Player Physics|Checkpoint Re-spawning|Particle Effect

Basic Rigidbody2D properties

- **Mass:** the weight of an object/character
- **Gravity:** you can make it behave exactly like in the real world, or you could make a world with little to no gravity. Gravity affects how things move, how high character can jump, and how an object reacts to a force against it
- **Linear Drag:** air resistance when moving along x/y axes
- **Angular Drag:** air resistance when rotating along x/y axes
- **Fixed Angle:** can the object spin when hit by something else?
- **Kinematic:** gravity doesn’t affect the object

Basic Collider2D properties

- Colliders can be box-shaped, circular, edge, or polygonal. Properties vary from one type to another
  - **Material:** property to assign a type of material to affect objects that collide with it
  - **Radius:** how big or small the circle collider is around the game object
  - **Size:** how big or small the box collider is around the game object
  - **Offset:** how far away the collider is from the game object itself
  - **Is Trigger:** If checked, a GameObject can trigger an event when it collides with another GameObject. This is useful for setting up events you want to happen when your player comes in contact with them.
    - Example: when the player runs into an enemy, it dies

Hinge Joint Constraint

- The most common use of a hinge joint in Unity is to create a *trapdoor* for your player character to fall through

- Unlike your player character, you don’t want your platform or ground to fall away due to gravity!

- So we add a hinge constraint to tell the physics engine that the object has a point on it that will constrain it to its position on the screen but allow it to rotate with the force of gravity without actually falling. There are a few different constraint types you are able to add to a Rigidbody2D Component.

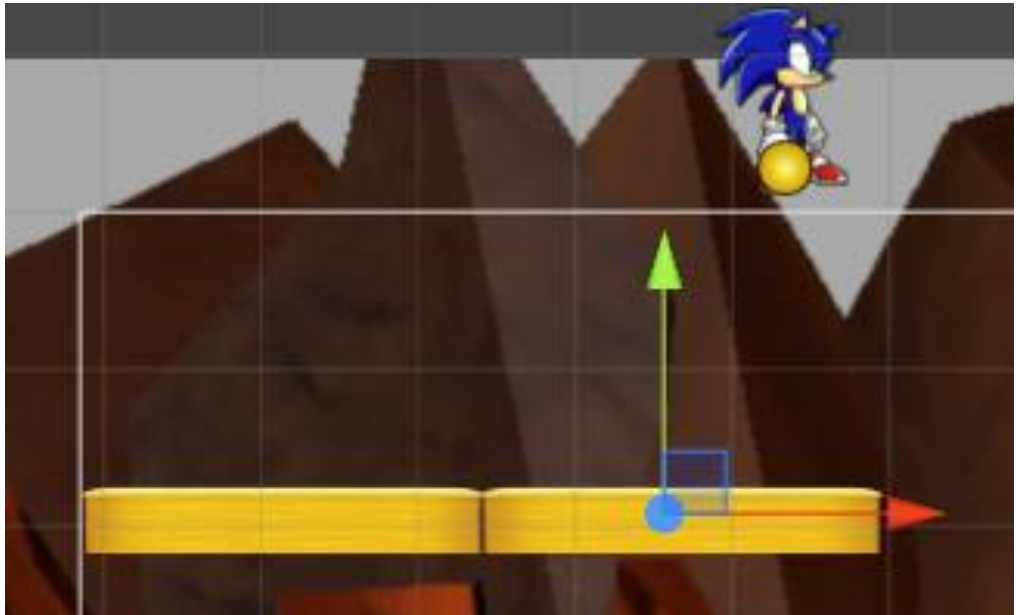
- A Hinge Joint can be triggered by a collision from another game object

Other Types of Joint Constraints

- There are other types of joints other than the Hinge Joint. Some of them are:
  - **Spring Joint 2D:** Creates a spring-type effect with two colliding GameObjects
  - **Slider Joint 2D:** Constrains a GameObject along a set path, like an elevator door. Can also be set to activate upon collision or continually
  - **Wheel Joint 2D:** Constrains an object to a position but allows it to rotate

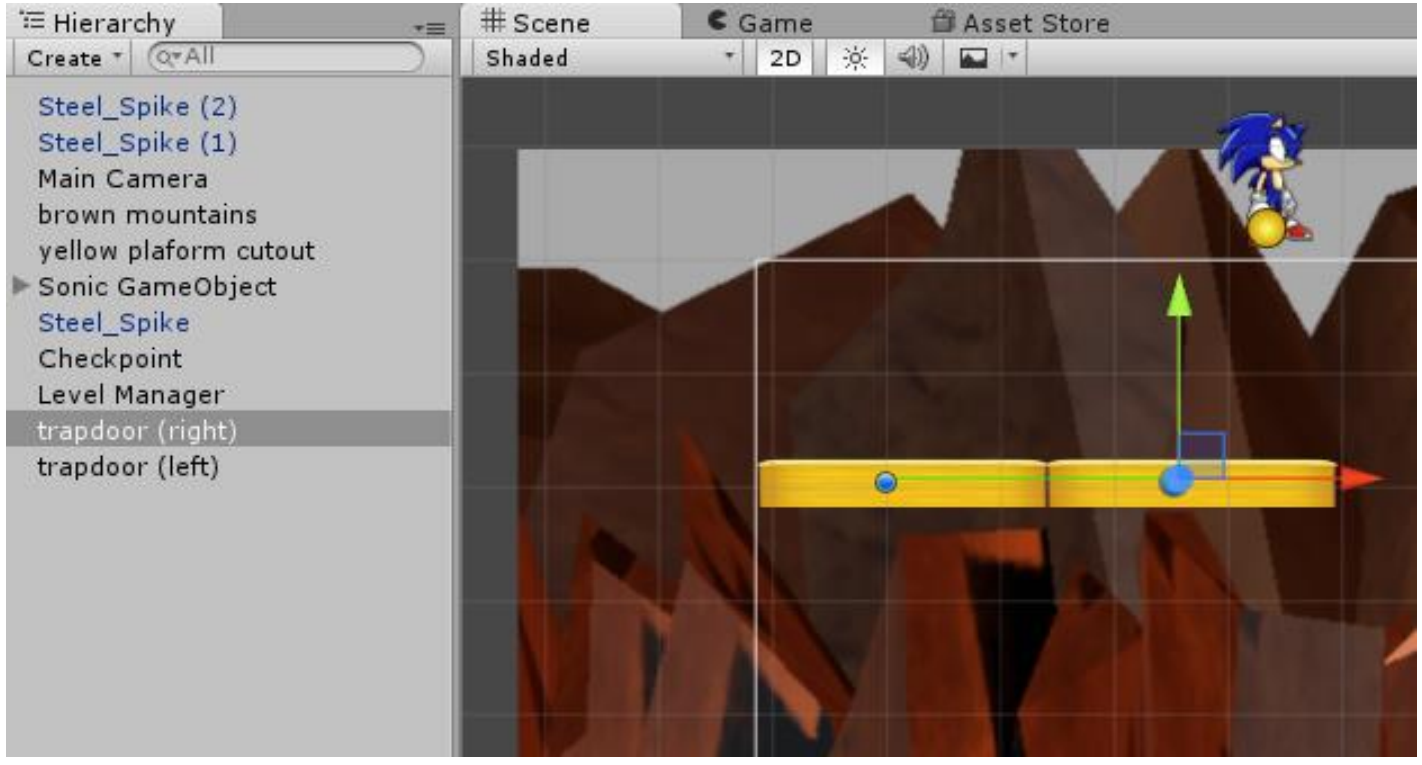
Exercise #14 – Create a Hinge Joint

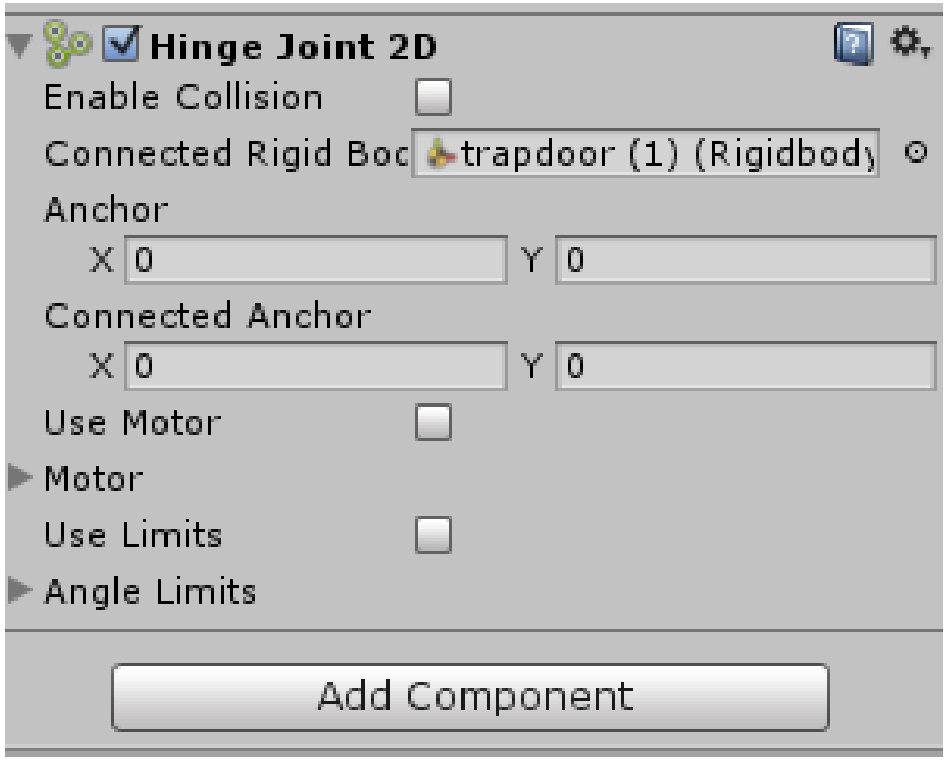
- Choose a couple platform game objects and attach a Rigidbody2D and BoxCollider2D to both of them. Attach a HingeJoint2D to the one on the right only
- In the example below, they are placed underneath Sonic because we want the one on the right to open *downwards* like a trap door when he collides with it



Exercise #14 – Create a Hinge Joint (Cont.)

- In the Hinge Joint2D component of the right-side platform in Inspector, change the value of the Connected Rigid Body property by dragging the platform game object on the left from the Hierarchy view and dropping it into the text box in the Inspector view

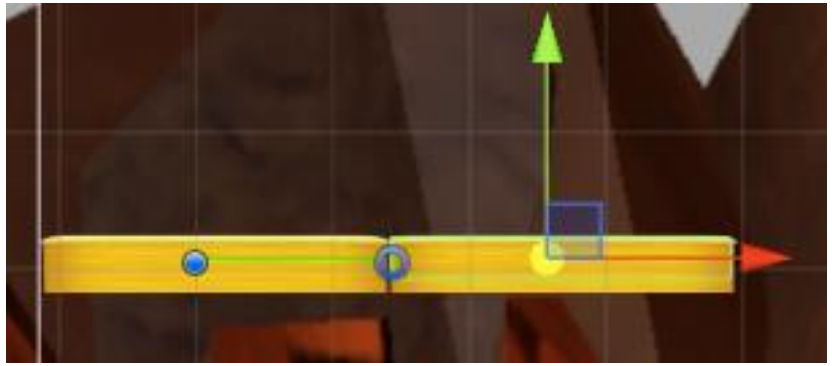


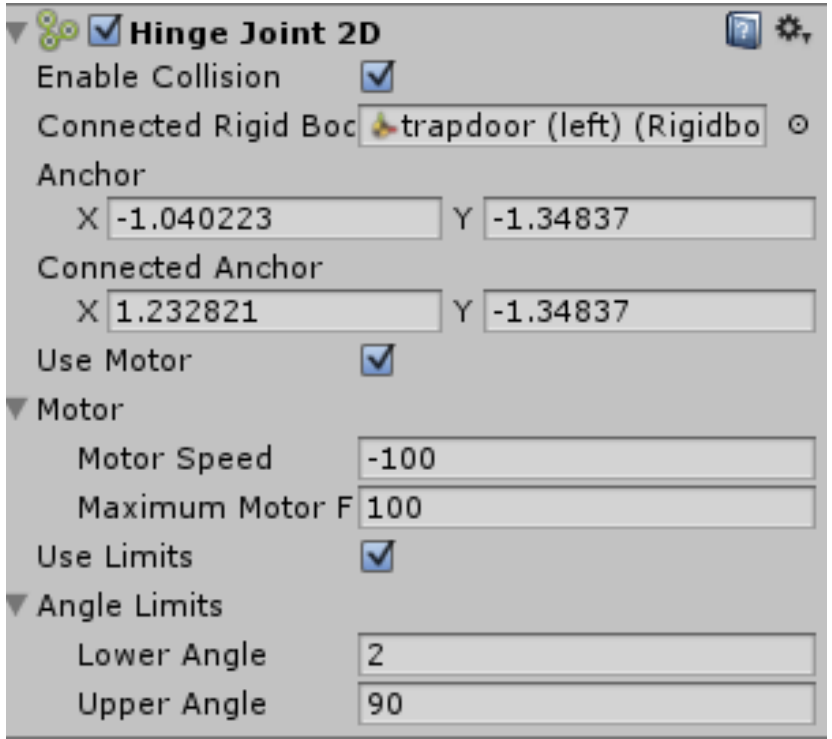


Exercise #14 – Create a Hinge Joint (Cont.)

- Understand what these properties do to get the hinge joint working the way you want it to:
  - **Anchor** – the hollow circle. It's the point your platform rotates around
  - **Connected Anchor** – the full circle. It indicates where in your scene the anchor is placed. If you're not sure where that is, experiment with its position till you get it the way you want
    - (In many cases, putting the full circle inside the hollow circle will be enough for you to get the effect you need)
  - **Enable Collision** – check this if you don't want your trapdoor to swing through the platform on the left as if it's invisible! You want them to *collide* with each other
  - **Use Limits** – check it if you don't want your trapdoor to rotate fully when something collides with it. Change the values of Lower Angle and Upper Angle until you get the effect you need. For this exercise, use degrees 2 and 90, respectively
  - **Use Motor** – check it to apply torque and make the platform rotate and control speed of rotation when the player collides with it. To understand the values *Motor Speed* and *Maximum Motor Force*, check Reference #1 at the end of this Powerpoint. For this exercise, use the values -100 and 100, respectively

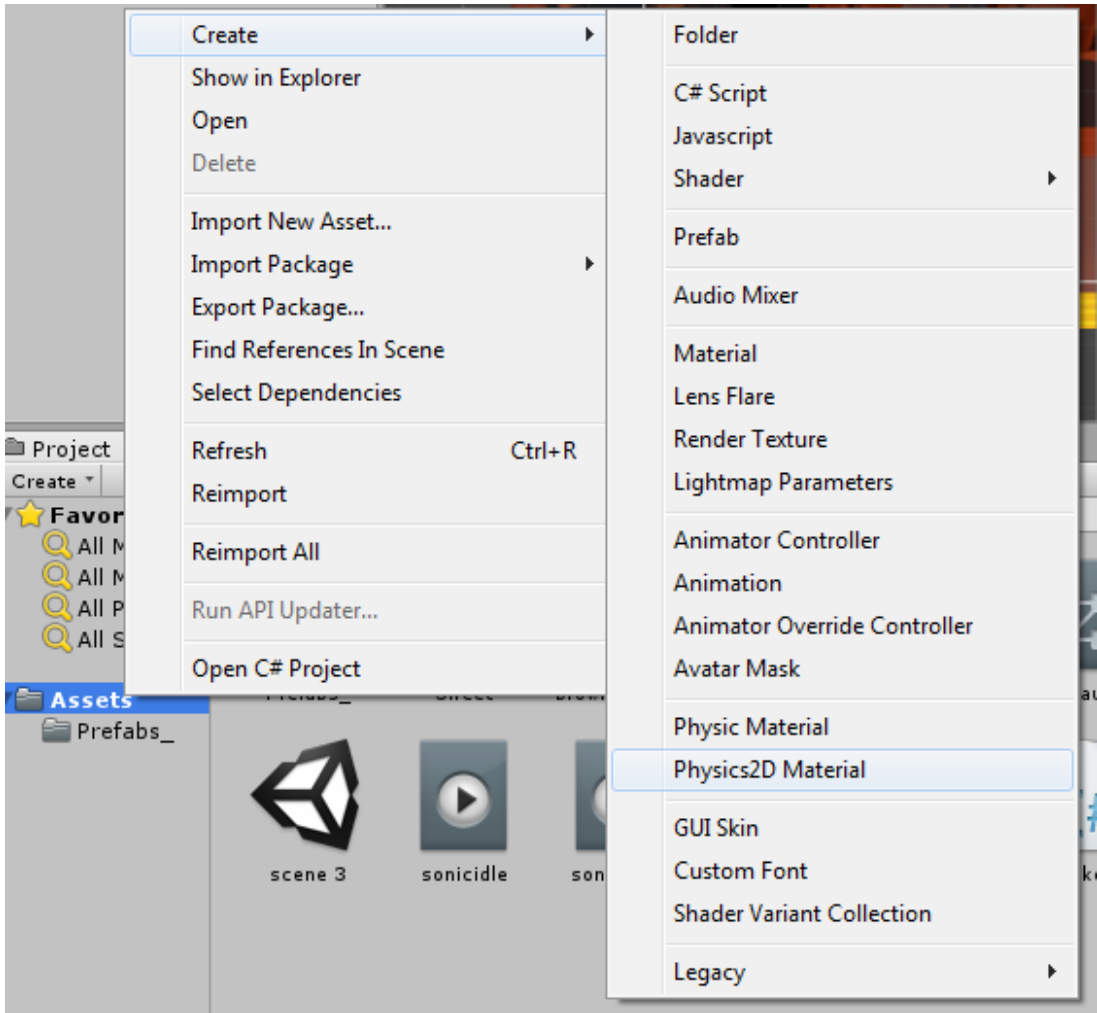
Exercise #14 – Create a Hinge Joint (Cont.)

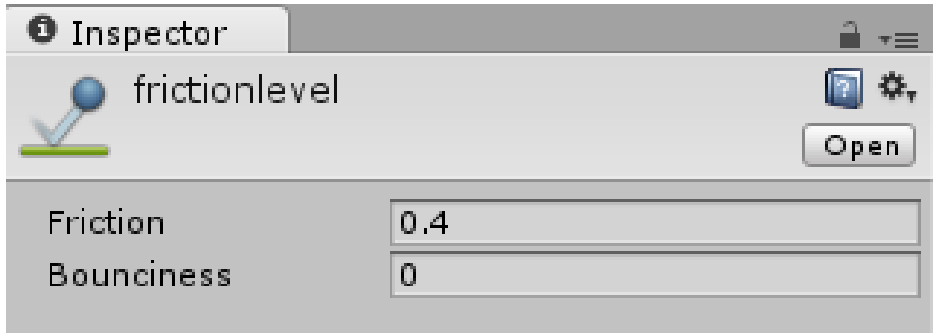




Exercise #15 - Creating Friction

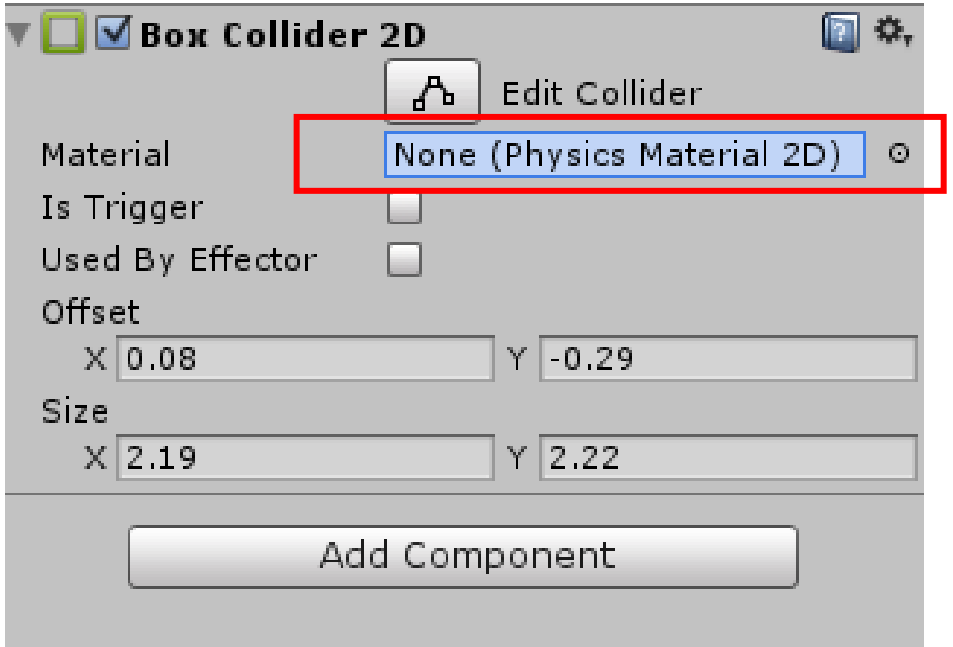
- If your ground/platform is much too slippery, you can make it rougher by controlling the Friction
- Right-click on Assets, and Choose Physics2D Material
- Name the new material and go to its Inspector view
- Change the value of friction to whatever suits you
    - Zero is slippery. One is very rough.





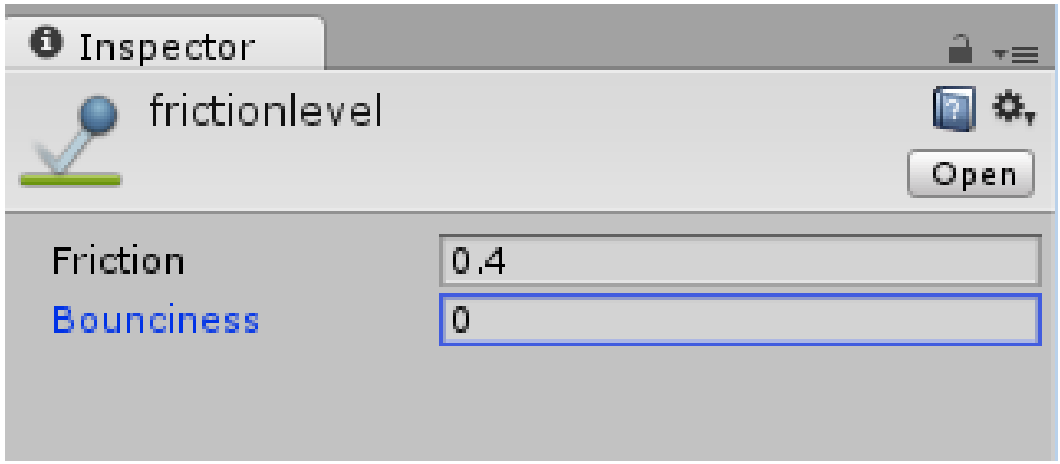
Exercise #15 - Creating Friction (Cont.)

- Now you must attach the material to your ground for the friction to work
  - Make sure your ground already has a box collider component attached!
- Select the material from your Assets folder, and drag it to the Material property inside the Box Collider component the Inspector window and drop it
- Now the ground will become as rough or as smooth as you need it to be



Exercise #15 - Creating Bounciness

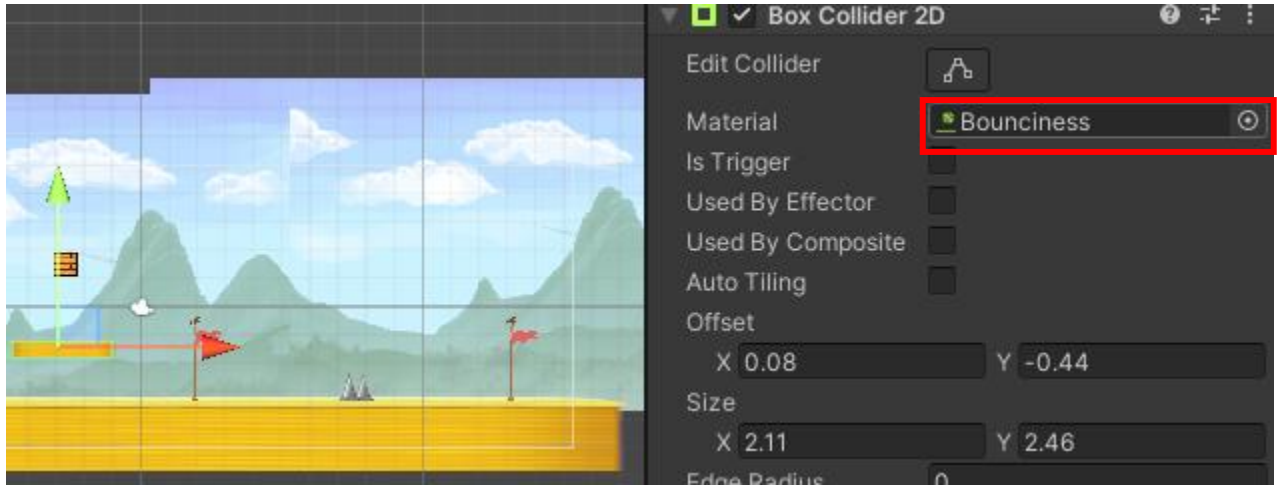
- Say you have a platform that the player can bounce on like a trampoline
- The procedure is the same as in the previous friction exercise
- Right-click on Assets, and Choose Physics2D Material
- Name the new material and go to its Inspector view
- Change the value of bounciness
  - Zero indicates no bounce at all



Exercise #15 - Creating Bounciness (Cont.)

- Make sure the platform has a collider2D component attached
- Drag the bouncy material from the Assets folder all the way to the Material property inside the Collider2D component the Inspector window and drop it

- Now the player will bounce upon landing on the platform

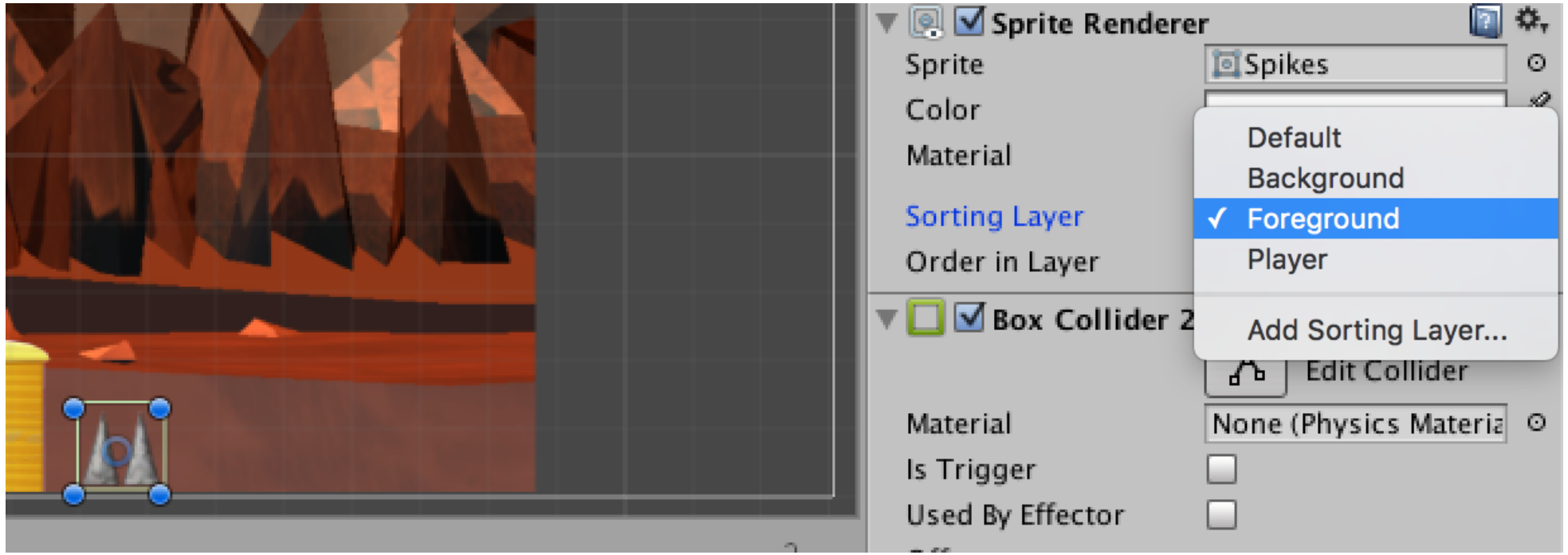


Checkpoints

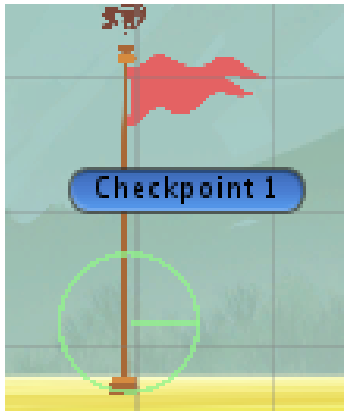
- We need to take into account that players can fall into the water or, eventually, get themselves knocked out by an enemy. Would we start them at the beginning of the level all over again? Of course not! We’re going to create a checkpoint system to account for such circumstances.

Exercise #16 - Checkpoints – Step 1: Add Spikes game object

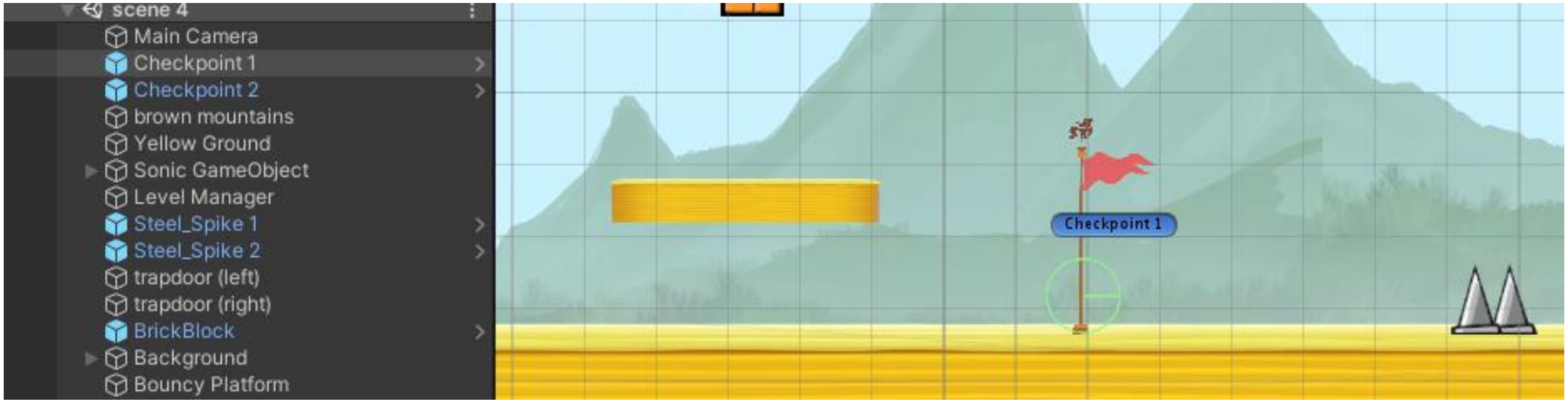
- Add Spikes game object to your scene
- Change it's sorting layer to Foreground
- Add a Box Collider 2D component to it



Exercise #16 - Checkpoints – Step 2: Add a Checkpoint GameObject

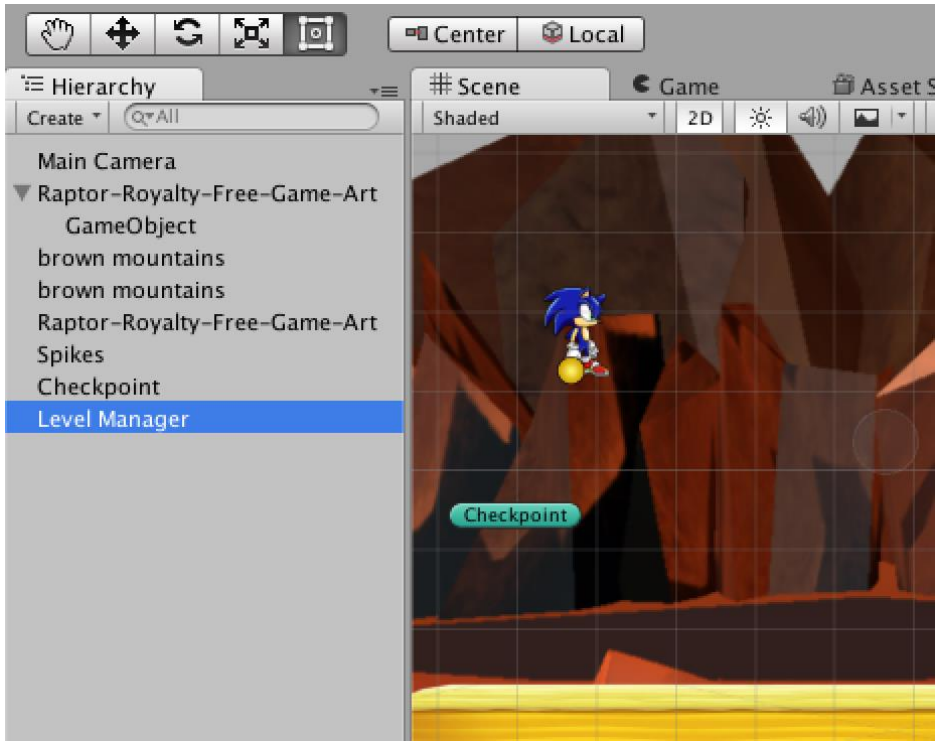


- Create a new gameObject and call it Checkpoint
- Give it a relevant icon or sprite (e.g. Flag)
- Add a Collider2D
- (Make sure its z position is 0, or otherwise not behind the background)



Exercise #16 - Checkpoints – Step 3: Add a Level Manager GameObject

- Create another empty game object to be used as a level manager, rename it to *Level Manager*
- Level Manager will be responsible for all logic related to this game level, but it doesn't need to be visible on the scene like Checkpoint and everything else.



Exercise #16 - Checkpoints – Step 4: Add three script files

- Create three script files for each of our 3 newly-created game objects
  - Spikes
  - Checkpoints
  - LevelManager

Exercise #16 - Checkpoints – Step 6: Write LevelManager script Logic

- LevelManager script is responsible for tracking the current checkpoint, it will use the current checkpoint's position to return our player to it when respawned.

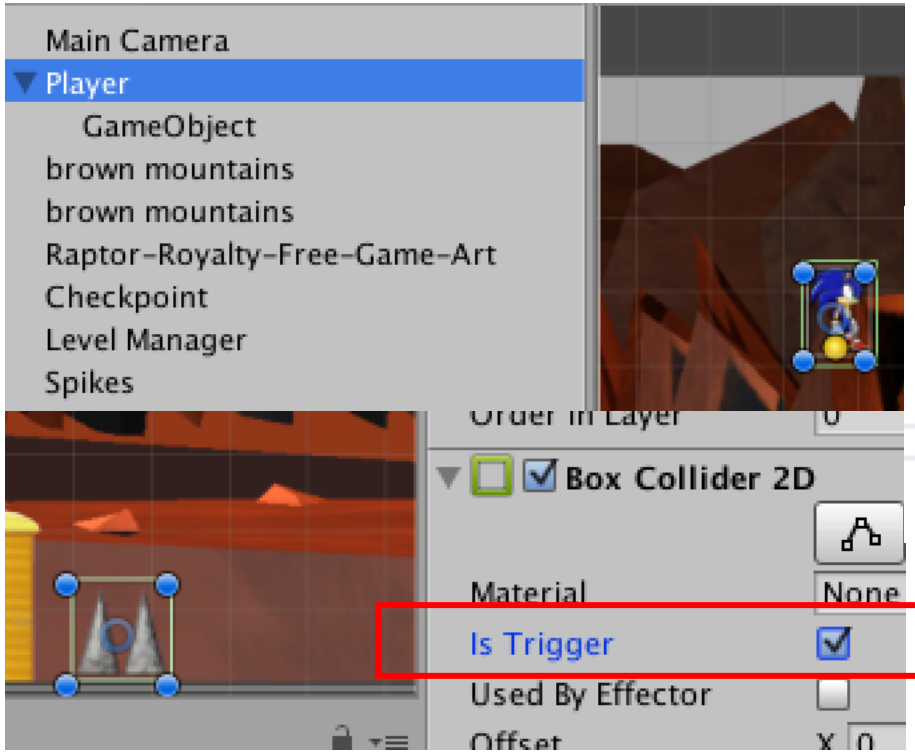
```
public GameObject CurrentCheckpoint; //so we can update the current checkpoint from within Unity
// Use this for initialization
void Start () {

}

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

}

public void RespawnPlayer()
{
    FindObjectOfType<Controller>().transform.position = CurrentCheckpoint.transform.position;//Search for the asset/object called
    //Controller (your player's script code name whatever it is). Once you've found it, change its player game object's position
    //to be at thew last checkpoint the player passed through before s/he died
}
```

Exercise #16 - Checkpoints – Step 6: Write LevelManager script Logic

- CurrentCheckpoint is a public variable, so we can change it from inside Unity. We'll change it from Unity to set the first checkpoint by dragging and dropping, then it will be updated dynamically when player passes by the next checkpoints.

Exercise #16 - Checkpoints – Step 7: Write Spikes script Logic

- Spikes script is responsible for detecting collision of player with it
- Make sure to give the Player a ‘Tag’ in the Inspector view. We’ll use it to make if-conditions easy
- Make sure that “Is Trigger” property of Spikes game object is checked, so that it can be overlapped with other game objects with box colliders




```
//function that executes upon trigger (when something collides with the spikes)
void OnTriggerEnter2D(Collider2D other)
{
    if  (other.tag == “Player”)
    //if the collider of the object whose name is Sonic GameObjects touches the spike collider

     FindObjectOfType<LevelManager>().RespawnPlayer();
    //go to the Level Manager script, and execute the Respawn Player function
}
```

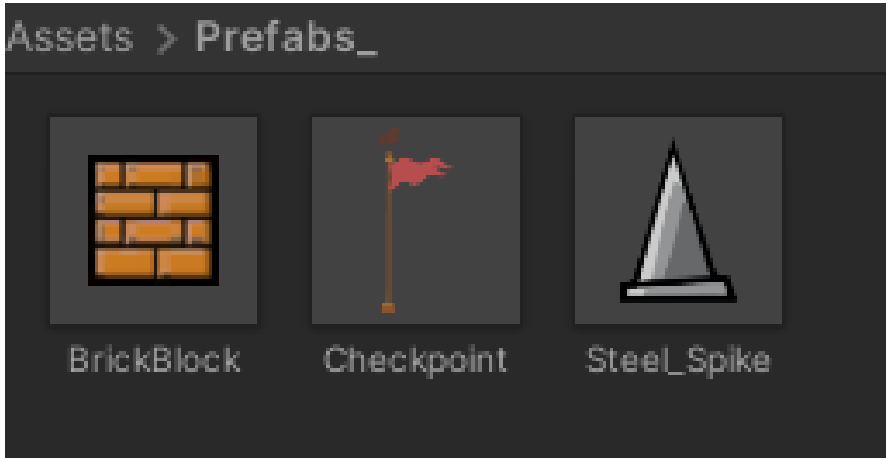
- Now play the scene. Let the player run through the checkpoint first, then run at the spikes. The moment he collides with them, he’ll immediately respawn back at the checkpoint

Exercise #15 - Checkpoints – Step 8: Add collider to checkpoint (for multiple checkpoints)

- Add Circle Collider 2D to the checkpoint game object and check ‘Is Trigger’ so that the player can interact with it

Exercise #16 - Checkpoints – Step 9: Add Spikes and Checkpoints Prefabs (for multiple checkpoints)

- Drag your Spikes and Checkpoints game objects to your Prefabs folder in Assets, so that you can create multiple Spikes and Checkpoints with same characteristics



Exercise #16 - Checkpoints – Step 10: Write Checkpoints script logic (for multiple checkpoints)

- Checkpoints script is responsible for updating the currentCheckpoint variable of the LevelManager when player passes by it

```
void OnTriggerEnter2D (Collider2D other)
{
    if  (other.tag == “Player”)
        //if the collider of the object whose name is Sonic GameObjects touches the checkpoint's circle collider


       FindObjectOfType<LevelManager>().CurrentCheckpoint = this.gameObject;
    //go to the Level Manager script, and update the value of CurrentCheckpoint to become the new Checkpoint the player has
    //just passed through. This is necessary when you have several checkpoints in a level
}
```

- Now when player interacts with any checkpoint, Checkpoint script will update the currentCheckpoint of the LevelManager so that the player will be returned to the currentCheckpoint (the last checkpoint the player interacts with) when spawned

Exercise #17 – Particle Effect (Animation)

- Particle effect is a unique tool that can add interactivity and responsiveness to your games. They add to the fine-tuning and detail of a game's look and animation

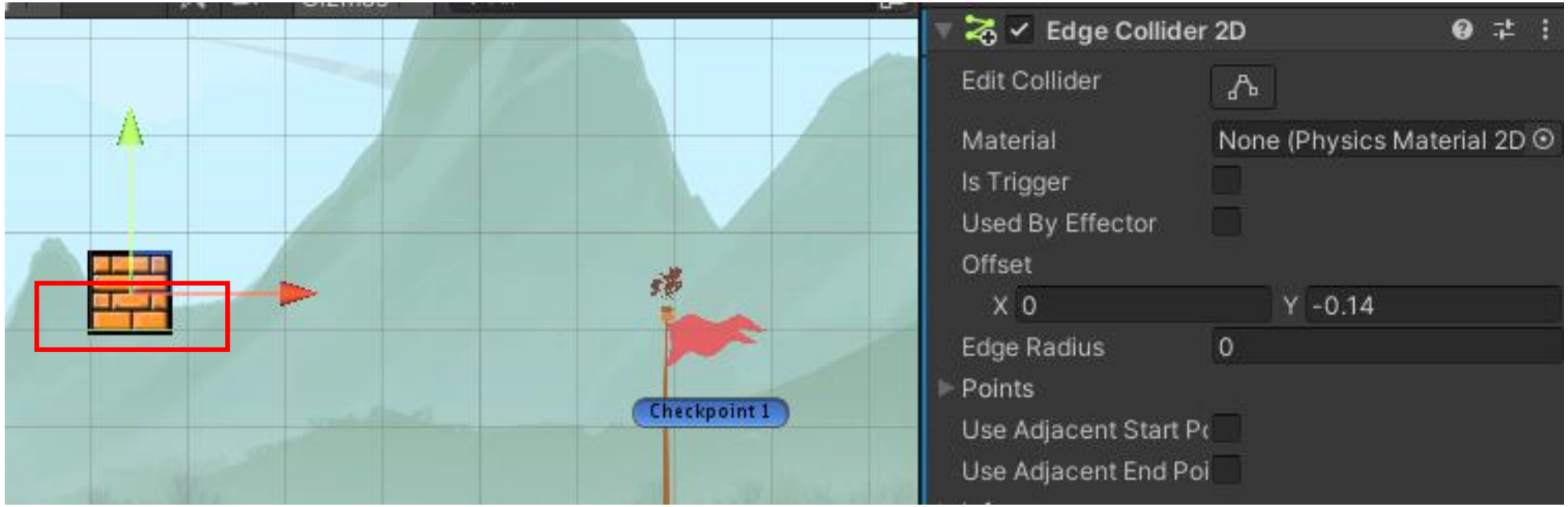
- As in Mario games, the player can break bricks by colliding with them

- The idea we'll play with is to swipe one sprite with another to create the illusion of animating a particle effect

Exercise #17 – Particle Effect – Step 1: Add Brick game object

- Add a Brick game object and place it on a suitable Sorting Layer

- Add component EdgeCollider 2D to its bottom side



Exercise #17 – Particle Effect – Step 2: Create script “Brick”

- Create script “Brick” and attach it to your brick game object.

```
public class Brick : MonoBehaviour {

    // Used to change the sprite
    private SpriteRenderer sr;

    // The sprite to change into
    public Sprite explodedBlock;

    // Use this for initialization
    void Start()
    {
        sr = GetComponent<SpriteRenderer>();
    }
```

Exercise #17 – Particle Effect – Step 2: Create script “Brick” (Cont.)

- In the If-condition, we check to see if the player game object has the correct name, and make sure he only needs to hit that block once, hence the index number zero

- Logically, the player’s y value should always be *smaller* than the block’s, so we put that in the condition as well

```
// Called when something hits the BrickBlock
void OnCollisionEnter2D(Collision2D other)
{
    // Check if the collision hit the bottom of the block
    if (other.tag == “Player” && other.GetContact(0).point.y < transform.position.y)
    {
```

Exercise #17 – Particle Effect – Step 2: Create script “Brick” (Cont.)

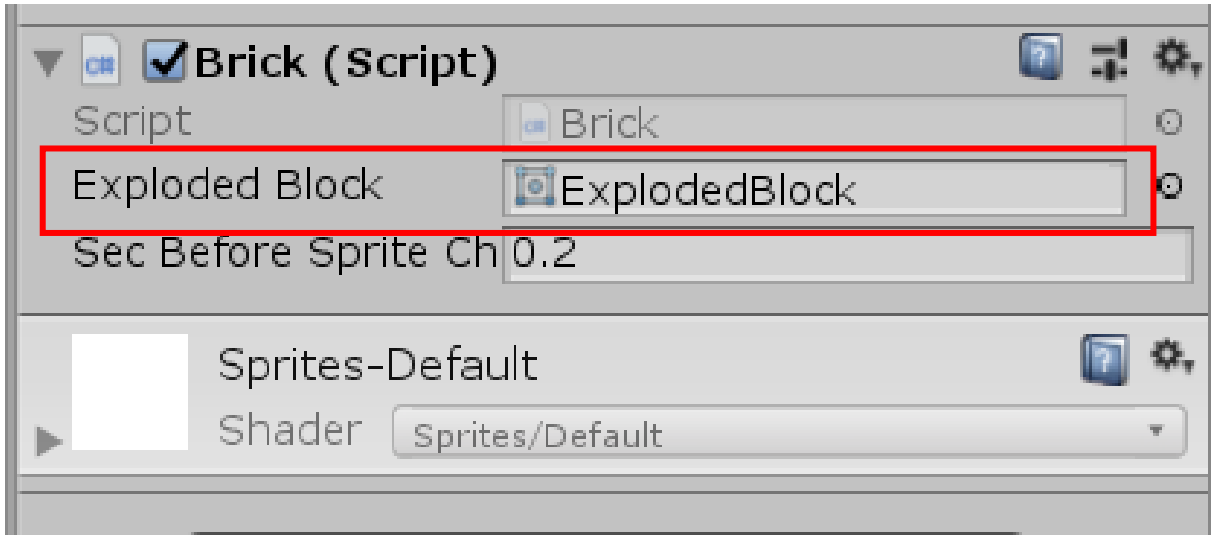
- If the condition is true, the Sprite Renderer will replace the BrickBlock sprite with the ExplodedBlock sprite

- Then the block will be destroyed altogether after a fraction of time

```
    if (other.tag == “Player” && other.GetContact(0).point.y < transform.position.y)
    {
        // Change the Block sprite
        sr.sprite = explodedBlock;

        // Wait a fraction of a second and then destroy the BrickBlock
        Object.Destroy(gameObject, .2f);
    }
}
}
```

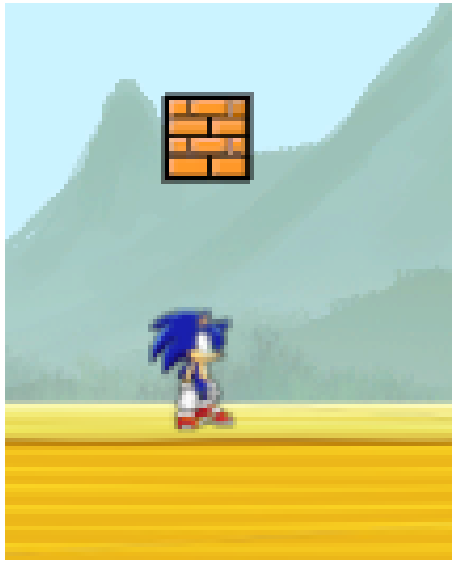
Exercise #17 – Particle Effect – Step 3: Making modifications after finishing the script

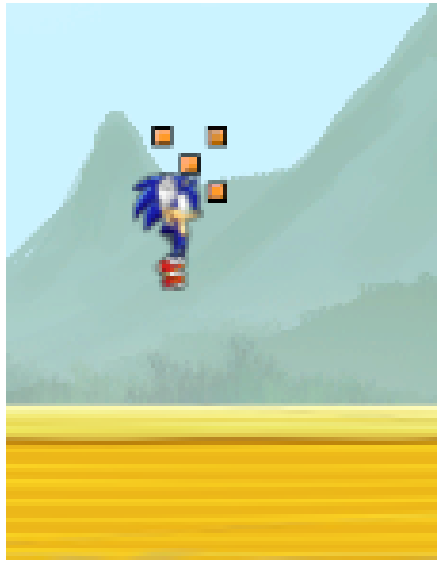
- Go back to Unity and drag and drop your ExplodedBlock sprite to its variable.



Exercise #17 – Particle Effect – Result

- Now when the player game object hits the brick game object from below, it will result in the brick exploding, then disappearing





References

- **Hinge Joint2D Tutorial. URL retrieved from:** https://www.youtube.com/watch?v=l6awvCT29yU
- **Endless Runner Game in Unity. URL retrieved from:** https://www.youtube.com/watch?v=GrQalFLtQT4&list=PLiyfvmtjWC_XmdYfXm2i1AQ3lKrEPgc9-
- **Checkpoints & Respawning - Unity 2D Platformer Tutorial - Part 3. URL retrieved from:** https://www.youtube.com/watch?v=ndYd4S7UkAU
- **HOW TO MAKE 2D PARTICLE EFFECTS - UNITY TUTORIAL. URL retrieved from:** https://www.youtube.com/watch?v=_z68_OoC_0o