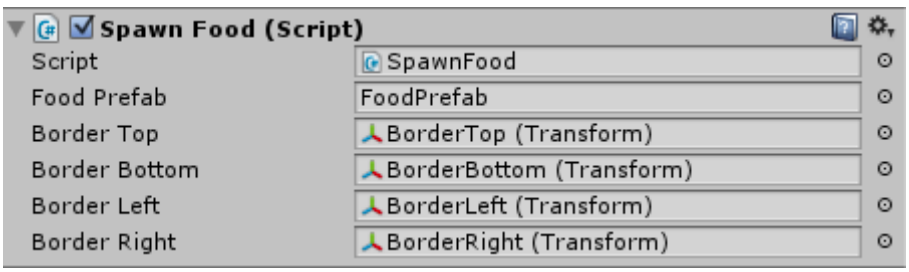
Now let's make sure that our Script calls the **Spawn** function every few seconds. We can do so by using InvokeRepeating:

```
// Use this for initialization
void Start () {
    // Spawn food every 4 seconds, starting in 3
    InvokeRepeating("Spawn", 3, 4);
}
```

Here is our full script:

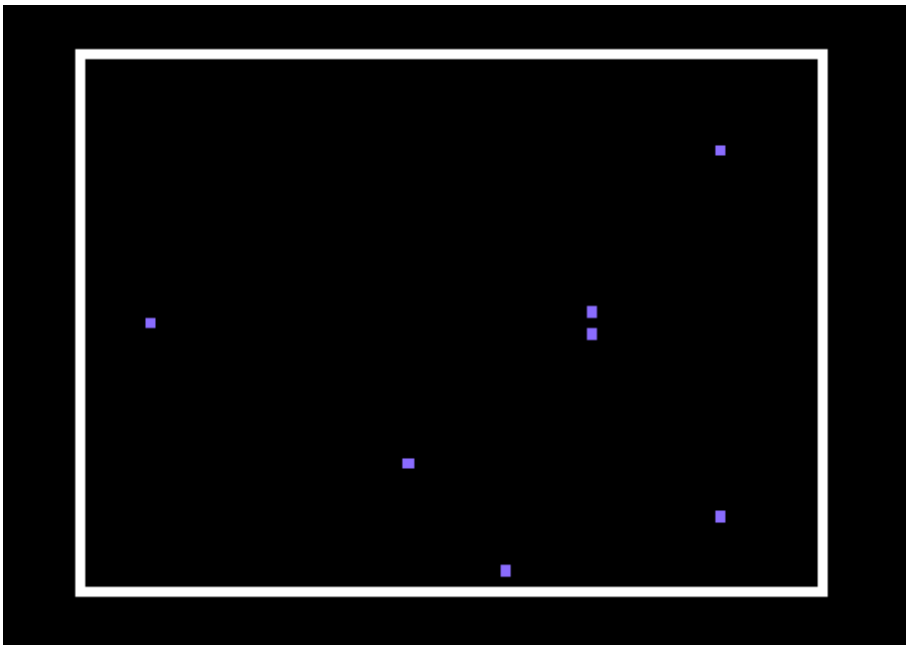
```
using UnityEngine;
using System.Collections;

public class SpawnFood : MonoBehaviour {
    // Food Prefab
    public GameObject foodPrefab;

    // Borders
    public Transform borderTop;
    public Transform borderBottom;
    public Transform borderLeft;
    public Transform borderRight;

    // Use this for initialization
    void Start () {
        // Spawn food every 4 seconds, starting in 3
        InvokeRepeating("Spawn", 3, 4);
    }

    // Spawn one piece of food
    void Spawn() {
        // x position between left & right border
        int x = (int)Random.Range(borderLeft.position.x,
                                  borderRight.position.x);

        // y position between top & bottom border
        int y = (int)Random.Range(borderBottom.position.y,
                                  borderTop.position.y);

        // Instantiate the food at (x, y)
        Instantiate(foodPrefab,
                    new Vector2(x, y),
                    Quaternion.identity); // default rotation
    }
}
```

If we save the Script and select the **Main Camera** then we can see that all our public variables are now shown in the Inspector. They are still **None**, so let's drag the **FoodPrefab** from the **Project Area** into the **Food Prefab** variable and the **Borders** from the **Hierarchy** into their corresponding slots:



Note: we can drag something into a Script's variable by literally dragging it with the mouse from the Hierarchy or Project Area into those slot things that can be seen in the above picture.

Alright, now it's time to press **Play** and wait a few seconds. We should be able to see some new food spawn in between the borders:

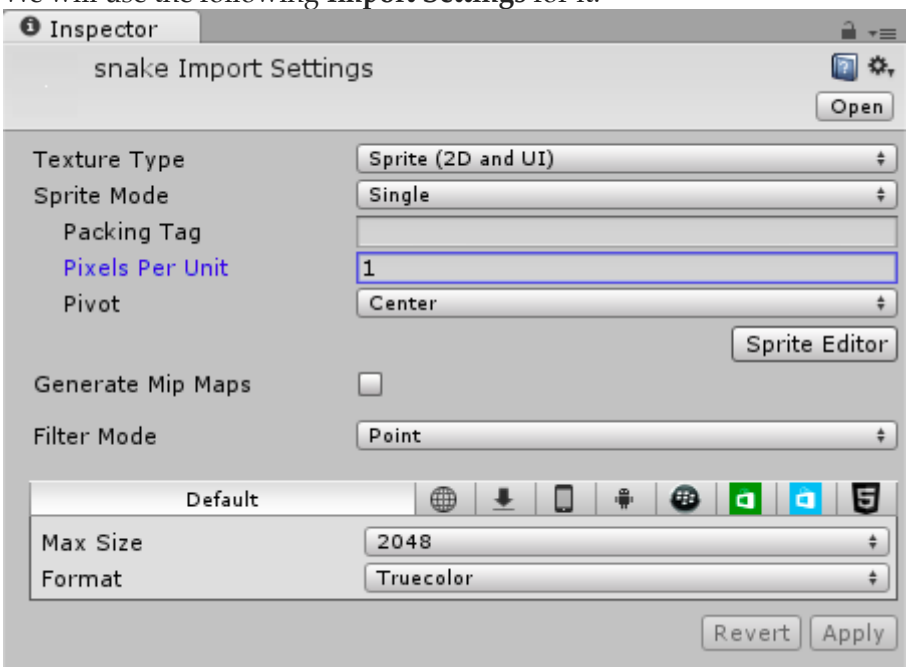


Creating the Snake

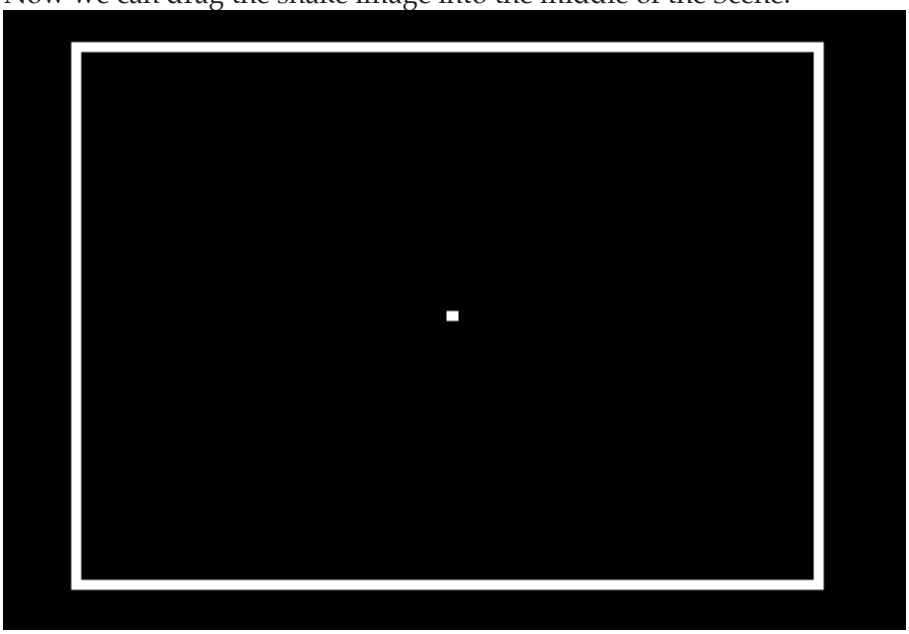
Let's finish the main part of the our game: the Snake. As usual we will start by drawing the snake image, which is just a 1 x 1 pixel texture:

- snake.png
  *Note: right click on the link, select **Save As...** and save it in the project's **Assets** folder.*

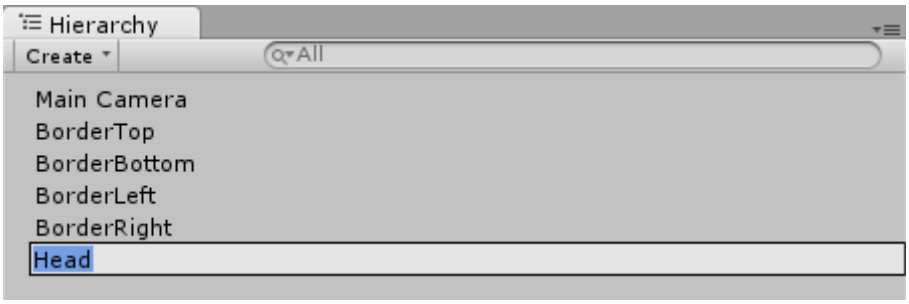
We will use the following **Import Settings** for it:



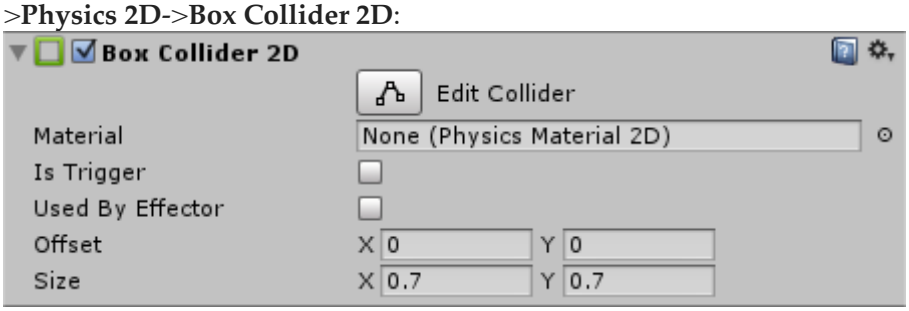
Now we can drag the snake image into the middle of the Scene:



So far it's just the Snake's head, so let's rename it to **Head** to keep things clean:



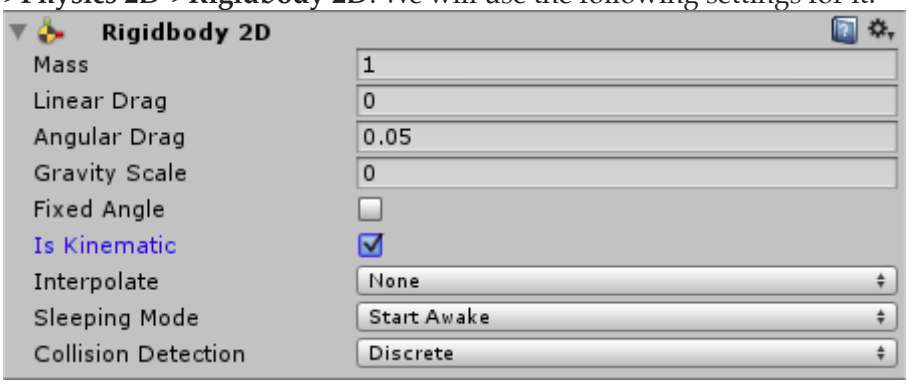
The snake should be part of the physics world, which means that we need to add a Collider to it again, so let's select **Add Component->Physics 2D->Box Collider 2D**:



*Note: the Collider has the size **(0.7, 0.7)** instead of **(1, 1)** so that it doesn't collide with other parts of the snake that are right next to it. We simply want to give Unity some space.*

Now the snake is also supposed to move around. As a rule of thumb, everything in the physics world that is supposed to move, needs a **Rigidbody**. A Rigidbody takes care of things like gravity, velocity and movement forces. We can add one by selecting **Add Component-**

>**Physics 2D->Rigidbody 2D**. We will use the following settings for it:



Notes:

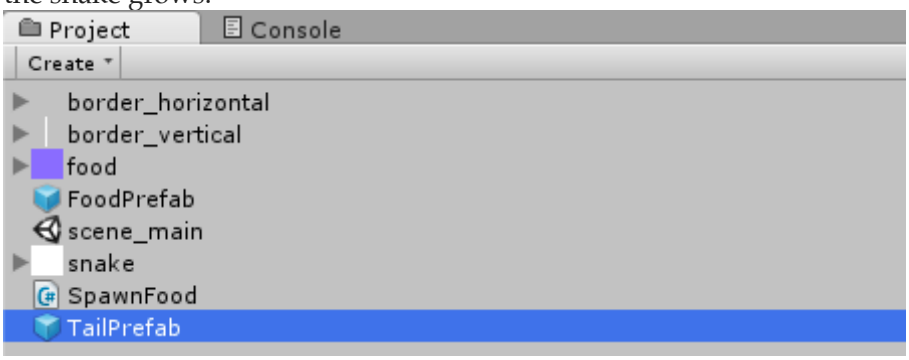
- The Rigidbody's **Gravity Scale** is **0** because we don't want the snake to fall towards the bottom of the screen all the time.
- The **Is Kinematic** option disables the physical behavior of the Rigidbody, so it doesn't react to gravity or collisions. We only need to know if the snake collided with something. We don't need Unity's physics to push things around in case of collisions. More info: Rigidbody2D Unity Docs.

The final snake will consist of many little elements. There will always be the **Head** at the front and then there will be several **Tail** elements like here:

```
ooooox
```

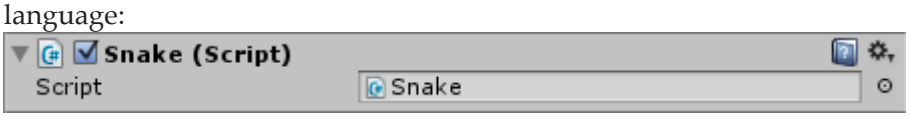
The only difference between the **Tail** elements and the **Head** is that the head does all the thinking. We will add a Script to it later.

Let's drag the snake **Head** from the **Hierarchy** into the **ProjectArea** to create a Prefab and then name it **TailPrefab** so we can load it whenever the snake grows:



*Note: some Unity versions will automatically rename GameObject in the **Hierarchy** too, so make sure that the one in the Hierarchy is still named **Head**.*

Alright, let's select the snake **Head** in the **Hierarchy** again and click on **Add Component->New Script**, name it **Snake** and select **CSharp** as the language:



We can open the Script by double clicking it in the **Project Area**:

```
using UnityEngine;
using System.Collections;

public class Snake : MonoBehaviour {

    // Use this for initialization
    void Start () {
```

```
    }

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

    }
}
```

Let's modify the top of the Script to include some List functionality that we will need later on:

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

public class Snake : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

The Snake should always move exactly one unit into whatever direction it wants to move. Now if we would allow it to move in every **Update** call, then it would be *really* fast. Instead we will only allow movement every 300 milliseconds by using Unity's InvokeRepeating function. It's like we create our own **Update** method that only gets called every 300 ms instead of every frame:

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

public class Snake : MonoBehaviour {

    // Use this for initialization
    void Start () {
        // Move the Snake every 300ms
        InvokeRepeating("Move", 0.3f, 0.3f);
    }

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

    }

    void Move() {
        // Do Movement Stuff..
    }
}
```

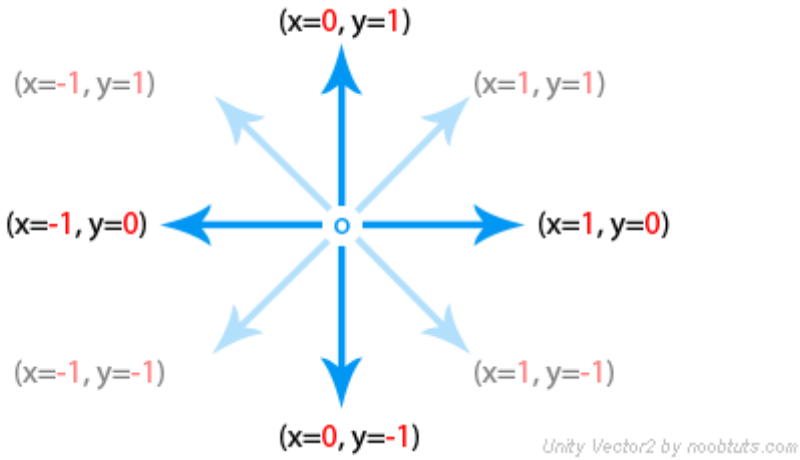
The Snake should **always** be moving into some direction, it should never stand still. So let's define a direction variable and use it to move the snake in the **Move** function:

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

```
public class Snake : MonoBehaviour {
    // Current Movement Direction
    // (by default it moves to the right)
    Vector2 dir = Vector2.right;

    // Use this for initialization
    void Start () {
        // Move the Snake every 300ms
        InvokeRepeating("Move", 0.3f, 0.3f);
    }

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

    }

    void Move() {
        // Move head into new direction
        transform.Translate(dir);
    }
}
```

*Note: transform.Translate means **'add this vector to my position'**.*

The direction variable is of type **Vector2**, which means that it has an **x** and **y** value. The following image shows different directions for a Vector2:



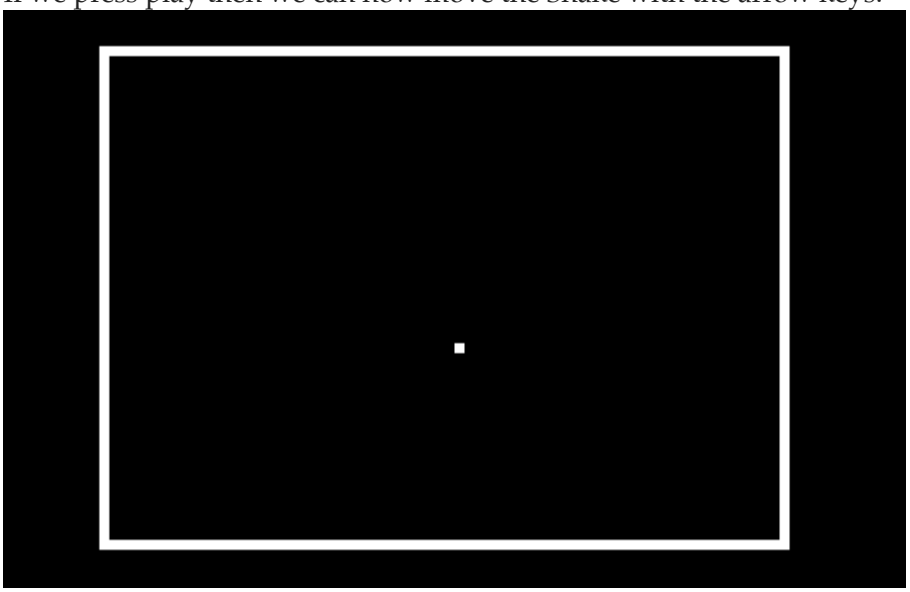
If we press play then we can already see the Snake move to the right:

The user should be able to change the movement direction by pressing one of the arrow keys. Now we could just check for key presses in our **Move** function, but that would make the game feel laggy because then we would only detect key presses every 300 ms. Instead we will use the **Update** function to detect key presses all the time:

```
// Update is called once per Frame
void Update() {
    // Move in a new Direction?
    if (Input.GetKey(KeyCode.RightArrow))
        dir = Vector2.right;
    else if (Input.GetKey(KeyCode.DownArrow))
        dir = -Vector2.up;    // '-up' means 'down'
    else if (Input.GetKey(KeyCode.LeftArrow))
        dir = -Vector2.right; // '-right' means 'left'
    else if (Input.GetKey(KeyCode.UpArrow))
        dir = Vector2.up;
}
```

*Note: if you are not sure why we used **Update** and **Move** instead of just **Update** or just **Move**, feel free to put the code from **Move** into **Update** or the code from **Update** into **Move**, then you will see that the snake moves rapidly fast or that the key presses are detected only rarely.*

If we press play then we can now move the Snake with the arrow keys:



The Snake's Tail

Let's think about how the Snake's tail will work. First of all, let's assume we have a snake with one head and three tail elements:

```
oox
```

Now as soon as the head moves to the right, the obvious thing would be to move every tail element to where its previous tail element was, like this:

```
step 1: oox   // snake didn't move yet
step 2: oo x  // head moved to the right
step 3: o ox  // first tail follows
step 4:  oox  // second tail follows
step 5:  oox  // third tail follows
```

This would work, but it would also be some complicated code. Let's use a neat little trick to make our lives easier. Instead of moving one tail element after another, we will simply move the last tail element into the gap like here:

```
step 1: ooox   // snake didn't move yet
step 2: ooo x  // head moved to the right
step 3:  ooox  // last tail element moved into the gap
```

Now that sounds like an easy algorithm. In every movement call, all we have to do is move the last tail element to where the head was before.

At first we will need some kind of data structure to keep track of all the tail elements that we will add later on:

```
// Keep Track of Tail
List<Transform> tail = new List<Transform>();
```

*Note: it's really important to add '**using System.Collections.Generic;**' and '**using System.Linq;**' to the top in order for lists to work.*

Let's get to the code that takes the last tail element, removes it from the back and puts it into the gap mentioned above:

```
void Move() {
    // Save current position (gap will be here)
    Vector2 v = transform.position;

    // Move head into new direction (now there is a gap)
    transform.Translate(dir);

    // Do we have a Tail?
    if (tail.Count > 0) {
        // Move last Tail Element to where the Head was
        tail.Last().position = v;

        // Add to front of list, remove from the back
        tail.Insert(0, tail.Last());
        tail.RemoveAt(tail.Count-1);
    }
}
```

*Note: **Translate** simply means 'add this vector to my position'. Afterwards we check if there is anything in the tail list, in which case we change the last tail element's position to the gap position (where the head was before). We also have to keep our list order, hence the **Insert** and **RemoveAt** calls at the end. They make sure that the last tail element is now the first element in the list, too.*

And that was the only slightly complicated part of our Unity 2D Snake Tutorial. Now we are almost done.

Feeding the Snake

We will use the OnTriggerEnter2D function to receive collision information *(which will happen whenever the snake walks into food or into a border)*.

Whenever it runs into food, we will use the exact same mechanics that we used for our movement above, except that this time instead of removing the last tail element and moving it into the gap, we will only **Instantiate** a new element into the gap:

```
ooo x  // gap
oooox  // gap filled with new element
```

It's important to understand that we will not make the Snake longer immediately after it eats something. Just like with our arrow key presses, we will wait until it actually moves. Therefore we will need a new variable that we will set to **true** whenever the Snake ate something:

```
// Did the snake eat something?
bool ate = false;
```

We will also need a public variable that let's us assign the **TailPrefab** later on:

```
// Did the snake eat something?
bool ate = false;

// Tail Prefab
public GameObject tailPrefab;
```

Note: the two variables are defined at the top of our Snake script.

Now let's get to the **OnTriggerEnter2D** function. This one will be straight forward again. We will find out if the Snake collided with food, in which case we set the **ate** variable to true and destroy the food. If it didn't collide with food, then it either collided with itself or with a border:

```
void OnTriggerEnter2D(Collider2D coll) {
    // Food?
    if (coll.name.StartsWith("FoodPrefab")) {
        // Get longer in next Move call
        ate = true;

        // Remove the Food
        Destroy(coll.gameObject);
    }
    // Collided with Tail or Border
    else {
        // ToDo 'You lose' screen
    }
}
```

*Note: we use **coll.name.StartsWith** because the food is called **'FoodPrefab(Clone)'** after instantiating it. The more elegant way to find out if **coll** is food or not would be by using a **Tag**, but for the sake of simplicity we will use string comparison in this Tutorial.*

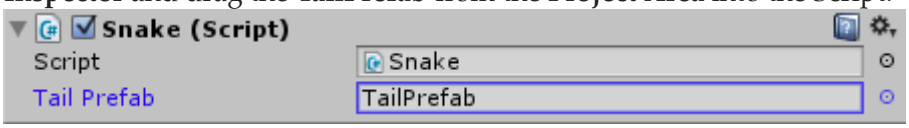
Alright, let's modify our **Move** function so it makes the Snake longer whenever **ate** is true:

```
void Move() {
    // Save current position (gap will be here)
    Vector2 v = transform.position;

    // Move head into new direction (now there is a gap)
    transform.Translate(dir);

    // Ate something? Then insert new Element into gap
    if (ate) {
        // Load Prefab into the world
        GameObject g =(GameObject)Instantiate(tailPrefab,
                                              v,
                                              Quaternion.ident

        // Keep track of it in our tail list
        tail.Insert(0, g.transform);

        // Reset the flag
        ate = false;
    }
    // Do we have a Tail?
    else if (tail.Count > 0) {
        // Move last Tail Element to where the Head was
        tail.Last().position = v;

        // Add to front of list, remove from the back
        tail.Insert(0, tail.Last());
        tail.RemoveAt(tail.Count-1);
```

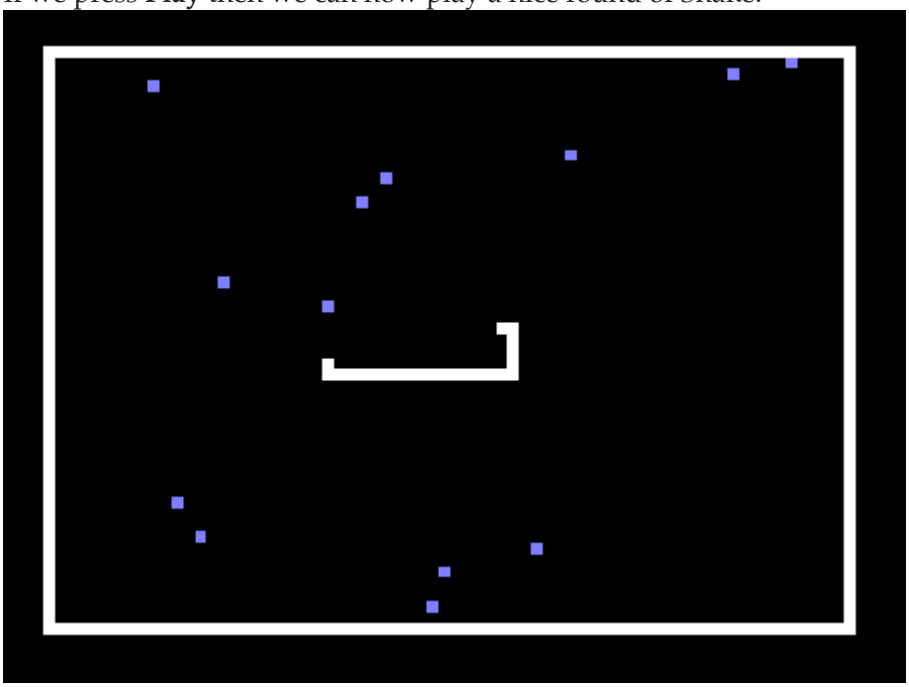
```
    }
}
```

*Note: all we did was check if **ate** is true, then **Instantiate** the tail prefab at the position **v** with the default rotation (**Quaternion.identity**). Afterwards we add it to the **tail** list and reset the **ate** flag. The rest of the code was already there.*

Now we can select the snake **Head** in the **Hierarchy**, take a look at the **Inspector** and drag the **TailPrefab** from the **Project Area** into the Script:



If we press **Play** then we can now play a nice round of Snake:



Summary

Snake is one awesome game for a Unity Tutorial. There is a lot of value in understanding how to do pixel exact games like this and how to add movement with **InvokeRepeating**. Yet again we saw how amazingly easy 2D games are with Unity's 2D features that were introduced not very long ago.

Now it's up to the reader to make the game fun! There are tons of improvements to be made like a win/lose screen, a better looking snake texture, multiple levels, power-ups, increasing speed, high-scores, multiplayer and so on.