MegaMan Engine for Unity

Made by MegaChibisX

Boss Creation

There is no one specific way to make a boss for a game. This tutorial will mainly go over creating and including a boss in the engine, however the main focus will be the scripts needed and used to make the boss compatible with the engine. The patterns, designs and animations are mostly up to you. In this tutorial Star Man will be implemented, but his behavior will be expanded, hopefully to give you a better idea of how the process will work. You can follow along with your own boss.

You can look at Star Man's script as a reference if you want to. Simply navigate to Assets>Scripts>Bosses>Bo_StarMan.

The Scripts:

Before you start, you should be aware of some basics. All bosses are expected to use the **Boss** script. To derive from it, you will need to name your class something like

```
public class MyBoss : Boss
{

}
```

For a script to derive from another, it means that the child script (**MyBoss**) will be using methods and variables from another script (**Boss**), and will be expanding on them. You may notice that Unity scripts derive from **MonoBehaviour** by default. That allows these objects to appear in the scene, among other things.

The **Boss** script already derives from the **Enemy** script. It borrows the following variables:

- **health**: It keeps track of the health of the boss.
- **damage**: The contact damage the boss does on the player.
- **canBeHit**: Will be set to false when the boss needs to be invincible, like right after being hit.
- **shielded**: To not be confused with **canBeHit**, this should be used when the boss can't be hit due to having a barrier of some sort active.
- **destroyOutsideCamera**: This is only used by enemies. Make sure to turn off for your boss, although it should be disabled anyway.
- **center**: This is used to set the sprite center of the boss. It is used for deciding things like the local position the explosion should spawn at when an enemy dies.
- **deathExplosion**: The explosion that shows when the boss dies.
- **body**: The **Rigidbody2D** component of the boss, which all bosses need to have.

The boss script expands on the variables by adding the following:

- **deathExplosionBoss**: The gameover explosion to make when dead.
- **healthBarFull** and **healthBarEmpty**: The sprites used for the healthbar UI. Examples can be found under **Assets>Resources>Sprites>Menus>Bars**.
- **fightStarted**: Checks if the fight has started to prevent the boss from playing their intro multiple times.
- **invisTime**: The timer that counts how long until you can hit the boss again.
- **endStageAfterFight**: Checks if MegaMan should teleport away from the stage once the fight is done.

It also borrows some methods:

- **Start()**: It is called right when the boss is spawned in the game. That means it fires at the beginning of the stage, not when the boss fight starts, as the boss is one of the enemies that don't typically respawn.
- **OnDrawGizmosSelected()**: It displays visual information on the **Scene** view.
- **Damage(Pl_Weapon weapon)**: Receives damage when hit by a weapon. Weaknesses and resistances should be handled in here.
- **Damage(float dmg, bool ignoreInvis)**: The actual method for damage. Will do the amount of damage

given, taking invis into account unless **ignoreInvis = true**.

- **Kill(bool makeItem)**: Destroys the enemy. If **makeItem = true**, the enemy will spawn a random item, using the **Item.GetObjectFromItem()** method. Changing drop rates can happen here by changing the 1s in **Item.GetRandomItem(1, 1, 1, 1, 1, 1, 1, 1)** to the desired rates.
- **Despawn()**: Normally despawns the enemy if they exit the scene. This is not needed for bosses.
- **Shoot(...)**: Will create a normal shot (unless otherwise specified).
- **LookAtPlayer()**: Looks left or right depending on the player's position.
- **GetClosestPlayer()**: Returns the player.

Furthermore, the **Boss** script has some of its own methods bosses will use:

- **OnGUI()**: Shows sprites on the player's screen. This is mainly used to show the boss's health, but if something else needs to be drawn on the screen, it will be done here.
- **StartFight()**: Is fired when the fight starts. This will be used before the intro plays, and usually just calls the intro.
- **PlayIntro()**: This holds the entire intro. By being an **IEnumerator**, you are able to put delays in this method and have it stop and play again later in the game.
  yield return null will give an one frame delay.
  yield return new WaitForEndOfFrame() works the same way.
  yield return new WaitForFixedUpdate() will give an one physics tick delay.
  yield return new WaitForSeconds(time) will give a delay of the time given in seconds. Affected by **Time.timeScale**, so if the game moves at half speed, the delay will also double, and if it is paused, the delay will also be extended indefinitely.
  yield return new WaitForSecondsRealtime(time) will give a delay of the time given in seconds.
  yield break will stop the Coroutine at its place, like **return** does otherwise. **Return** does not work here.

There are furthermore some variables you may need to use, depending on your needs.

- **private Animator anim;**
  This will hold the boss's Animator component. The creation and use of the animator component will not be covered deeply here, as there are tons of excellent tutorials on **Sprite Sheet Animation** for Unity2D which cover the topic much better and more extensively than I can. Just keep in mind that playing animations is so done through scripting in this project, so you don't need to make transitions in the animator itself. You can still use them if you know how and find it easier.
- **private BoxCollider2D col;**
  This is your boss's collider component. An array **private BoxCollider2D[] cols;** can be used if multiple colliders are used by your boss.
- **private SpriteRenderer spr;**
  This is your boss's **SpriteRenderer**, aka the component that draws your boss's sprite on screen.
- **private AudioSource aud;**
  This is your boss's **AudioSource**. Like the **SpriteRenderer**, it allows sound to be played. Only one **AudioClip** can be played at a time, and you usually won't need more than one, but if you do, make multiple.
- **public Transform leftCorner;** and **public Transform rightCorner;**
  These are empty **GameObjects** that keep track if the room's left and right corner. It gives the boss a way to sense their environment.
- **public GameObject** shotOrObjectName;
  You need to keep track of the shoots and objects your boss can shoot. These are made as prefabs first and then assigned to these fields. It's easy to give them names you can understand.

Preparing sprites and animations:

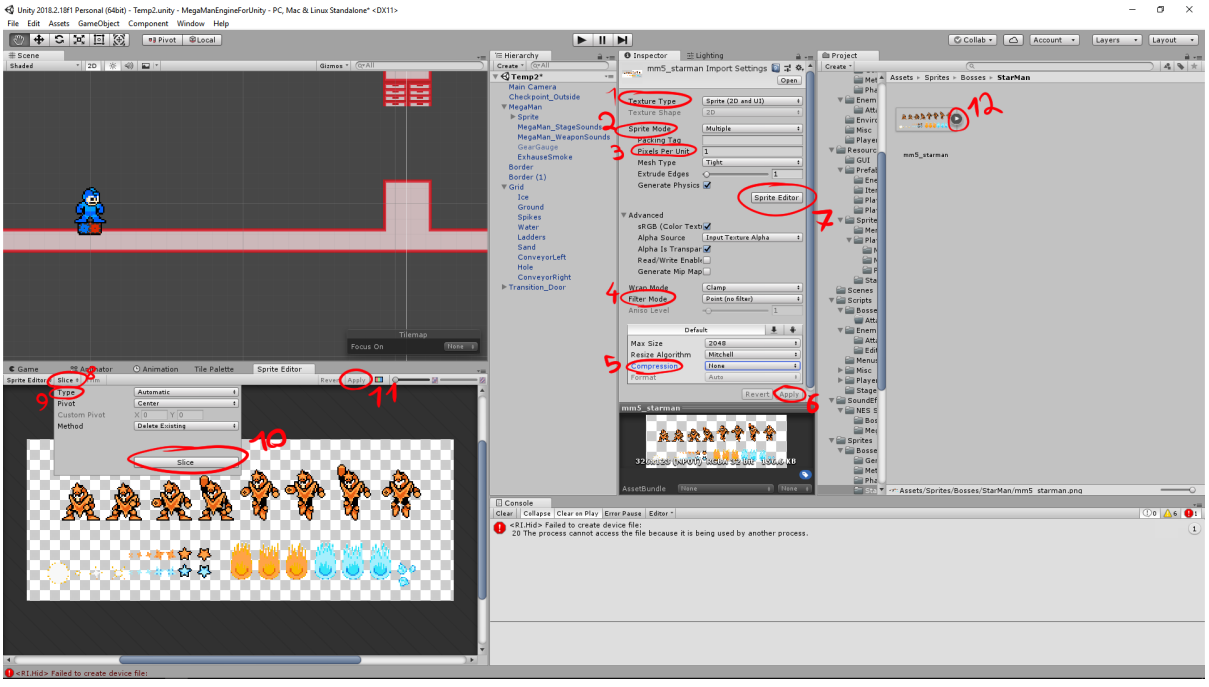
Sprites:
Now we can start! First of all, you need to make a sprite sheet. You can use your favorite software for that, but make sure the background is fully transparent. You can also use official sprite sheets, or someone else's sprites (assuming you have the rights to the image). **GIF** formats don't work with Unity, so make sure you export the sprite sheet as **.png**.

In Unity, navigate to **Assets>Sprites>Bosses**, right click in an empty spot inside the **Project** window, and click **Create>Folder**. Give the folder your boss's name, or something else appropriate. Drag the image file into the folder in the **Project** window.

(You can rename the file by pressing F2. To replace the file with a newer version without losing your progress here, right click on your image, select "**Show in Explorer**" and copy-paste your new file over the current one. DO NOT delete the file in the Unity Editor or click on the Unity Editor with the file missing and not replaced, as this might erase your current progress. Changes you've made to the file in the Editor are held in the **.meta** files Unity creates by default.).

For your image file to work with the engine, there are a few settings you need to change once you import your image.

- (1) Change **Texture Type** to "**Sprite (2D and UI)**".
- (2) Change **Sprite Mode** to "**Multiple**".
- (3) Change **Pixels Per Unit** to **1**.
- (4) Change **Filter Mode to "Point (no filter)**".
- (5) Change **Compression** to "**None**".
- (6) Click on the "**Apply**" button.



Next you need to cut your sprite sheet into slices. First of all, open the **Sprite Editor (7)**. Unity can automatically cut your sprite sheet into individual pieces. Click on **Slide (8).** Make sure **Type** is set to "**Automatic**" **(9)** and click **Select (10)**. To apply all the changes to Unity, press "**Apply**" **(11)**.

You may be done, but depending on your sprite sheet, Unity might have made some mistakes. If the sheet contains many small pieces close together or sprites with nearby borders, Unity could count them as one sprite. Unity could also count one sprite as multiple if it contains many small pieces. Click on a problematic square and resize it to fit only one sprite entirely. If there are extra squares left, click on them and press the **Delete** button. If a sprite is left bare, drag your mouse to create a new square and fit it around the sprite. When you are done, press "**Apply**".

An **arrow (12)** should have appeared next to your image. If not, go to the previous folder in the **Project** window and return. If it still doesn't appear, you may have forgotten to click **Apply (11)**.

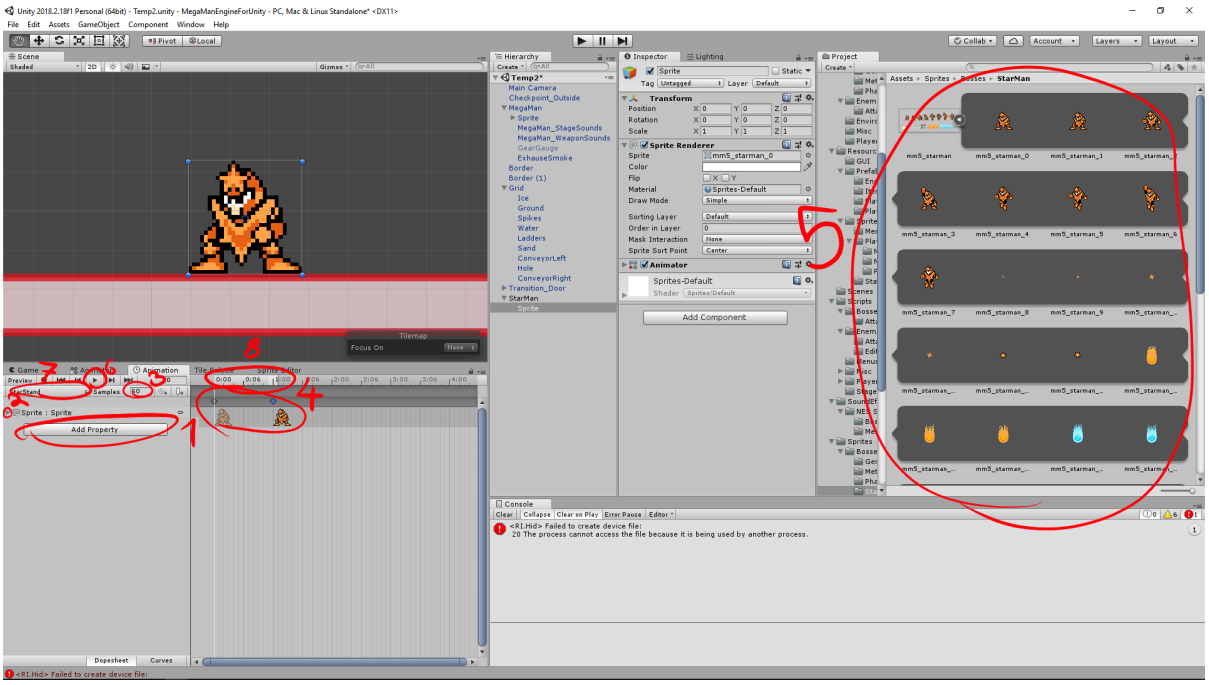
Animations:

On to the animations. First of all, click the **arrow (12)** to view all the sprites you've created. Click on the first frame of your standing animation sprite and drag it into the screen. Press the **F** button on your keyboard to focus on the sprite, then **right click** in an empty spot in the **Hierarchy** window and press "**Create Empty**". If the new GameObject is parented to your sprite or another object, click on its name and drag it to some empty space.

Rename the **Empty** GameObject to your boss's name, or another naming convention you may have. Click on the name of the GameObject that contains your sprite, and drag it right on your **Empty** GameObject's name. Now the **Sprite** GameObject should be parented to the **Empty** one. Rename the empty one to "**Sprite**" for convenience. Set the **Sprite**'s **Position** and **Rotation** to **(0, 0, 0)** in the transform component. The **Scale** should also be at **(1, 1, 1)**.

Open the **Animation** window if it is not open already. You can find it in **Window>Animation>Animation**. Drag it to a place that makes it convenient for you to work with. You need to have both the **Animation** and **Project** windows open at the same time. It is also convenient to have the **Scene** window open to be able to test your animations without moving the camera around in the Editor. The engine should come with an arrangement helpful for animation work.

Select your **Sprite** GameObject to be active, and press the "**Create**" button on your animation window. You will be prompted to name and save your animation. Name it **BossNameStand.anim** (or your own naming convention) and save it in the same folder as your sprite. Two new objects should have gotten created, the **Animation Clip** with the name of your animation and the **Animator** with the name of your GameObject. Rename the **Animator** to **BossNameAnimator**, or follow your naming convention.



Your **Animation** window should look something like this. Make sure the **Sprite** GameObject is selected again. Click on **Add Property>Sprite Renderer>Sprite (1)**. Do the same with **Add Property>Transform>Position** for future usage. Click on the **arrow (2)** and set the **Samples (3)** to 12. If you need your animation speed to be different, feel free to change the **Samples (3)**. 12 is just a value that seems to work well with this project. You should notice Unity has created 2 **Keyframes (4)** by default. Delete the last one and drag **Sprites (5)** from the **Project** window into the **Animation** window. When your animation is ready, press the **Play (6)** button and test your animation. It may look a bit clunky for now, but just make sure the speed and timing seem fine.

If your standing pose only had one frame, that's alright, just keep one **Keyframe** in your animation and move on to the next animation.

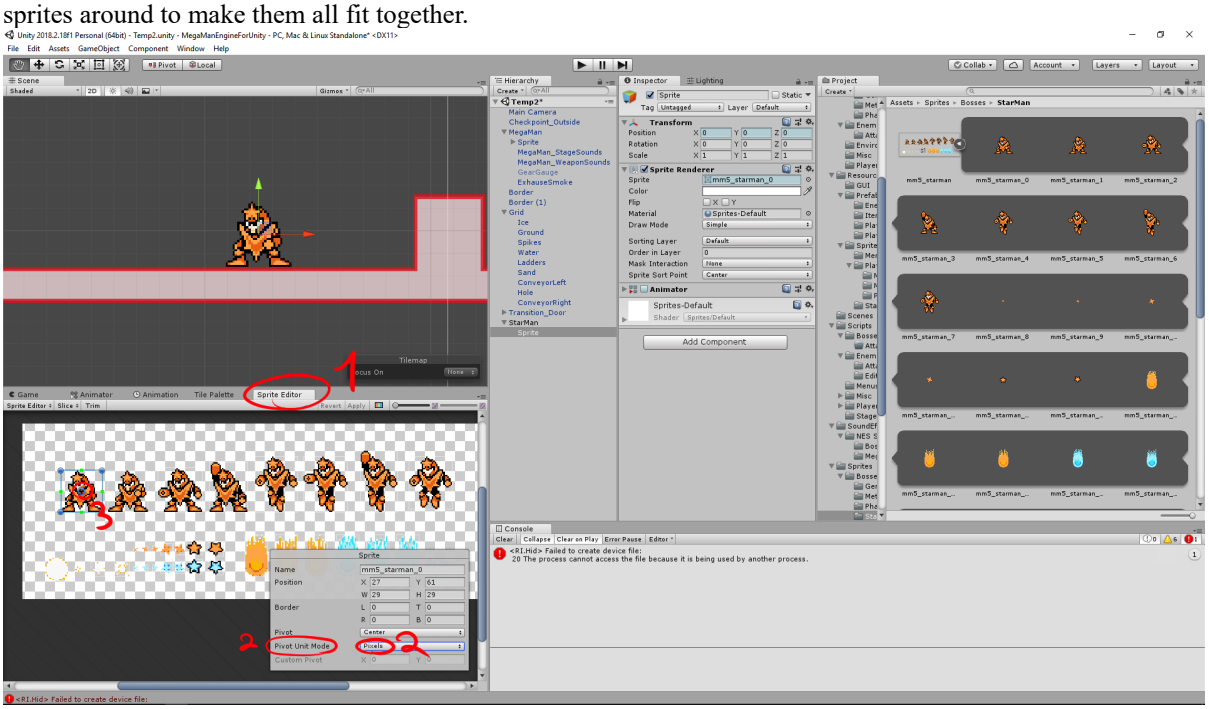
To create a new animation, click on the **Small Button with your current Animation's name (7)** and select **Create New Clip**. Now follow the previous steps again and make all the animations you will need for your boss's behavior. On top of them, create an animation for your boss's intro, and if you desire, one of your boss's death.

The intro animation will include a few extra steps. First of all press the button with the big red circle that's covered by **number 7** in the image above. Move your **Sprite** GameObject a few blocks above to be outside of the in-game camera's view. 260 units should be enough. Now move your current frame to 0:05 by right clicking on the **Timeline (8)** and reset the **Sprite**'s position. Finally, select all **Keyframes (4)** in both positions by dragging your mouse and select them all. Finally right click on the left side and select **Both Tangents>Linear** to make the boss fall smoothly in the area. Test your animation again to make sure they fall smoothly. Now continue after this part as normal. When the game starts, the boss will drop in, play the rest of the animation and then start filling their health

Some animations will need to be looped when they end, some will not. For those that do not need to be looped, click on each animation in the **Project** window and untick the "**Loop Time**" button.

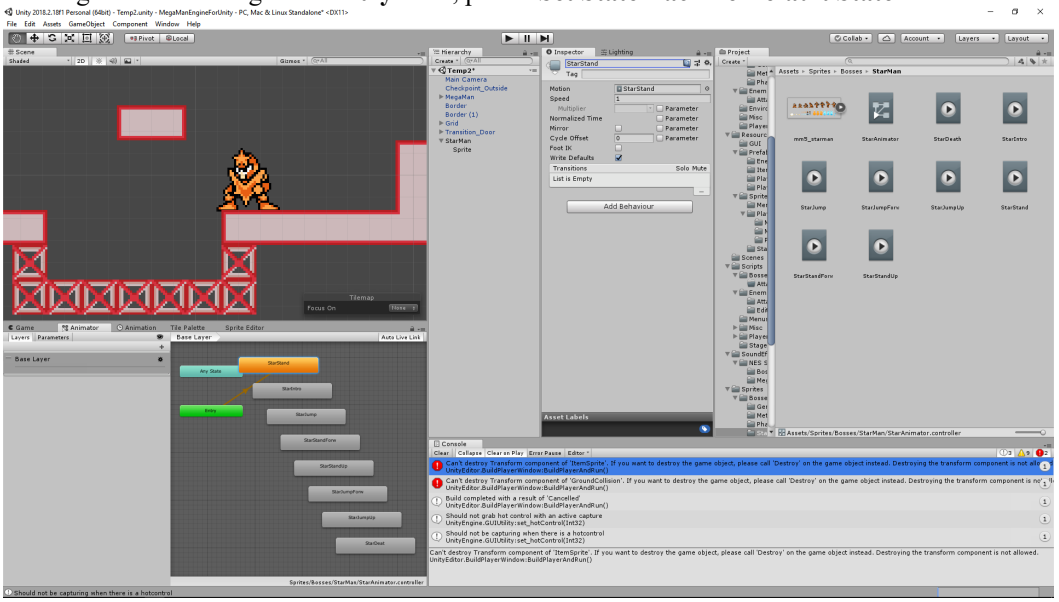
You have likely noticed your animations moving around wobbly. To fix that, you will need to move the **pivot (3)** of your sprites. The pivot of your sprite is the center of it. When Unity shows a sprite, the origin of the sprite will be on the same position as the GameObject that renders it is at.

During animations, you need to make sure that the **pivots** align in a way so that the final product looks smooth. To do that, go to your **Sprite Editor (1)**, set the **Pivot Unit Mode** to **Pixels (2)** and move the **Pivot (3)** of each of your sprites around to make them all fit together.



Finally open the **Animator** window by going to **Window>Animation>Animator**. You should find all the animations you created there. You can make any additions and deletions of leftover clips in the animator, as the **Animator** holds the clips you created earlier in it, but doesn't register changes. You can also simplify the names here as you don't have to worry about name conflicts with other objects of the game. You can also sort them nicely.

Also right click on the green "**Entry**" box, press "**Set StateMachine Default State**" and select the **Intro** animation.



Coding:

It is finally time for coding. Whether you love it or hate it, someone has to do it. You should have already read the fundamentals of how the **Boss** script works. How it's time to implement everything.

First of all, you need to make the script. Go to **Assets>Scripts>Bosses** and create a new **C# Script**. It is a good idea to put a prefix on your script name to signify it is a **Boss**. Double click on it and open it in your favorite IDE.

Your code should look something like the following. Replace the "**MonoBehaviour**" part with "**Boss**" to make your script inherit from the **Boss** script.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Bo_StarMan : MonoBehaviour
{

	// Use this for initialization
	void Start () {

	}

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

	}
}
```

You should remove the two lines that start with "//", as these are comments Unity writes by default. You should put comments in your code to remember what does what, but there is no reason to keep these two lines. Also add the "**protected override**" keywords before the "**void Start()**".

You can also remove the "**System.Collections;**" and "**System.Collections.Generic;**" lines, as these libraries aren't used yet. Some IDEs will show you if you have libraries you don't use in your script. However, you may need them in the future, so if you aren't very familiar with libraries, you can leave them on.

Now that your script inherits from the **Boss** one, you should add it to your boss GameObject. Go to the Unity Editor, select your **Parent** GameObject and click the "**Add Component**" button. Find your script and add it to your boss. It should be added to the GameObject, alongside a **Rigidbody2D** component. There are a few changes you need to make.

On the **Rigidbody2D** Component:

- Set the **Gravity Scale** to **100**.
- Set the **Body Type** to **Kinematic**.
- Open **Constraints** and click the **Freeze Rotation** button called "**Z**".

On your **Boss** Component:

- Assign the object called "**Explosion_Big**" object from the assets to the "**Death Explosion**" field.
- Assign the object called "**Deathplosion**" object from the assets to the "**Death Explosion Boss**" field.
- Assign an appropriate **Bar** sprite for for "**Health Bar Full**" and "**Health Bar Empty**".

Intro:

The boss intro usually plays the boss's intro animation and fills their health when they join. There are however a few more things that need to happen in the background when a boss fight starts. Take Pharaoh Man's intro as a reference.

- If the boss fight has already started, then don't play the intro.

```
if (fightStarted)
    yield break;
```

- Set the boss's variables to make them pre-battle-ready.

```
health = 0;
canBeHit = false;
fightStarted = true;
body.isKinematic = true;
```

- Increment the number of bosses active in the GameManager.

```
GameManager.bossesActive++;
```

- Freeze the player.

```
if (Player.instance != null)
    Player.instance.CanMove(false);
```

- Play the animation from the animator. Rename "Intro" to what your intro animation is called in the animator. (You will get an error on the anim.Play("Intro"); line right now, but ignore it for the time being).

```
anim.gameObject.SetActive(true);
anim.Play("Intro");
yield return new WaitForSeconds(1f);
```

- Slowly and dramatically increase the boss's health from 0 to max(28).

```
for (int i = 0; i < 28; i++)
{
    health++;
    yield return new WaitForSeconds(0.05f);
}
```

- Allow both the boss and playermovement.

```
body.isKinematic = false;

if (Player.instance != null)
    Player.instance.CanMove(true);
```

- Allow the boss to be hit.

```
canBeHit = true;
```

- Start the boss's behavior. This isn't usable yet, so add it but comment it out, or you'll get an error.

```
StartCoroutine(Behavior());
```

Unless your intro has something special, you can likely just copy-paste the code from **Bo_PharaohMan.cs** into your own boss's script. As the **PlayIntro()** method is inherited from the **Boss** script, you should use the **override** keyword to tell Unity this is the same method. Write **public override IEnumerator PlayIntro()** and copy the code from **Bo_PharaohMan.cs**, use the code from above or write your own. No shame in reusing old code.

Components:

For the components to be usable, you will need to assign them to the script yourself. There are multiple ways to do that. In the **Enemy** script the **Rigidbody2D** component is already defined by adding the variable "**protected Rigidbody2D body;**" line to the script. To assign it a value, go to the **Start()** method inside your boss's script. Inside the body of the method, add the line "**body = GetComponent<Rigidbody2D>();**". This will tell Unity to look for the **Rigidbody2D** component inside and only on the **Parent** GameObject, and assign it to the **body** variable.

Now go to the top, inside the class but before any methods. Add the variables "**protected Animator anim;**" and "**protected SpriteRenderer rend;**".

To assign values to these variables, a similar method needs to be called. Since the **Animator** component is not on the **Parent** GameObject, but rather one of its **Children**, the method **GetComponentInChildren** needs to be called instead. Add it to the **Start()** method.

The **Sprite Renderer** component is on the same GameObject as the **Animator** one. Since the **anim** variable has already been assigned, you can search for the component only on that GameObject to find it faster, as well as distinguish between multiple components found in different GameObjects. In this case, you will need to use **GetComponent** from the **anim** variable. To do that, use **anim.GetComponent**. Assign this variable as well.

Add the line "**anim.gameObject.SetActive(false);**" under the two assignments. This will hide the boss's sprite until the fight starts and the intro plays. Otherwise the intro will play before the fight starts, and the boss will briefly appear before the intro plays.

To play audio, you will also need a component called **AudioSource**. Add the variable "**protected AudioSource audio;**" and assign it with "**aud = GetComponent<AudioSource>();**".

To play an **Audio Clip**, define it with the other variables as "**public AudioClip clipName;**". When you want to play the clip, use the line "**Helper.PlaySound(clipName, forcePlay);**". Since only one clip can be played by an **AudioSource** at a time, only if **forcePlay** is true the clip currently playing will be replaced by **clipName**.

The error in the **Intro** method should be fixed now too. You can now try it out. Drag your **Boss** GameObject to a **Transition Door**'s "**Door To Activate**" field and move through the door.

Collision:

If everything was done correctly, here is what should have happened: MegaMan enters the boss door. The boss drops in and plays their intro. They then fill their healthbar to the top. Finally, they drop through the floor. Everything was done correctly, but there is one thing still missing. Most interactive objects in the game will need a **Collider**.

Right click on the **Sprite** GameObject in the **Hierarchy** window and create an **Empty** GameObject. It should be parented to the **Sprite** GameObject. Rename the object to "**Collider**". Click on "**Add Component>Physics 2D> Box Collider 2D**". Now change the **Offset** and **Size** variables to give your character a hitbox appropriate of their size. It shouldn't be too large or too small, to make it fair for the player to hit and be hit by. You should judge this on your own. Be sure the **Size** values are positive. Go to the top right of the **Inspector** window, click on the dropdown menu and set the **Layer** to **9:CollideWithWallOnly**. Now the boss is able stand on the ground, but in typical MegaMan fashion, the player can damage boost through them. Not that they can hit or be hit by the boss. To do that, the boss will also need a **Trigger**.

Click on the **Collider** GameObject on the **Hierarchy** window, and press **Ctrl+D** to duplicate the GameObject. Name the new GameObject to "**Trigger**". Change the layer back to "**0: Default**". Finally, go to the **Box Collider 2D** Component and tick the "**Is Trigger**" field. This should also the player to hit the boss, as well as be hit by them.

Damage:

The boss can technically be hurt at this point, but the damage doesn't work the same way damage in a boss fight should work. Bosses usually have an invisibility time after they are hit, as well as a single image that flashes to display this state.

The invisibility timer has already been defined in the **Boss** script as "**protected float invisTime = 0.0f;**". The hurt sprite will need to be defined in your own boss's script. Go under the variables you defined earlier and add "**public Sprite hurtSprite;**".

To have the flashing sprite overwrite the current animation, the **rend**'s sprite needs to be changed, but inside a specific Unity method called **LateUpdate()**. Without going into detail, this method is triggered in every frame, after Unity has decided which sprite needs to be shown, but before it has passed that information to the renderer.

Rename the already existing **Update()** method to **LateUpdate()**. If it's already used for something, create a new **private void LateUpdate()** under it. Don't forget to add the brackets for its body.

Inside the body, add the following lines:

```
    if (invisTime % 0.2f > 0.07f)
        rend.sprite = hurtSprite;
```

This will decide if the flashing frame should appear when the **invisTime** is active. It gets the **mod** value of the **invisTime**, and depending on its result, it will flash for ~2/3 of the time.

The flashing animation has been set, but once it depends on the **invisTime** lowering. To do that, simply add the following code right under:

```
    if (invisTime > 0.0f)
        invisTime -= Time.deltaTime;
```

Finally, you need to assign an invis frame to the **hurtSprite** variable. Save your progress, go back to the Unity Editor and wait for it to apply the changes. A new field should have appeared for your **hurtSprite**. Click on it and search for the sprite called **Damage_Frame**.

Weaknesses:

This is also a good opportunity to define weaknesses. Damage multipliers are handled in the **public override void Damage(Pl_Weapon weapon)** method. Simply add the following in your boss script:

```
public override void Damage(Pl_Weapon weapon)
{
    float damage = weapon.damage;

    if (shielded)
    {
        weapon.Deflect();
        return;
    }
    Damage(damage, weapon.ignoreInvis);
}
```

Simply check the **Weapon Type** before the **Damage(float damage, bool ignoreInvis)** method is called, and multiply the **damage** variable. For example, this is how the Gemini Laser does double damage to the boss:

```
    if (weapon.weaponType == Pl_Weapon.WeaponTypes.Gemini)
        damage *= 2;
```

Prefab:

We have done some good progress with the boss so far. You have likely saved the scene already a few times at this point. If not, remember to save **frequently**. But either way, it is time to save the GameObject as a **Prefab**. This will allow you to drag & drop your boss wherever you want in whichever scene you want at any point. It will also save your boss's settings and everything beyond the Scene you are currently working on.

Navigate to **Assets>Prefabs>Bosses** and create a new folder. Give it the name of your boss. Now drag the **Parent** GameObject from the **Hierarchy** window to your folder in the **Project** window.

And this is it, your boss is now a **Prefab**! You may notice three new buttons under the **Tag** and **Layer** dropdown menus you saw earlier. You should press "**Apply**" fairly often after making changes to your boss. This is like saving the scene, and will save the changes you've made to your boss. If you don't like the changes you've made and want to go back to the current **Prefab** instead, simply press the "**Revert**" button. You can also use "**Select**" as a shortcut to your prefab in the **Project** window.

(Note: Versions 2018.3 and afterwards have made changes to how prefabs work. It's best to learn the basics by finding better tutorials online.)

Behaviors:

It's time to get creative again, as it's time to work on the boss behaviors.

First of all, the way the boss's behavior works in this engine is simple enough. A coroutine is used that has an infinite loop in it, which decides which attack the boss should use, waits for it to finish, and then starts all over again. Infinite loops can be used with coroutines without an exit condition, as long as there is one or more delays inside the loop. Otherwise, the game and the entire Editor will freeze.

Start by defining the coroutine and giving it a body under the **PlayIntro()** coroutine.

```
public IEnumerator Behavior()
{

}
```

Now add the infinite loop inside the body of the **Behavior()** coroutine.

```
    while (true)
    {

    }
```

If you try to run the game now, it will freeze and you will have to reset the editor. To make sure that doesn't happen, even by mistake after you have fully implemented the boss behavior, add a single frame of delay inside the infinite **while** loop. The keywords are **yield return null;** in case you forgot.

Since you have your coroutine prepared, you should tell your boss to use it as soon as the intro is done playing. To do that, add **StartCoroutine(Behavior());** to the end of your intro coroutine. If you already have it there, but as a comment, just remove the comment parts and let it run. You should get no errors.

(**Important**: There is another way to play a coroutine, if you want to run it from inside another coroutine. Using the keywords **yield return CoroutineName();** will pause the current coroutine and wait until **CoroutineName()** is done to continue. This should be used inside the infinite **while** loop to use an attack, wait for it to finish, then use another attack, and continue forever.)

If everything was done correctly, you should be able to run the game without your game freezing. It should work, but you won't see too much happen yet. To make sure your coroutine works, you should change your boss's animation to the standing pose as soon as the **Behavior()** starts. Reminder, use **anim.Play("Stand");** to change your boss's animation. If your standing animation isn't called **Stand** inside the boss's **Animator**, then give it the right name.

Running the game, what should happen is that the boss heals, then after about half a second, changes to their stand pose.

From this point onwards it's up to you to decide how you are going to program your boss. You can check the boss scripts that already exist to check how everything works. This tutorial will cover the basics regardless.

Let's start by keeping track of the corners corners of the room. This will be done with two **Transforms**, one for each side. Go to the top of the class, and under the variables you added previously add these two lines:

```
public Transform cornerLeft;
public Transform cornerRight;
```

Now go back to the Editor. Create two **empty** GameObjects and give them the names "**CornerLeft**" and "**CornerRight**". Move them around the scene to be approximately at each corner of the boss room. Now drag them each into the right

This should work in this scene, but you will have to add them to every new scene you put your boss in, and this can be tedious and even counter-productive. Furthermore, you can't have multiple GameObjects share on prefab on their own. To do that, you will need to parent all GameObjects to one **empty** parent object, and then replace the boss prefab you made earlier with this new one.

Focus on your boss by clicking their name in the **Hierarchy** window and press the **F** button to focus the camera on them. Now right click on an empty spot in the **Hierarchy** window and select **Create Empty**. In case the new object isn't in the same position as your boss, copy-paste the position from the boss's **Transform** component into the **Empty** GameObject's.

Rename the GameObject to **BossNameParent** and drag the boss GameObject from the **Hierarchy** window into it. It should now be parented to the new GameObject. Do the same for the two corners.

To replace the prefab with the new **Parent** GameObject, navigate to your boss's location in the **Project** window. Reminder, you can use the **Select** button. Click on your boss's prefab and delete it. Now click on the **Parent** GameObject's name and drag it to the old Prefab's old place. This should make a new prefab that contains your boss and the two corners.

Jump Across Room:

Now that the boss keeps track of the room's corners, let's make a coroutine that jumps to either corner. The position where the boss jumps on can be anything, and by changing the **jumpPos** variable we will define soon, you can make the boss jump in any position using the same script.

Start by defining the coroutine under the already defined **Behavior()**:

```
public IEnumerator JumpAcrossRoom(bool leftSide)
{

}
```

Now the coroutine is defined, and it has one parameter called **leftSide**. If true, the boss will jump to the **CornerLeft** transform. Otherwise, it will jump towards the **CornerRight** one.

Start by defining the variable **jumpPos**, which is the position the boss should jump at. Check if **leftSide** is true, and assign **jumpPos** to have either the **cornerLeft**'s position or **cornerRight**'s position as a value.

```
Vector3 jumpPos;

if (leftSide)
    jumpPos = cornerLeft.position;
else
    jumpPos = cornerRight.position;
```

Normally to get an exact jump at a specified position, you will need to do some fancy coding and know how to use physics. Thankfully this has already been figured out, so you don't have to.

The engine comes with a class called **Helper**. This class holds many simple methods that many different classes may need to use. In this case, the method **LaunchVelocity(Vector3 origin, Vector3 target, float angle, float gravity)** can be used.
(Note: I do not take credit for the creation of this script, it was found after searching for some time. I have only done minor changes. Unfortunately I can't find a source to give proper credit. It was found in a site like the Unity forums, as an answer to a question. If someone knows the source, let me know to give credit to the original thread.)

To make the boss jump, you need to set the **Rigidbody2D**'s velocity with **LaunchVelocity()** with the following line:

```
body.velocity = Helper.LaunchVelocity(transform.position, jumpPos, 50.0f, 1000f);
```

This will launch the boss towards the target position with a 50 degree angle. You can change the angle (1-89) as you see fit.

With the following lines your boss should look towards the position they're jumping towards, as well as play the jump animation:

```
anim.Play("Jump");
if (jumpPos.x < transform.position.x)
    transform.localScale = new Vector3(1, 1, 1);
else
    transform.localScale = new Vector3(-1, 1, 1);
```

If your sprite sheet faces right, replace the second line with the following:

```
if (jumpPos.x > transform.position.x)
```

Add a 4 second delay after this line temporarily. This will allow you to see the jumps happen and give the boss time to jump around. Try to remember the code for a 4 second delay.

Finally, go back to the **Behavior()** coroutine, go inside the infinite **while** loop and under the frame delay add the following:

```
        yield return JumpAcrossRoom(true);
        yield return JumpAcrossRoom(false);
```

If everything was done correctly, the boss should jump across the room, stick to a corner in their jump animation, then jump back, and keep doing that forever. The boss needs to have a way of checking when they hit the ground.

First of all the boss needs to keep track of its collider. Since there are multiple colliders, you need to assign the solid collider from the Editor itself.

Add the line **public BoxCollider2D col;** under the **Animator** and **SpriteRenderer** definitions. Now go to the Editor, find the **Collider** GameObject in the **Hierarchy** window and drag it into the **Col** field.

Go back to your script and, at the very bottom of the class, add this:

```
  private bool isGrounded
  {
    get
    {
      return body.velocity.y <= 0 &&
        Physics2D.OverlapBox((Vector2)transform.position + col.offset - (Vector2)transform.up, new
Vector2(col.size.x * 0.9f, col.size.y), 0, 1 << 8);
    }
  }
```

This should return **true** if the boss is touching the ground and **false** if they aren't.

Replace the delay **yield return new WaitForSeconds(4f);** with the following:

```
    while (!isGrounded)
      yield return null;
```

This should give an one frame delay every frame until the condition is met. When the condition is met, it will continue. You can try the boss fight again, but now the boss should be zooming across the room without a second thought. After a jump is finished, the boss should rest for half a second after they hit the ground. They should also play the standing animation again. Try to change the animation and add the delay on your own.

Shoot a Projectile:

There is a simple code in the **Enemy** script that will allow the boss to shoot either a default bullet or a custom one. The **Shoot()** method will shoot a default bullet, unless otherwise specified, at the desired direction and speed, with the desired damage.

Make a new coroutine and call it **public IEnumerator ShootAtDir(Vector2 direction)**. Change the animation to a shooting one and add a delay. If your boss has both an aerial and ground shooting animation, you can use **isGrounded** to change to the proper animation.

Example:

```
if (isGrounded)
    anim.Play("StandShoot");
else
    anim.Play("JumpShoot");
```

You can freeze your boss mid-air by changing their **bodyType** to **Kinematic**. You should also set the **Rigidbody2D**'s **velocity** to zero, otherwise, once you reset the **bodyType** to **Dynamic**, the boss will be launched.

Example:

```
body.bodyType = RigidbodyType2D.Kinematic;
body.velocity = Vector2.zero;
```

Also flip the boss's sprite with the following code, flipping the x position's 1s if needed.

```
transform.localScale = new Vector3(direction.x > 0 ? -1 : 1, 1, 1);
```

Give the boss a small delay after they change their sprite, to give the player time to react. Say 1 second, for example.

You can now use the **Shoot()** method to shoot towards the player. Here are the parameters:

- **shootPosition:** Will spawn the bullet on the specified position. To shoot one from the center of the boss, set this to **transform.position**. To further move the shot towards the position of the buster, set it to **transform.position + Vector2(x, y)**, with **x** and **y** being values you find on your own. And finally, to make the position flip based on the direction the boss is shooting at, it should be like **transform.position + Vector2(x * transform.localScale.x, y)**, or **transform.position + Vector2(x * -transform.localScale.x, y)**
- **shootDirection:** The bullet will move and look towards the direction defined.
- **shotDamage:** The damage the bullet will do to the player.
- **shotSpeed:** The speed the bullet will travel with. Default is 200.
- **obj:** The custom bullet to use. If this is set to **null** or not included, then a default bullet will be shot.

After the delay, put the following code in, adjusting the position to fit your boss's sprite.

```
Shoot(transform.position + new Vector3(10 * -transform.localScale.x, 0), direction, 2, 200, null);
```

Now add a little extra delay, like half a second. You can also change the boss's animation back to the standing one.

```
if (isGrounded)
    anim.Play("Stand");
else
    anim.Play("Jump");
```

Finally reset the **Rigidbody2D** to allow the boss to move again.

```
body.bodyType = RigidbodyType2D.Dynamic;
```

To make the boss shoot, simply go to **Behavior()** and add the line:

```
yield return ShootAtDir(direction);
```

The direction should be a Vector2, but you can set it to any Vector2 you want. If you aren't familiar with how Vectors can be used as directions, then this might help.

- **new Vector2(transform.localScale.x, 0)**: This will make the boss shoot in front of them, regardless of the direction they are facing. If their sprite originally faces left, then "**-transform.localScale.x**" should be used instead.
- **new Vector2(1, 0)** and **new Vector2(-1, 0)**: These will shoot to the right and left perspectively, regardless of the boss's orientation.
- **new Vector2(0, 1)**: This will make the shot go upwards.
- **new Vector2(2 * transform.localScale.x, 1)**: The shot will follow the boss's orientation, with a ~35 degree angle upwards.
- **new Vector2(Quaternion.AngleAxis(45, Vector3.forward) * Vector2.right)**: Using the **Quaternion** class, Vectors can be rotation. In this case, the **Vector2.right** vector will be rotated 45 degrees counter-clockwise. Orientation will be ignored, and the shot will move to the right and upward. Vector3.forward needs to be used to rotate a **Vector2**.
- **new Vector2(Quaternion.AngleAxis(45, Vector3.forward) * Vector2.left)**: Again, this will rotate the **Vector2.left** vector 45 degrees counter-clockwise. This time the bullet will move to the left and downward.
- **new Vector2(Quaternion.AngleAxis(45, Vector3.forward) * Vector2.right * transform.localScale.x)**: This will change the starting vector, and the bullet will move either to the right and up or left and down, depending on the boss's orientation.
- **new Vector2(Quaternion.AngleAxis(45 * transform.localScale.x, Vector3.forward) * Vector2.right * transform.localScale.x)**: Multiplying the angle as well as the right Vector will make the bullet move upwards regardless of orientation.
- **new Vector2(Quaternion.AngleAxis(45 * transform.localScale.x, Vector3.forward) * Vector2.Scale(Vector2.right, transform.localScale))**: Finally, using **Vector2.Scale** any Vector can be made to face the same way the boss does, and using **transform.localScale.x** the angle can also be changed accordingly.

Assuming you have two points on the stage, two positions, named posA and posB. If you want to travel from posA to posB, then the direction you need to follow will be **posB - posA**. To get the player's position, **GameManager.playerPosition** can be used. To shoot a bullet towards the player, **GameManager.playerPosition - transform.position** can be used.

Back to the **Behavior()**, you can make your boss shoot after each jump. To make them jump, then shoot at the player, then jump back and shoot again, your code should look something like this:

```
yield return JumpAcrossRoom(true);
yield return ShootAtDir(GameManager.playerPosition - transform.position);
yield return JumpAcrossRoom(false);
yield return ShootAtDir(GameManager.playerPosition - transform.position);
```

Custom Bullets:

So far your boss can shoot default bullets. Making your custom ones can be fairly simple.

Let's start with a reskin of the default bullet. Start by creating a new Sprite GameObject from the **Hierarchy** window (**Right Click>2D Object>Sprite**). Give your bullet a name. Assign a sprite in the **Sprite Renderer**. If your bullet is animated, this will only serve as a preview. Otherwise, it will be its actual sprite.

Click the **Add Component** button and find the **EnWp_Shot** script. It can be found easily by using the search bar. This should add a **Rigidbody2D** and a **EnWp_Shot** component to the GameObject.

Set the **GravityScale** to 0 in the **Rigidbody2D** component.
In the **EnWp_Shot** component there are a few different things to play attention to. First of all, it derives from the **Enemy** script, just like the **Boss** one. For that reason, the **health, canBeHit, shielded** and **center** variables should be ignored. Just make sure to set the **canBeHit** variable to false, and leave everything else as it is.

On the variables you should pay attention to:

- **Damage**: This is the damage the bullet will do to the player upon contact. Will be changed if called by the **Shoot()** method.
- **Destroy Outside Camera**: This decides if the bullet will be destroyed when it exits the camera's view. It should be **true** to avoid bullets from lingering in the scene long for too long, but if for some reason you want the bullet to stay active after exiting the camera view, then the **DestroyAfterTime** component should be added, to make sure the bullet gets destroyed after a while. The **DestroyOnWall** variable can also be used, as long as it's certain that the bullet will hit a wall in the future.
- **Center**: The center of the bullet. If a **DeathExplosion** object is set, it will spawn from the position set by the **Center** variable.
- **DeathExplosion**: This is the object that will be created once the bullet hits something. In the assets, you can find and use the prefabs **Explosion_Small, Explosion_Big** and **Explosion_Big_Damaging**. The last one will explode and do damage to the player, the other two are just for better visuals.
- **Direction**: The direction the bullet will travel towards. Again, can be changed by the **Shoot()** method.
- **Speed**: The default speed of the bullet. Last one that will be changed by the **Shoot()** method.
- **DestroyOnWall**: If this is **true**, then the bullet will be destroyed once it hits a wall. For that to happen, you will need to give the bullet a child GameObject with a **Collider** that is not marked as **trigger**. It's layer should also be changed to **9:ColliderWithWallOnly**
- **xFrameSpeed**: The speed of the animation. If your bullet isn't animated, feel free to ignore it.
- **Sprites:** If you want your bullet to be animated, there is no reason to make a new **Animator** and **AnimationClip**. Simply set the **Size** of this array to the number of frames your animation has, and assign the frames in order in this array.

There is one last component that needs to be added, if you want your bullet to collide with the player. Go to **Add Component** and navigate to **Physics 2D> Box Collider**. Unity should automatically set the size of the collider to fit the sprite of the **Sprite Renderer**. Modify the **size** and **offset** accordingly, and set **isTrigger** to **true**.

Finally, in the **Project** window, navigate to the folder your boss is in, and drag the bullet from the **Hierarchy** window into the **Project** one. This should turn the bullet into a prefab. Delete the one in the **Scene**.

To make the boss shoot your bullet, go back to the script and define a new variable under the **hurtSprite**. Call it **public GameObject bulletName;**. Now go to the **ShootAtDir()** coroutine, and find the **Shoot()** method. The last variable should be set to **null**. Replace it with the name of your bullet. This should now replace the default bullet with the **EnWp_Shot** you made.

Make sure you assigned the prefab to your boss in the Editor and applied. Try the game out.

There is one more thing you can do to expand the usage of the **ShootAtDir()** coroutine. Go to the parameters, and after **Vector2 direction**, add the variable **GameObject** shot.

```
public IEnumerator ShootAtDir(Vector2 direction, GameObject shot)
```

Go to the **Shoot()** method in the coroutine again, and once again change the last variable. Replace your bullet with **shot**. Now go to the **Behavior()** coroutine, and add your bullet where you called the **ShootAtDir(dir, shot)** coroutine.

Example:

```
        yield return JumpAcrossRoom(true);
        yield return ShootAtDir(GameManager.playerPosition - transform.position, bulletName);
        yield return JumpAcrossRoom(false);
        yield return ShootAtDir(GameManager.playerPosition - transform.position, bulletName);
```

You can also define and use multiple bullets this way, just by using different names with **ShootAtDir()**. Furthermore, you can do the same with other variables, like the speed and damage.

Non-Bullet Attacks:

Now this is the part where you let your creativity flow. At this point experience with Unity helps a lot. This tutorial will go through the basics. Not much detail will be given either. You can check the **Shoot()** method inside the **Enemy** script to learn how to spawn attacks and access their components, if you need help with either.

Star Shield:

For the Star Shield, an **Enemy** script will be added instead of an **EnWp_Shot**. **Shielded** has been set to true, **canBeHit** has been set to true and a **CircleCollider2D** has been added.

The Star Shield would normally be parented to Star Man, until he decides to shoot it. However, we run into a problem when we want to parent a Rigidbody into another Rigidbody. For that reason, the engine comes with a script called **Misc_FollowTransform**. This component is also added to the Star Shield, and a **target** will be assigned through script by Star Man.

The Star Shield has been saved as a **prefab**. Now to the script. Star Man will make a shield, then jump around a few times, and at the peak of the final jump shoot the shield towards the player.

The coroutine **StarShield** handles all this. The number of jumps can be adjusted, and has been given as a parameter.

To create the shield, it needs to be **Instantiated**. That means a copy of the shield will be created into the scene. To also place it over Star Man, its **transform.position** needs to be set. Furthermore, it is set to follow Star Man, and is given a rotation, which won't slow down because the **Rigidbody2D**'s **anglularDrag** has been set to 0.

With the shield up, Star Man will jump 4 blocks towards the player, like when he jumped towards either corner of the room. A small delay is also given after each jump.

Once it's time for the last jump, Star Man needs to reach the peak of their jump. Since **LaunchVelocity** will give the boss a positive **velocity.y**, while the boss is still going up, it will remain positive. Once that value is no longer positive, the boss will start falling. At that point, Star Man freezes and shoots the shield. Afterwards, he waits to fall.

Shooting Star:

Star Man's abilities have been expanded for demonstrative purposes. Once of his new moves is the shooting star. He Star Man will shoot a few stars upwards, and then they will drop back down on the player's position.

First of all, the **Shooting Star** itself. While the **Enemy** script on its own lacks some of the functions needed, **EnWp_Shot** has everything it needs. With the direction and speed set, the meteor will automatically move downwards as soon as it spawns. All that needs to be done now is spawn at the right place.

The coroutine **ShootingStar()** will make Star Man shoot a few bullets up. Every third bullet will be an ice one. The **Shoot()** method is all that's needed to shoot the stars upwards. To bring them back down, however, another coroutine will be used. The stars serve as an extra challenge, and will rain down while Star Man does his regular patterns. The **DropStars()** coroutine will run alongside everything else. To not interrupt the **ShootingStar()** or **Behavior()** coroutines already running, **StartCoroutine()** will be used again.

Finally, to spawn the meteors, Star Man simply instantiates them and places them 240 blocks above the player. The **EnWp_Shot** script handles everything else. A little camera shake is also added.

Behavior:

In the **Behavior()** coroutine so far you've probably managed to play a few different attacks in a row.

To add some randomness to the behavior, **Random.Range()** can be used. Assuming you want an attack to have a 3/4 chance of playing, the following code can be used:

```
if (Random.Range(0,4) < 3)
    yield return Attack();
```

To play an attack only after some health has been depleted, use:

```
if (health < 16)
    yield return Attack();
```

Death:

Everything should work as expected now. Only problem, the player will stay in the stage after the fight, and will even be able to move. On top of that, the doors are working again. If this was a mini-boss, this would be all you need to do. Stage bosses, however, need a few extra things.

When the boss dies, the method **Kill()** will be called:

```
public override void Kill(bool makeItem)
{

}
```

When the boss dies, first of all, they need to stop moving around. Since this is done through coroutines, and coroutines only handle the boss's behavior, simply calling **StopAllCoroutines()** will do.

Afterwards, you need to make a new coroutine that handles the boss's death animation. In theory you could cram everything in a single method, but that doesn't give you the satisfying boss death that all MegaMan games show.

```
public IEnumerator PlayDeathShort() { }
```

The first thing that needs to happen is to check if the boss actually died. If this coroutine was called by accident before the boss fight, then it should be canceled. If the **fightStarted** variable is **false**, then **yield break;** should be used to stop exit the coroutine.

```
    if (!fightStarted)
        yield break;
```

Once it is certain that the boss died, we need to tell the **GameManager** that the boss is dead. The **GameManager** is a static class that handles everything game-related, like which bosses the player has beaten, which checkpoint is active and what items the player has. You likely noticed that class earlier when we needed to access the player's position.

When a boss fight starts, a variable called **bossesActive** in the **GameManager** is increased by one. When the boss dies, that variable needs to go back down to 0. To do that, simply use **GameManager.bossesActive--;**.

Another thing that the **GameManager** needs to be updated on is the boss that just died. You will need to define a new variable to hold whether your boss is dead or not. To do that, go to the **GameManager**. It can be found in **Assets>Scripts>Misc>GameManager**.

Under the **// Boss related variables** field you should be able to see all the related variables. Add your own, which should look like this:

```
public static bool bossDead_BossName;
```

You should also set that to false in the class's **constructor**. Find the method called **static GameManager()**, and search for the other **bossDead** variables. Add **bossDead_BossName = false;** with the rest.

Back to the boss's death coroutine, don't forget to set the variable to true.

To make the boss explode, you need to create an explosion object in the scene. Since the variable **deathExplosionBoss** is already defined in the Editor, and it only needs to be used once, we can instantiate it with the following line:

```
deathExplosionBoss = Instantiate(deathExplosionBoss);
```

If that confuses you, just use **GameObject exp = Instantiate(deathExplosionBoss);**. It's the same thing. Also set its position to the boss's position.

After that, hide the GameObject containing the sprite. We still need the boss script to be active because of the coroutine, but the boss shouldn't appear. To hide the sprite, simply use **rend.gameObject.SetActive(false);**.

The entire code should look like this:

```
    deathExplosionBoss = Instantiate(deathExplosionBoss);
    deathExplosionBoss.transform.position = transform.position;
    rend.gameObject.SetActive(false);
```

At the same time, the player needs to stay on the alive after their victory. To do that, use the following, checking whether a player exists or not:

```
    if (Player.instance != null)
      Player.instance.canBeHurt = false;
```

Give the coroutine a small delay, to allow the player move around on for a few seconds.

Now freeze the game and the player.

```
    Time.timeScale = 0.0f;
    if (Player.instance != null)
    {
      Player.instance.CanMove(false);
      Player.instance.SetGear(false, false);
      Player.instance.RefreshWeaponList();
    }
```

Finally, after a smaller delay, tell the player to play their outro animation. Everything else is handled by the player.

```
Player.instance.Outro();
```

If you want the player to keep moving around after this, in case there are multiple bosses around, for example, use the following instead:

```
    if (Player.instance != null)
    {
      if (GameManager.bossesActive > 0)
      {
        Player.instance.CanMove(true);
        Player.instance.canBeHurt = true;
        Time.timeScale = 1.0f;
      }
      else
      {
        Player.instance.Outro();
      }
    }
```

Now the boss should play a classic death animation. For the more fancy animation you may have seen, simply copy it from **Bo_PharaohMan.cs** and change which boss gets killed in the **GameManager**.

Stage Select:

One last thing you may want to do is make your stage selectable in the stage select. To get to the stage select script, navigate to **Assets>Scripts>Menus** and open the **Menu_StageSelect** script.

In the variable definitions, look for the **"Main Stage Select Elements"** header. You should see some variables called **public SpriteRenderer bossnameIcon;**. Under them add your own.

Open the scene called **StageSelect.** This contains everything Go to the inspector, find the spot where you want your boss to be at. The GameObject called "**StageSelect**" should contain the **SpriteRenderers** for all the 8 boss icons.

Find the one you want and replace the sprite with your own boss's.

Now go back to the script. Find a method called **public void ChangeRooms()**. At the very top, inside a **switch** statement, there should a section that contains statements like this:

```
if (GameManager.bossDead_bossName)
    bossIcon.enabled = false;
```

Add your own for the **bossDead_BossName** and **bossnameIcon** you created earlier.

Remember the position you assigned your icon on in the Editor. The names are of the format **IconXY**. For example, an icon with the name **Icon20** would be considered to be on the position **X = 2, Y = 0**.

Find the **GoToSelectedScene()** method. This should check the position of the cursor and go to the right stage. The cursor is a **Vector2** called **stageIndex**. To check its position and go to the right stage, you can use the following code:

```
if (stageIndex.x == X && stageIndex.y == Y)
{
    Helper.GoToStage("StageName");
    return true;
}
```

Simply replace the capital **X** and **Y** with the position you selected and **StageName** with the stage you created earlier. Make sure the stage is added to the build. To do that, simply open your stage, go to **File>Build Settings** from the top menu and press the **Add Open Scenes** button in the little window that pops up.