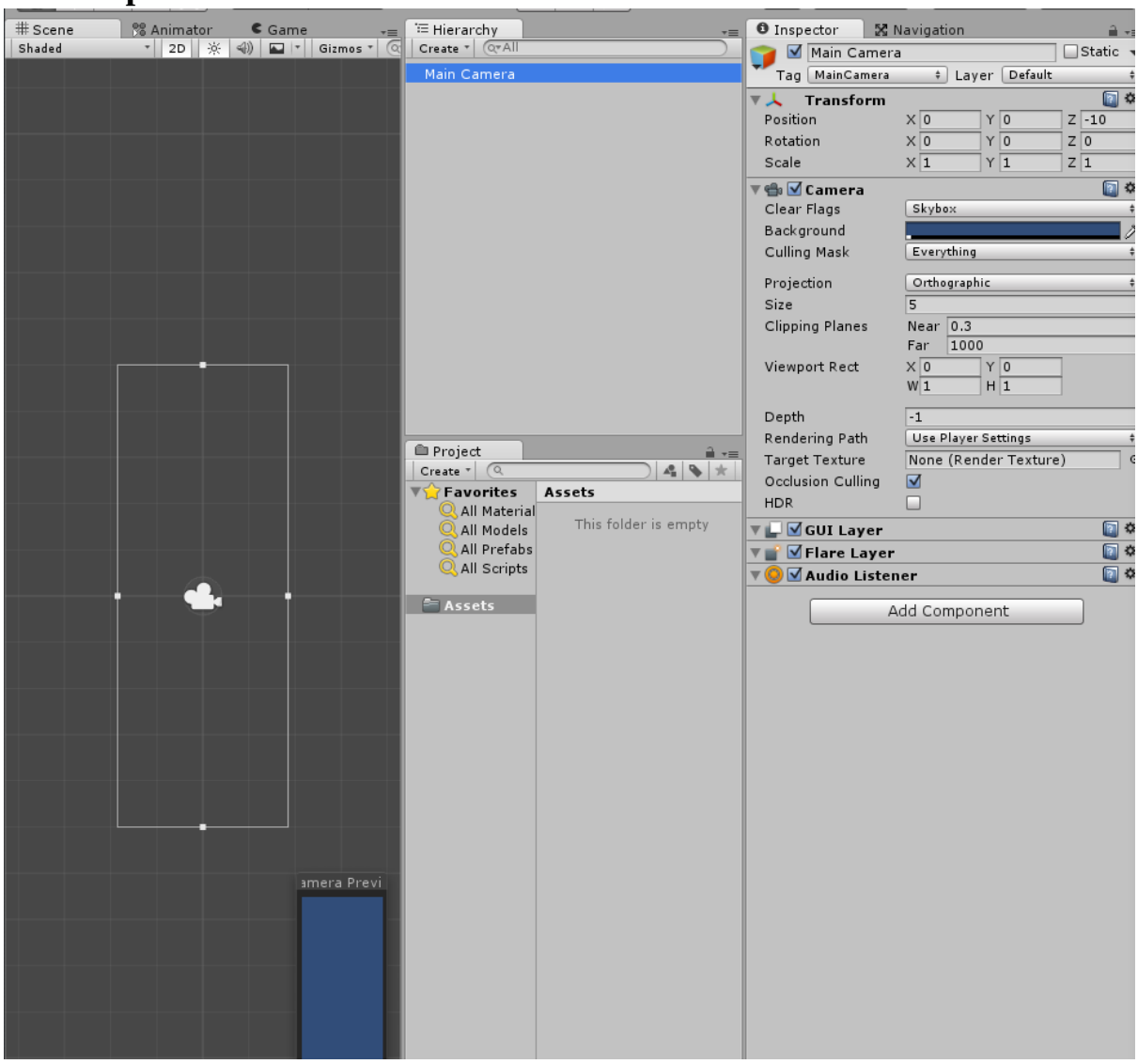
Flappy Bird Creation

Flappy Birds was/is a popular game that became THE most downloaded iOs app in January 2014, it made the creator thousands and got him death threats.

As it's a simple game it's a nice tutorial and entry point into Unity and a great example of the strength of the engine.

Create a new Unity Project, set it to 2D and name it something suitable (Flappybird is a great name).

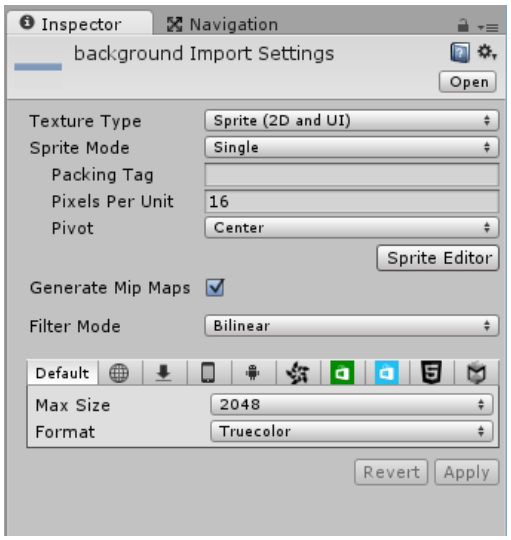
First steps



You can see the Main Camera in the hierarchy, selecting it will show you the Main Camera in the inspector. We can set the **Background Color** to a light blue *(R=190, G=210, B=230)* for the sky and adjust the **Size** (the amount of the screen shown) to 10.

Background

We start off by using a simple sky, we could use a block colour as a background but a little artistry goes a long way. Import the background file and save it in a new folder called “sprites”.

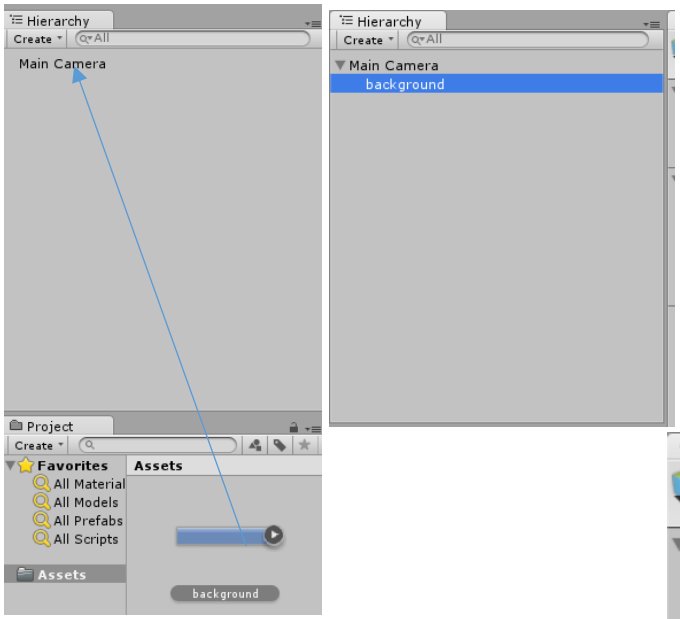


Click on the background in the project view and you should see the similar view to that to the side. Two things should change, the Pixels per Unit, and the Format.

Pixels per unit means how many pixels will represent one “unit” in Unity. The bird will be 16 pixels and we want that to be one unit so we set everything to 16 by 16 to keep the same scale.

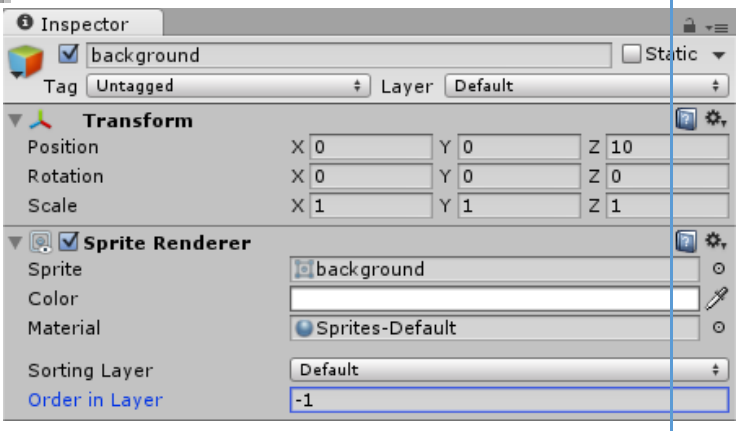
The format is just whether we compress or not, it’s a small file, we keep it perfect colour wise so truecolor. Do this and then drag it from project to hierarchy.

In game we want the background to keep fixed with the camera. We could accomplish this with complex code, or we could just grab the background in Hierarchy and drop it on Main camera. Now where the camera goes the background does.

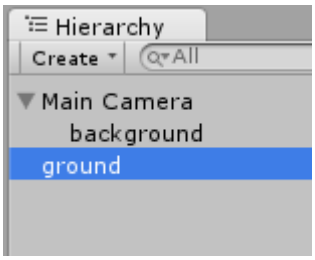


There is one more adjustment to be made here we want the background to be behind everything. Unity uses a Sorting Layer and Order in Layer properties to decide which parts of a game should be in front of which other parts.

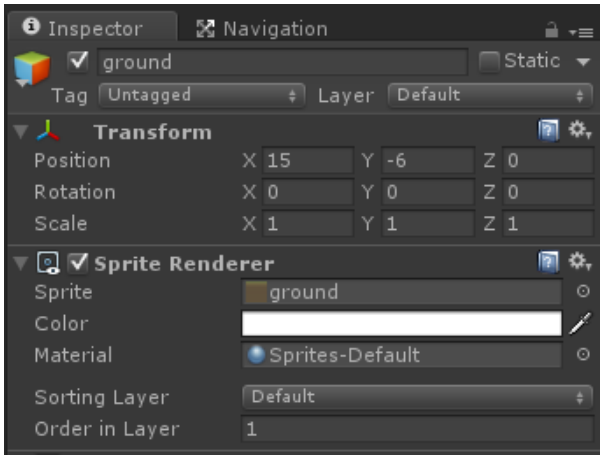
We will simply set the Order in Layer to -1 so that everything else will be drawn in front of it, the higher the number, the further forward it is:



The ground



If you press play now you should see the sky, but now we need the ground, import it and set it the same again, 16 PPU and truecolor compression. Drag it onto the scene, this time not onto the camera but separate.

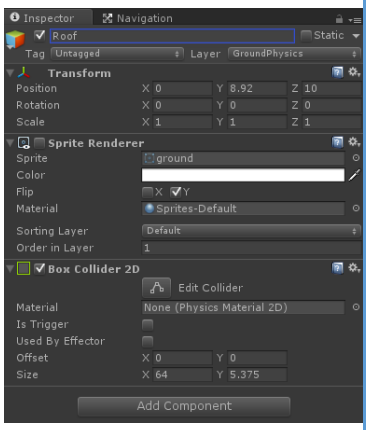


Now, change the transform to X 16 Y -6 Z 0 this should put the ground around the bottom of the sky and give us a nice look. This time we set an order in layer of 1 so it's further forwards than the bird.

Physics

We need the ground to be physical, this means it needs to be a physics object, not just an image.

We want the ground to be solid so the bird can collide with it. With the ground selected click **Add Component**->**Physics 2D**->**Box Collider 2D** in the **Inspector**:



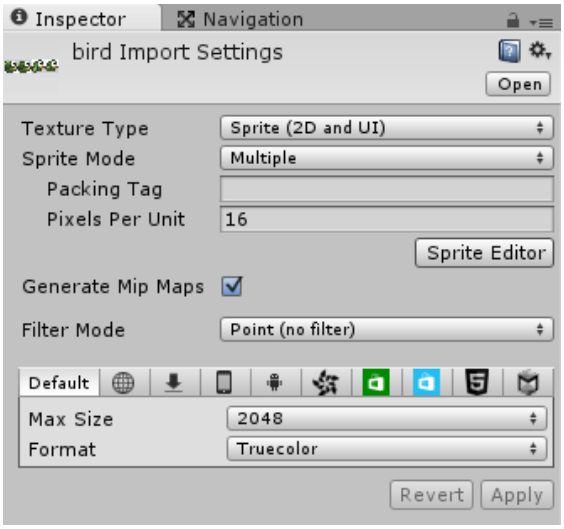
This adds a Box around the ground, when something hits that box a physics event is generated. There'll be a couple more things we need to do with physics later but we'll come back to those.

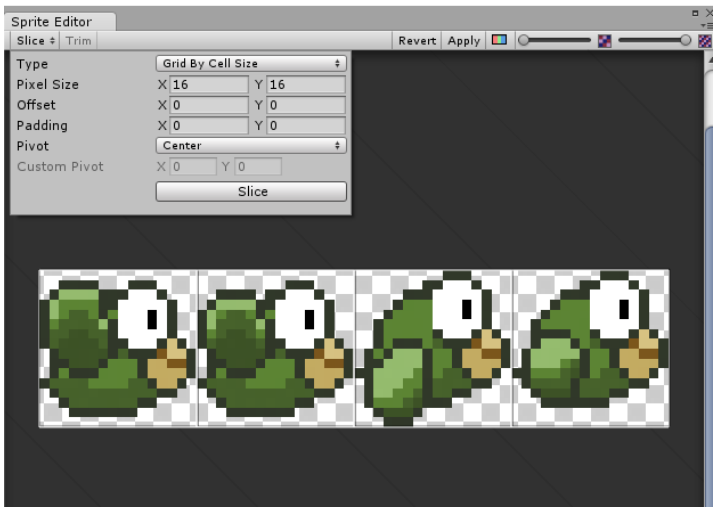
The problem is that, if you were to play the game like this, there's nothing stopping you from flying high- above the obstacles and easily just keeping going. Instead, what we do is simple, we duplicate the ground, moving it up to a suitable position and, because having ground in the air looks weird, we disable its renderer, essentially making it disappear. Don't forget, because it's set as ground – if they touch it, they'll die (we're being evil here).

The Bird is the word

We import the bird sprite and make sure to configure it as before, this time though, the bird is a sprite sheet. So we switch sprite mode to multiple, this lets us reconfigure it to choose the sprites we want.

Hit Sprite editor and we slice it into a 16x16 grid.

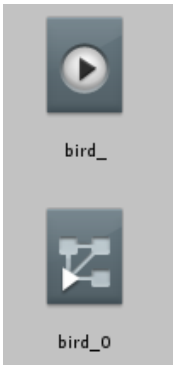




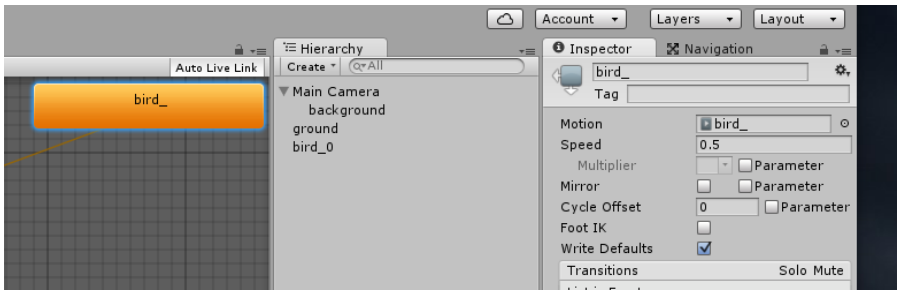
Hit slice and then Unity will prompt you about Unapplied Import settings, hit apply and it will slice the image into four. If you look in project view and hit the little arrow you'll see them clearly split up.

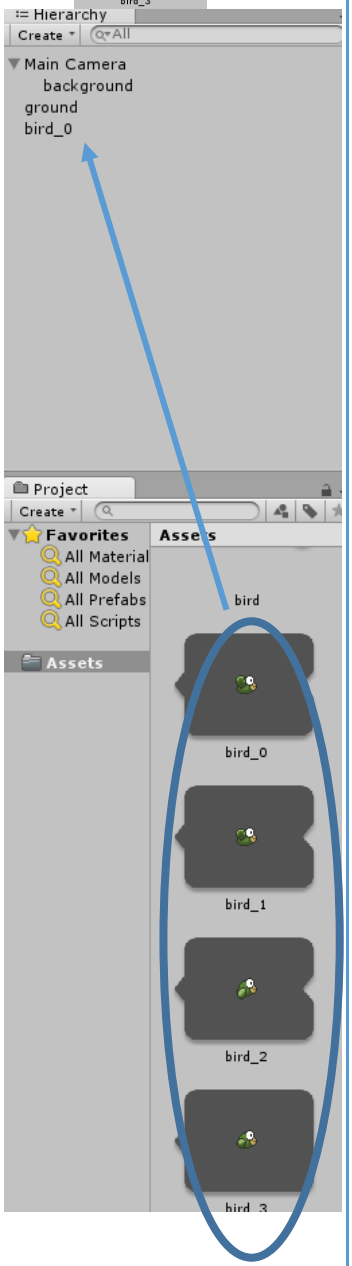
Animating the bird

Select the four new separate images (but not the one that shows all four in one image) and drag them into the hierarchy. You'll see a new single object created, and, if you look in the project view you'll see a new set of files. These are an animation containing those images and a controller, you don't need to know or worry too much about these as we only have one, Unity does it all for us.



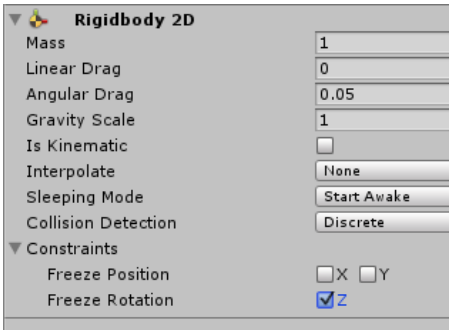
What we do need to do it open the controller (the bottom one to the right) and find the bird_ animation from there. Drop the speed (shown below) to 0.5, this slows it down so our bird isn't working as hard and looks a bit better. Press play and see it in action, how easy was that?





More Physics

Our bird is also physical, it can go splat, this time we add a circle collider 2D AND, we also add a rigidbody which makes it even more physical, basically it allows Unity to simulate things like forces, gravity etc. Add a Rigidbody2D and expand constraints, clicking "Freeze Rotation" this means physics are simulated and the bird can't be rotated- stopping some weird things like our bird spinning round and round. Give it a try, the bird should now fall.



Actual Physics

Create a new script (Right click -> Create -> C# Script) name it bird, we're going to do a couple things in this script- make the bird move right, make it jump up when you click (or tap) and make it die when you collide with the ground.

Because we made the bird a physics object we can use forces to make our life easier. Inside the new script add the following code;

```
6       // Movement speed
7       public float speed = 2;
8
9       // Flap force
10      public float force = 300;
11
12      // Use this for initialization
13      void Start () {
14              // Fly towards the right
15          GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
16      }
17
```

Line 7 and 10 add two new variables, speed and force, this is the speed at which it will move right and the amount of force that pushes up each time we click (or tap!).

Void Start() is run at start and simply gets the rigidbody attached so we can use physics, gets the velocity of it (speed in a direction) and sets it to a speed in the right hand direction. Now we want to make the bird fly up when the user clicks. We use GetMouseButtonDown(0) (LMB clicked) because this should also work if the user taps on the screen, meaning we can deploy to mobile easily. We check if the button is pressed, if it does we push the bird up with a force.

```
18      // Update is called once per frame
19      void Update () {
20
21          if (Input.GetMouseButtonDown(0)) {
22              // Flap
23                  GetComponent<Rigidbody2D>().AddForce(Vector2.up * force);
24          }
25
26
27          }
```

Finally, we want the bird to crash and restart if we hit anything, because we made the ground (and will make the other stuff) physical we can just check if we hit anything with a collider.

```
29      void OnCollisionEnter2D(Collision2D coll) {
30          // Restart
31          Application.LoadLevel(Application.loadedLevel);
32      }
33      |
34
35  }
36
```

Application.LoadLevel can be used to load a Scene and **Application.loadedLevel** is the currently loaded Scene. So we load the currently loaded screen.

Following the bird

Currently the camera doesn't follow the bird, if you go off screen the camera loses the bird, let's fix that.

Select the **Main Camera** in the **Hierarchy** and then add a new script, name it **CameraScript**.

We want the camera to follow its target (which will be the bird), so, we need a variable to hold the bird of type Transform (it's location rotation scale). Instead of using Update, which runs at different points in the scene, we want to use LateUpdate, which always runs after everything else has been updated.
In LateUpdate we want to;
Get the cameras position and set it equal to the bird's position, we can't directly do that, but we can copy the bird's position to a new Vector3 and copy that in. The code is below.

```
4 public class CameraScript : MonoBehaviour {
5
6     public Transform target;
7
8     void LateUpdate () {
9         transform.position = new Vector3 (target.position.x,
10                                           transform.position.y,
11                                           transform.position.z);
12    }
13 }
```

Once you've done this, you can drag the bird from the hierarchy, into the script's target variable in inspector.

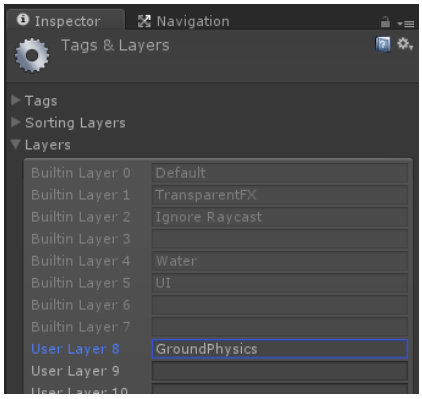
Obstacles

Import the obstacle and go from there, set the PPU and the compression as we have before and we can drag a few into the scene to place them.

We want to position it somewhere where it looks like it's sticking out of the ground, around 3, -5, 0 will work. Give it a try.
If you press play now you will find that your bird clips straight through, you need to add a collider to the obstacle, add a BoxCollider2D and try again.

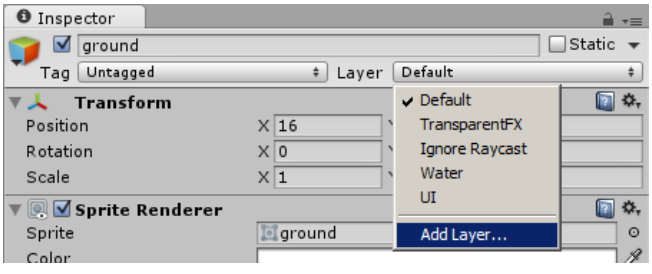
Obstacle Physics

We want the obstacles to move upwards and downwards into the ground because the obstacles have Colliders and are both "physical" they can't be in the same place at the same time. So, we need to tell Unity that although they're both physics objects we want to ignore collisions between them. We do this by creating a new physics **Layer** that we use for the ground and the obstacles.
We can create a new layer by selecting **Add Layer** in the **Inspector**:



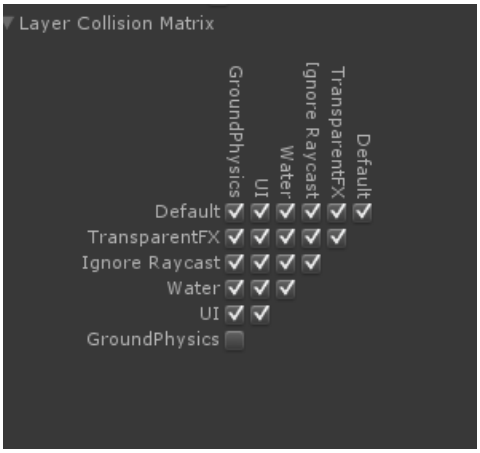
Then we add a new User Layer called GroundPhysics.

Now we can select the ground in the **Hierarchy** again



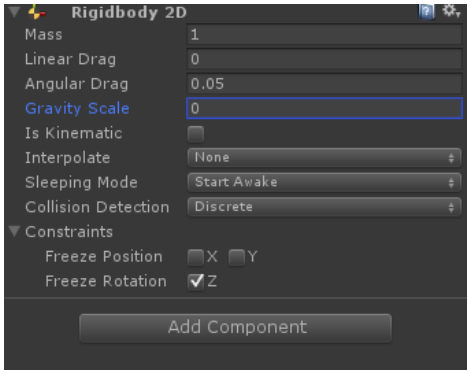
assign the **GroundPhysics** layer:

Afterwards we select **Edit**->**Project Settings**->**Physics 2D** from the top menu and uncheck
the **GroundPhysics** vs. **GroundPhysics** collisions in the **Layer Collision Matrix**:
Now the ground will never collide with any of the obstacles.



We talked about how the obstacles might end up inside the ground and how we don't want the two to collide with each other, so let's make it part of our **GroundPhysics** layer, too. Now, because we disabled collisions between two groundphysics things the bird will collide with everything but the obstacles won't collide with the ground.

We can scattered objects and we have a playable game, but, let's make things move to up the difficulty. Everything in the physics world that is supposed to move will need a Rigidbody, so we select **Add Component**->**Physics 2D**->**Rigidbody 2D** in the **Inspector** again. We don't want it to move under gravity so we set gravity to 0 and we check the constraint Freeze Rotation Z to stop it spinning.



Obstacle Movement

Moving can be implemented with a Script again select **Add Component**->**New Script**, name it **Obstacle and open.**

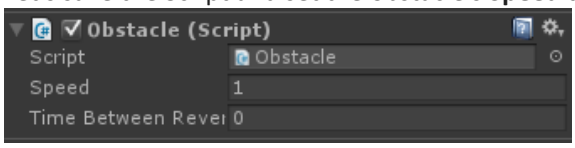
We will begin by adding a **speed** variable and then setting the Rigidbody's **velocity** so that the obstacles move upwards with the **speed** like we did with the bird.

```
4  public class Obstacle : MonoBehaviour {
5
6      public float speed=0;
7
8      // Use this for initialization
9      void Start () {
10
11     }
12
13     // Update is called once per frame
14     void Update () {
15         GetComponent<Rigidbody2D>().velocity=Vector2.up*speed;
16     }
17 }
```

Now Unity's InvokeRepeating function to reverse that velocity every few seconds, InvokeRepeating calls that function every x seconds. So we write a function that switches the direction of the velocity and call it every few seconds.

```
4 public class Obstacle : MonoBehaviour {
5
6     public float speed=0;
7     public float timeBetweenReverse=0;
8
9     // Use this for initialization
10    void Start () {
11
12    }
13
14    // Update is called once per frame
15    void Update () {
16        GetComponent<Rigidbody2D>().velocity=Vector2.up*speed;
17
18        // Switch every few seconds
19        InvokeRepeating("Reverse", 0, timeBetweenReverse);
20    }
21
22
23    void Reverse() {
24        GetComponent<Rigidbody2D>().velocity *= -1;
25    }
26
27
28 }
```

Let's save the Script and set the obstacle's **Speed** to **1, make sure to change the time too!!**:



We can have static objects too, we just set the speed to 0 if we press **Play** then we can see our obstacle moving up- and downwards.

But we want more!- right click the obstacle in the **Hierarchy**, select **Duplicate** and move it left or right.

We can also duplicate one or two and set the **Scale.Y** property to **-1**, this means it looks nice when we put it upside down. Experiment with switching up the speed and time between reverse properties.

There's a lot of scope to make things more or less difficult and extend the game, one good thing might be to add a timer or score so you know how long you've been going. Give it a try, another nice thing is, if you've got an android device for instance- it doesn't take much to export it and get it working on there.