Unity for Babies

JANUARY 02, 2021

Learning Objectives: Basic Game Components

- Creating animator and animation clip
- Transition between animations
- Setting up parameters for animation
- Timing animations and creating events
- Using coroutines: to execute methods over a number of frames or seconds
- Exploring the 2D physics engine: Colliders, 2D effectors, 2D joints
- Adding sound effects
- Creating physics materials and scripting

Introduction

We will continue where we left off last week by trying to polish our game a little bit better with sound effects, animation, and platforms. As usual, Unity pros can jump straight to the Checkoff heading to find out more information about the required final state of this Lab without having to read all the details.

Rigidbody2D Constraints

Notice how we always need Mario to stand upright, and not toppling when moving too fast. In order to do this, we need to place **constraints** on Mario's Rigidbody2D component:

- Go to Mario's Inspector window, and look for `Constraints` property under its Rigidbody2D component
- Select `Freeze Rotation: Z` and that's it!

We can do the same to constrain position as well, which will come handy later when we create movable obstacle for the game.

Animation

Mario's animation is broken down into **four** main state:

1. Idle state, when he's not moving at all
2. Running state, when he's moving left or right
3. Skidding state, when he switches direction while running
4. Jumping state, when he's off the ground

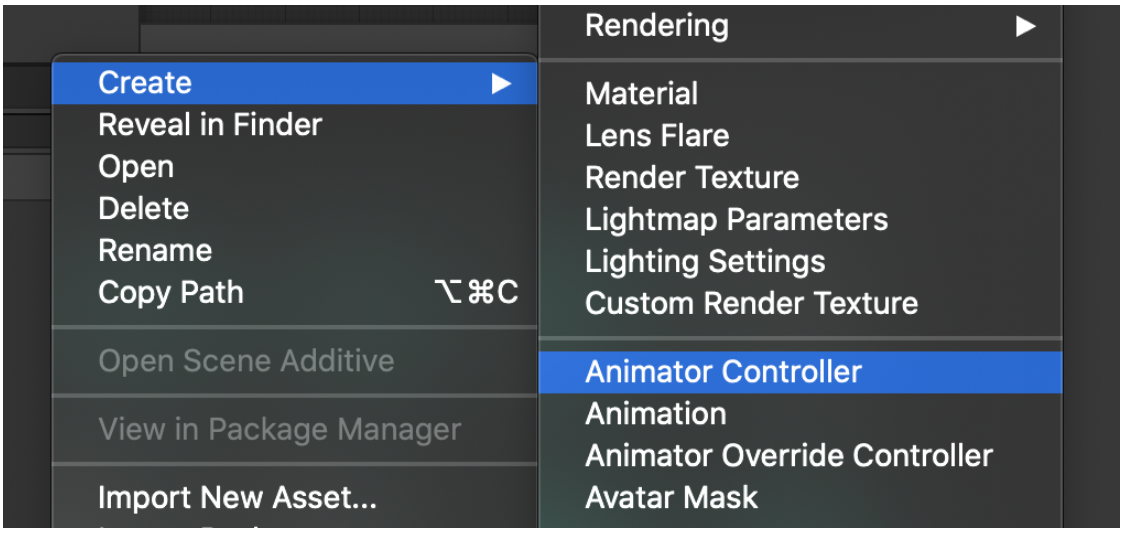
The Mario sprite given in the starter asset already contain the corresponding sprite that's suitable for each state, mainly `mario_idle`, `mario_jump`, `mario_run1`, `mario_run2`, `mario_run3`, and `mario_skid`.

To begin animating a GameObject, we need:

- An **Animator** element attached to the GameObject,
- An **Animator Controller**,
- and several **Animation Clips** to be managed by the controller.

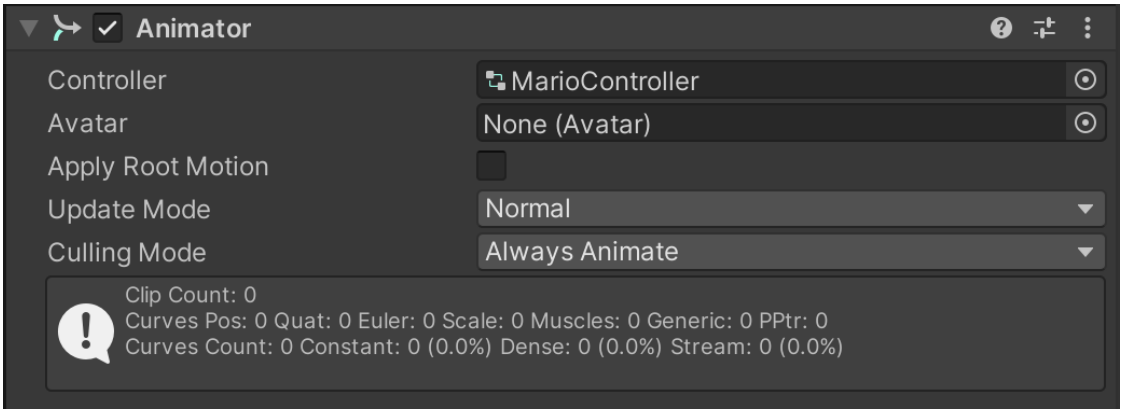
Animator Controller

Right click inside the Animation folder in your Assets, and create an Animator Controller. Name it `MarioController`.

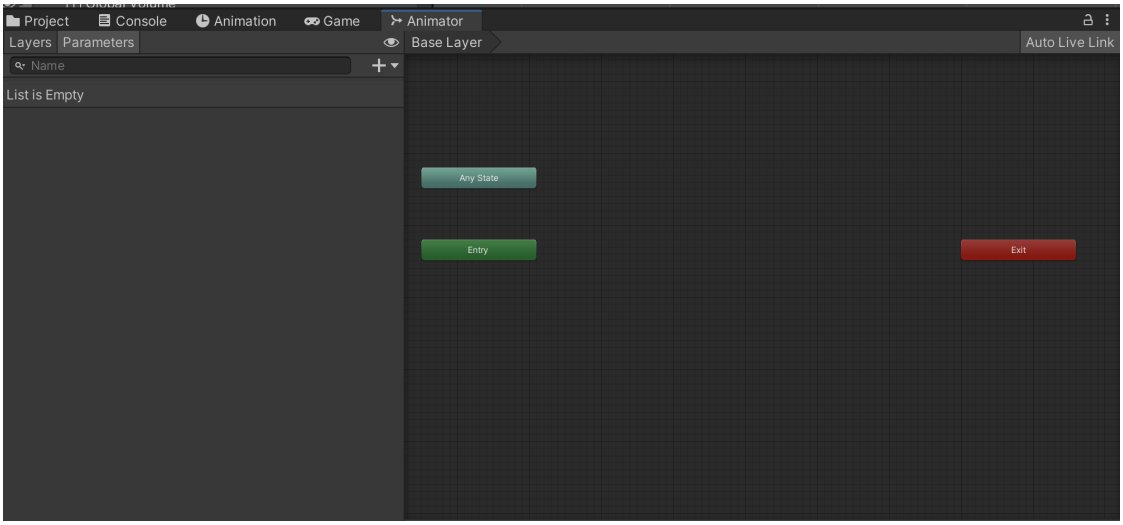


Animator Element

Then, **add an Animator element** to Mario, and load `MarioController` as the Animator controller, as shown:

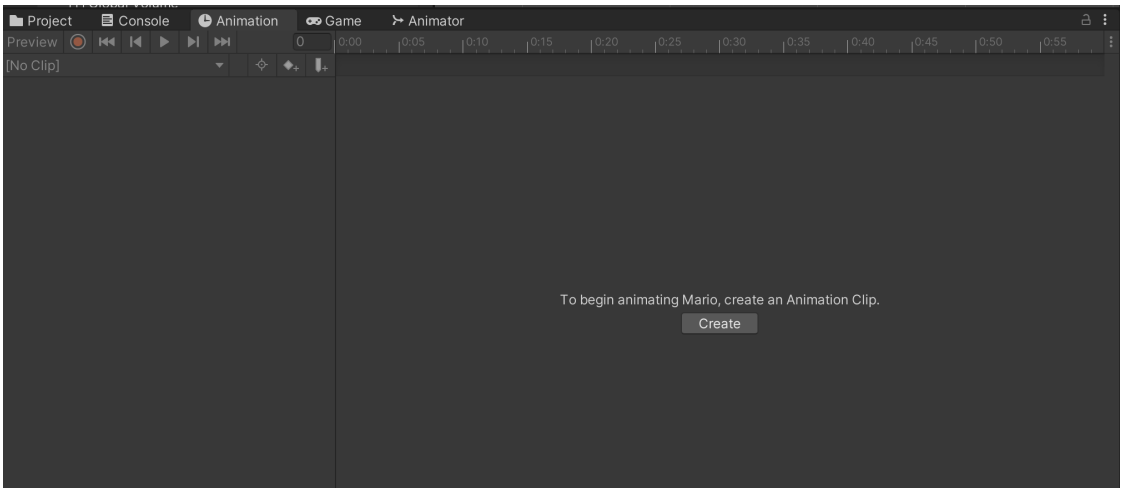


Open the **Animation tab** and you can see some kind of *state machine*. This will the place for us to edit the Animator and dictate which Animation Clip to play under certain condition.

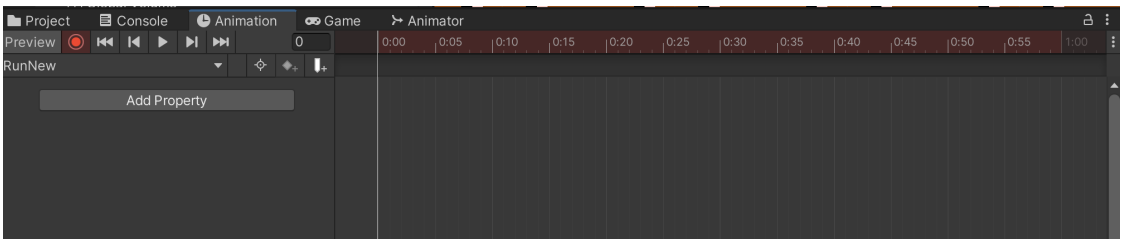


Animation Clips

Now let's create some clips. The first clip we need to do is to *animate Mario running*. Click the **Animation tab** instead, and click on the *Create* button.



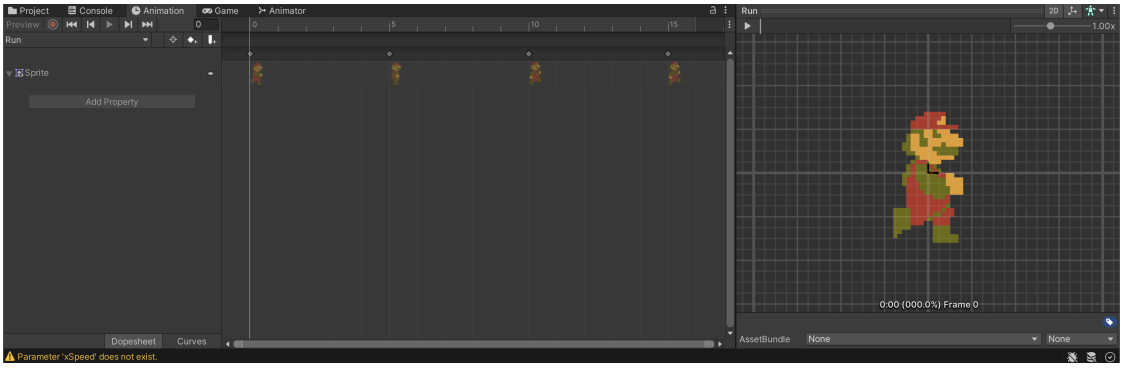
Now, click the **record** button, and you should see that the window turns red.



The values on the right side represents the frame (60 frames per second). We can change any element on Mario and it will be automatically recorded. For example, we can do the following actions:

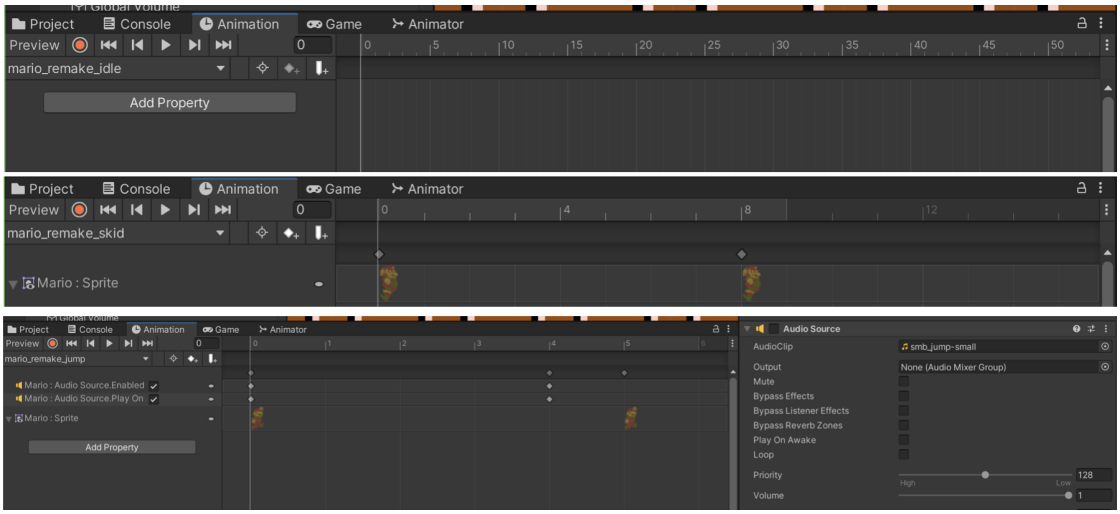
1. Click on Frame 0, and then change Mario's sprite into `mario_run1`
2. Click on Frame 5, and then change Mario's sprite into `mario_run2`
3. Click on Frame 10, then again change Mario's sprite into `mario_run3`
4. Click on Frame 15, and keep Mario's sprite at `mario_run3`

At the end, you should see something like on the Animation tab. Click play on the right side (inspector window) to observe the animation. You're free to edit which frame to show whichever sprite, as long as you're happy that the output is smooth.



To create more clip, click the dropdown containing the animation clip name (e.g: *Run* in the screenshot above) on the left, just under the record button.

Do the same to obtain the skidding, jumping, and idle animation. For idle animation, you simply don't have to record anything. Below is the screenshot of suggested animation clips:



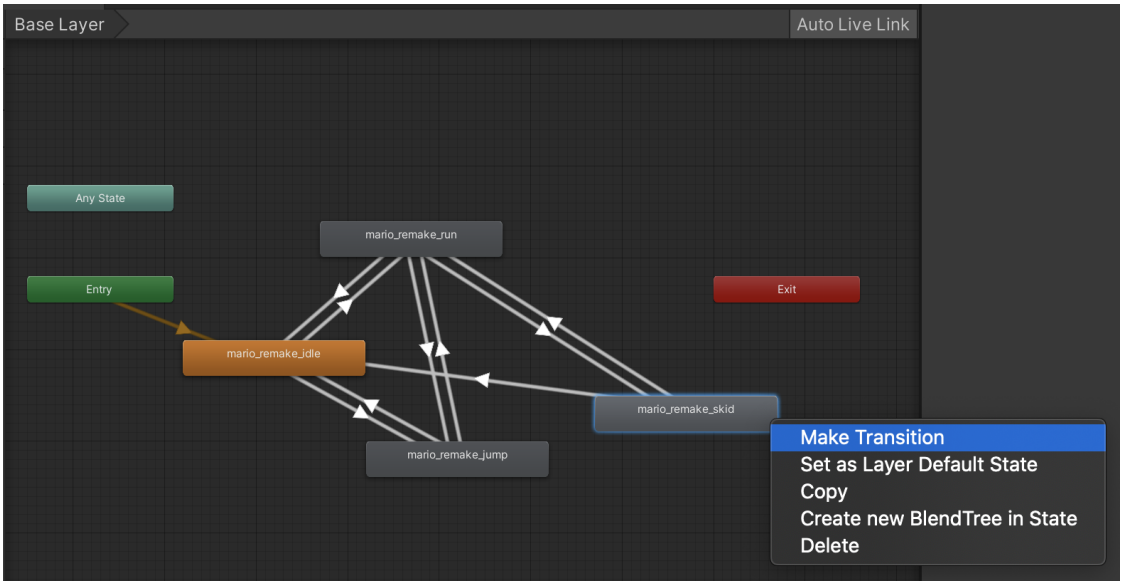
Sound Element

For the jumping animation, we need to do more than just a sprite change. We want to have a **sound effect** as well:

1. Add AudioSource element to Mario, and load the sound `smb_jump-small` to the AudioClip. Disable it and ensure that Play On Awake is also disabled
2. During the first frame of the clip, enable it and also click Play On Awake
3. During the fourth frame of the clip, disable it.

Transition

Now head to the Animator tab, and you now see all the clips as separate **states**. We need to now draw the *transitions*. You can right click on each state to make transition, and click on the destination state. Make something like this:

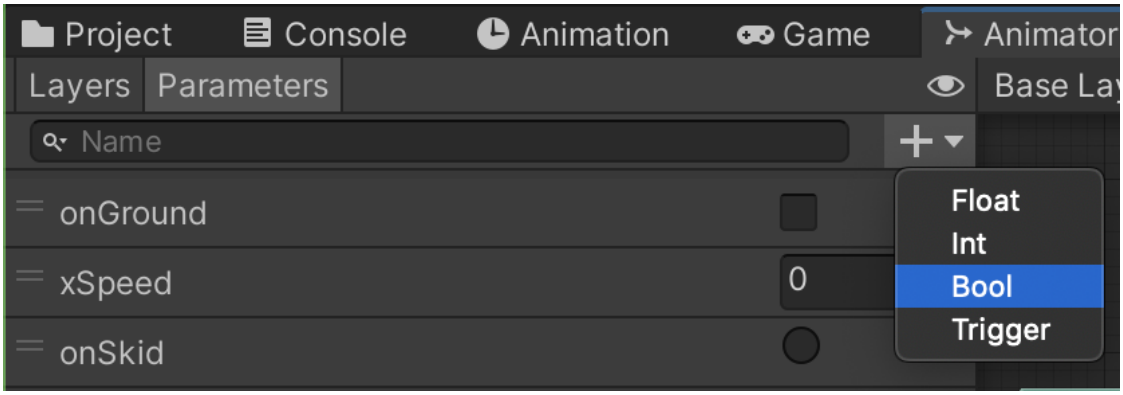


Animator Parameters

To enable correct transition conditions, we need to create **parameters**. These parameters will be used to trigger transition between each animation clip (motion). Create these three parameters:

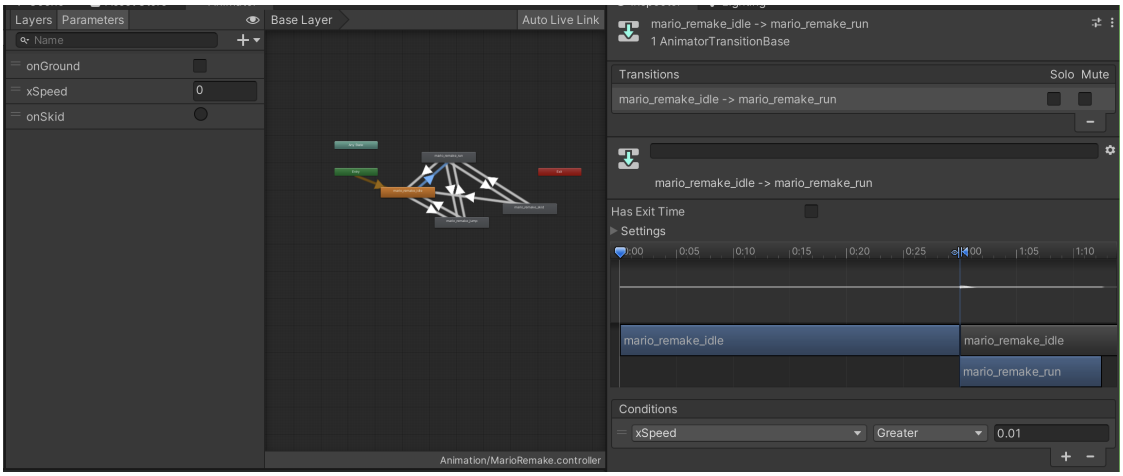
1. `onGround` of type `bool`
2. `xSpeed` of type `float`
3. `onSkid` of type `trigger` (a boolean parameter that is **reset** by the controller when consumed by a transition)

Here's all the parameters that you should have in the end:



Conditional Transitions

Now it's time to manage the transitions. Let's consider the transition from **idle** state into **run** state. This transition should happen if Mario's speed is larger than a certain number. Click on the arrow pointing from idle state to run state, and set its inspector to be the following:



Pay attention to the following settings:

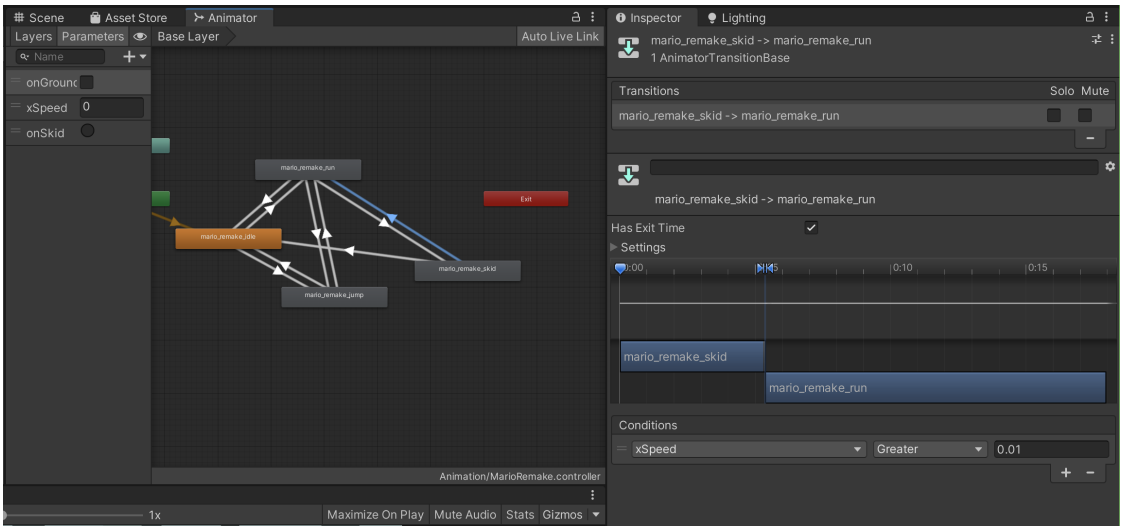
1. **Conditions**: Utilises the *Parameters* to trigger this transition. It is right now set to be triggered when `xSpeed` parameter value is greater than 0.01. We will of

course control this value via `PlayerController.cs` script later on.

2. **Has Exit Time**: Make sure this is unticked, so that transition happens **immediately**. Otherwise, we need to wait for the entire clip to finish (however many frames you set it to be).
3. **The transition graph**: It tells us whether to fade or mix between the current clip and the next clip *if exit time is enabled*. Right now the setting is such that we transit **abruptly** so the graph doesn't matter.

Let's take a look at another transition where we **want to have an *exit* time**, that is the transition between **skidding state** and **running state**. What we want is for the entire skidding state to **complete** (all frames played) before transitioning to the running state. The transition itself takes **no time**.

You can read more about transition properties here.



For the rest of the transition arrows, make use of the parameters in a way that you seem fit, for example, transition between jump and run should happen when `onGround` is `false` and when `xSpeed` is greater than `0.01`, and so on.

Controlling Animator Parameters via Script

Open `PlayerController.cs` script that you have attached on Mario and add the Animator variable:

```
private  Animator marioAnimator;
```

Then under `Start()` method, get its component as usual. This gives us **reference** to its current animator:

```
marioAnimator  =  GetComponent<Animator>();
```

Now our job is to **manipulate the Animator's parameters**: `onGround`, `xSpeed`, and `onSkid` when Mario's jumping, running, or skidding respectively. Mario will only skid as long as the key `a` or `d` is **pressed** down. To handle the skidding, enable the `onSkid` trigger under the `Update()` function, inside the clauses where you check `a` or `d` is pressed:

```
if (Mathf.Abs(marioBody.velocity.x) >  1.0)
        marioAnimator.SetTrigger("onSkid");
```

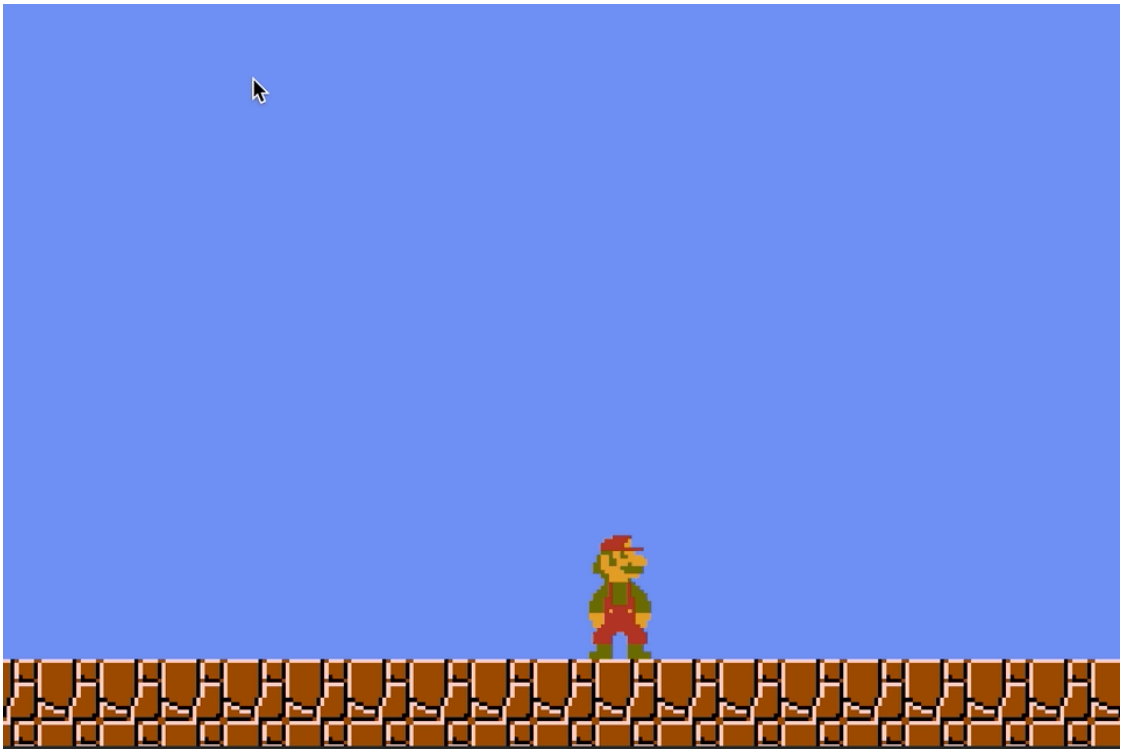
And the following line anywhere inside the `Update()` function to always update the `xSpeed` parameter to match Mario's current speed along the x-axis.

```
marioAnimator.SetFloat("xSpeed", Mathf.Abs(marioBody.velocity.x));
```

To handle Mario's jumping state, set the animator's `onGround` parameter to match the current `onGroundState` value whenever it's changed in the script, e.g:

```
marioAnimator.SetBool("onGround", onGroundState);
```

At the end of the day, your Mario should smoothly move around as shown. Ignore the Camera's auto follow feature for now. We will do that later.



Adding Events in Animation Clips

Notice how the jump sound effect is sorta got *cut* because the transition between jump animation state and run/idle animation state is **abrupt**? In other words, the AudioSource may already disabled ***before*** the clip finished playing, so the jump sound effect doesn't play fully.

We can improve this by adding **events** in the jump clip instead of primitively enabling/disabling the AudioSource like we did above. First, we need to write the function that will **handle** this jumping event that we about to create.

Open `PlayerController.cs` and add the following function:

```
void  PlayJumpSound(){
        marioAudio.PlayOneShot(marioAudio.clip);
}
```

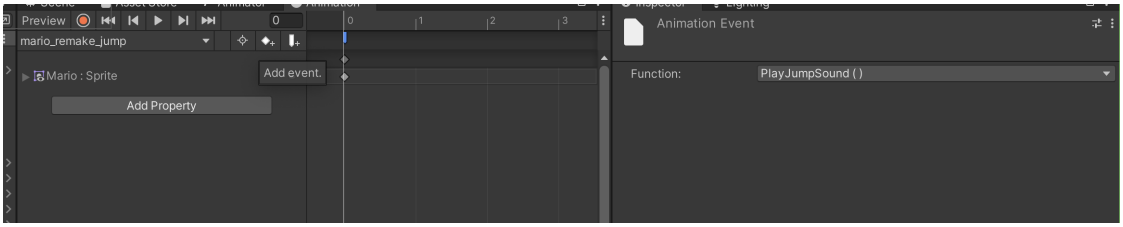
where `marioAudio` is a private variable of type `AudioSource` **(declare it yourself!)**, which you set via `GetComponent<AudioSource>` in the `Start()`

function. You should be familiar with this by now. Make sure the AudioSource component in Mario is now **enabled**, but **untick** the **Play On Awake** property.

> You can also implement this via `marioAudio.Play()`, but `PlayOneShot()` can play multiple sounds **without** cutting each other off.

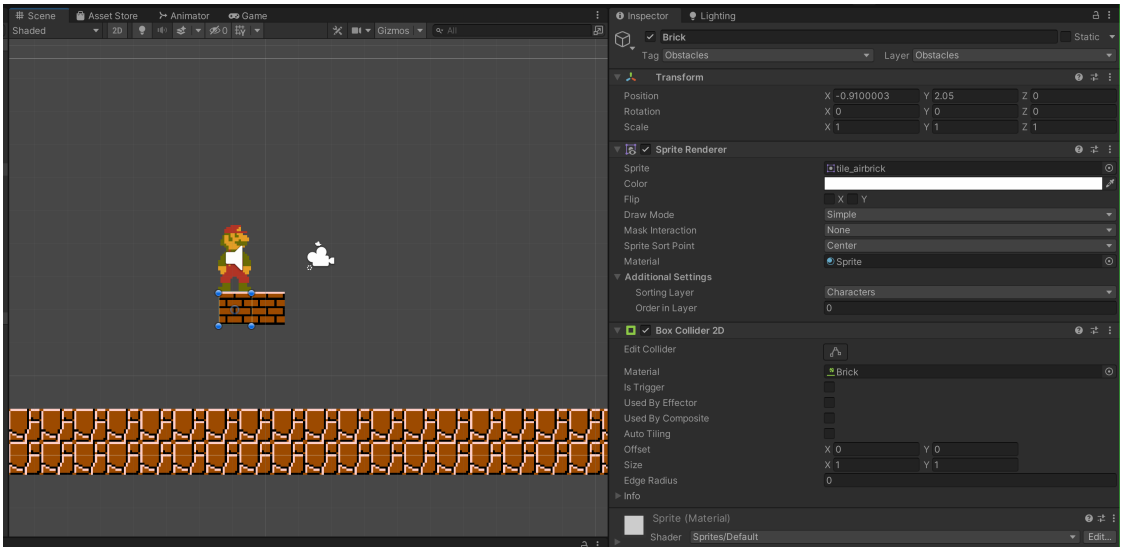
Now you can **add an event** in the jump animation clip by clicking the **Add Event** button in the desired frame (frame 0 in this case). In the inspector, select the `PlayJumpSound()` function.

It will *automatically* detect all custom functions of the scripts attached to the same GameObject that has the Animator as well.



Unity Physics2D

Now that our Mario can move around smoothly with proper animations, it's time we add some platforms. Adding platforms that Mario can jump on is easy: simply create a 2D object with Sprite Renderer and Box Collider 2D element on it:



Make that brick into a prefab so that you can have a master copy because we are going to spawn many of these in the scene.

> You might notice that Mario's animation doesn't get reset to "idle" state after he jumps onto the brick, because it is now colliding with a brick and not ground. Fix `PlayerController.cs` to consider the case where he can jump onto obstacles like this as well.
>
> A cheap and easy way will be to add a **new** Tag to the brick, e.g: `Obstacles` and add that check as well when resetting the `onGround` animator parameter.

Effector2D

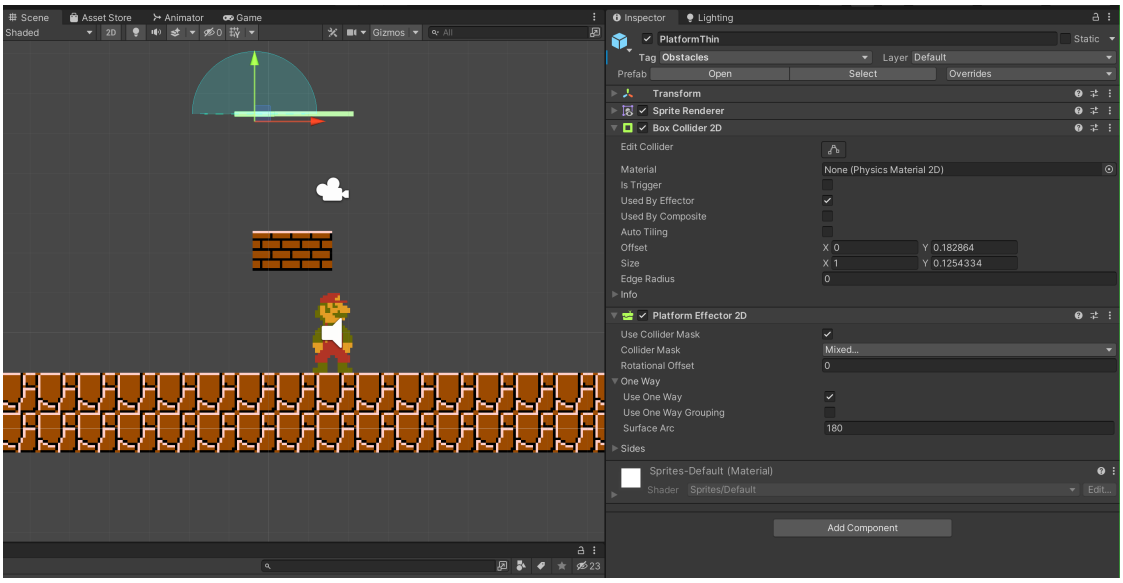
Suppose you want to create a platform that allows only one-way collision. You can upgrade your 2DColliders to be used with a new component called **effectors**.

To demonstrate how this works, let's create a platform where the character can "jump in" from underneath it but can stay upright on top of it.

- Create a new 2D GameObject » Sprite » Square and name it PlatformThin.
- Change its sprite to `tiles_187`

- Add BoxCollider2D component and edit its Collider to match the thin sprite.
  - **Tick** the Used By Effector property
- Now, add PlatformEffector2D component
  - **Tick** Use One Way
- Make PlatformThin a prefab, and spawn a two more in your scene, making a long edge

You should have something like this now:

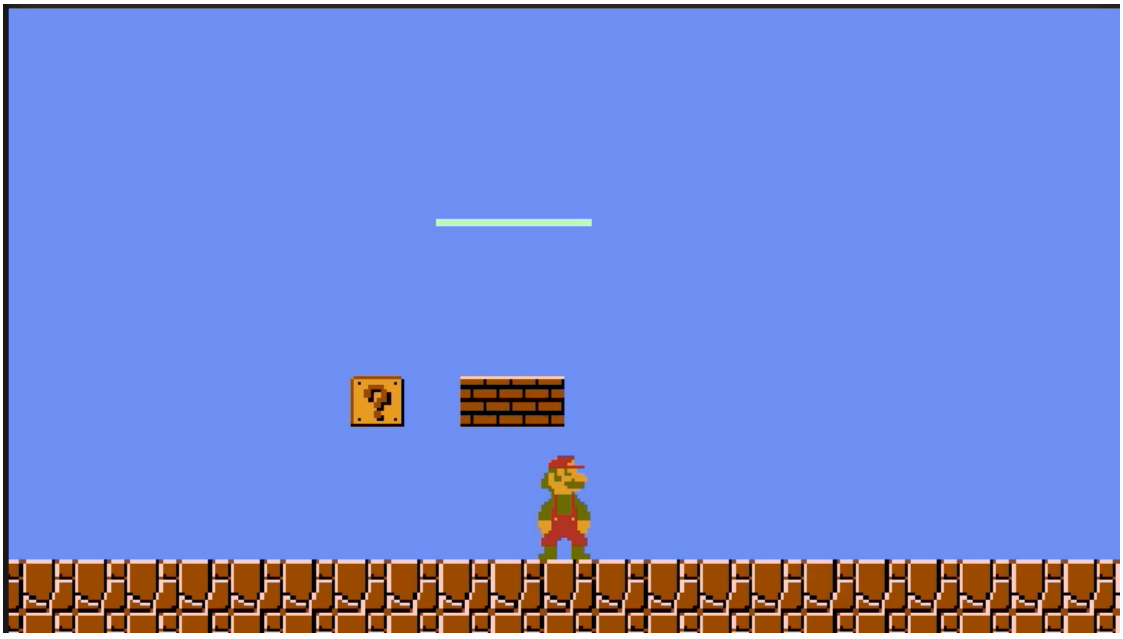


Try jumping onto the brick and onto the platform **from right underneath the platform**. You should notice that Mario can't jump onto the brick from underneath it, while he can do so on the platform.

You shall experiment with other effectors and their properties as well so that you know what features are supported or suitable for your game idea: some kind of boost, buoyancy, etc. This is what you can use if you try to implement a pinball game with many different areas that can affect the kinematics of the ball .

PhysicsJoint2D

Another interesting component of Unity's physics engine is the joints. It will safe you so much time if you want to implement any basic joints: hinge, spring, slider, wheel, etc. For example, the following is possible due to **SpringJoint2D** applied:



Follow these steps to create that Springy Question-box:

1. Create an **empty** GameObject with **two children** GameObjects. Name them:
   - **HittableSimple**
     - **TopCollider**
     - **EdgeDetector**
2. For the `EdgeDetector`, add **four** components:
   - SpriteRenderer, with `tile_questionblock_0` as its Sprite
   - Rigidbody2D: with `Mass=0.0001`, zero Linear and Angular Drag, and Constraints to Freeze `X` position, as well as `Z` rotation.
   - EdgeCollider2D: **edit the Collider** to align nicely with the bottom edge of the sprite
   - SpringJoint2D (will edit its properties later)
3. For the `TopCollider`, add **two** components:
   - Rigidbody2D: with Body Type of `static` (we don't want it to be affected by Physics, we will only use it as an anchor for the SpringJoint in EdgeDetector GameObject).
   - BoxCollider2D: edit the Collider to align nicely with the question box sprite

SpringJoint2D

Now time to configure the SpringJoint2D inside EdgeDetector GameObject. It has a few properties, but we mainly are interested in setting up the Spring's anchor. If you're not familiar with how Spring works, you basically need to endpoints to make a spring, its called:

1. **Anchor**: Where the end point of the joint connects to *this* object.
2. **Connected Anchor**: Where the end point of the joint connects to the *other* object.

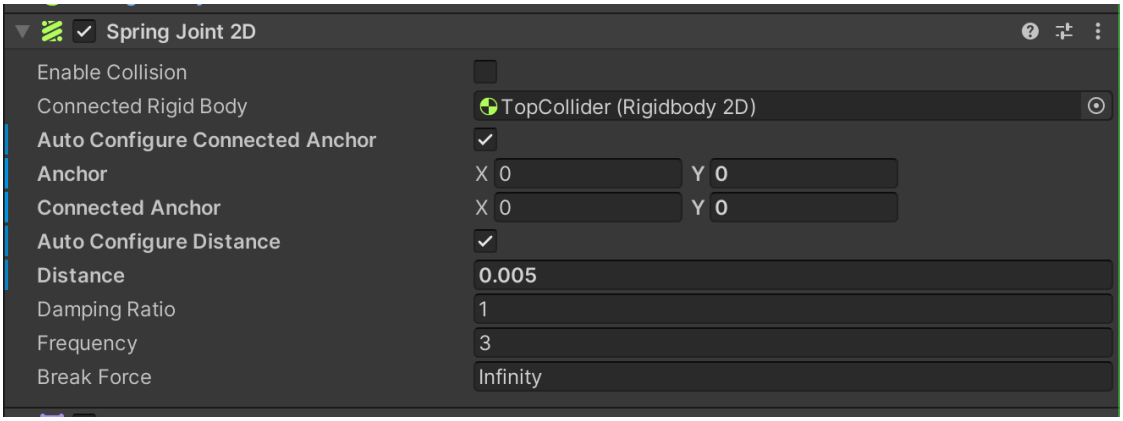
You also need to configure:

1. **Distance**: the spring will try its best to keep the length of the spring (distance between Anchor and Connected Anchor) to be that value that you set.
2. **Damping Ratio and Frequency**: to set the behaviour of the spring (hopefully now you find the stuffs taught in your Freshmore year useful).

To create a nice bouncy spring,

1. **Set** the TopCollider's Rigidbody2D as the Connected Rigid Body property of the spring.
2. **Tick** Auto Configure Connected Anchor and Auto Configure Distance properties.
3. Set `Damping Ratio` to 1, and `Frequency` to 3
4. Ensure that EdgeDetector's RigidBody2D **mass** is very small at 0.0001 as we set above, because we don't want so much force to nudge it, or allow it to *sink* when the game starts.

You should have these settings at the end:



Layer and Collision Matrix for Physics2D

Before we can test, we need to first ensure that the BoxCollider2D in TopCollider GameObject **does NOT** collide with the EdgeCollider2D in EdgeDetector GameObject.

- The former is used to *collide with Mario* when he climbs on top of this box so he can *stand* on the box,
- while the latter is used to *bounce the box* (excite the spring) when he hits it from below.

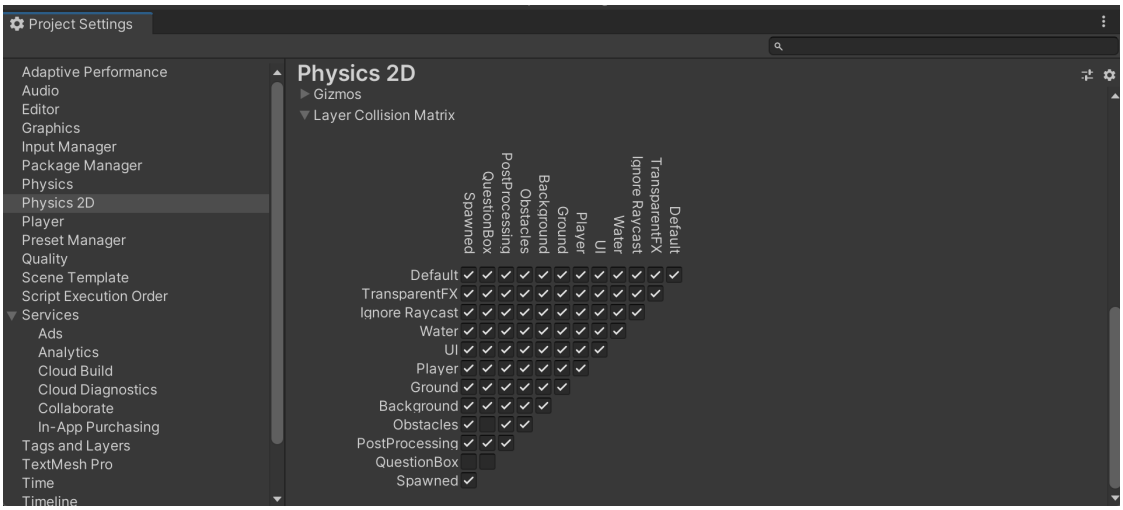
In order do have fine-grained collision tuning, we need to define the `Layer` of each object and set the engine's **Physics2D Collision Matrix**. On the top right hand corner of any GameObject inspector, notice there's a property called `Layer`. Just like a Tag, you can create your own Layer. It will be used by the Physics engine to determine **who can collide with each other.**

- Create a new layer called `QuestionBox` and assign it to the `EdgeDetector` GameObject
- Assign the `Obstacles` layer you have created earlier to the `TopCollider` GameObject

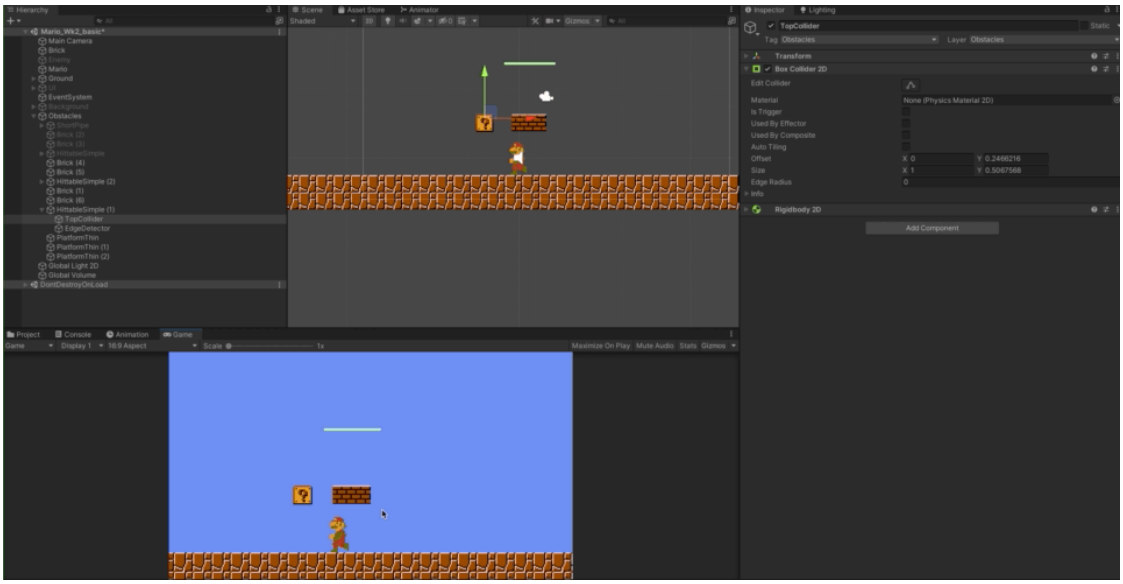
Then, go to **Edit » Project Settings » Physics2D**. You should see some kind of Collision Matrix depending on how many different Layers you have set in your

project. Right now you must only have the basics + `Obstacles` that you have created above.

Untick collision between QuestionBox and Obstacles to disable collisions between the two.



Now you can test your spring. While in Play mode, you can dynamically change the properties of the SpringJoint2D. Spend some time to play around with it to see how it works. You can also go to your Scene tab and dynamically drag around the TopCollider. Experiment around by unticking the auto configure distance and auto configure anchor properties to get a hang on how the spring works:



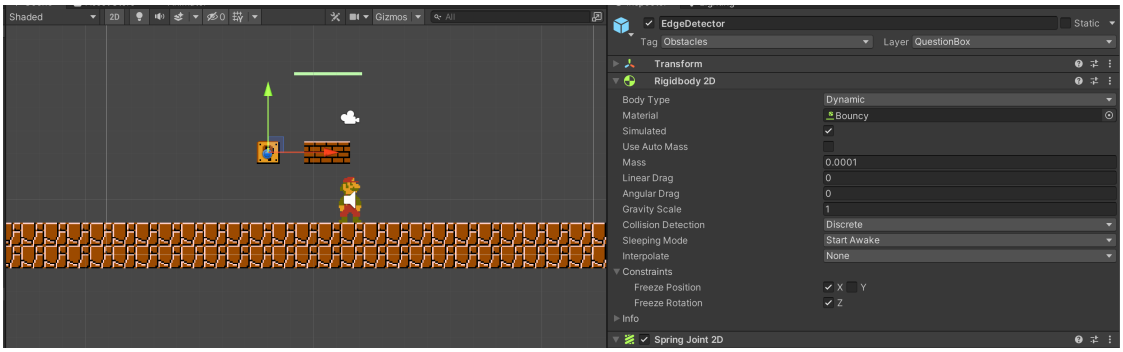
Once you're satisfied, save the HittableSimple object as prefab.

Physic Material

To get a more **bouncy** impact, you can create `Physics Material` and apply it on the Rigidbody of the EdgeDetector. It dictates the amount of **friction** or **bouncing** effects of colliding objects.

Go to the Project tab, and under Materials folder, right click » Create » Physic Material. Name it `Bouncy` and set the `Bounciness` property to some positive value, e.g: 0.3.

Afterwards, apply this Physic Material onto the Material property in Edge Detector's Rigidbody2D component:



Test run to see if the box now looks *bouncier*.

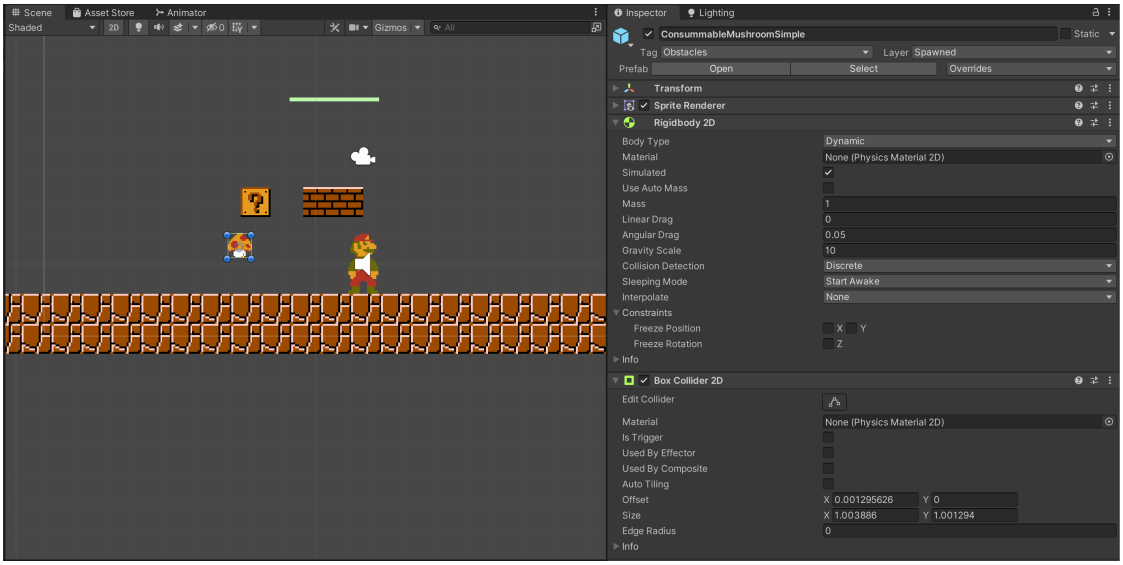
Spawning GameObjects at Runtime

Now what's left is to spawn something at the top of the HittableSimple object when Mario hits its bottom edge. We can begin by creating the prefab of the object that we are going to spawn.

Create a new 2D GameObject » Sprites » Square, name it `ConsummableMushroomSimple`:

- Change the Sprite property into any mushroom, e.g: `items_0`
- Add Rigidbody2D component (`Dynamic` type, with `Gravity Scale` of 10)
- Add BoxCollider2D component and edit the collider to match the sprite
- Create a new `Layer` called `Spawned` and assign it to the mushroom
- Change its scale to be of the right size, then drag it to the Prefab folder

You should have something like this in the end:



Now **delete** it from the Scene since we will only spawn it from the script and not in the beginning of the game.

Instantiate Prefab at Runtime

Double click on the `HittableSimple` **Prefab** (not the one on your scene!), and add a new C# Script component to its `EdgeDetector` GameObject, name the script: `QuestionBoxController.cs` .

The logic for this script is:

1. If the EdgeDetector has collided with Mario, then it has to *bounce*
2. And spawn the `ConsummableMushroomSimple` instantly, exactly **once**
3. *When the box has finished bouncing*, it has to change Sprite (so the player wont hit it again)

4. And we have to disable the Spring as well (the box turns into a stationary object, just like the regular Brick)

It is easy to do (1) and (2) above, as you might've guessed: simply write something inside `OnCollisionEnter2D` callback. Doing (3) requires us to *check* if the QuestionBox has turned stationary *after* briefly bouncing from being hit by Mario, and this is **not that trivial**, depending on how you choose to solve the problem. We will explain why later.

Declare the following variables in `QuestionBoxController.cs` :

```
public  Rigidbody2D rigidBody;
public  SpringJoint2D springJoint;
public  GameObject consummablePrefab; // the spawned mushroom prefab
public  SpriteRenderer spriteRenderer;
public  Sprite usedQuestionBox; // the sprite that indicates empty box instead of a
private bool hit =  false;
```

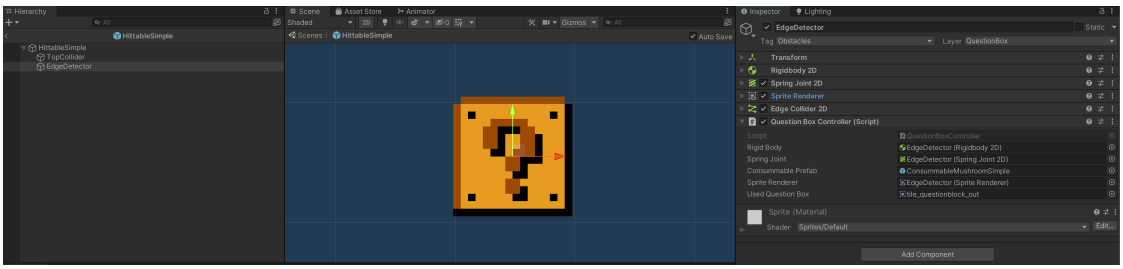
Then implement `OnCollisionEnter2D` callback function:

```
void  OnCollisionEnter2D(Collision2D col)
{
        if (col.gameObject.CompareTag("Player") &&  !hit){
                hit  =  true;
                // spawn the mushroom prefab slightly above the box
                Instantiate(consummablePrefab, new  Vector3(this.transform.position.
        }
}
```

This will allow you to spawn the Mushroom right above the box when it's hit by Mario.

- Attach this script to `EdgeDetector` GameObject under `HittableSimple` prefab (again, in prefab mode as shown in screenshot below! Not your game scene),
- Link up all the required public variables with all the assets.

- All assets except `Consummable Prefab` must be those in the Prefab's Scene and not the Assets tab.
- You can find the mushroom prefab to link here under the Assets tab.



Consumable Mushroom

Now create a simple script yourself for the mushroom to dictate its behaviour as follows once spawned by the box:

- It will randomly move to the left or to the right at a **constant speed**
- It **will not topple** (Z-rotation is constrained)

> Set its Rigidbody constraint properly

- However it will **change direction** when colliding with other objects (e.g: the Pipe. Quickly create some pipe prefabs of your own. You only need BoxCollider2D and a few sprites – the pipe body and the pipe top – for that)
- It will **stop moving** when it collides with Mario (Player)

> Use `CompareTag()` function for the two points above

- It will collide normally with `Obstacles` (the bricks) or `Ground`, so it will travel above the bricks or ground normally
- It **will not collide** with the `TopCollider` GameObject (child of `HittableSimple`) so as **not** to affect the spring motion of the box

> Ensure the mushroom's Layer (whatever you set it to be), do not collide with the `TopCollider` 's Layer

- It **springs** out of the box once instantiated (upwards)

The gif below summarises the required behavior of the spawned mushroom:

You probably can do all the above except the **first**: move the mushroom at constant speed (either to the left or to the right) and the last point (to give initial impulse force upwards).

Rigidbody2D MovePosition

To do the first point, we need to compute the supposed position of the mushroom in the next frame in the mushroom's script (whatever you name the script to be). Given `float speed` , current `Vector2 currentPosition` of the Mushroom, and `Vector2 currentDirection` (`{1,0}` or `{-1,0}` , indicating movement towards the right or the left), we can compute the mushroom's next position as:

```
Vector2 nextPosition = currentPosition + speed * currentDirection.Normalize() * Time
```

where `Time.fixedDeltaTime` is simply the **interval in seconds** at which Physics frame rate updates.

Afterwards, we can set the mushroom's Rigidbody2D position directly to be this `nextPosition` under `FixedUpdate()`:

```
Vector2 nextPosition = currentPosition + speed * currentDirection.Normalize() * Time
rigidBody.MovePosition(nextPosition);
```

Note: although not written, it is explicit for you to **declare** `rigidBody` as `this` mushroom's rigidBody and instantiate it at the `Start` method using `GetComponent()` as usual. </mark>

Rigidbody2D AddForce

Finally, it will be nice to add an **impulse** force upwards for the mushroom so it looks like it springs out of the box. Under the mushroom script's `Start()` method, you can add the instruction:

```
rigidBody.AddForce(Vector2.up  *  20, ForceMode2D.Impulse);
```

where `20` is just a value which you can change as you like depending on the upwards force effect that you want.

You might wonder what is the meaning behind `ForceMode2D.Impulse` and why didn't we use this mode instead:

```
rigidBody.AddForce(Vector2.up  *  20, ForceMode2D.Force);
```

If you use `ForceMode2D.Force`, you might observe straight away that the effect is *not immediate* because the net amount of upwards force `Vector2.up * 20` we

wrote to be applied on the mushroom is automatically set as the amount of **TOTAL force to be applied over ONE second** (50 **physics** frame).

> By default on desktop, **Unity** runs the FixedUpdate at 50 **FPS** and the **Update** at 60 **FPS**

However it DOES NOT mean that the physics engine continuously apply this force over 1 whole second. It will apply **only over the exact frames** when you call it, e.g: over **0.02 seconds** if you only call it for a single frame.

> Therefore, if you were to call `rigidBody.AddForce(Vector2.up * 20, ForceMode2D.Force)` continuously for **FIFTY** times (over 1 second), then the **total amount of force applied** on the mushroom body would've been the same as calling `rigidBody.AddForce(Vector2.up * 20, ForceMode2D.Impulse)` for **ONE** frame (one time).

In other words, `ForceMode2D.Impulse` tells the Unity Physics engine that you want to apply this much force **NOW**, where `ForceMode2D.Force` means that you want to apply this much force in total IF we were to call the `addForce()` method continuously, for exactly 50 times over 1 second.

Coroutines

Finally, we need to **disable** the `HittableSimple` 's spring and Rigidbody2D ***after*** it has finished bouncing. The problem with this is that we do not know *when* exactly the box has finished bouncing. We can continuously check under its script's `Update()` method after `hit == true` , or we can use an alternative called **Coroutines**.

> A coroutine is like a function that has the ability to **pause** execution and return control to Unity but then to **continue where it left off on the following frame**. A normal function like `Update()` cannot do this and must run into completion before returning control to Unity.

You can declare a Coroutine like this:

```
IEnumerator functionName(){
        //implementation here
}
```

and call it like this: `StartCoroutine(functionName());` .

Paste the following Coroutine and regular function inside `QuestionBoxController.cs` :

```
bool ObjectMovedAndStopped(){
        return Mathf.Abs(rigidBody.velocity.magnitude)<0.01;
}

IEnumerator DisableHittable(){
        if (!ObjectMovedAndStopped()){
                yield return new WaitUntil(() => ObjectMovedAndStopped());
        }

        //continues here when the ObjectMovedAndStopped() returns true
        spriteRenderer.sprite = usedQuestionBox; // change sprite to be "used-box'
        rigidBody.bodyType = RigidbodyType2D.Static; // make the box unaffected by

        //reset box position
        this.transform.localPosition = Vector3.zero;
        springJoint.enabled = false; // disable spring
}
```

The instruction `yield return new <something>` returns control to Unity until that `<something>` condition happens. We can wait for a few seconds: `yield`

`return new WaitForSeconds(0.1f)` , or wait until end of frame, etc. It will continue with the **next** instruction when resumed, which is `spriteRenderer.sprite = usedQuestionBox;` for the above example.

Another common way to `yield` is `yield return null` .

> **`yield return null`** waits for the next frame and continue execution from this line

For example, consider this sample code that gradually fades the color of an object,

```
IEnumerator Fade() {
        for (float ft = 1f; ft >= 0; ft -= 0.1f)
        {
                Color c = renderer.material.color;
                c.a = ft;
                renderer.material.color = c;
                yield return null;
        }
}
```

Without `yield return null` , Unity runs the loop **fully for the amount of time in one frame**, therefore blocking the editor and the intermediate values will never be seen. The object will disappear instantly.

> Hence, in order to see the fading effect, the effect of each loop must be seen per frame – rendered out to the viewer. Using `yield return null` returns the control back to Unity, and the instruction (`for` loop check) will be executed back in the next frame.

Now back to our game, we can call the Coroutine inside `OnCollisionEnter2D` , right after we instantiate the mushroom inside `QuestionBoxController.cs` :

```
void  OnCollisionEnter2D(Collision2D col)
{
        // Debug.Log("OnCollisionEnter2D");
        if (col.gameObject.CompareTag("Player") &&  !hit){
                hit  =  true;
                Instantiate(consummablePrefab, new  Vector3(this.transform.position.
                StartCoroutine(DisableHittable());
        }
}
```

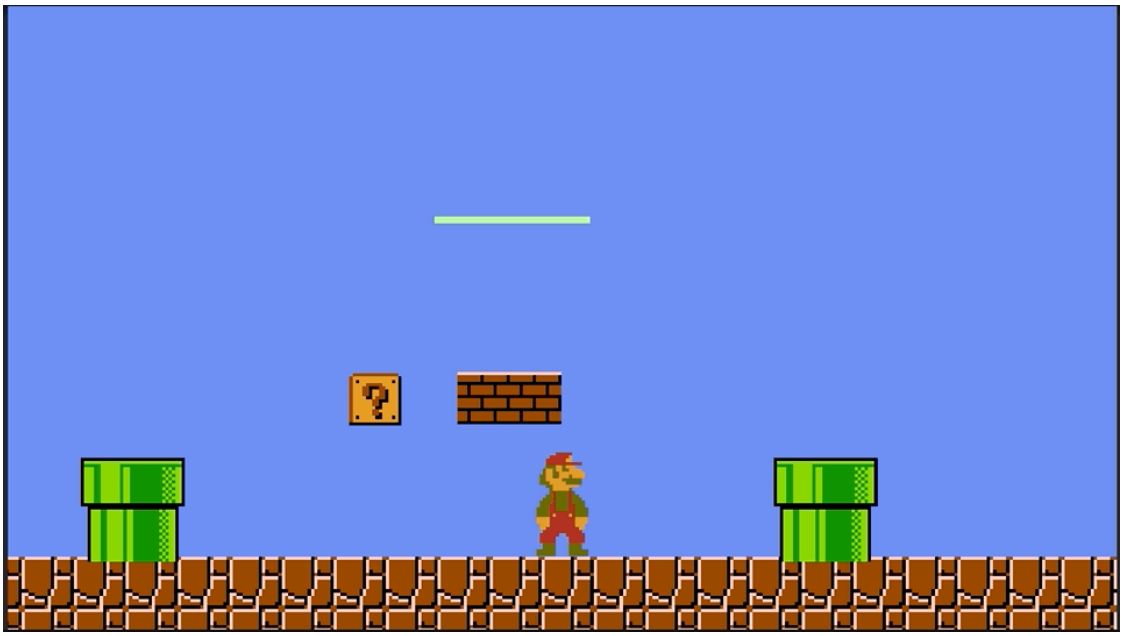
Has the Question Box moved?

One problem with the above implementation is that we cannot know for sure if the box has bounced up (has moved) *sufficiently high* before `ObjectMovedAndStopped()` returns **true**, depending on *where* Mario hits the box. It can be the case that Mario so very slightly collides with the question box's edge but didn't give sufficient energy to bounce the spring attached to the hittable box.

To fix this quickly, we can add **more** upwards force when collision is detected, and be sure that the `HittableSimple` box bounces at least of a certain amount regardless of Mario's momentum. The complete `OnCollisionEnter2D` implementation is as follows:

```
void  OnCollisionEnter2D(Collision2D col)
{
        if (col.gameObject.CompareTag("Player") &&  !hit){
                hit  =  true;
                // ensure that we move this object sufficiently
                rigidBody.AddForce(new  Vector2(0, rigidBody.mass*20), ForceMode2D.I
                // spawn mushroom
                Instantiate(consummablePrefab, new  Vector3(this.transform.position.
                // begin check to disable object's spring and rigidbody
                StartCoroutine(DisableHittable());
        }
}
```

If everything's implemented correctly, you should have this nice bouncy box effect that's no longer moving or spawn anything after it was hit once:



> **Disclaimer**: Note that this is not the only way to implement the entire bouncing effect of the box. There's a lot of other alternatives out there, which you are free to choose and implement. These specific methods of using SpringJoint2D, Rigidbody2D methods (addForce, etc), and Coroutines are selected because we want to teach the Unity newbies these concepts by incorporating it into our sample game.

Moving the Camera

The second thing to do to complete the game for this tutorial is to let the Camera follow the player, but clamped so that it doesn't go out of screen too much the left or too much the right. Create a new script called `CameraController.cs` and declare the following variables:

```
public  Transform player; // Mario's Transform
public  Transform endLimit; // GameObject that indicates end of map
```

```
private float offset; // initial x-offset between camera and Mario
private float startX; // smallest x-coordinate of the Camera
private float endX; // largest x-coordinate of the camera
private float viewportHalfWidth;
```

In the `Start()` method, we instantiate a few things, but the highlight lies on the part where we need to get the **world coordinate** of the bottom-left point of the Camera's **viewport** using `ViewportToWorldPoint`. The reason we do this is because we need to find out what exactly is the world coordinate (x-coordinate specifically) of the leftmost point of the viewport.

> We use it later on to prevent the camera to move *too much to the left.*

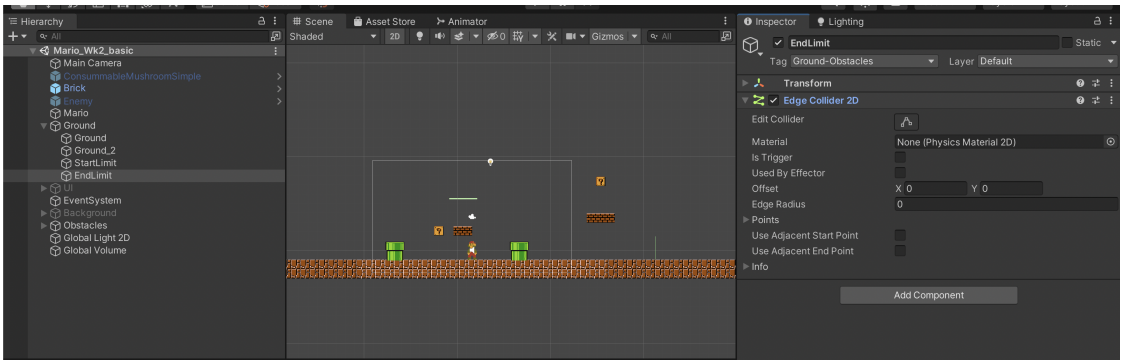
```
void Start()
{
        // get coordinate of the bottomleft of the viewport
        // z doesn't matter since the camera is orthographic
        Vector3 bottomLeft = Camera.main.ViewportToWorldPoint(new Vector3(0, 0, 0)
        viewportHalfWidth = Mathf.Abs(bottomLeft.x - this.transform.position.x);

        offset = this.transform.position.x - player.position.x;
        startX = this.transform.position.x;
        endX = endLimit.transform.position.x - viewportHalfWidth;
}
```

Then under the `Update()` method, the camera constantly follow the player unless it has reached the ends of the game map:

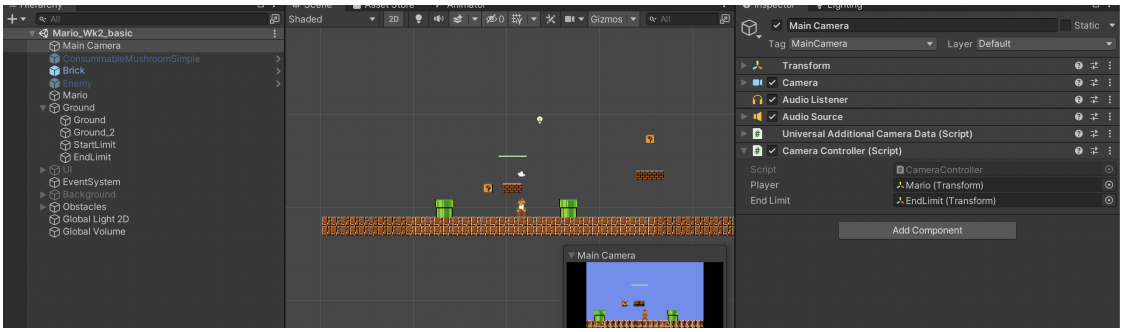
```
void Update()
{
        float desiredX = player.position.x + offset;
        // check if desiredX is within startX and endX
        if (desiredX > startX && desiredX < endX)
        this.transform.position = new Vector3(desiredX, thi
}
```

Create an empty GameObject called `EndLimit` with an EdgeCollider2D to prevent Mario from going over too much to the right:



> You can do the same for the left side as well, right at the left side of the Camera's ViewPort. In the screenshot above, we name it `StartLimit`.

Then in the Camera's inspector, link up the references of Mario's Transform and EndLimit's Transform:



Test run and you shall have the Camera gloriously following Mario around the Scene.

Destroy GameObject

One final thing to do is to ensure that the `ConsummableMushroomSimple` Prefab is **Destroyed** once it **goes out of view.** It is very simple to do this as there's a callback dedicated for it: `OnBecameInvisible`.

> From Unity's documentation: `OnBecameInvisible` is called when the **renderer** is no longer visible by any camera. This message is sent to all scripts attached to the renderer. OnBecameVisible and OnBecameInvisible is useful to avoid computations that are only necessary when the object is visible.

Open the script controlling `ConsummableMushroomSimple` and add the following method, and we're done! The mushroom disappears once it's no longer visible by the camera. This callback obviously only works on gameObjects that have `Renderer` attached to it.

```
void  OnBecameInvisible(){
        Destroy(gameObject);
}
```

> Some newbies will make the mistake `Destroy(this)` thinking that `this` is the `gameObject`. This is not true. `this` refers to `this (script) instance` instead.

Checkoff

For this checkoff, you're required to implement everything you can see in the demo .gif below (you can just gauge and estimate the placement of each obstacle).

Everything is mostly covered in this lab, **except the script that controls the `ConsummableMushroomSimple`** and the addition of **background objects** as shown in the demo .gif below. In summary, for this checkoff you must (but you can do more):

- Write a script so the spawned mushroom can behave exactly like the .gif shown.
  - You can use the suggested method above, or use other means that gives the same visual effect. You can also use any Mushroom sprite, it doesn't have to be this green one or the red one above.
  - Note that you are also **free to use any alternative method** to create the bouncy question box (by using Animator, or manually from script). You don't have to follow our method of using the SpringJoint2D.
- Create various GameObjects to render some nice background: the small hills in the background as well as the background *wallpaper*. You have to organise your Sprite Renderer's **Sorting Layer** properly here. Obviously all these objects do not affect the game, so you'll only need to have Transform and Sprite Renderer components attached to these background GameObjects.

> Get used to the creation and organisation of various Layers, Tags, and Sorting Layers, and understand the application of each property:
>
> - **Layer**: for 2D collision matrix and *Camera culling mask (next week)*
> - **Sorting Layer**: to determine what is rendered in front of what when they're on the same XY-plane
> - **Tags**: to quickly prototype and identify the `gameObject` (although you can also search by its *name*).

Refer to our course handout as usual to find out the standard protocol on how to submit your lab checkoff.

Next

We improve on a few things this time round, but we still lack a few features: the enemies, counting of scores and coin collection, having power-ups effect on the character, and arranging the world to match Super Mario Bros World 1-1. However with your skills now, it should be clear how to implement them (at least to get it to work) so we will not put it as a priority at this point. In the next Lab we will learn new things instead, that is how to polish the looks of this game: adding VFX and basic SFX.

NEXT POST
Unity for Newborns