Unity 2D Pong Game

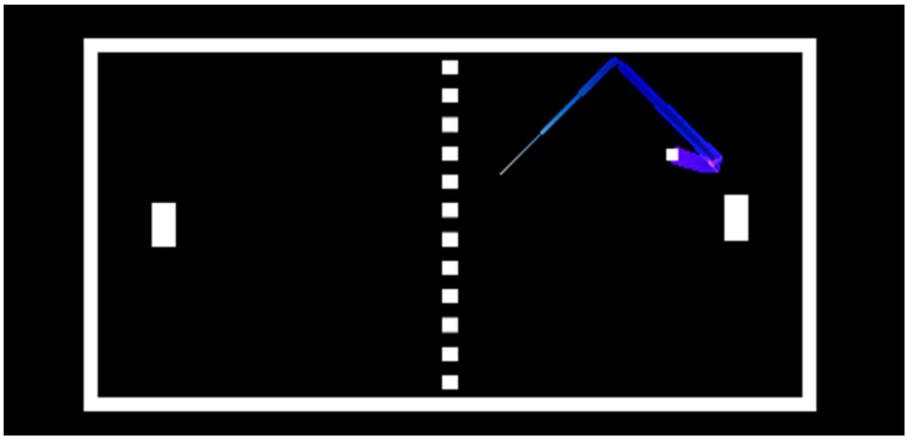
Intro

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

We will cover how to install and use Unity, create a basic Scene with just some textures, use Unity's Physics and create Scripts to add custom game mechanics.. The programming for this tutorial is only 38 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).

Note: Optional portions of this tutorial will be written in **Red**.

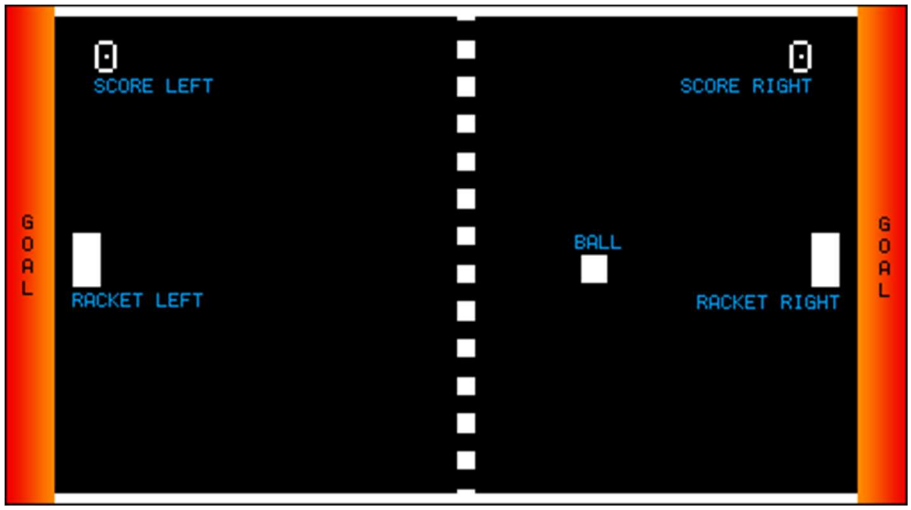
For our first tutorial we will be recreating the classic game Pong! Here is a preview:



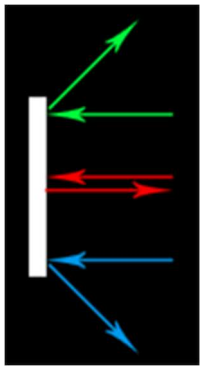
After opening the project folder you will notice two scenes (these are the things with the unity icon), one called "**New Scene**" another called "**finished_game**". You should currently have "**New Scene**" selected (if not double click to open that scene).

"**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**".

Game Concept



The game rules should be very similar to the original Pong. The right player scores a point if the ball hits the left wall. The left player scores a point if the ball hits the right wall. If it hits the top or bottom wall then it bounces off. Each player will have a racket that can be moved up and down to hit back the ball.
Our rackets should have an influence on the ball's outgoing angle:



- If the racket hits the ball at the **top** corner, then it should bounce off towards our top border.
- If the racket hits the ball at the **center**, then it should bounce off towards the right, and not up or down at all.
- If the racket hits the ball at the **bottom** corner, then it should bounce off towards our bottom border.

About the Unity Engine

In order to make our idea a real game, we will have to download and install the Unity Engine first.

While Unity is downloading, let's talk about the Engine a bit. If someone would have to summarize Unity in one word, it would be **simplicity**.

Unity is the first Game Engine that is really simple to use *(even for normal people)*, while still being perfectly suitable for professional games that demand high performance and realistic

graphics. In Unity everything is really simple, no matter if it's creating Animations, making a car explode, creating 2D and 3D Games or just making a world with realistic Physics.
One of the greatest features about Unity is the deployment: after creating our game once, we can deploy it to Windows, Mac, Linux, Android, iOS, XBox, PS3 and more. What usually takes months of hard work can be done with just one click in Unity.

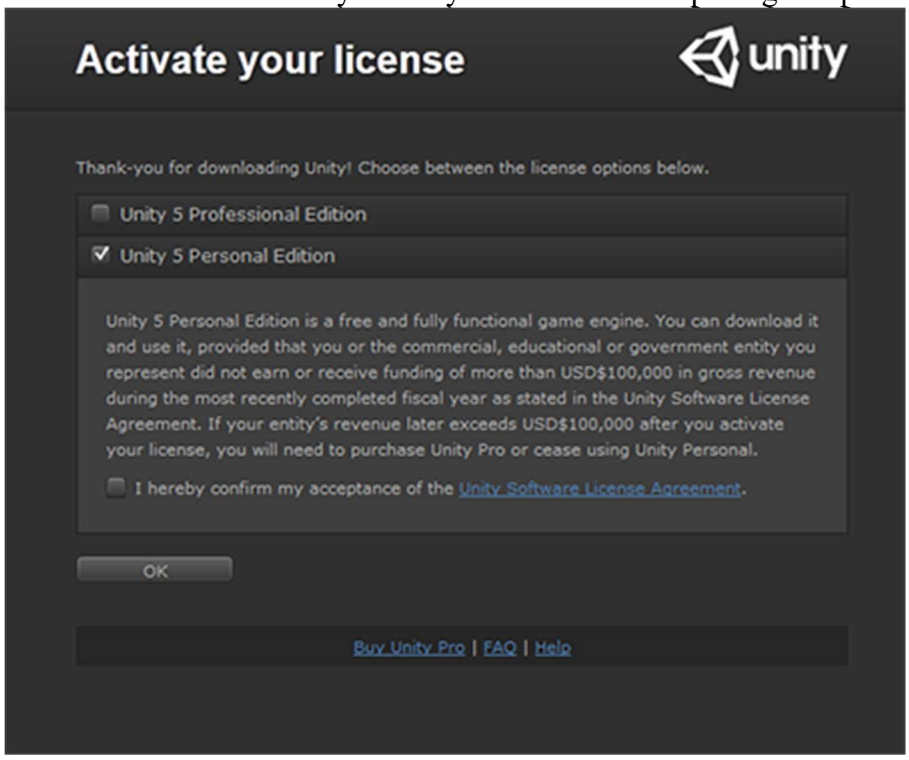
Every game we create will involve the same simple steps: we create a Project, we put a few things into our Scene and move them around interactively until they are where we want them to be, and then we do some Scripting to implement game logic, respond to events or control our player(s).

When it comes to Scripting we have several choices: C#, Javascript and Boo. In this Tutorial we will use C#, because it is the most commonly used and it is a very structured programming language that develops good programming habits. It is also most similar to Java, which is the language most CS students at Cal Poly learn first.

After the Unity Download was finished, we open the downloaded file and progress through the Installation.

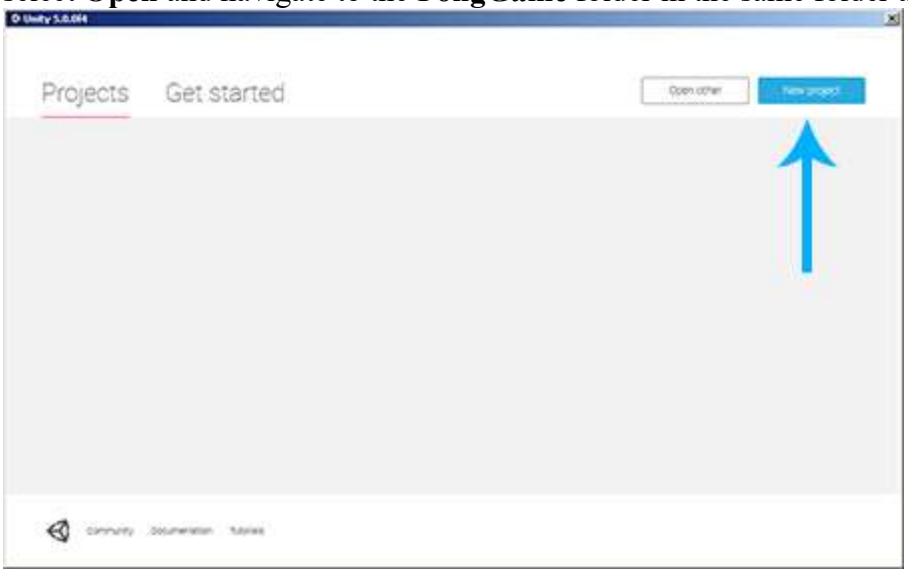
Setting Up Unity

Now it's time to start Unity. What you will see after opening it is probably something like this:

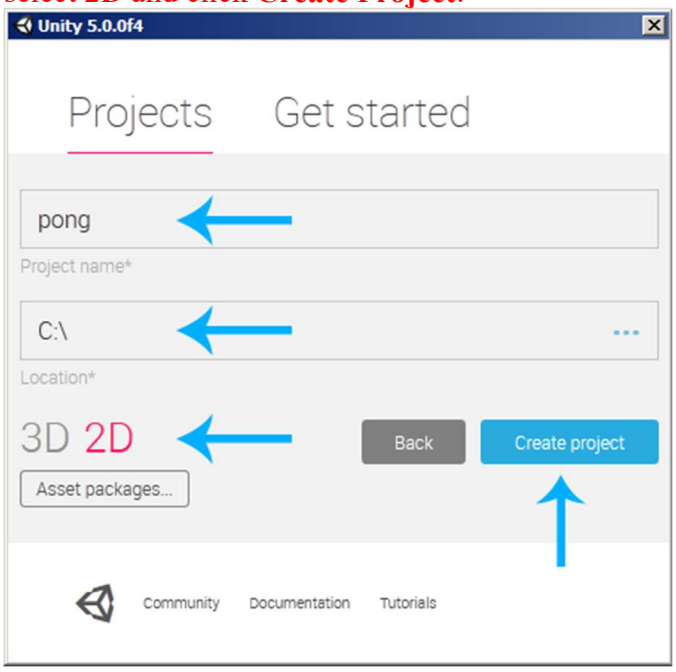


Simply select **Unity 5 Personal Edition**, accept the **License Agreement**and press **OK** afterwards.
Note: the Personal Edition is free, can be used forever, and you can create and release your Games without a problem.

If you are now asked to login into your Unity account, just create one and then login with it, afterwards the Project Wizard will start. Here you could create a New Project, but in this case select **Open** and navigate to the **PongGame** folder in the same folder as this tutorial file.

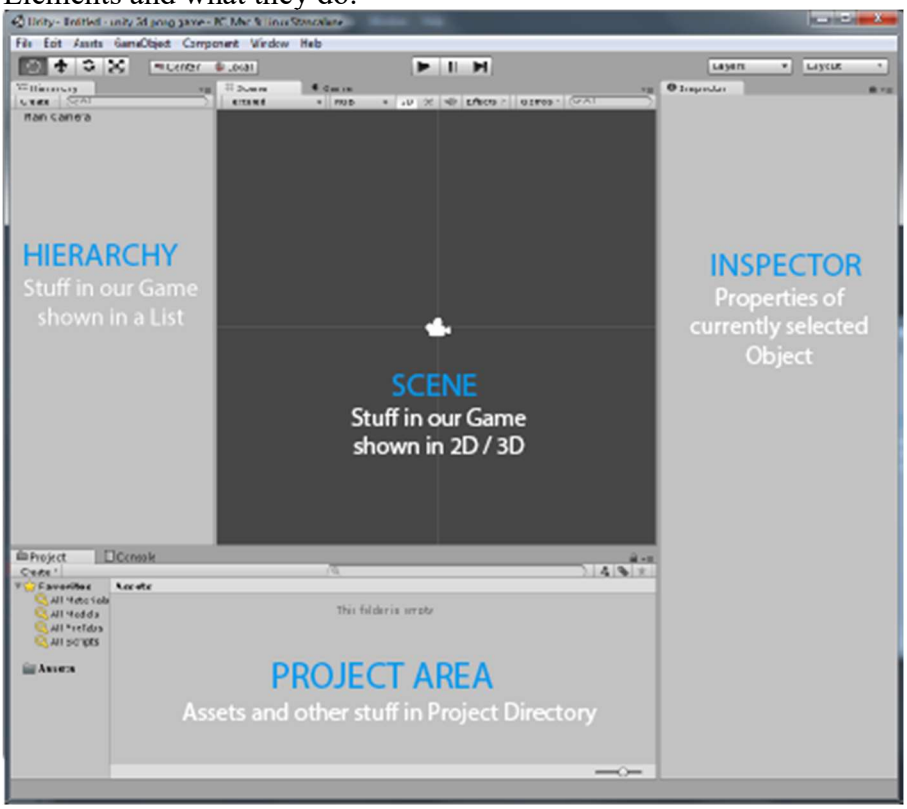


If we wanted to create a **New Project** we would name it **pong**, select any location like **C:**, select **2D** and click **Create Project**:



Exploring Unity

Let's close the Welcome screen and get familiar with the Engine a bit. Here are the main Elements and what they do:



Hierarchy:

On the left side we can see the **Hierarchy**, which contains a list of all the things that are currently in our game. As we can see, currently it's only the **Main Camera**, which was added by Unity automatically to make our life easier.

Scene:

We can see the game world in the **Scene** view. We can find any game object in the Scene by double clicking it in the Hierarchy. To navigate around the Scene, we just click into it once and then use the left, right and middle mouse buttons. Don't be afraid to play around with it a bit until you mastered it.

Inspector:

The Inspector shows the properties of the currently selected object(s). For example, if we select the Main Camera in the Hierarchy then we can see its Position, Rotation, Name and all other things that are relevant about it.

Unity is like many other Game Engines, Component-Based. Everything is just an empty Object first. Then things are added to it, for example a Light, a Position, a Texture, a 3D Model and more. In case of the Camera, the Components are Transform, Camera, GUILayer, Flare Layer and Audio Listener. All those Components are put onto one Object, and all together make our

Main Camera which can view something, hear sounds, change its position and so on. Don't worry if this sounds confusing, we will do it over and over again and soon we will see that this is the easiest way possible to create games.

Project Area:

The Project Area contains all kinds of Assets like Textures, 3D Models or Scripts. If we want, we can use those Assets in our game by dragging them into the Hierarchy or into the Scene.

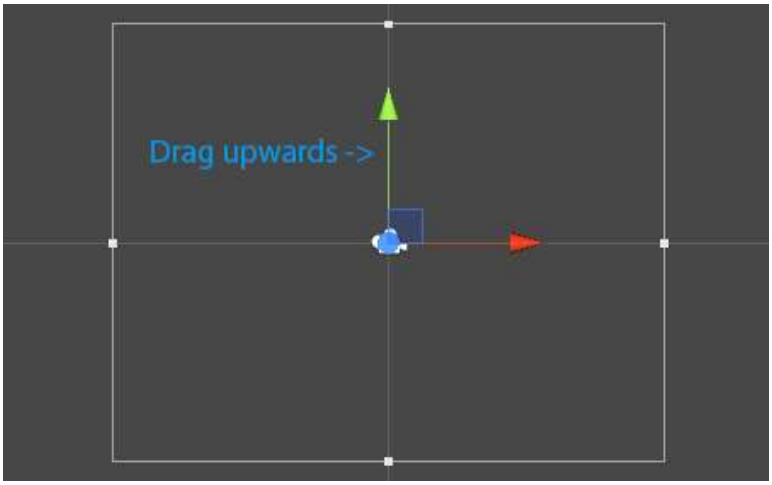
A scene is where you will place all objects for your game. A scene could be your entire game or it could be one level of your game.

Unity Controls

Moving, Rotating, Scaling:

Let's have some fun and move around the Camera. We can either change the position, rotate it or make it bigger *(or smaller)*. That's what those buttons are for:

The Hand Button is to navigate around in the Scene, the Cross-Arrows are for moving objects around *(Hotkey: **W**)*, the Circle-Arrows are for rotating them *(Hotkey: **E**)*, and the thing in the right is to scale them, so we can make it bigger or smaller *(Hotkey: **R**)*.
Just click one of those Buttons and then interact with the Blue/Green/Red lines around the Camera in the Scene View:



The Play Button

Let's run our game and see how it looks by pressing the **Play** Button:

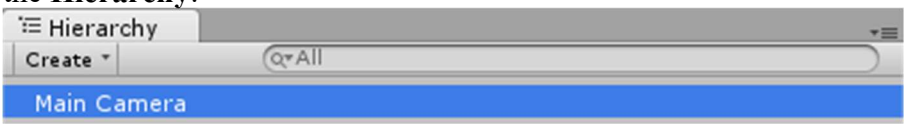
When pressing Play, we always see the Game World through the eyes of our Camera. So far we only see the blue background because there is nothing in front of the camera yet.

Note: to go back to the editing mode, press the Play button again (to stop playing).
That's it, now we know everything about Unity that we need to know to make a 2D Pong Game. Don't forget to Save the scene *(Top Menu: **File->Save Scene**)*.

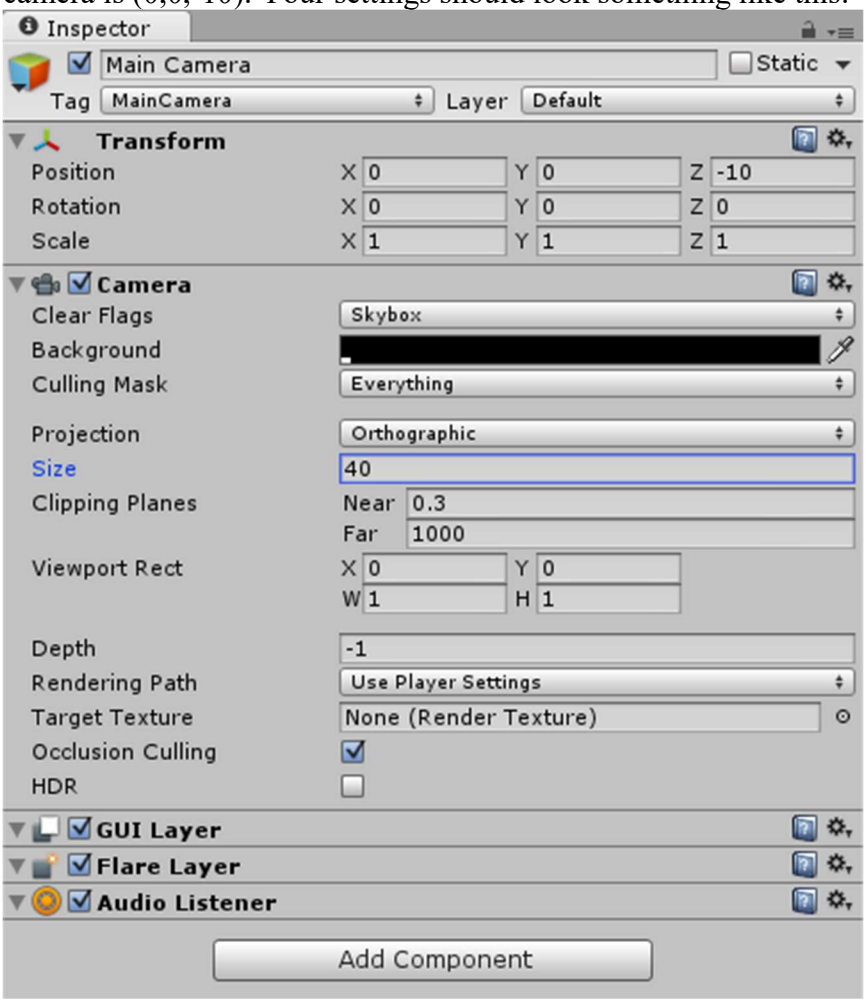
Lets start making our game!

Camera Setup

Alright, we will begin by modifying the Camera so that it shows the game in the correct size and with the right background color. We can modify the Camera settings by first selecting it in the **Hierarchy**:



Afterwards we can see the settings in the **Inspector**. We will change the **Background** color to black and adjust the **Size** so that it fits our game later on. Also make sure the position of the camera is (0,0,-10). Your settings should look something like this:



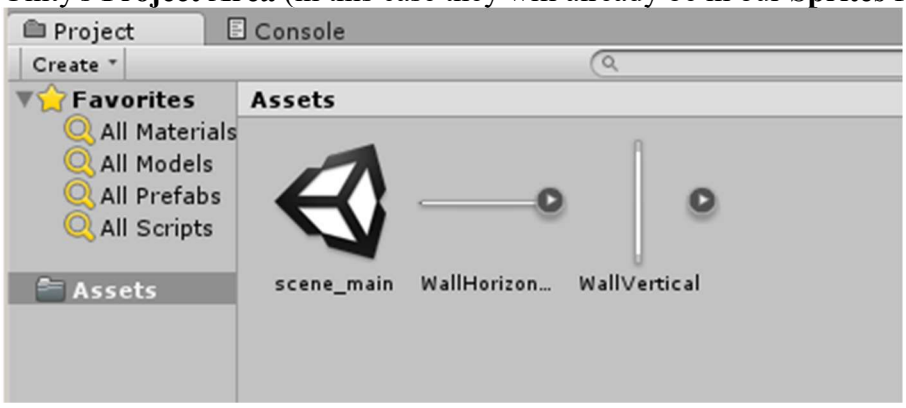
Creating the Walls

The Wall Texture

Let's add four walls to our game. All we need to make a wall is a so called **Sprite**, or in other words: a Texture.
We will use one horizontal Sprite for the top and bottom walls and one vertical Sprite for the left and right walls:

*Note: You can import textures, objects, and scripts from anywhere by simply saving them in the project's **Assets** folder. They will then be immediately ready to use in your project.*

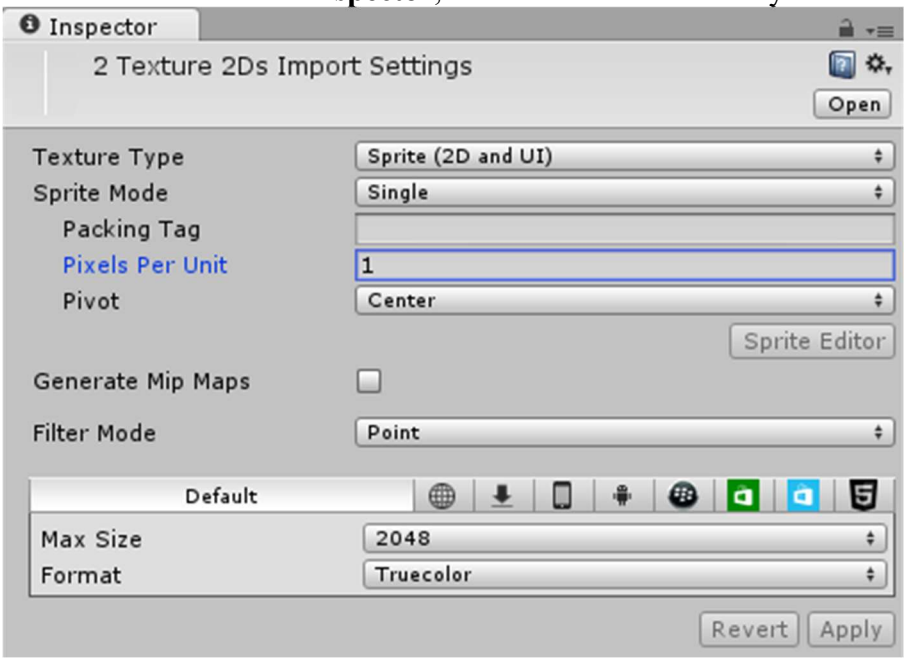
All of our assets for this tutorial should already be imported. You can see them in Unity's **Project Area** (in this case they will already be in our **Sprites** folder):



*Note: select **Assets** instead of **Favorites** in the Project area in case you don't see them.*

The Wall Import Settings

If we take a look at the **Inspector**, our texture should already have these **Import Settings**:



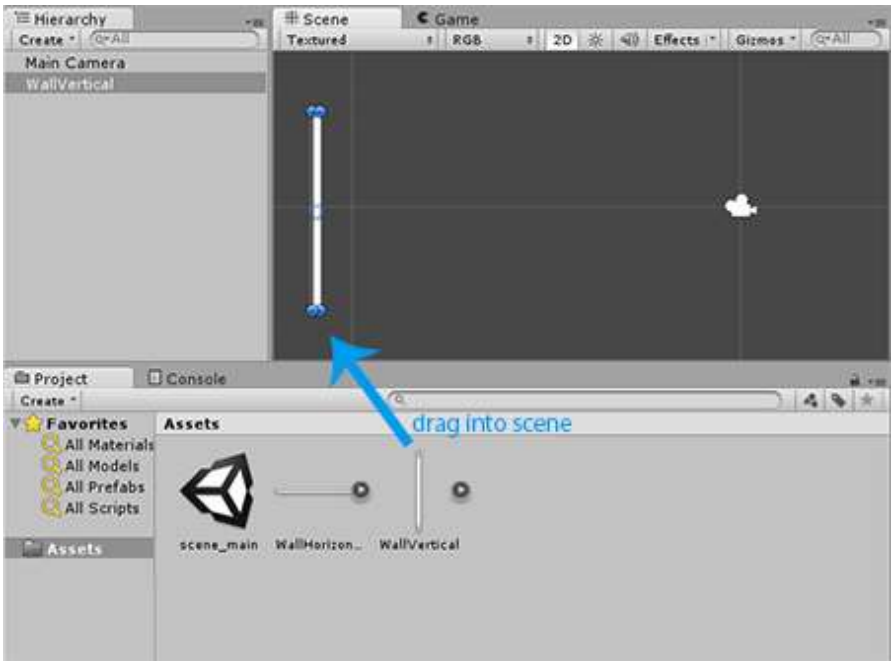
*Note: the **Filter Mode** and **Format** can be used to decide between quality and performance.*

*A **Pixels Per Unit** value of **1** means that **1 x 1** pixels will fit into one unit in the game world. We will use this value for all our textures, because the ball will be **1 x 1** pixels later on, which should be exactly one unit in the game world.*
Modifying the **Import Settings** might seem like a weird thing to do if you are new to Unity. As a matter of fact, our game would work just fine without ever touching the Import Settings at all. However in 2D games it's usually a good idea to modify those settings so that the world size is something reasonable *(we don't want a 100 meter huge racket, this could make the physics somewhat tricky)*.

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.

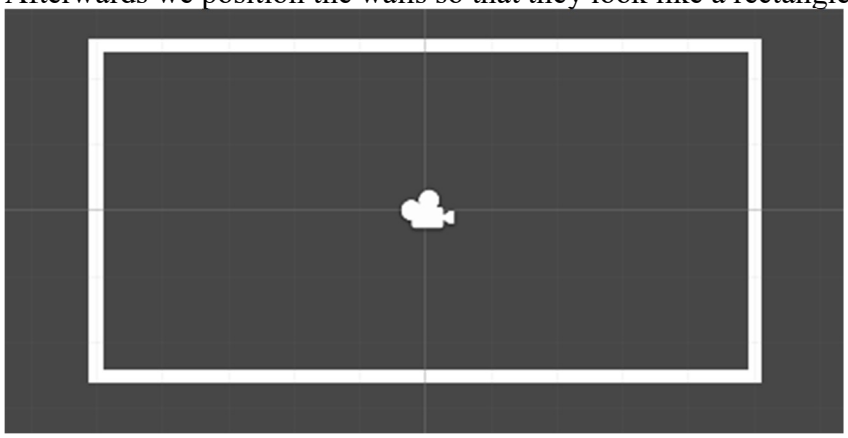
Adding the Walls to the Game World

So in order to add the Walls to our game, all we have to do is drag them from the **Project Area** into the **Scene**:



Select our **WallHorizontal** and **WallVertical** textures from the Sprites folder and drag each texture into the Scene **twice** so that we have 4 walls:

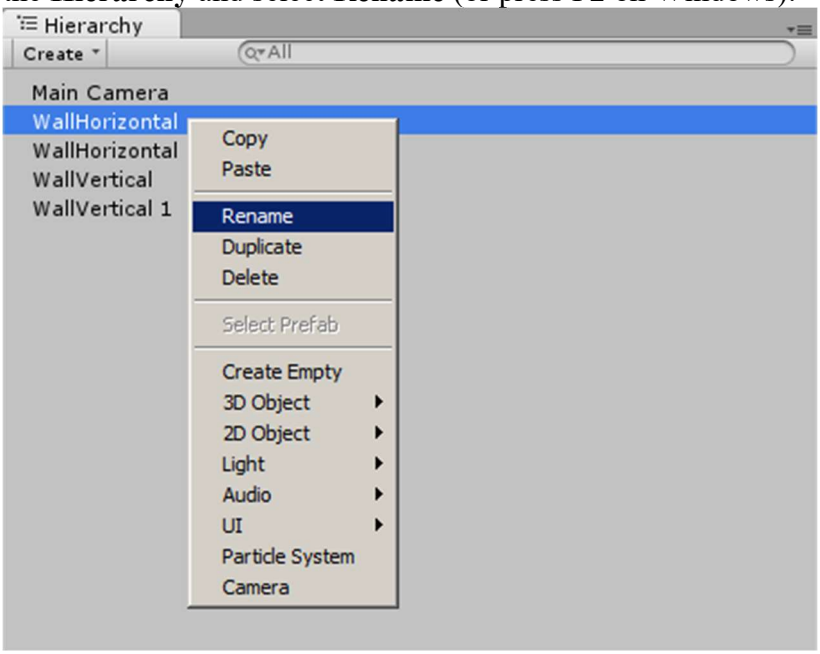
Afterwards we position the walls so that they look like a rectangle with the Camera in the center:



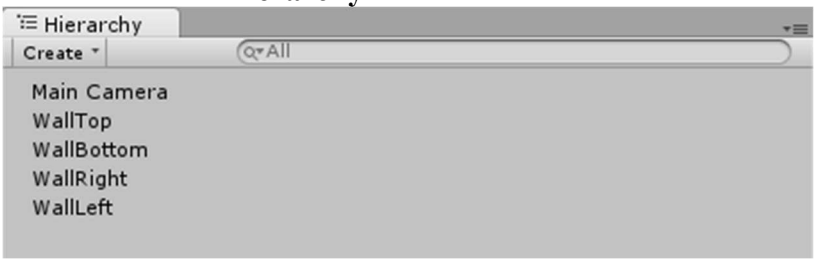
*Note: we can position the walls by either dragging them around, or by selecting them and then changing their **Position** in the **Inspector**.*

Renaming the Walls

We will also rename the Walls to **WallLeft**, **WallRight**, **WallTop** and **WallBottom** so that we don't lose the overview later on. Renaming is very easy, all we have to do is right click a wall in the **Hierarchy** and select **Rename** (or press F2 on Windows):



Here is what our **Hierarchy** looks like afterwards:

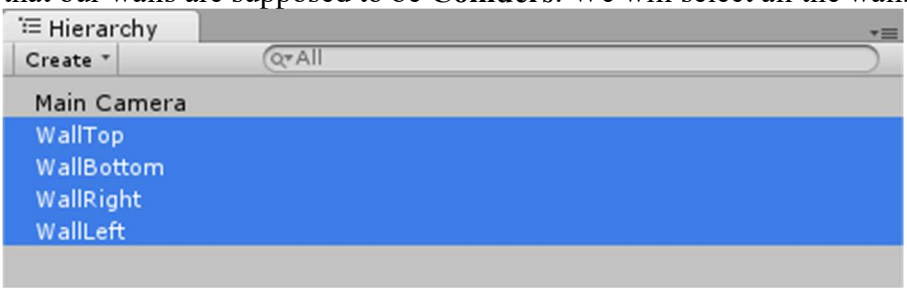


Wall Physics

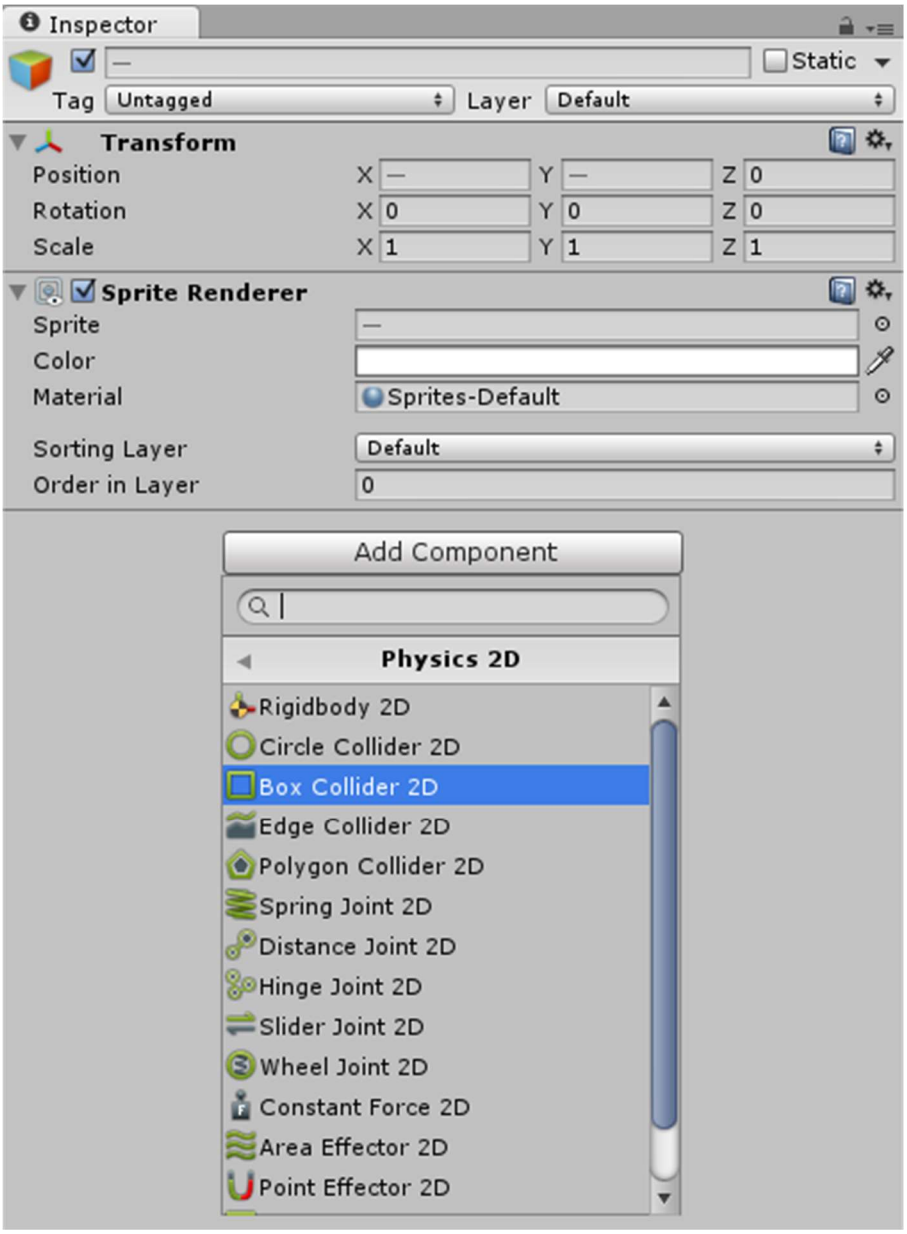
Right now we can see the walls in the game, but they aren't real walls yet. They are just images in the game world, a purely visual effect.

We want the walls to be real walls so that the Rackets and the Ball will collide with them instead of just going right through them.

Unity comes with an incredibly powerful physics engine, and all that we have to do is tell Unity that our walls are supposed to be **Colliders**. We will select all the walls in the **Hierarchy**:

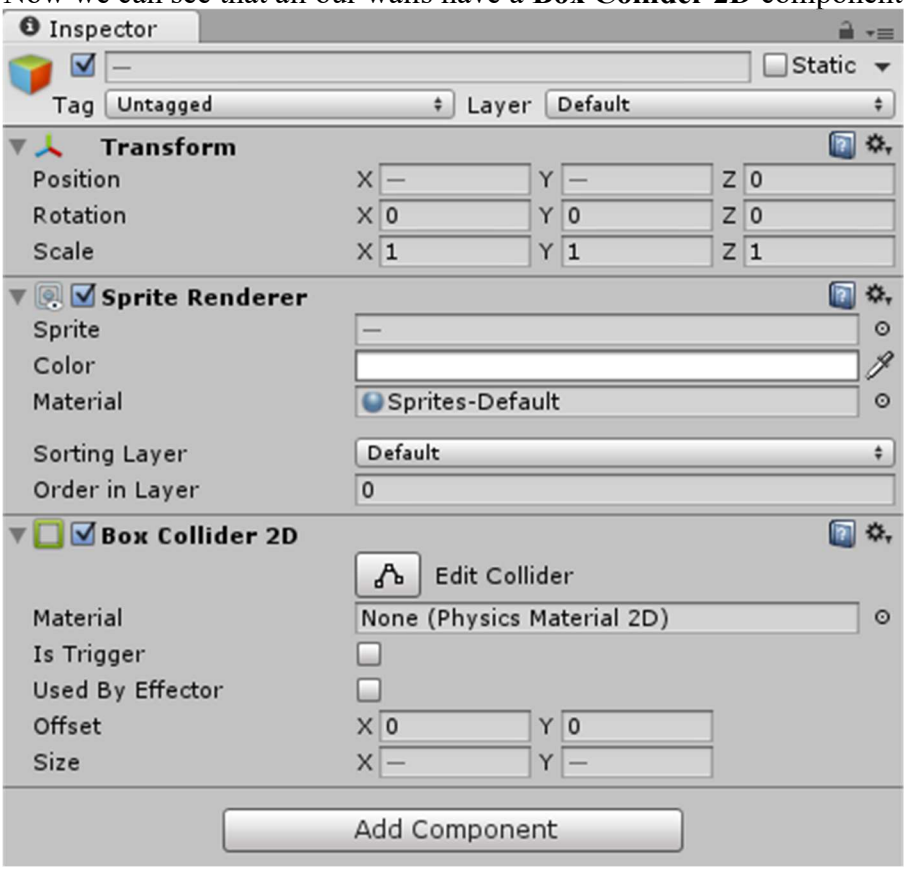


Afterwards we click on the **Add Component** button in the **Inspector** and then select **Physics2D->Box Collider 2D**:



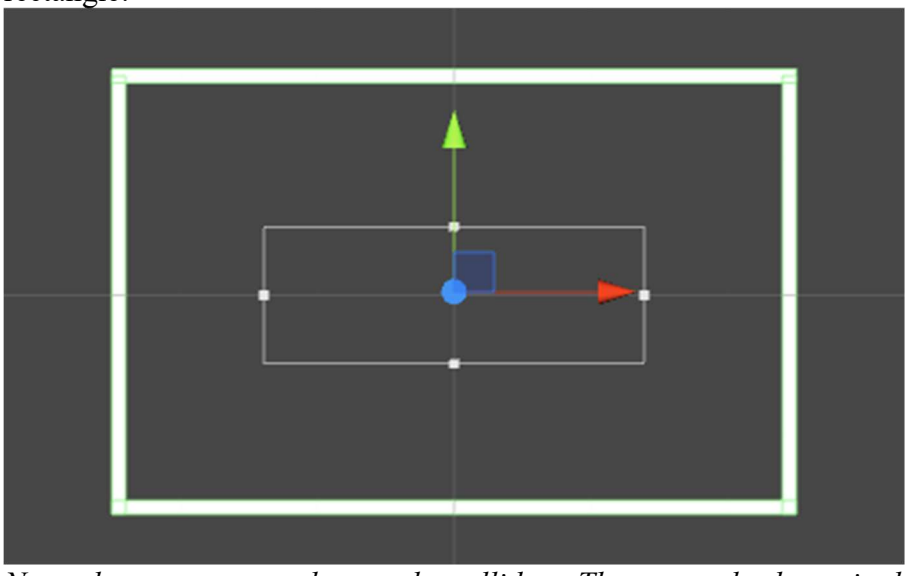
Note: whatever we do in the Inspector will be done for all objects that are selected in the Hierarchy. And because we selected all four walls, they will all have a Collider now.

Now we can see that all our walls have a **Box Collider 2D** component in the Inspector:



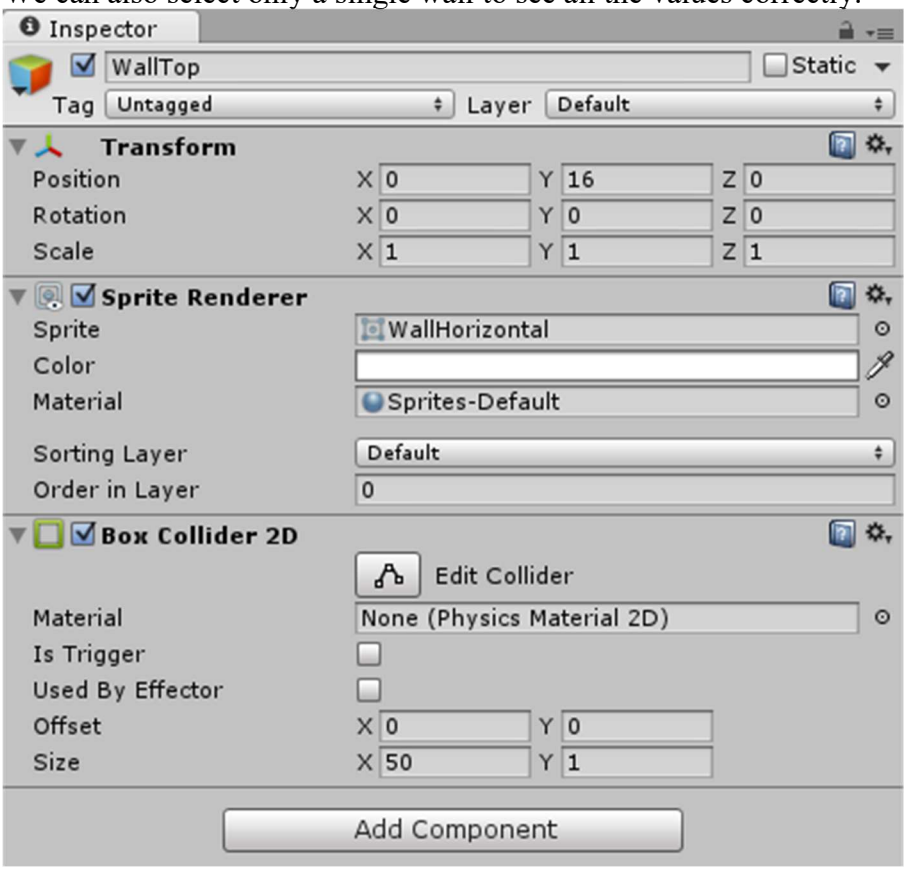
*Note: the **--** values are the ones that are different between the selected GameObjects.*

If we take a look at the **Scene** then we can also see that each wall is now surrounded by a green rectangle:



Note: the green rectangles are the colliders. They are only shown in the Scene and not in the final game.

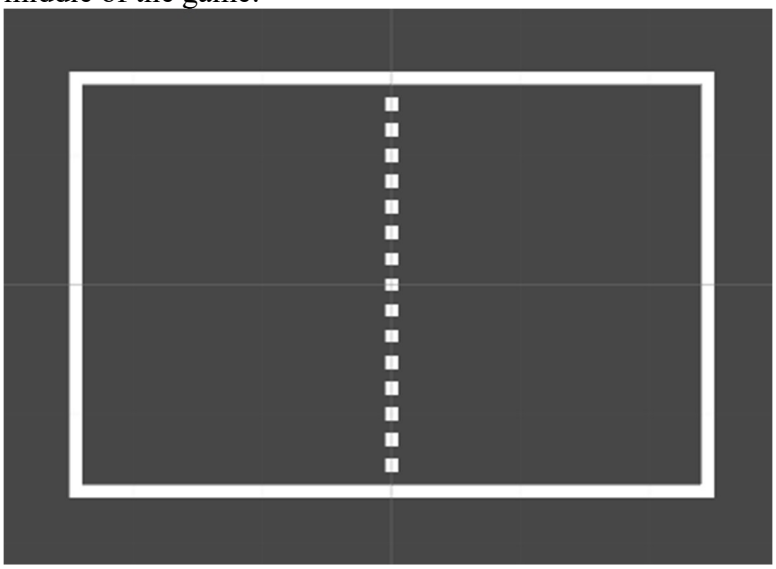
We can also select only a single wall to see all the values correctly:



Adding the Dotted Line

Alight let's add the dotted line in the middle. Select the **DottedLine** texture in the **Project Area**.

Afterwards we can drag it from the **Project Area** into the **Scene**. We will position it in the middle of the game:



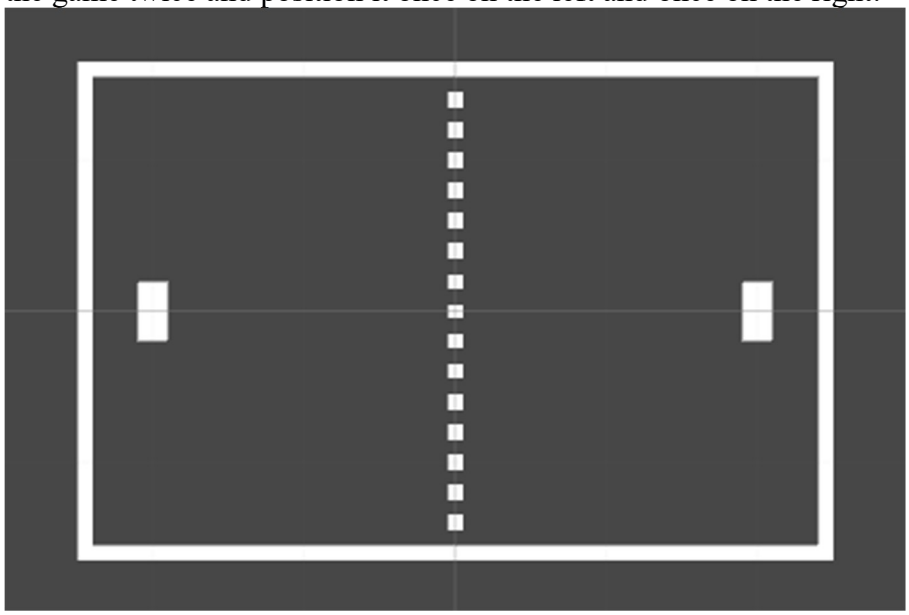
*Note: the dotted line is a great example to understand how Unity's Physics work. Right now the dotted line is just a texture. Just something that we can **see**. The ball will not collide with the dotted line unless we add a **Collider** to it (which we won't, because the ball is not supposed to collide with it).*

Creating the Rackets

The Racket Texture

We will use yet another white texture for the rackets. Select the **Racket** texture in the **Project Area**.

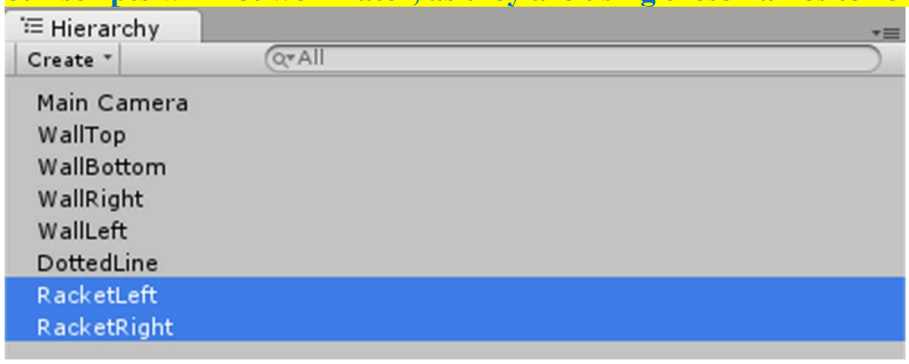
Our game will have two players, one on the left and one on the right. So let's drag the racket into the game twice and position it once on the left and once on the right:



Renaming the Rackets

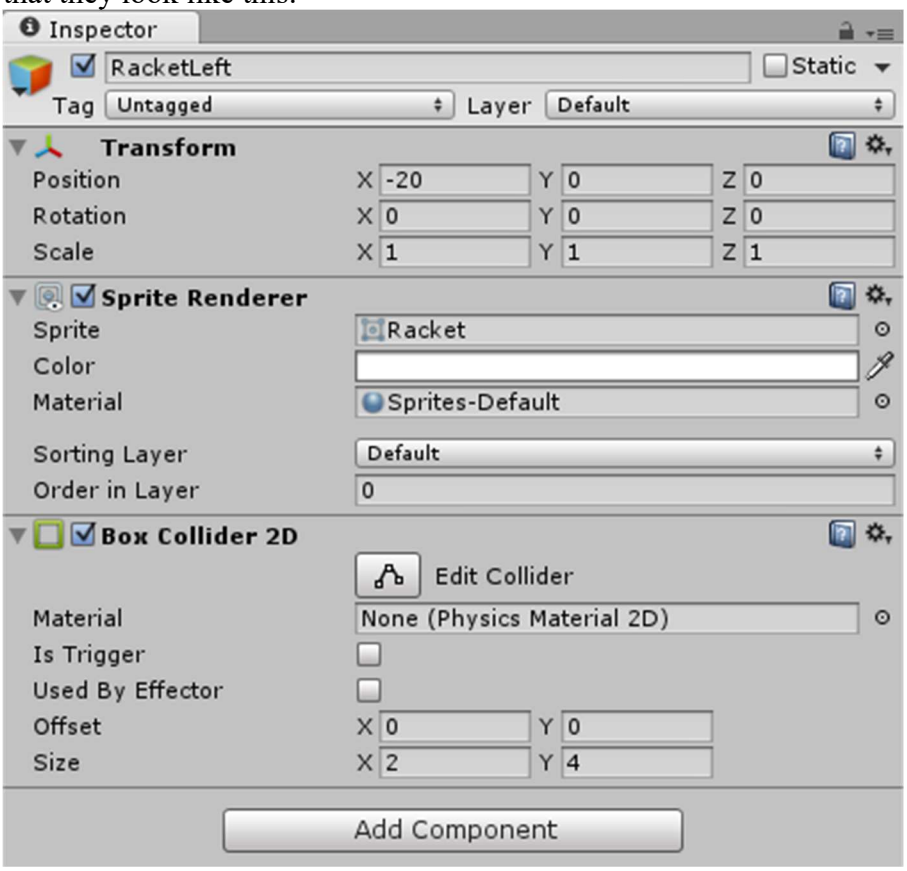
Again to make our lives easier later on, we will rename the two rackets to **RacketLeft** and **RacketRight** in the **Hierarchy**:

Note: in this case the name is important. The rackets must be named exactly as above or our scripts will not work later, as they are using these names to reference the rackets.



Racket Physics

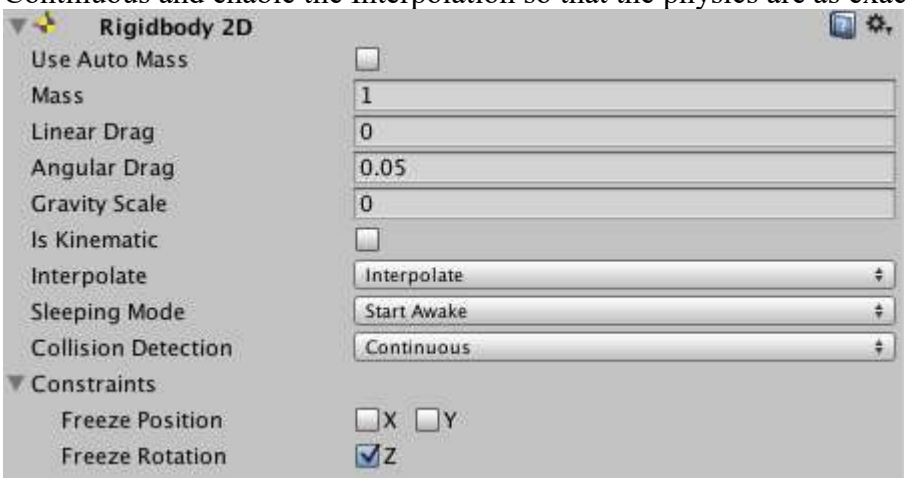
Okay so our Rackets should make use of Unity's Physics Engine. At first they should be able to collide with the wall, hence why we add colliders again by selecting both Rackets in the Hierarchy, then pressing **Add Component->Physics 2D->Box Collider 2D** in the **Inspector** so that they look like this:



The player should also be able to move the Racket upwards and downwards later on. But the Rackets should stop moving upwards *(or downwards)* when they collide with a wall.
What sounds like some complicated math will be ridiculously easy in Unity, because a **Rigidbody** does just that. It always adjusts an object's position so it's physically correct. For example, it can automatically apply gravity to the object, or it can make sure that our Rackets will never move through a wall.
*Note: as a rule of thumb, everything physical that moves through the game world will need a **Rigidbody**.*

To add a Rigidbody to our Rackets we just select both of them in the Hierarchy again, then take a look in the **Inspector** and press **Add Component->Physics 2D->Rigidbody 2D**. We then modify the **Rigidbody 2D** to disable **Gravity** *(because there is no gravity in a pong game)*, enable **Freeze Rotation Z***(the rackets should never rotate)* and set the **Collision Detection** to

Continuous and enable the Interpolation so that the physics are as exact as possible:

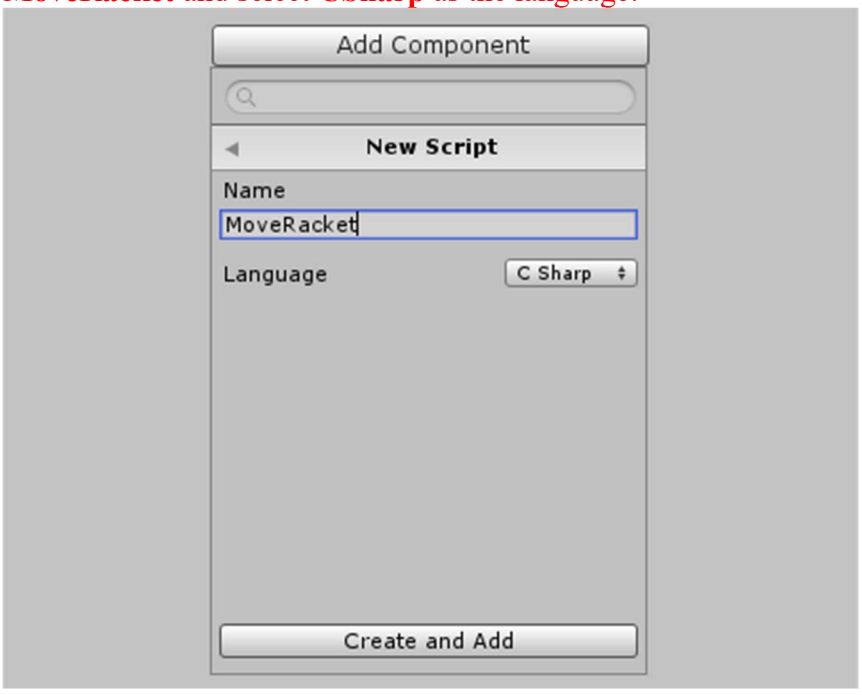


Racket Movement

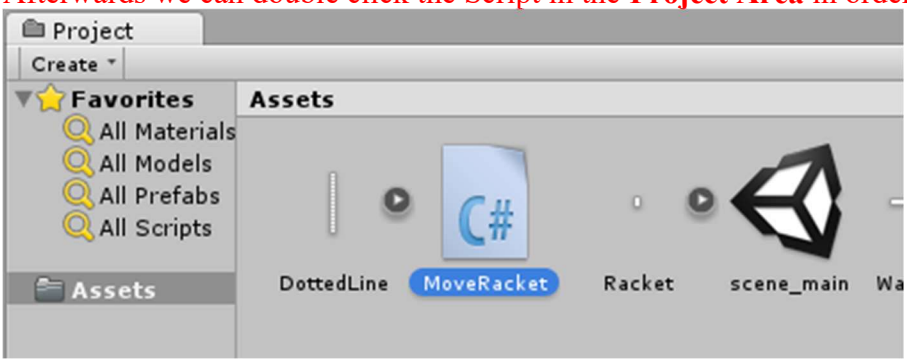
Let's make sure that players can move their rackets. That kind of custom behavior usually requires **Scripting**. With both rackets still selected, we will click on **Add Component->MoveRacket**:

*Note: The following instructions may include some programming, if you wish to skip the programming portion of the tutorial just ignore all instructions in **red**. However, you are highly encouraged to check out the code though because it is not very complicated and will give you a better understanding of how Unity physics and scripting works.*

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



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



Here is what our Script currently looks like:

```
using UnityEngine;
using System.Collections;

public class MoveRacket : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

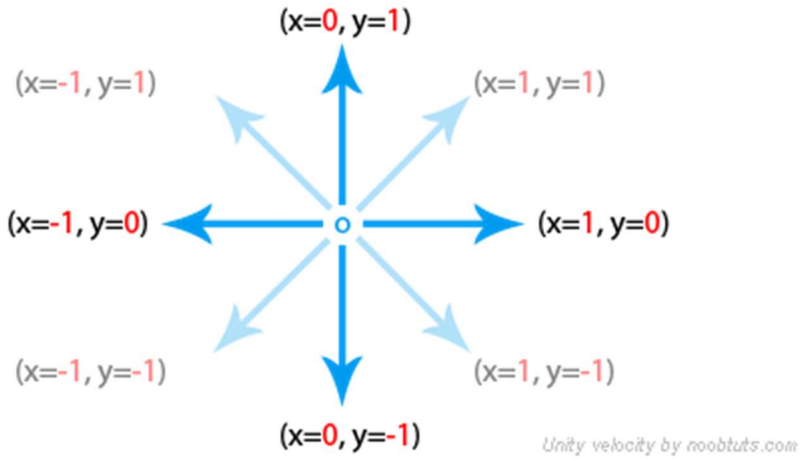
The **Start** function is automatically called by Unity when starting the game.
The **Update** function is automatically called over and over again, roughly 60 times per second. But there is another **Update** function, it's called **FixedUpdate**. This one is also called over and over again, but in a fixed time interval. Unity's Physics are calculated in the exact same time interval, so it's usually a good idea to use **FixedUpdate** when doing Physics stuff *(we want to move Rackets that have Colliders and RigidBodys, hence Physics stuff)*.

Okay so let's remove the **Start** and **Update** functions and create a **FixedUpdate** function instead:

```
using UnityEngine;
using System.Collections;

public class MoveRacket : MonoBehaviour {

    void FixedUpdate () {

    }
}
```

*Note: it's important that we name it exactly **FixedUpdate** because this is the name that Unity expects. We can also make functions with different names, but they wouldn't automatically be called by Unity then.*

The rackets have a **Rigidbody** component and we will use the Rigidbody's **velocity** property for movement. The velocity is always the movement **direction** multiplied by the **speed**.
The direction will be a **Vector2** with a **x** component *(horizontal direction)*and
a **y** component *(vertical direction)*. The following image shows a few Vector2 examples:



The rackets should only move upwards and downwards, which means that the **x** component will always be **0** and the **y** component will be **1** for **upwards**, **-1** for **downwards** or **0** for **not moving**. The **y** value depends on the user input. We could either check for all kinds of key presses *(wsad, arrow keys, gamepad sticks and so on)*, or we could simply use Unity's GetAxisRaw function:

```
using UnityEngine;
using System.Collections;

public class MoveRacket : MonoBehaviour {

    void FixedUpdate () {
        float v = Input.GetAxisRaw("Vertical");
    }
}
```

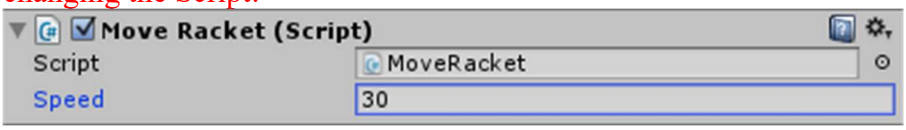
*Note: we use GetAxisRaw to check the vertical input axis. This will return **1**when pressing either the **W** key, the **UpArrow** key, or when pointing a gamepad's stick upwards. It will return **-1** when using the **S** key, the **DownArrow** key, or when pointing a gamepad's stick downwards. It will return **0** when none of those keys are pressed. Or in other words, it's exactly what we need.*
Now we can use **GetComponent** to access the racket's **Rigidbody** component and then set its **velocity**:

```
using UnityEngine;
using System.Collections;

public class MoveRacket : MonoBehaviour {

    void FixedUpdate () {
        float v = Input.GetAxisRaw("Vertical");
        GetComponent<Rigidbody2D>().velocity = new Vector2(0, v);
    }
}
```

We will also add a **speed** variable to our Script, so that we can control the racket's movement speed:

```
using UnityEngine;
using System.Collections;

public class MoveRacket : MonoBehaviour {
    public float speed = 30;

    void FixedUpdate () {
        float v = Input.GetAxisRaw("Vertical");
        GetComponent<Rigidbody2D>().velocity = new Vector2(0, v);
    }
}
```

We made the **speed** variable public so that we can always modify it in the **Inspector** without changing the Script:



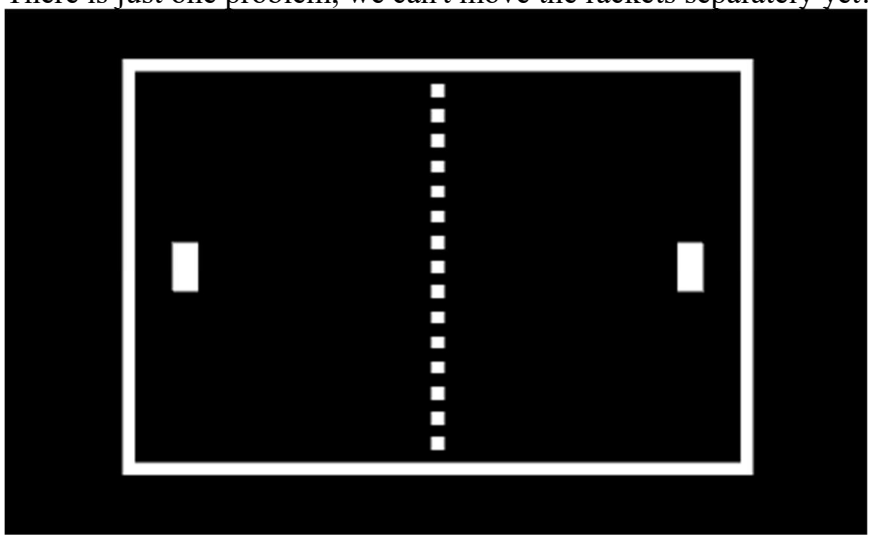
Now we can modify our Script to make use of the **speed** variable:

```
using UnityEngine;
using System.Collections;

public class MoveRacket : MonoBehaviour {
    public float speed = 30;

    void FixedUpdate () {
        float v = Input.GetAxisRaw("Vertical");
        GetComponent<Rigidbody2D>().velocity = new Vector2(0, v) * speed;
    }
}
```

*Note: we set the **velocity** to the **direction** multiplied by the **speed**, which is exactly the velocity's definition.*

If we save the Script and press **Play** then we can now move the rackets:

There is just one problem, we can't move the rackets separately yet!



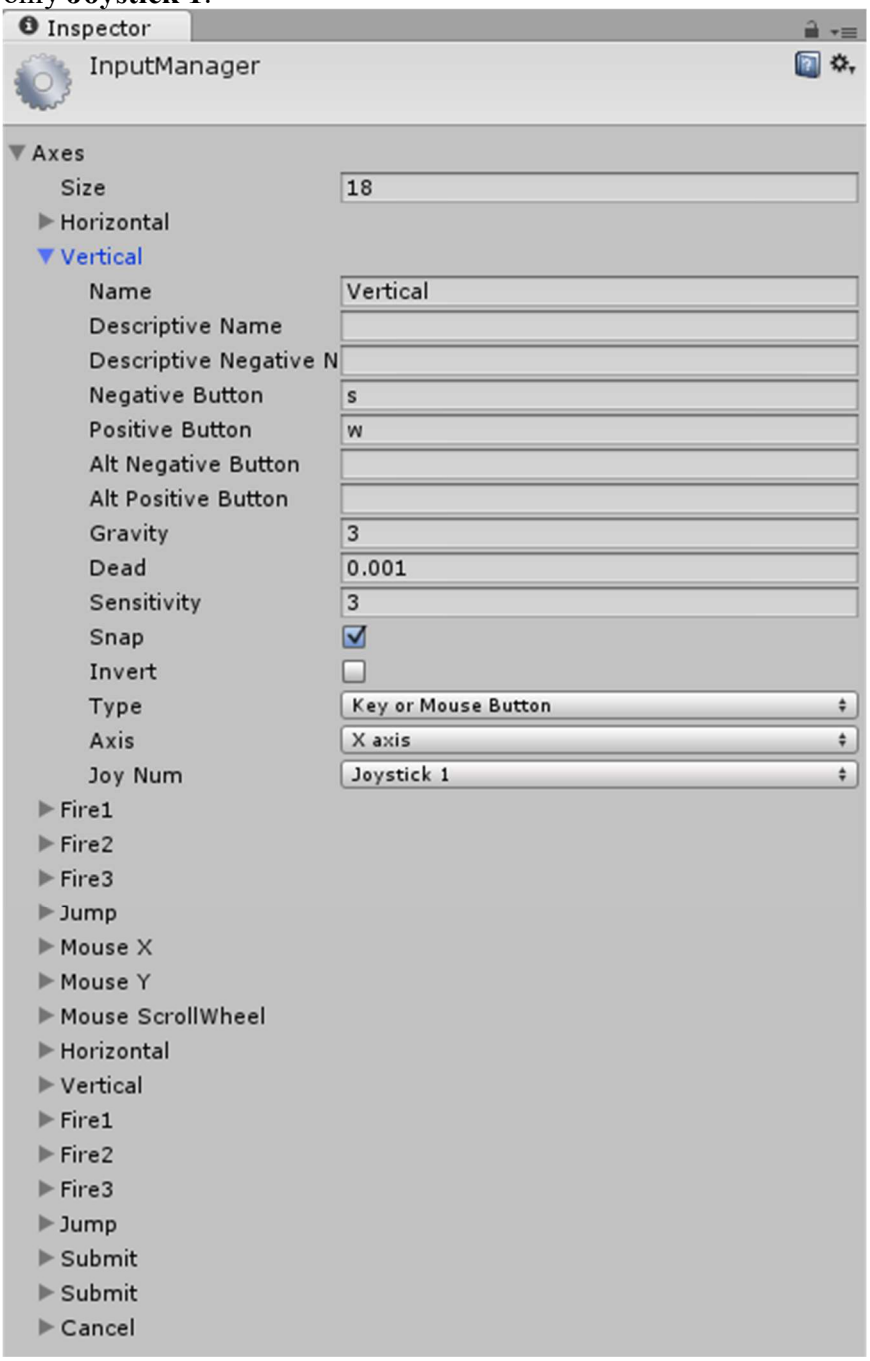
Adding a Movement Axis

Right now, both our Scripts check the **"Vertical"** Input axis for the movement calculations. Let's create a new **Axis** variable so that we can change the Input Axis in the Inspector:

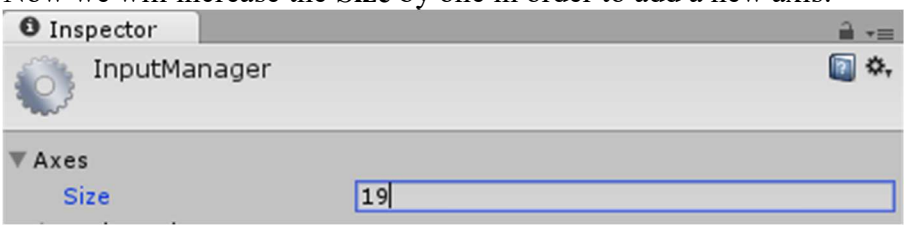
```
using UnityEngine;
using System.Collections;

public class MoveRacket : MonoBehaviour {
    public float speed = 30;
    public string axis = "Vertical";

    void FixedUpdate () {
        float v = Input.GetAxisRaw(axis);
        GetComponent<Rigidbody2D>().velocity = new Vector2(0, v) * speed;
    }
}
```

Let's select **Edit->Project Settings->Input** from the top menu. Here we can modify the current **Vertical** axis so that it only uses the **W** and **S** keys. We will also make it use

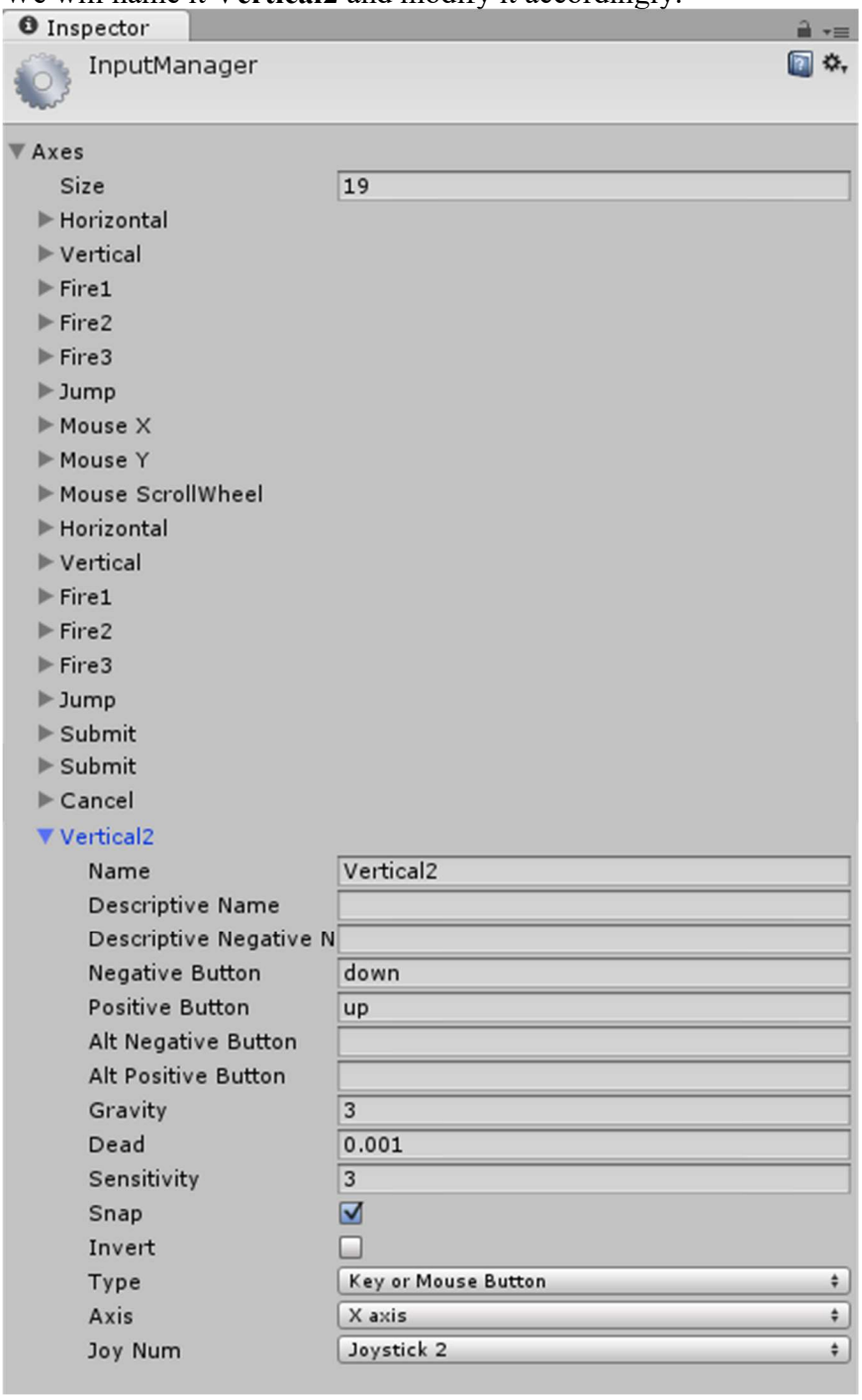
only **Joystick 1**:



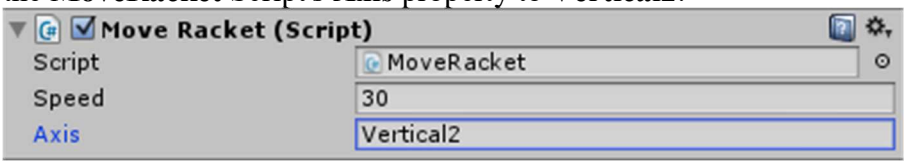
Now we will increase the **Size** by one in order to add a new axis:



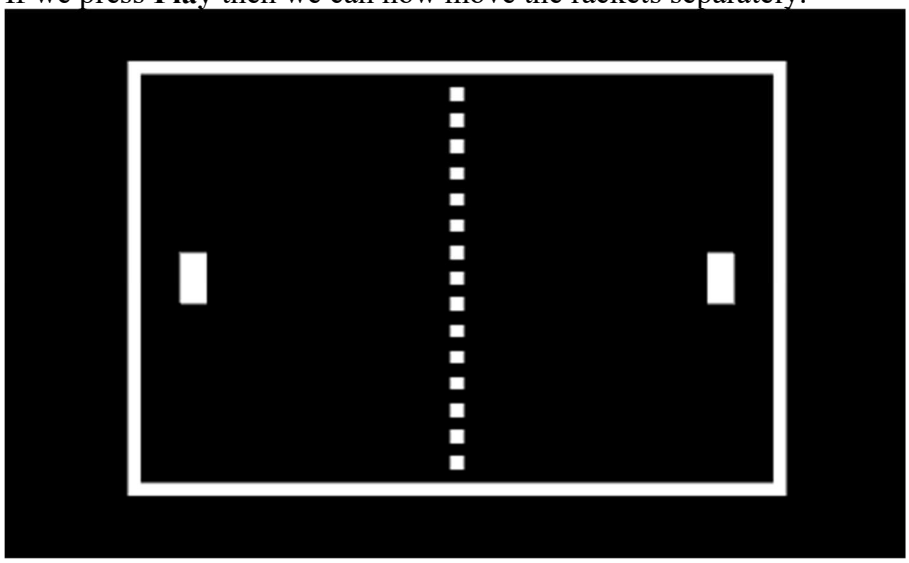
We will name it **Vertical2** and modify it accordingly:



Afterwards we will select the **RacketRight** GameObject and change the **MoveRacket** Script's **Axis** property to **Vertical2**:



If we press **Play** then we can now move the rackets separately:



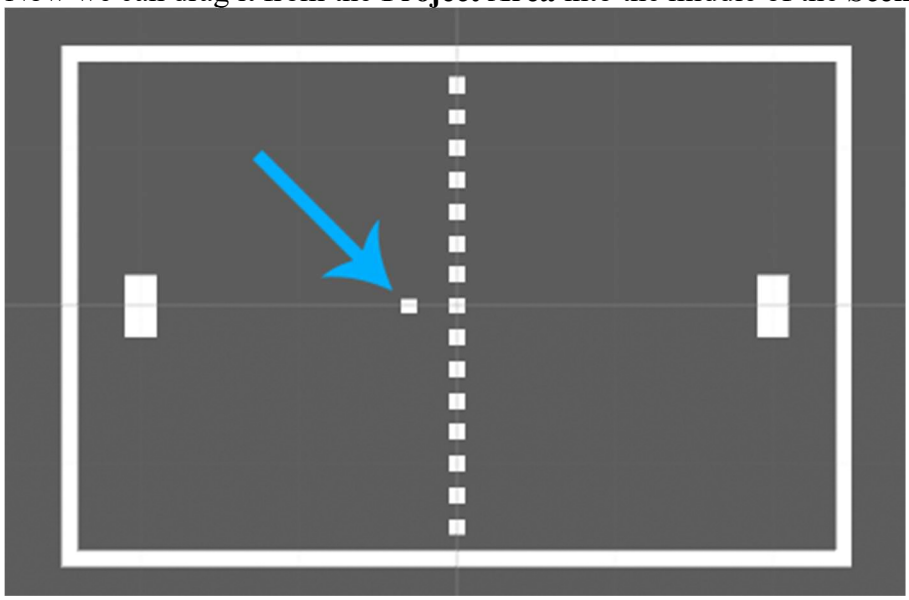
*Note: instead of using axes, we could also create a **up** and **down** key variable in our Script and then set it to **w/s** for the left racket and **UpArrow/DownArrow**for the right racket. However by using axes we end up with far less code and perfect gamepad/joystick/keyboard support.*

Creating the Ball

The Ball Texture

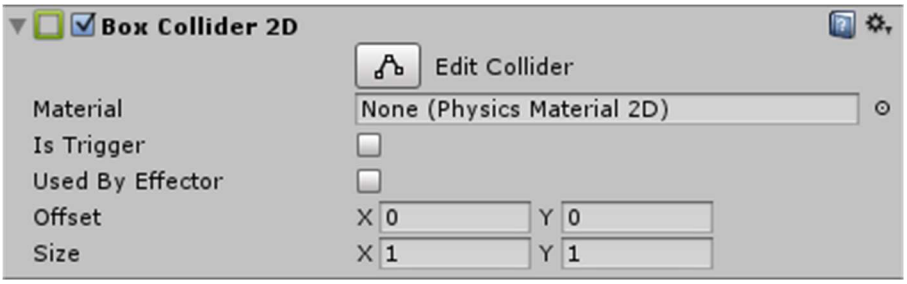
Okay the Ball will be easy again. Select the **Ball** texture in our Project's Assets folder.

Now we can drag it from the **Project Area** into the middle of the **Scene**:



The Ball Collider

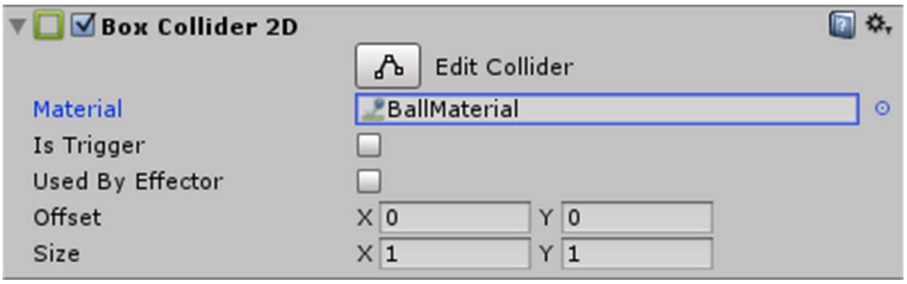
Our Ball should make use of Unity's Physics again, so let's select **Add Component->Physics 2D->Box Collider 2D** to add a Collider:



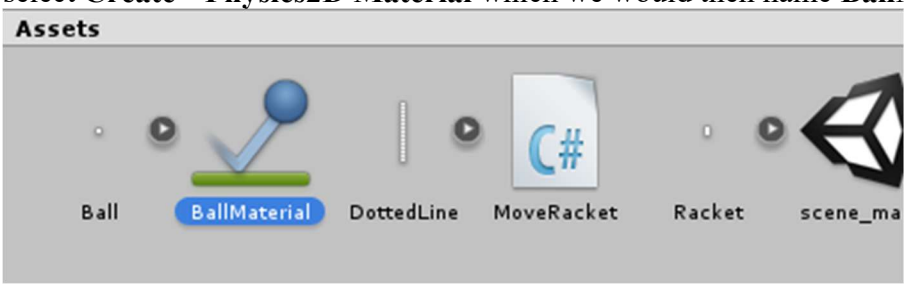
Our ball is supposed to bounce off walls. For example when flying directly towards a wall it should bounce off in the exact opposite direction. When flying in a 45° angle towards a wall, it should bounce off in a -45° angle and so on.

This sounds like some complicated math that could be done with Scripting. But since we are lazy, we will just let Unity take care of the bouncing off thing by assigning a **Physics Material** to the Collider that makes it bounce off things all the time.

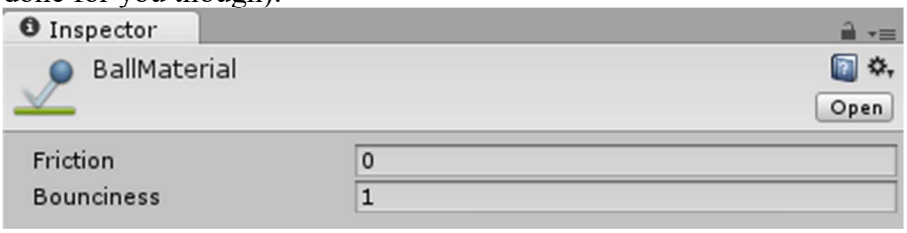
Drag the **BallMaterial** material from the **Project Area** into the **Material** slot of the Ball's Collider:



If we wanted to create a new material we would right click in our Project Area and select **Create->Physics2D Material** which we would then name **BallMaterial**:



Then we could modify our material in the Inspector to make it bounce off (all of this is already done for you though):



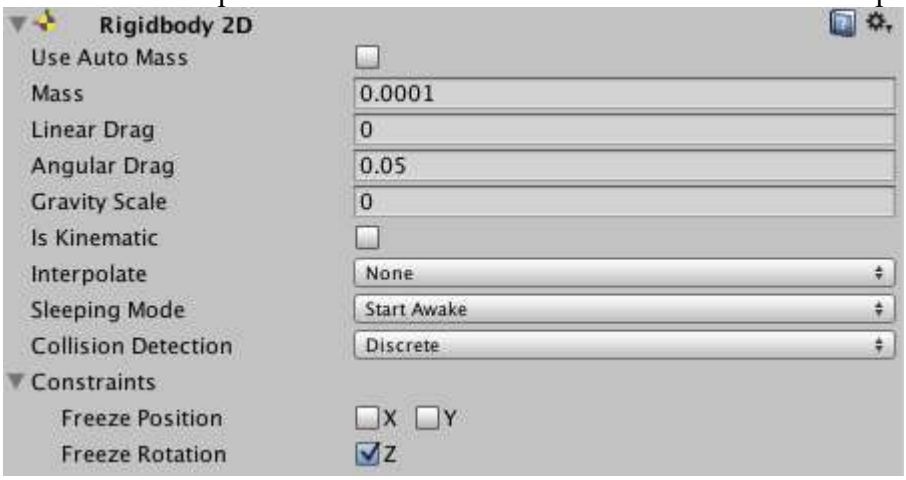
And that's all. Now the ball will bounce off in case it collides with things in the game.

The Ball Rigidbody

In order to make our ball move through the game world, we will add a **Rigidbody2D** to it again by selecting **Add Component->Physics 2D->Rigidbody 2D**.
Note: remember, every Physics thing that is supposed to move through the game world will need a Rigidbody.

We will modify the Rigidbody component in several ways:

- We don't want it to use Gravity
- We want it to have a very small Mass so it doesn't push away the Rackets when colliding
- We don't want it to rotate
- We use Interpolate and Continous Collision Detection for exact physics



Note: those modifications are not very obvious to beginners. The usual workflow is to add a Rigidbody, test the game and then modify the Rigidbody in case of undesired effects like a too big mass.

Okay so there is one more thing to do before we see some cool ball movement. We will select **Add Component->Ball** which will get our ball moving at the start of the game and tell it how to behave when it is hit by rackets.

To create a new script, we select **Add Component->New Script** name it **Ball** and select **CSharp** for the language. Afterwards double click the Script in the **Project Area** in order to open it:

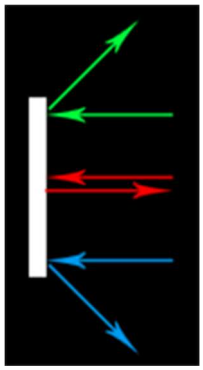
```
using UnityEngine;
using System.Collections;

public class Ball : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

Let's remove the **Update** function because we won't need it. Instead we will use the **Start** function to give the ball some initial velocity. Yet again we will use a **direction** multiplied by a **speed**:

```
using UnityEngine;
using System.Collections;

public class Ball : MonoBehaviour {
    public float speed = 30;

    void Start() {
        // Initial Velocity
        GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
    }
}
```

The Ball <-> Racket Collision Angle

Our game already looks a lot like Pong, but there is one more important modification to be done. We explained in the very beginning that the ball's outgoing angle should depend on where it hit the racket:

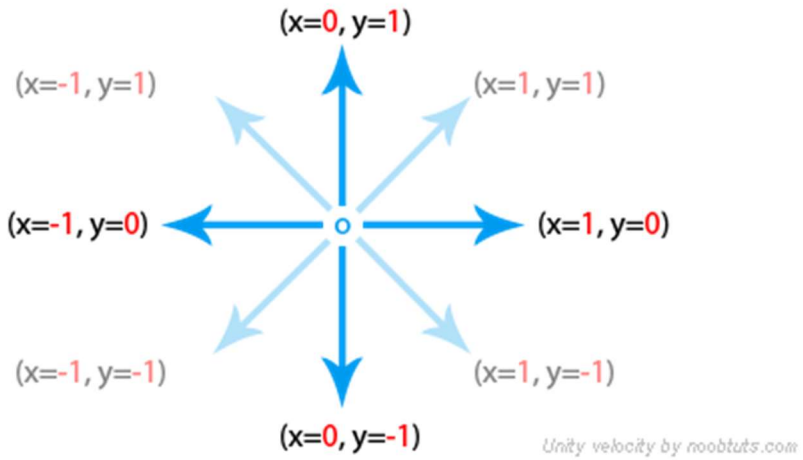


This way the players can shoot the ball into whatever direction they please, which adds a huge tactical component to the game.

Let's modify our Ball Script to use the OnCollisionEnter2D function that is automatically called by Unity upon colliding with something else:

```
using UnityEngine;
using System.Collections;

public class Ball : MonoBehaviour {
    public float speed = 30;

    void Start() {
        // Initial Velocity
        GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
    }

    void OnCollisionEnter2D(Collision2D col) {
        // Note: 'col' holds the collision information. If the
        // Ball collided with a racket, then:
        //   col.gameObject is the racket
        //   col.transform.position is the racket's position
        //   col.collider is the racket's collider
    }
}
```

So now we need a function that calculates the ball's velocity depending on where it hit the racket. The following image shows the Vector2 for several movement directions again:



Now the **x** value is obvious, it's **-1** in case it bounces off the right racket and it's **1** in case it bounces off the left racket. What we need to think about is the **y** value, which will be somewhere between **-1** and **1**. All we really need to calculate is this:

```
||   1 <- at the top of the racket
||
||   0 <- at the middle of the racket
||
|| -1 <- at the bottom of the racket
```

Or in other words: we just have to find out where the ball is in relation to the racket. Or in other words: we just have to divide the ball's **y**coordinate by the racket's **height**. Here is our function:

```
float hitFactor(Vector2 ballPos, Vector2 racketPos,
                float racketHeight) {
    // ascii art:
    // ||  1 <- at the top of the racket
    // ||
    // ||  0 <- at the middle of the racket
    // ||
    // || -1 <- at the bottom of the racket
    return (ballPos.y - racketPos.y) / racketHeight;
}
```

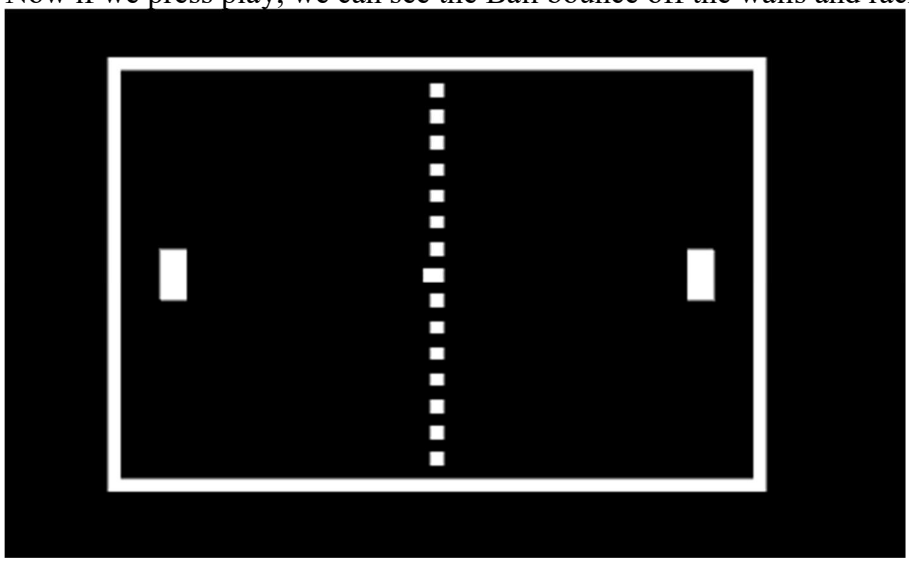
Note: we subtract the racketPos.y from the ballPos.y to have a relative position.

Here is how our final **OnCollisionEnter2D** function looks:

```
void OnCollisionEnter2D(Collision2D col) {
    // Note: 'col' holds the collision information. If the
    // Ball collided with a racket, then:
    //   col.gameObject is the racket
    //   col.transform.position is the racket's position
    //   col.collider is the racket's collider

    // Hit the left Racket?
    if (col.gameObject.name == "RacketLeft") {
        // Calculate hit Factor
        float y = hitFactor(transform.position,
                            col.transform.position,
                            col.collider.bounds.size.y);

        // Calculate direction, make length=1 via .normalized
        Vector2 dir = new Vector2(1, y).normalized;

        // Set Velocity with dir * speed
        GetComponent<Rigidbody2D>().velocity = dir * speed;
    }

    // Hit the right Racket?
    if (col.gameObject.name == "RacketRight") {
        // Calculate hit Factor
        float y = hitFactor(transform.position,
                            col.transform.position,
                            col.collider.bounds.size.y);

        // Calculate direction, make length=1 via .normalized
        Vector2 dir = new Vector2(-1, y).normalized;

        // Set Velocity with dir * speed
        GetComponent<Rigidbody2D>().velocity = dir * speed;
    }
}
```

Note: please read through the comments in order to understand what's going on.
If we press play, we can now influence the ball's bouncing direction depending on where we hit it with the Racket.

Now if we press play, we can see the Ball bounce off the walls and rackets:



Again, we didn't have to worry about any complicated math. Unity took care of most of it for us with its powerful Physics engine.

Congratulations! You just completed your first game in Unity. Enjoy it with a friend and show off your new skills.

Summary

In this Tutorial we learned how to install and use Unity, create a basic Scene with just some textures, use Unity's Physics and create Scripts to add custom game mechanics.

You are encourage to continue to make improvements to the game on your own! Here are some ideas:

- Add a Trail Effect like shown at the top screenshot
- Add some “Power Ups” (like a bigger racket, or a sticky racket)
- Add the old Pong Sound that we all love
- Add a Score system
- Increase the Ball's speed over time
- Add an AI enemy
- Add a menu and a credits screen

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.