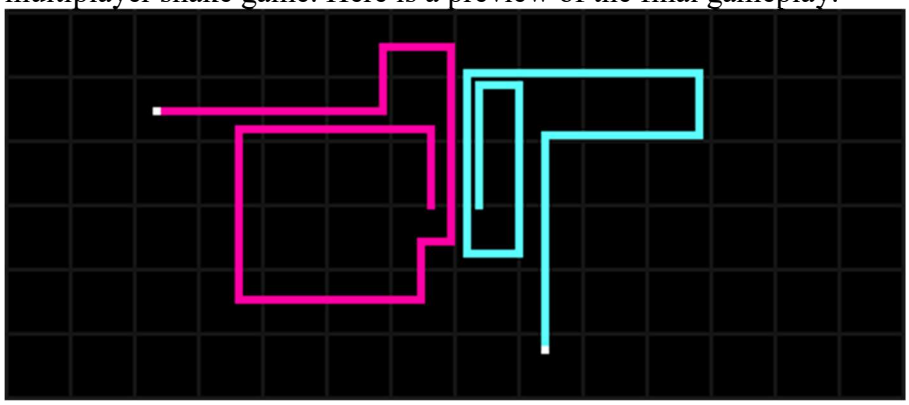
Unity 2D Tron Light-Cycles Tutorial

Intro

This workshop is intended for total Unity noobs. As always, everything will be explained step-by-step so anyone can understand it.

We will expand upon our previous knowledge of Unity and go more in depth with GameObjects, Prefabs and Transforms. The programming for this tutorial is only 60 lines of code and it is completely optional as all the scripts are already provided (they are also commented so you know what each line of code does). However, this time you are encouraged to take a look at the code as it will help you better understand Unity objects can be manipulated.

This time, let's make a Tron style 2D game in Unity. Two players will be able to compete with each other. The goal is to move your lightcycle in a way that traps the other player, kinda like a multiplayer snake game. Here is a preview of the final gameplay:

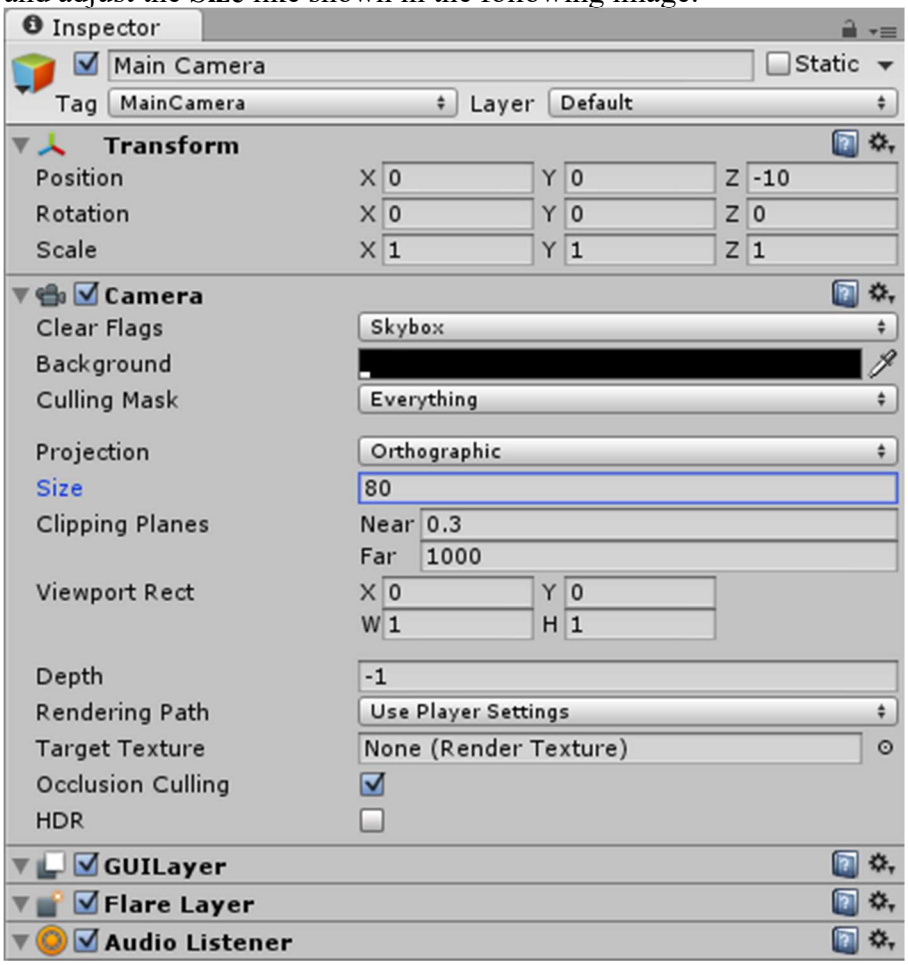


After opening the project folder you will notice two scenes (these are the things with the unity icon), one called "**New Scene**" another called "**finished_game**". You should currently have "**New Scene**" selected (if not double click to open that scene).

"**New Scene**" is an empty scene where we will create our game from scratch. "**finished_game**" is the completed game with all the objects placed, components added, scripts attached, etc… You can select "**finished_game**" to view the final project and see how things should be set up, but we will be recreating the game from scratch in "**New Scene**".

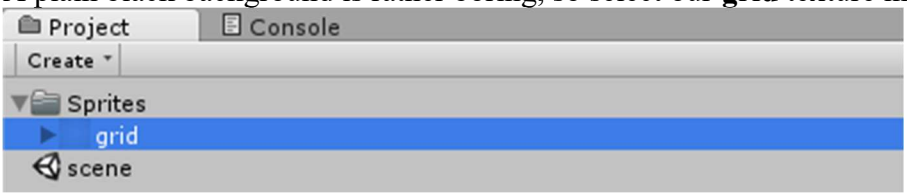
Camera Setup

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

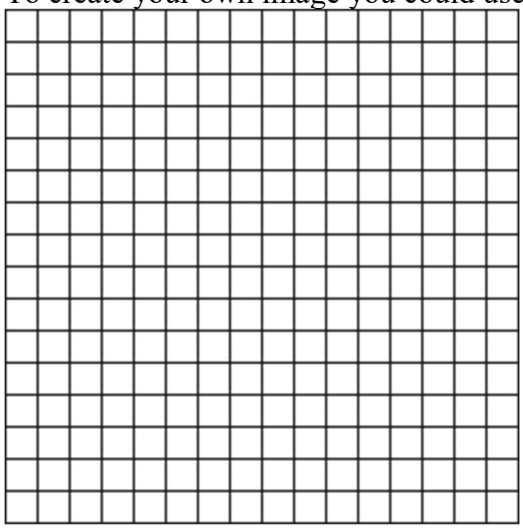


The Background Image

A plain black background is rather boring, so select our **grid** texture in the **Project Area**.

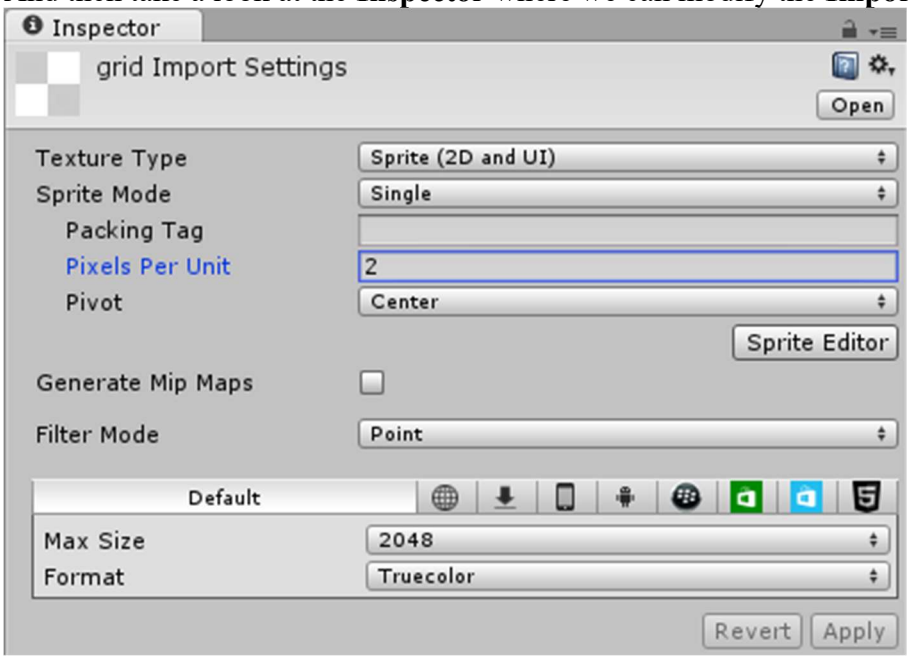


To create your own image you could use any drawing tool of your choice:



*Note: select **Save As...**, navigate to the project's **Assets** folder and save it in a new **Sprites** folder.*

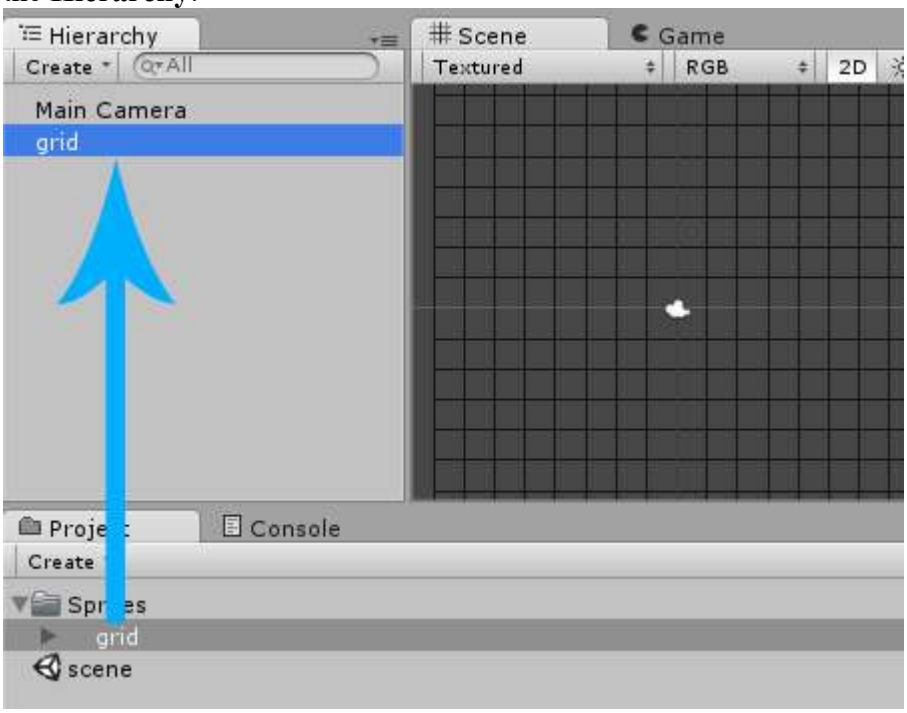
And then take a look at the **Inspector** where we can modify the **Import Settings**:



*Note: a **Pixels Per Unit** value of **2** means that 2 x 2 pixels will fit into one unit in the game world. We will use this value for all our textures, because the player sprite will have the size of 2 x 2 pixels later on. The other settings are just visual effects. We want to make the image look perfectly sharp, without any compression.*

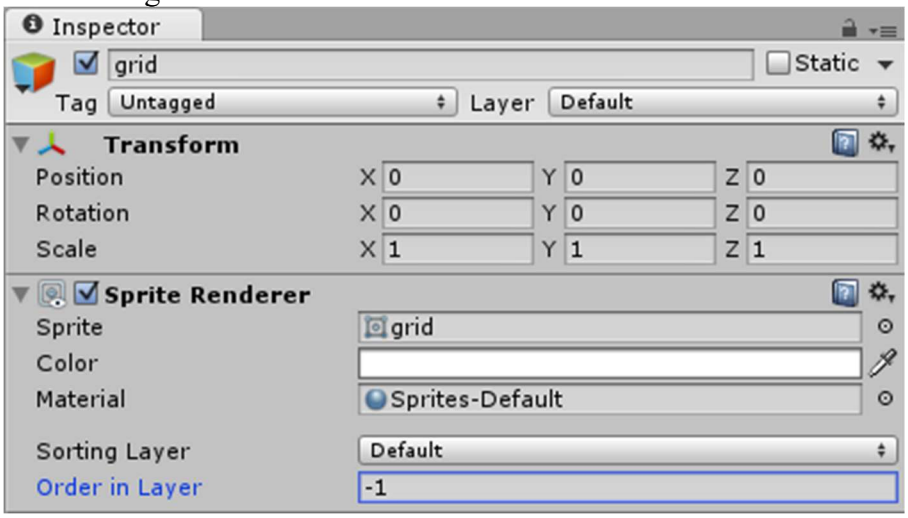
The Import Settings have already been set up for this project, but if you make your own simple 2D game in the future you will want to edit them yourself to be something like we have here.

Now we can add the grid to our game world by simply dragging it from the **Project Area** into the **Hierarchy**:



*Note: we can also drag it from the **Project Area** into the **Scene**, but then we also have to re-adjust the position to **(0, 0, 0)**.*

We will also change the grid's **Order in Layer** property to **-1** to make sure that it's always drawn in the background later:



*Note: usually we would create a whole new **Background** Sorting Layer, but for such a simple game, using the **Order in Layer** property is enough.*

If we press **Play** then we can already see the grid in game:

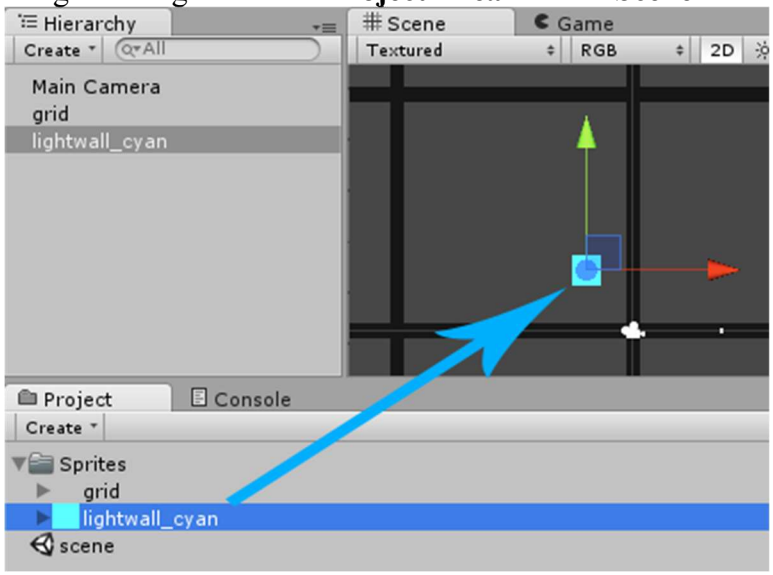
The Lightwalls

The Players should leave a lightwall wherever they move, so let's create the first one.

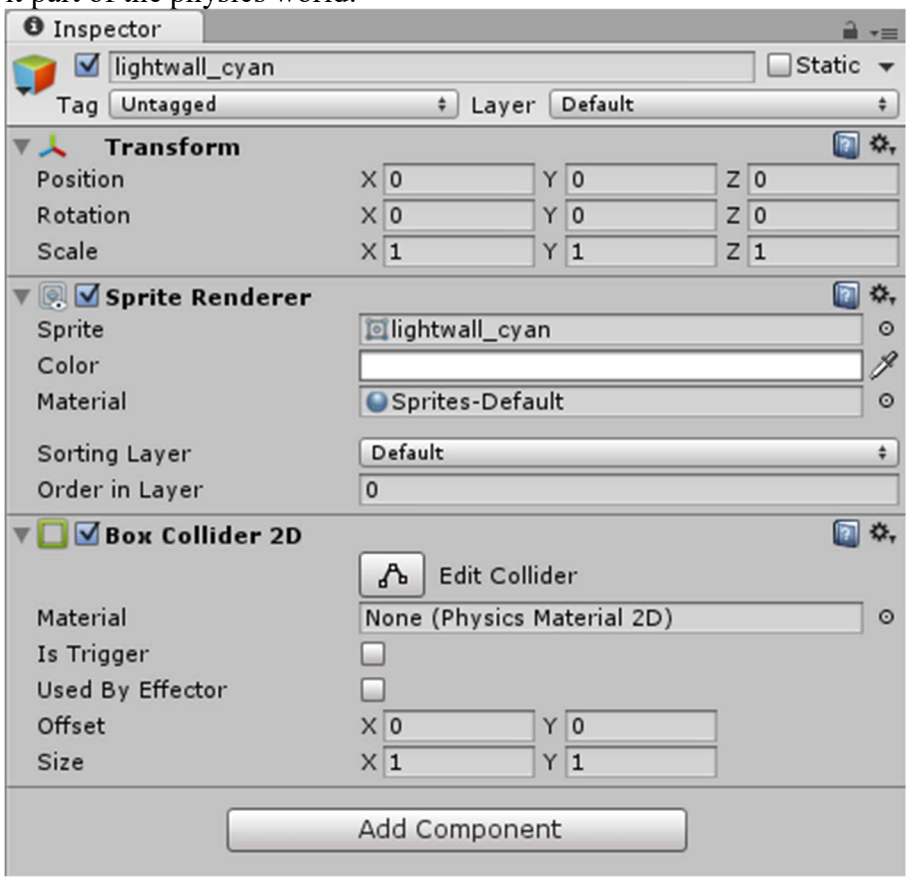
The Cyan Lightwall

- Select our 2x2 px **lightwall_cyan** texture in the **Project Area**.

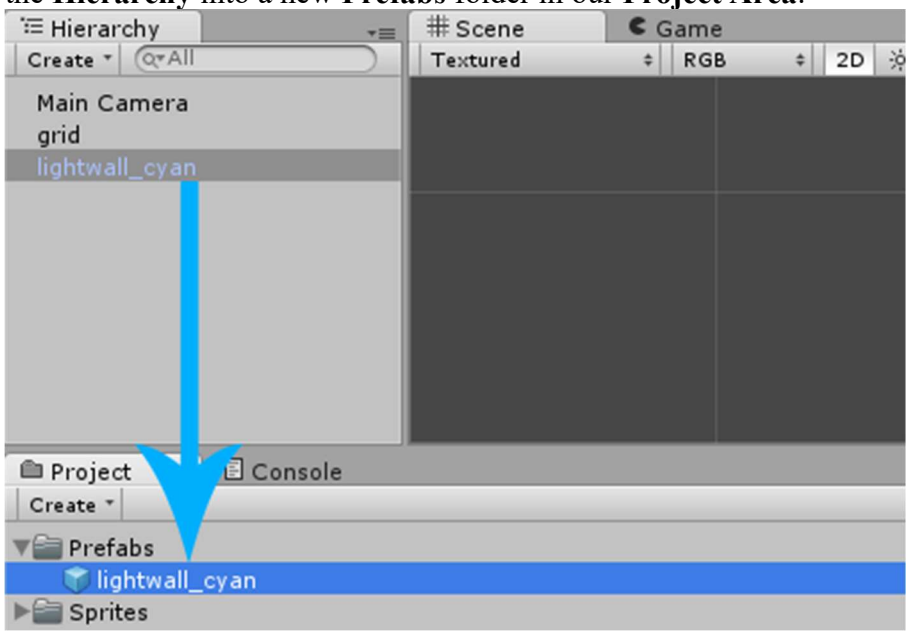
Drag the image from the **Project Area** into the **Scene** in order to create a **GameObject** from it:



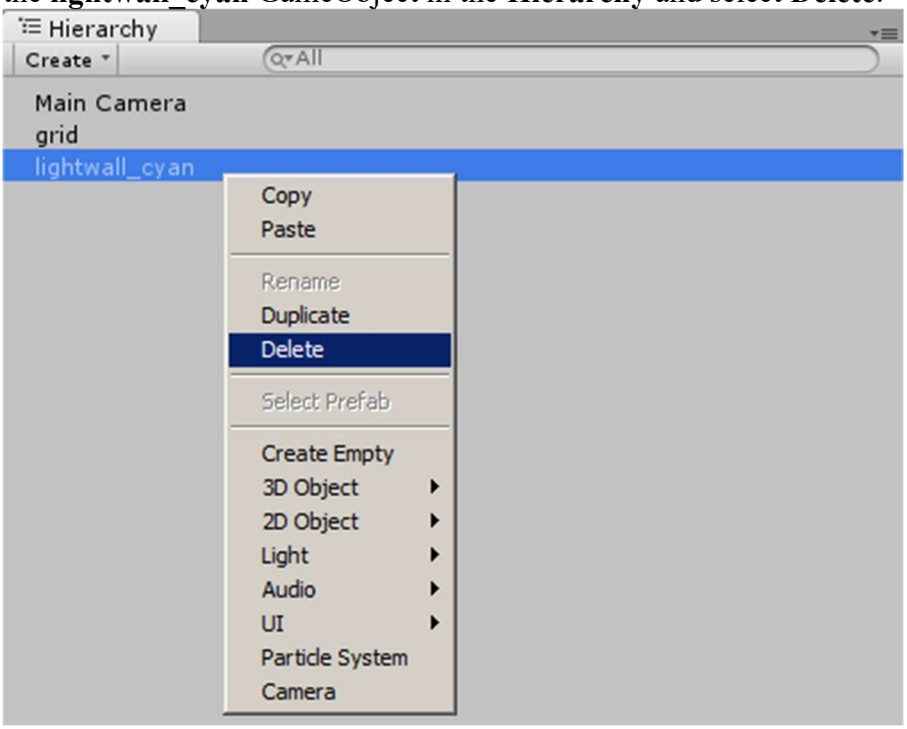
Right now the Lightwall is really just an image in the game world, nothing would collide with it. Let's select **Add Component->Physics 2D->Box Collider 2D** in the **Inspector** in order to make it part of the physics world:



Now the Lightwall is finished. Let's create a Prefab from it by dragging it from the **Hierarchy** into a new **Prefabs** folder in our **Project Area**:



Having saved the Lightwall as a Prefab means that we can load it into the game whenever we want. And since we don't need it to be in the game just yet, we can right click the **lightwall_cyan** GameObject in the **Hierarchy** and select **Delete**:

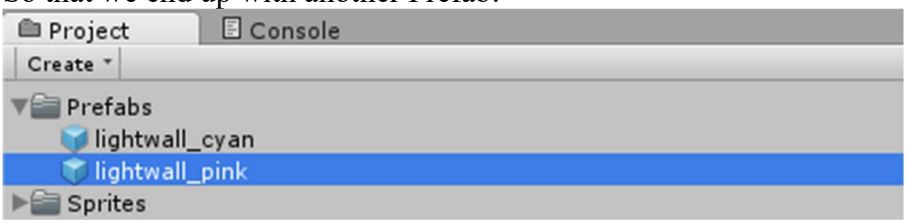


The Pink Lightwall

Let's repeat the above workflow for the pink Lightwall image:

- Select our 2x2 px **lightwall_pink** texture in the **Project Area**.

So that we end up with another Prefab:

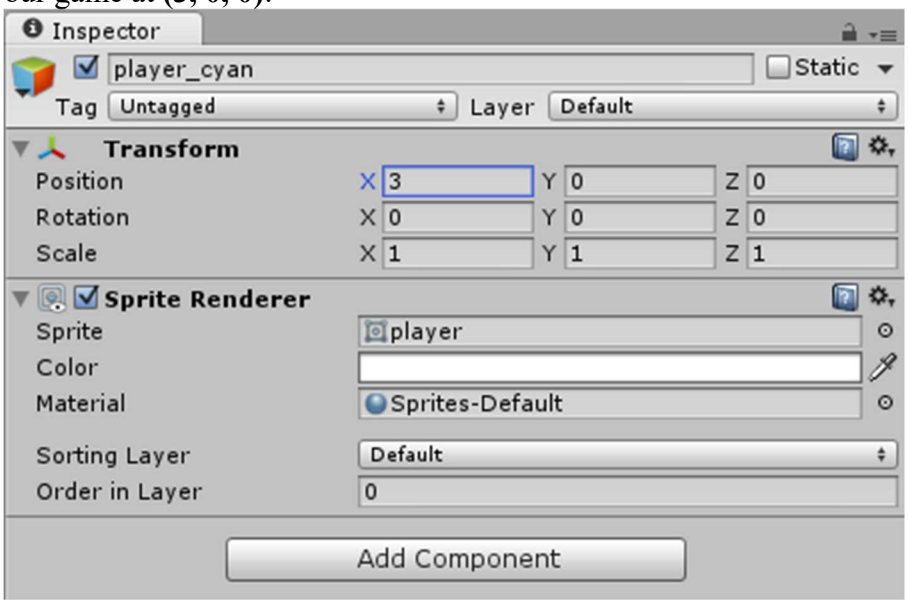


The Player

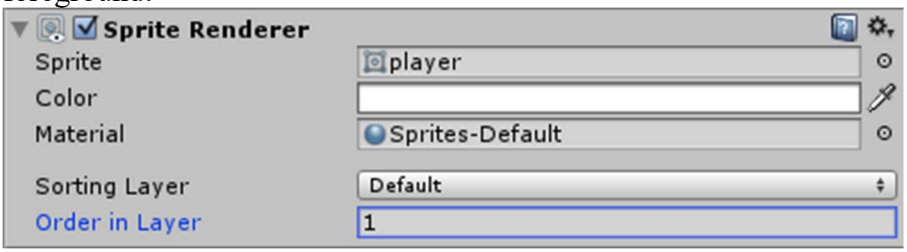
Now it's time to add the Player. The Player should be a simple white square that is moveable by pressing some keys. The Player will also drag a Lightwall everywhere he goes.

- Select our 2x2 px **player** texture in the **Project Area**.

Now we can drag the image from the **Project Area** into the **Scene** in order to create a GameObject from it. We will then rename it to **player_cyan** and position it at the right center of our game at **(3, 0, 0)**:



Let's also modify the **Order in Layer** property again to make sure that the player will be in the foreground:

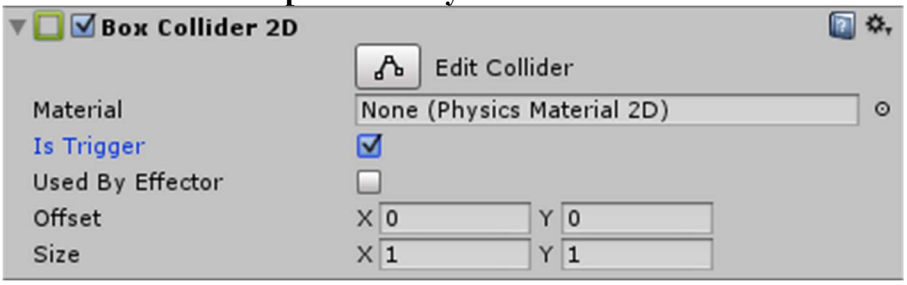


*Note: just like mentioned before, we would usually use a Sorting Layer for this. However since we will only have 3 elements in our game: the background, the player and the Lightwalls, we will keep it simple and just use three different **Order in Layer** values.*

Player Physics

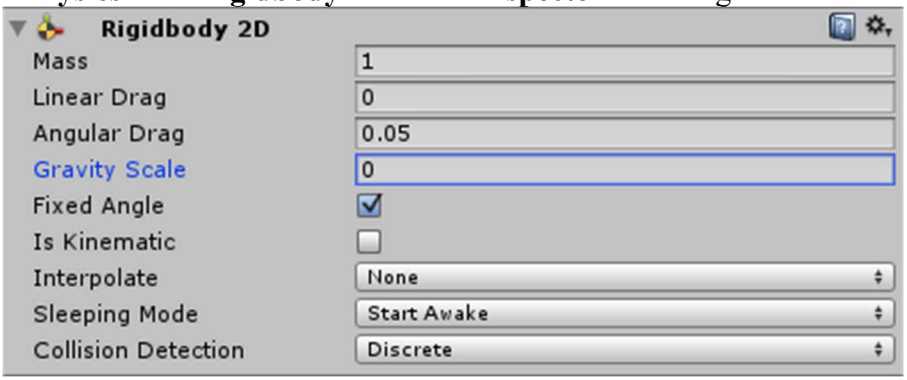
Right now the player is not part of the physics world, things won't collide with him and he can't move around. We will need to add a **Collider** again in order to make him part of the physics world.

Let's select **Add Component->Physics 2D->Box Collider 2D** in the **Inspector**:



*Note: we enabled **IsTrigger** to avoid collisions with the player's own Lightwall later on. As long as we have **IsTrigger** enabled, the player will only receive collision information, without actually colliding with anything. This will make sense very soon.*

The player is also supposed to move around. A Rigidbody takes care of stuff like gravity, velocity and other forces that make things move. As a rule of thumb, everything in the physics world that is supposed to move around needs a **Rigidbody**. Let's select **Add Component->Physics 2D->Rigidbody 2D** in the **Inspector** and assign the following settings to it:



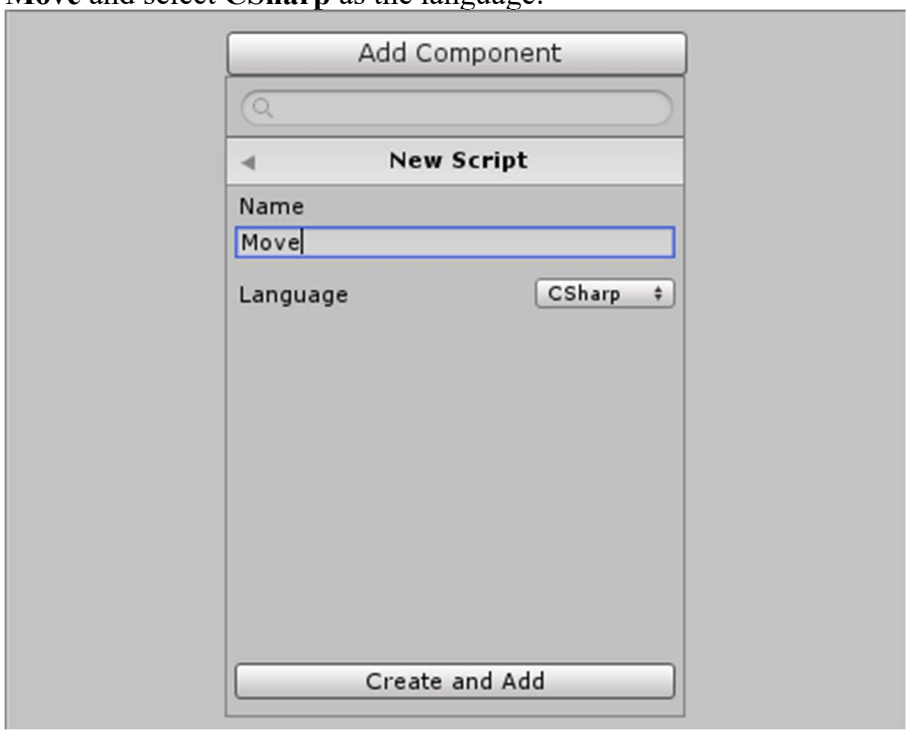
*Note: we set the **Gravity Scale** to **0** because we don't need any gravity in our game. Furthermore we enabled the **Fixed Angle** property to prevent the player from rotating around.*
Our player is now part of the physics world, it's simple as that.

Player Movement

We will use Scripting to make the player move. Let's select **Add Component->Move**, to attach the Move script to the player.

Our Script will be rather simple for now, all we have to do is check for arrow key presses and modify the Rigidbody's **velocity** property. The Rigidbody will then take care of all the movement itself.
Note: the velocity is the movement direction multiplied by the movement speed.

If we were creating a brand new script we would select **Add Component->New Script,** name it **Move** and select **CSharp** as the language:



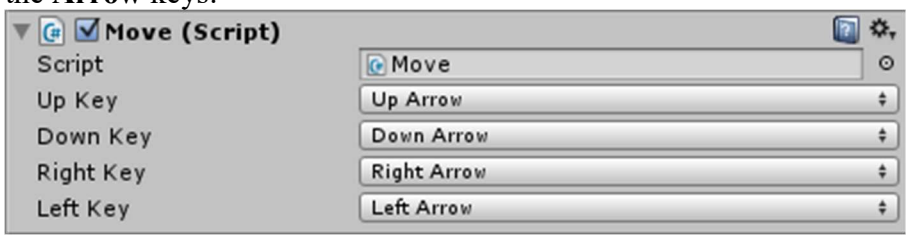
Afterwards we can double click the Script in the **Project Area** in order to open it:

```
using UnityEngine;
using System.Collections;

public class Move : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

First of all we want to find out if the movement keys were pressed. Now we only want to create one movement Script for both players, so let's make the movement keys customizable so that we can use the **Arrow** keys for one player and the **WSAD** keys for the other player:

```
using UnityEngine;
using System.Collections;

public class Move : MonoBehaviour {
    // Movement keys (customizable in Inspector)
    public KeyCode upKey;
    public KeyCode downKey;
    public KeyCode rightKey;
    public KeyCode leftKey;

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

If we save the Script and take a look at the **Inspector** then we can set the key variables to the **Arrow** keys:



Alright, let's check for key presses in our **Update** function:

```
// Update is called once per frame
void Update () {
    // Check for key presses
    if (Input.GetKeyDown(upKey)) {
        // Do stuff...
    }
    else if (Input.GetKeyDown(downKey)) {
        // Do stuff...
    }
    else if (Input.GetKeyDown(rightKey)) {
        // Do stuff...
    }
    else if (Input.GetKeyDown(leftKey)) {
        // Do stuff...
    }
}
```

Now as soon as the player presses any of those keys, we want to make the player move into that direction. As mentioned before, we will use the Rigidbody's **velocity** property for that. The velocity is always the **movement direction** multiplied by the **movement speed**. Let's add a movement speed variable first:

```
using UnityEngine;
using System.Collections;

public class Move : MonoBehaviour {
    // Movement keys (customizable in Inspector)
    public KeyCode upKey;
    public KeyCode downKey;
    public KeyCode rightKey;
    public KeyCode leftKey;

    // Movement Speed
    public float speed = 16;

    ...
}
```

The rest will be really simple. All we have to do is modify our **Update** function one more time to set the Rigidbody's **velocity** property:

```
// Update is called once per frame
void Update () {
    // Check for key presses
    if (Input.GetKeyDown(upKey)) {
        GetComponent<Rigidbody2D>().velocity = Vector2.up * speed;
    }
    else if (Input.GetKeyDown(downKey)) {
        GetComponent<Rigidbody2D>().velocity = -Vector2.up * speed;
    }
    else if (Input.GetKeyDown(rightKey)) {
        GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
    }
    else if (Input.GetKeyDown(leftKey)) {
        GetComponent<Rigidbody2D>().velocity = -Vector2.right * speed;
    }
}
```

*Note: **-Vector2.up** means **down** and **-Vector2.right** means **left**.*
Let's also modify our **Start** function really quick to give the player an initial velocity:

```
// Use this for initialization
void Start () {
    // Initial Velocity
    GetComponent<Rigidbody2D>().velocity = Vector2.up * speed;
}
```

Now there is just one problem with our script, we do not want our players to be able to reverse their direction 180 degrees and run back into the LightWall they have just created. To prevent this, we let's add one more variable to our script to keep track of a player's current direction.

```
using UnityEngine;
using System.Collections;

public class Move : MonoBehaviour {
    // Movement keys (customizable in Inspector)
    public KeyCode upKey;
    public KeyCode downKey;
    public KeyCode rightKey;
    public KeyCode leftKey;

    private string currentDirection = "up";

    ...
}
```

Let's add this new variable as a secondary condition to our key press checks and also update the variable whenever the player changes direction.

```
// Update is called once per frame
void Update () {
    // Check for key presses
    if (Input.GetKeyDown(upKey) && currentDirection != "down") {
       GetComponent<Rigidbody2D>().velocity = Vector2.up * speed;
        currentDirection = "up";
    }
    else if (Input.GetKeyDown(downKey) && currentDirection != "up") {
       GetComponent<Rigidbody2D>().velocity = -Vector2.up * speed;
        currentDirection = "down";
    }
    else if (Input.GetKeyDown(rightKey) && currentDirection != "left") {
       GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
        currentDirection = "right";
    }
    else if (Input.GetKeyDown(leftKey) && currentDirection != "right") {
       GetComponent<Rigidbody2D>().velocity = -Vector2.right * speed;
        currentDirection = "left";
    }
}
```

If we press **Play** then we can now move the player with the arrow keys:

Player Lightwalls

We want to add a feature that creates a Lightwall wherever the player goes. All we really need to do is create a new Lightwall as soon as the player turns into a new direction, and then always scale the Lightwall along where the player goes, until he moves into another direction.

We will need a helper function that spawns a new Lightwall. At first we will add two variables to our Script. One of them will be the Lightwall Prefab and the other will be the wall that is currently being dragged along by the player:

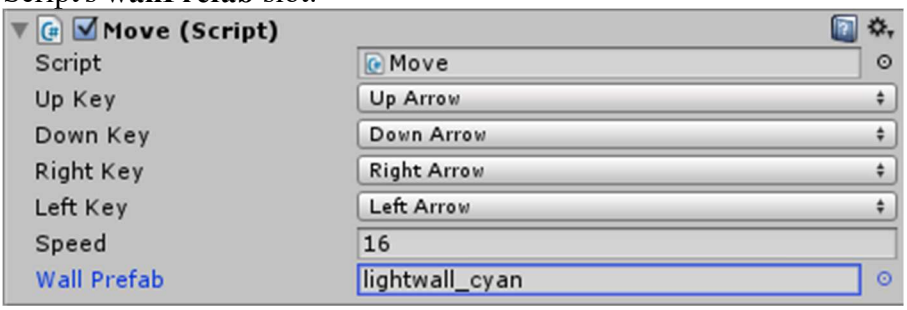
```
public class Move : MonoBehaviour {
    // Movement keys (customizable in Inspector)
    public KeyCode upKey;
    public KeyCode downKey;
    public KeyCode rightKey;
    public KeyCode leftKey;

    // Movement Speed
    public float speed = 16;

    // Wall Prefab
    public GameObject wallPrefab;

    // Current Wall
    Collider2D wall;

    ...
```

Now we can use Instantiate to create a function that spawns a new Lightwall at the player's current position:

```
void spawnWall() {
    // Spawn a new Lightwall
    GameObject g = (GameObject)Instantiate(wallPrefab, transform.position,
Quaternion.identity);
    wall = g.GetComponent<Collider2D>();
}
```

*Note: **transform.position** is the player's current position and **Quaternion.identity** is the default rotation. We also save the GameObject's **Collider2D** in our **wall** variable to keep track of the current wall.*

Let's save the Script and then drag the **lightwall_cyan** Prefab from the **Project Area** into the Script's **wallPrefab** slot:



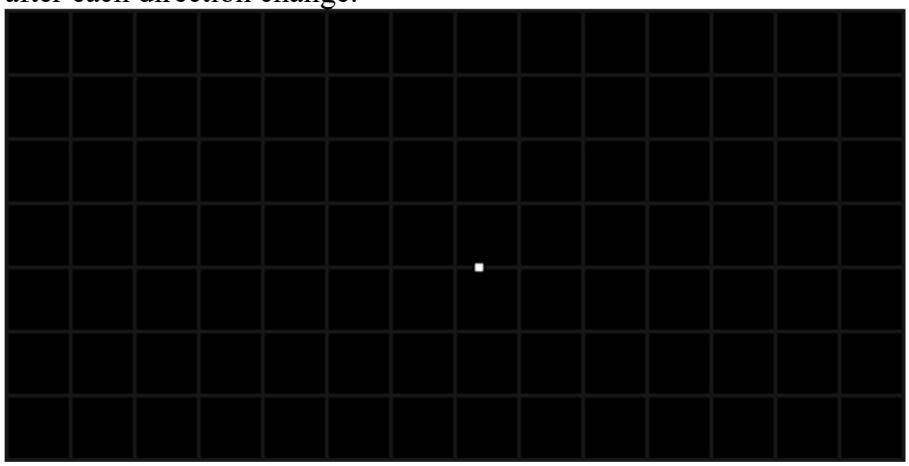
Alright, it's time to make use of our helper function. We will now modify our Script's **Update** function to spawn a new Lightwall after changing the direction:

```
// Update is called once per frame
void Update () {
    // Check for key presses
    if (Input.GetKeyDown(upKey) && currentDirection != "down") {
        GetComponent<Rigidbody2D>().velocity = Vector2.up * speed;
        currentDirection = "up";
        spawnWall();
    }
    else if (Input.GetKeyDown(downKey) && currentDirection != "up") {
        GetComponent<Rigidbody2D>().velocity = -Vector2.up * speed;
        currentDirection = "down";
        spawnWall();
    }
    else if (Input.GetKeyDown(rightKey) && currentDirection != "left") {
        GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
        currentDirection = "right";
        spawnWall();
    }
    else if (Input.GetKeyDown(leftKey) && currentDirection != "right") {
        GetComponent<Rigidbody2D>().velocity = -Vector2.right * speed;
        currentDirection = "left";
        spawnWall();
    }
}
```

We will also spawn a new Lightwall when the game starts:

```
// Use this for initialization
void Start () {
    // Initial Velocity
    GetComponent<Rigidbody2D>().velocity = Vector2.up * speed;
    spawnWall();
}
```

If we save the Script and press **Play**, then we can see how a new Lightwall is being spawned after each direction change:



So far, so good.

Right now the Lightwalls are only little squares, we still have to scale them. Let's create a new **fitColliderBetween** function that takes a Collider and two points, and then fits the Collider between those two points:

```
void fitColliderBetween(Collider2D co, Vector2 a, Vector2 b) {
    // Calculate the Center Position
    co.transform.position = a + (b - a) * 0.5f;

    // Scale it (horizontally or vertically)
    float dist = Vector2.Distance(a, b);
    if (a.x != b.x)
        co.transform.localScale = new Vector2(dist, 1);
    else
        co.transform.localScale = new Vector2(1, dist);
}
```

*Note: this function may look a bit confusing at first. The obvious way to fit a Collider between two points would be something like **collider.setMinMax()**, but Unity doesn't allow that. Instead we will simply use the **transform.position** property to position it exactly between the two points, and then use the **transform.localScale** property to make it really long, so it fits exactly between the points. The formula **a + (b - a) * 0.5** is very easy to understand, too. First of all we calculate the direction from **a** to **b** by using **(b - a)**. Then we simply add half of that direction to the point **a**, which results in the center point. Afterwards we find out if the line is supposed to go horizontally or vertically by comparing the two **x** coordinates. If they are equal, then the line goes horizontally, otherwise vertically. Finally we adjust the scale so the wall is **dist** units long and **1** unit wide.*

Let's make use of our **fitColliderBetween** function. We always want to fit the Collider between the end of the last Collider and the player's current position. So first of all, we will have to keep track of the end of the last Collider.

We will add a **lastWallEnd** variable to our Script:

```
public class Move : MonoBehaviour {
    // Movement keys (customizable in Inspector)
    public KeyCode upKey;
    public KeyCode downKey;
    public KeyCode rightKey;
    public KeyCode leftKey;

    // Movement Speed
    public float speed = 16;

    // Wall Prefab
    public GameObject wallPrefab;

    // Current Wall
    Collider2D wall;

    // Last Wall's End
    Vector2 lastWallEnd;

    ...
```

And set the position in our **spawnWall** function:

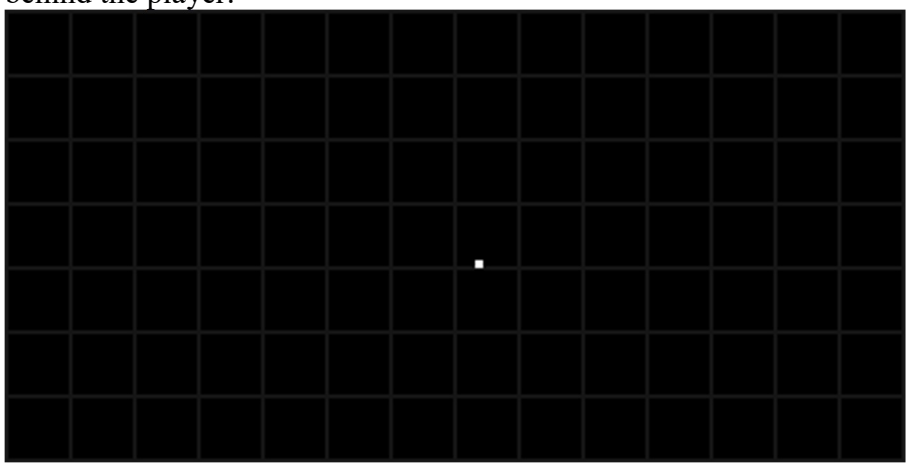
```
void spawnWall() {
    // Save last wall's position
    lastWallEnd = transform.position;

    // Spawn a new Lightwall
    GameObject g = (GameObject)Instantiate(wallPrefab, transform.position,
Quaternion.identity);
    wall = g.GetComponent<Collider2D>();
}
```

*Note: technically the last wall's position should be **wall.transform.position**, but we used the player's **transform.position** here. The reason is that when spawning the first wall, there was no last wall yet, hence why we couldn't set the **lastWallEnd** position. Instead we always set it to the player's current position before spawning the next wall, which pretty much ends up being the same thing.*

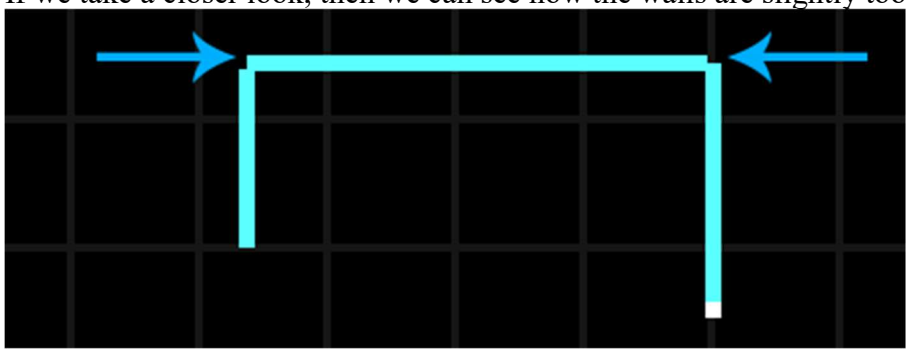
Almost done. Now we can modify our **Update** function again to always fit the current wall between the last wall's end position and the player's current position:

```
// Update is called once per frame
void Update () {
    // Check for key presses
    if (Input.GetKeyDown(upKey) && currentDirection != "down") {
        GetComponent<Rigidbody2D>().velocity = Vector2.up * speed;
        currentDirection = "up";
        spawnWall();
    }
    else if (Input.GetKeyDown(downKey) && currentDirection != "up") {
        GetComponent<Rigidbody2D>().velocity = -Vector2.up * speed;
        currentDirection = "down";
        spawnWall();
    }
    else if (Input.GetKeyDown(rightKey) && currentDirection != "left") {
        GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
        currentDirection = "right";
        spawnWall();
    }
    else if (Input.GetKeyDown(leftKey) && currentDirection != "right") {
        GetComponent<Rigidbody2D>().velocity = -Vector2.right * speed;
        currentDirection = "left";
        spawnWall();
    }

    fitColliderBetween(wall, lastWallEnd, transform.position);
}
```

If we save the Script and press **Play**, then we can see how the Lightwalls are being created behind the player:



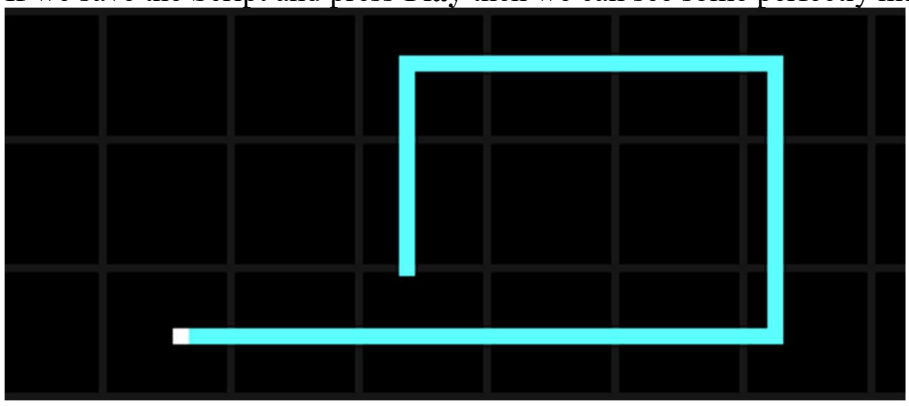
If we take a closer look, then we can see how the walls are slightly too short on the corners:



There is a very easy solution to our problem. All we have to do is go back to our **fitColliderBetween** function and always make the wall one unit longer:

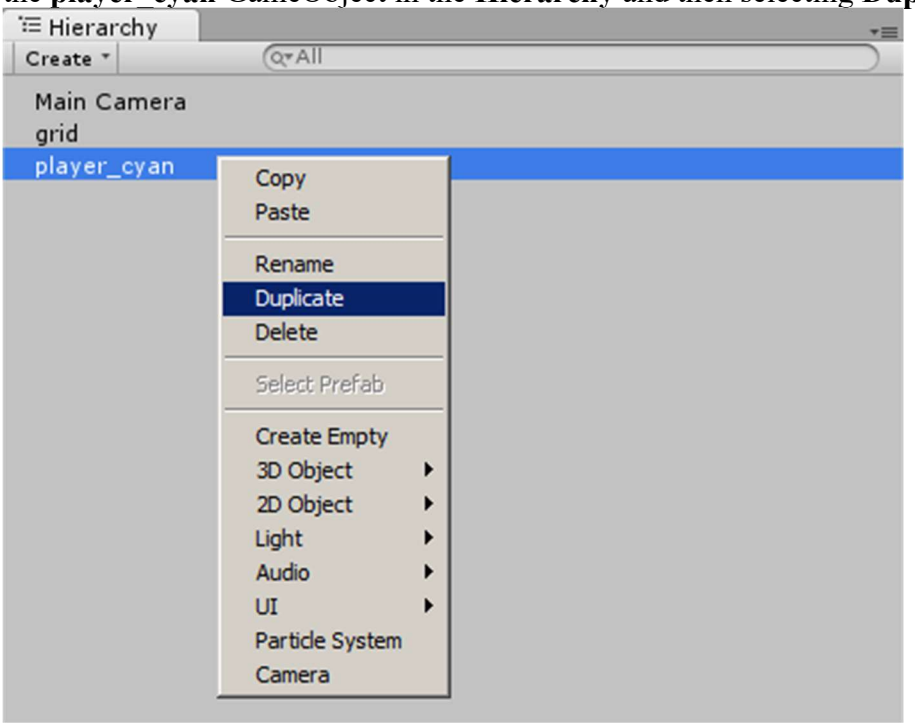
```
void fitColliderBetween(Collider2D co, Vector2 a, Vector2 b) {
    // Calculate the Center Position
    co.transform.position = a + (b - a) * 0.5f;

    // Scale it (horizontally or vertically)
    float dist = Vector2.Distance(a, b);
    if (a.x != b.x)
        co.transform.localScale = new Vector2(dist + 1, 1);
    else
        co.transform.localScale = new Vector2(1, dist + 1);
}
```

If we save the Script and press **Play** then we can see some perfectly matching corners:

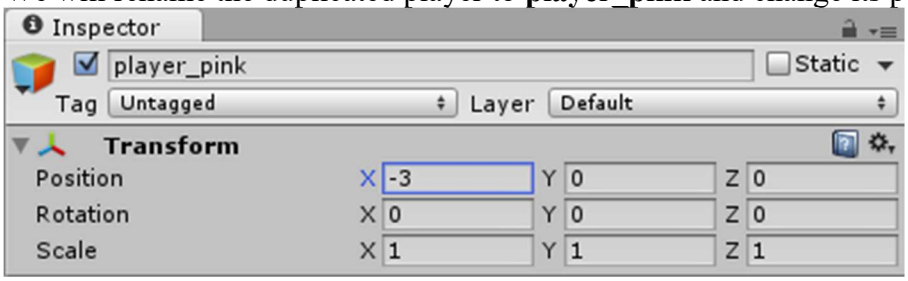


Adding another Player

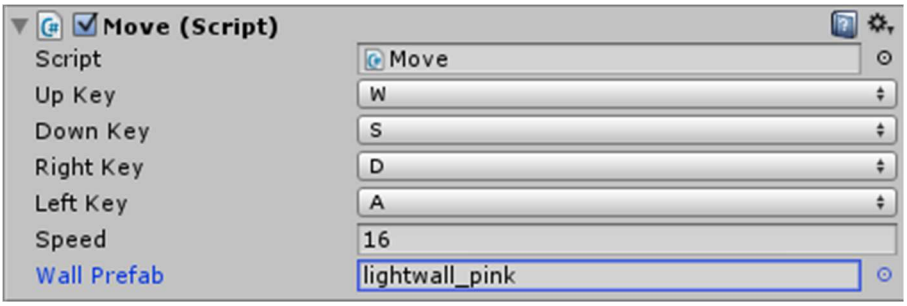
Alright, let's add the second player to our game. We will begin by right clicking the **player_cyan** GameObject in the **Hierarchy** and then selecting **Duplicate**:



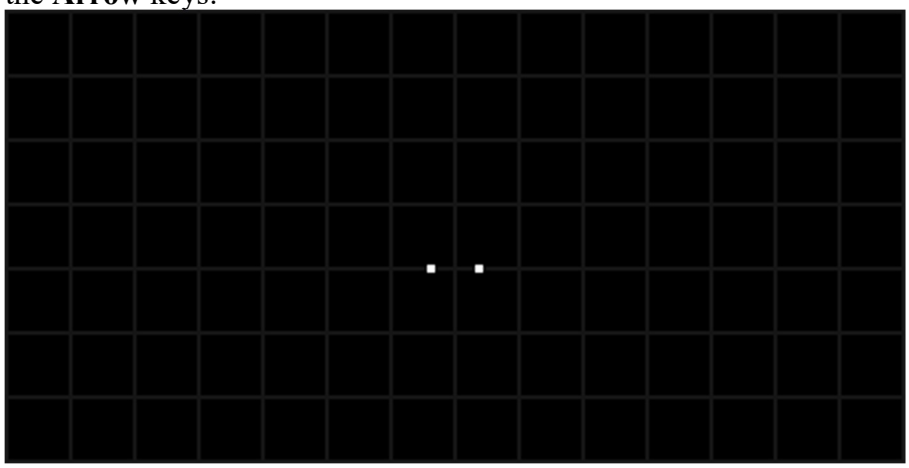
We will rename the duplicated player to **player_pink** and change its position to **(-3, 0, 0)**:



Let's also change the movement keys to **WSAD** and drag the **lightwall_pink** Prefab into the **Wall Prefab** slot:



If we press **Play** then we can now control two players, one with the **WSAD** keys and one with the **Arrow** keys:



Collision Detection

Alright so let's add a **lose** condition to our game. A player will lose the game whenever he moves into a wall.
We already added the Physics components *(Colliders and Rigidbodies)* to the Lightwalls and to the players, so all we have to do now is add a new **OnTriggerEnter2D** function to our **Move** Script. This function will automatically be called by Unity if a player collides with something:

```
void OnTriggerEnter2D(Collider2D co) {
    // Do Stuff...
}
```

The '**Collider2D co**' parameter is the Collider that the player collided with. Let's make sure that this Collider is not the wall that the player is currently dragging along behind him:

```
void OnTriggerEnter2D(Collider2D co) {
    // Not the current wall?
    if (co != wall) {
        // Do Stuff...
    }
}
```

In which case it must be any other wall, which means that the player lost the game. We will keep it simple here and only **Destroy** the player:

```
void OnTriggerEnter2D(Collider2D co) {
    // Not the current wall?
    if (co != wall) {
        print("Player lost:" + name);
        Destroy(gameObject);
    }
}
```

Note: feel free to add some kind of win/lose screen at this point.

Summary

We just created a Tron Light-Cycles style 2D game in Unity. We dove deeper into Unity GameObjects, Prefabs and Transforms, while hopefully learning some useful scripting along the way.

The game offers lot's of potential, there are all kinds of features that could still be added:

- Win/Lose Screen
- More than 2 Players
- Online Multiplayer
- AI
- Some “Power Ups” (like a speed power up)
- Some special effects
- Better Sprites

When making further improvements to the game, always remember: Unity is simple, you can do almost everything with just a few mouse clicks or a few lines of C# code.

There are also plenty of resources online to learn. Check out Unity’s official YouTube channel for detailed tutorials and walkthroughs, or search for other resources online. Unity also has a very active and passionate community and you can always check out their forums if you have any questions.