# MegaMan Engine for Unity

Made by MegaChibisX

## Enemy Creation

If you looked at the **Boss** Tutorial, you must have noticed that the **Boss** script inherits from the **Enemy** script. As a result, a few variables and methods are shared between the two scripts. Quick reminder of the basic stuff:

**Variables**:

- **health**: It keeps track of the health of the enemy.
- **damage**: The contact damage the enemy does on the player.
- **canBeHit**: Will be set to false when the enemy needs to be invincible. Not used by most enemies, as they lack invincibility frames.
- **shielded**: To not be confused with **canBeHit**, this should be used when the enemy can't be hit due to having a barrier of some sort active. Bullets will be deflected off the enemy.
- **destroyOutsideCamera**: This makes an enemy despawn when they are off-screen. Should be disabled for bosses, mid-bosses and enemies that aren't supposed to respawn when killed, or need to be placed outside of  the camera's view for one reason or another.
- **center**: This is used to set the sprite center of the enemy. 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 enemy dies.
- **body**: The **Rigidbody2D** component of the enemy, which all enemies need to have.

**Methods**:

- **Start()**: It is called right when the enemy appears in the scene. For enemies that respawn, it will be called when they get into the camera's view. For other enemies, it will be called as soon as the scene, and likely the stage, starts.
- **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.
- **Shoot(...)**: Will create a normal shot (unless otherwise specified).
- **LookAtPlayer()**: Looks left or right depending on the player's position.
- **GetClosestPlayer()**: Returns the player.
- **ChangeColorScheme(Color[] colors)**: Will change the enemy's color scheme, if possible.

**Adding an enemy to the scene:**

Adding an enemy to the scene is very simple. Simple navigate to “**Assets>Prefabs>Enemies**” and drag any of the already existing enemies into the scene. You can also make an empty **GameObject** and parent all enemies to it, to keep your scene more tidy.

Enemies added into the scene this way won’t respawn when they are killed. Furthermore, if the **DestroyOutsideCamera** variable is **true**, the enemy will be destroyed as soon as they are outside of camera’s view. If they spawn outside the camera, they won’t appear at all. For that reason, enemies need to be **spawned** into the game window using another method.

**Spawning:**

There is one script that goes hand-in-hand with the **Enemy** script, called the **EnemyBlueprint** script. This is what allows an enemy to spawn and respawn when they get inside the camera’s view, assuming the enemy isn’t already active in the room.

Create an empty **GameObject** in your scene and add the **_Blueprint** component (**MegaMan>Enemy>_Blueprint**). Drag an enemy from the **Project** window into the **Blueprint** field and press the “**Recalculate Bounds**” button.

Once this is done, the **BluePrint** should show a sprite of the enemy you selected, a small green box and a bigger cyan box. The green box shows the dimensions of the enemy, and the cyan one shows the area that, if the player enters, will make the enemy spawn. That can be changed by modifying the **Center** and **Size** boxes, but they will be reset if the “**Recalculate Bounds**” button is pressed again.

Some enemies come in many different colors. To change the colors of these enemies, simply give the **New Colors** array the new colors you want to assign. Try it out with the **JoeNorm** enemy
(Note: This only works with specific enemies. To allow your enemy to be recolored, you need to override the **ChangeColorScheme** method. It will be explored later in the tutorial.)

# Making a new enemy

The types of enemies you can add in the game are practically endless, it is up to you to decide what enemies you are going to make and add in your project. In this tutorial we will simply make a few enemies that already exist in the games, and add them to the project.

By this point you should be able to add your own sprites and animations to the project, and you should have a fair understanding of some basic components of the game. If not, look back to the **Boss** tutorial.

**As other tutorials have already covered the important parts already, from now on we will pick up the pace.**

You can find enemy-related folders under these paths:
**Sprites**: **Assets>Sprites>Enemies**
**Prefabs**: **Assets>Prefabs>Enemies**
**Scripts**: **Assets>Scripts>Enemies**

**Skullmet**:

For the Skullmet 3 animations will be made: [**Idle**, **Blink**, **Shoot**].

We will use an animator, so making a new folder will be useful to keep everything tidy. Selecting the frames of the first animation and dragging them into the **Scene** view, this prompts Unity to make a new animator, as well as the first animation. Both are placed in the new folder. The next animations should also be made now.

With the animations made, an enemy **GameObject** needs to be created. Start by creating an empty **GameObject**. Now drag the **Sprite** object into the empty one and name the parent appropriately.

Collision should also be added. Make a new **GameObject**, make it a child of the enemy, then add a **BoxCollider2D** component. Change the **size** and **offset** appropriately. Name it "**Solid**", then duplicate it and call the duplicate "**Trigger**".

The player should be able to move through the enemy, but get damaged. For that reason, a **Trigger** collider will need to be used. With a **trigger**, however, the enemy won't be able to interact with walls and the ground if it needs to move around. For that reason, a **solid** collider will also need to be added.

Set the **trigger** GameObject's "**Is Trigger**" variable to **true.** Set the layer of the **solid** GameObject to "**9: ColliderWithWallOnly**". Everything's set.

To make the **Script**, go to the **Scripts** folder  and make a new **Script**. **Enemy** scripts use the prefix “**En_**”, which helps with categorization.

Open the script and replace the **Monobehaviour** with **Enemy**. Also empty the body of the class. The script should look like this:

```
public class En_Skullmet : Enemy
{

}
```

The behavior of the **Skullmet**  will be very simple. It simply waits for a few seconds, blinks, then shoots at the player. An **IEnumerator** will be used.

```
public IEnumerator Behavior()
{
   while (true)
   {
      yield return null;

      anim.Play("Idle");
      yield return new WaitForSeconds(3.0f);

      anim.Play("Blink");
      yield return new WaitForSeconds(0.25f);

      anim.Play("Shoot");
      yield return new WaitForSeconds(0.125f);

         Vector3 shotOrigin = transform.position + transform.right * (transform.localScale.x > 0 ? -1 : 1) * 11 +
transform.up * 11;
         Vector3 vel = Helper.LaunchVelocity(shotOrigin, GameManager.playerPosition, 45, 500);

         GameObject shot = Shoot(shotOrigin, vel.normalized, 2, vel.magnitude);
         shot.GetComponent<Rigidbody2D>().gravityScale = 50;
   }
}
```

Override the **Start()** method from your **Enemy** script, and use **StartCoroutine(Behavior());**

Drag the script into the **Parent** GameObject, set the **Gravity Scale** to **100** and the **Freeze Rotation** constraint to **true**.

Set the **Death Explosion** to the **Explosion_Small** asset.

Drag the enemy **Parent** into the **Prefabs** folder. You should be able to use the enemy with a **Blueprint** now.

**Totem Polen**:

A Totem Polen won't need to use an animator. Instead, each phase will simply change the sprite to use through script, and no animations will need to be used.

Make a new empty **GameObject** and give it your enemy's name. This will be the **Parent** object. Drag the idle sprite into the **parent**, in the inspector window.

For collision, create another empty **GameObject**, and again parent it to the **Parent** object. Name it "**Solid**". Add a **BoxCollider2D** component. Normally the above process would be followed again. However, this enemy will instead prevent the player from moving through them. Simply add an **Enemy** script for now, and set the **Body Type** to **Kinematic**.

Remove the generic **Enemy** script and make a new one for your enemy.

**Changing Sprites:**

To switch between sprites, different variables will be needed. First of all, a **Sprite Renderer** needs to be referenced from the script. Second, different **Sprite** variables are needed, each referencing one of the possible sprites your enemy can have.

To assign the **SpriteRenderer** (called **rend** in this case), you need to use **GetComponentInChildren<SpriteRenderer>()** as soon as the enemy spawns. Don't forget to override the method from the **Enemy** script.

```
protected override void Start()
{
    base.Start();
    rend = GetComponentInChildren<SpriteRenderer>();
}
```

Make a new **Coroutine** for your enemy's behavior.

Inside an infinite loop, start by setting the **Sprite Renderer**'s **sprite** to the idle sprite. This is done with the line:
**rend.sprrite = spriteIdle;**

Don't forget to use **yield return new WaitForSeconds();** or **yield return null;**, otherwise an accidental infinite loop will freeze the Editor.

You can flip the enemy to face the player with the following code:

```
transform.localScale = new Vector3(GameManager.playerPosition.x > transform.position.x ? -1 : 1, 1, 1);
```

(Flip the comparison symbol if your sprite faces right).

Another **Couroutine** will be made that will allow the enemy to shoot. A variable will be used to decide how far down the Totem Polen will shoot from.

```
public IEnumerator Shoot(int head)
{
    if (head <= 0 || head > 4)
        yield break;

    yield return null;


    rend.sprite = spriteShoot[head - 1];

    yield return new WaitForSeconds(0.125f);

    Vector3 shootDir = transform.right * (transform.localScale.x > 0 ? -1 : 1);
    Shoot(transform.position + transform.up * 42f - transform.up * 16f * head, shootDir, 2, 200);

    yield return new WaitForSeconds(0.375f);
}
```

To  run one **Coroutine** from inside another, pausing the first one until the second is completed simply use
**yield  return Shoot();**

To shoot from a random head, **Random.Range()** will be used. For a random integer, use **Random.Range(x, y);**, which will return a number between **x** and (**y - 1**). (Ex: Random.Range(1,5) with return a number between 1 and 4).

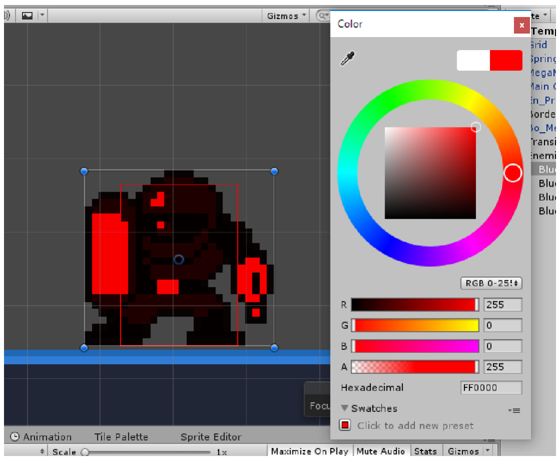
**Shield Attacker**:

The shield attackers should be very simple to make. They will use two animations, one for moving and one for turning, as well as a coroutine for turning.

The important thing with this enemy is that they will be able to change colors. A few enemies, like the Sniper Joes, can use multiple different colors with their sprites. MegaMan games usually do that in different stages, to make enemies fit different environments better and give the stage more variety.

Start by making your enemy. It doesn't matter how or what enemy you are making, go nuts. You can also edit one of the enemies above, or one you just made yourself. Just be aware you will need some space in their sprite sheet to edit it a little.

To change the color of a sprite in Unity, simply go to the **Sprite Renderer** of a GameObject and change the **Color** variable. Doing that, however, you may notice the sprite looking a little weird, like the one below.



Unity simply multiplies the color of the Sprite with the **Color** variable, and that doesn't work well with colors other than white. It also colors the entire sprite, and not just one color. For these reasons, a simple solution is to use another **Sprite Renderer** and draw the parts of the sprite we want to recolor above the main **Renderer**. This might not be the best solution out there, but it is very simple to do, and should be fairly effective for simple sprites like thes.

Making the Recolor Sprite is very simple:

- Go to your favorite image editor and open your enemy sprite
- Duplicate your enemy, and drag it into an empty space in the sheet. Preferably, place them near the original enemy, and only move them either horizontally or vertically.
- Keep track of the displacement, aka how many pixels you moved the duplicate in both directions.
- Erase all the colors of your sprite, minus the one you want to be recolored. If you have multiple colors, this process needs to be repeated for every single color.
- Recolor the entire thing to white.

Save and go back to the Unity Editor. Now you need to add these sprites in the game, so they can be used by your enemy script.

Go to the **Sprite Editor** window. For every single frame of your enemy, select it in the Sprite Editor and duplicate it (use the **Ctrl+D** shortcut). Now simply move it over the white-out piece you made before. They should all be shifted as many pixels as you moved them in your editor, so changing the **x** and **y positions** should suffice. As the white sprites can be hard to see, moving the rectangle manually can be hard. However, the Sprite Editor does have the ability to change the preview from RGB to Alpha, which can help if you do go that way.

After all the sprites have been duplicated and moved, press **Apply**.

To change the color of the sprite, first of  all, you are going to need a new **Sprite Renderer** displayed on top of your already existing one. It should also be parented to that **Source** Sprite Renderer. Select the **source** sprite renderer on the Hierarchy Window, right click on it and create an **Empty** GameObject, which should be parented to the Sprite Renderer. Rename the Empty GameObject to "**Color 1**", and add a **Sprite Renderer**  component to it. Also set its position to (0, 0, -1), so that the new renderer will always be displayed on top of the source one.

Now go to your enemy's script. You will need to add a few new variables. Specifically, one **Sprite Renderer** for the source, one for the new color, and an array which will hold pairs of sprites. The array will be used to check which sprite is active in the **source**, then change the **new** renderer's sprite to the matching one.

First, the following variables will be defined:

```
public SpritePair[] pairs;
public SpriteRenderer rendSource;
public SpriteRenderer rendOutput1;
```

Now go to the inspector and assign the S**ource** Renderer to the **rendSource** variable, the **Color 1** Renderer to **rendOutput**1, and finally open the **pairs** field, The **Size** should  be equal to the number of sprites your enemy has. Open up each Element, and with your sprite open in the **Project** window, drag each matching **Source** and **White** sprites into the **Key** and **Value** fields of the element. The Source sprite should be put in the Key field, and the Output1 should  be put in the Value.

Finally, add the following method to your script:

```
protected void ChangeSpriteColor(SpriteRenderer source, SpriteRenderer output,  SpritePair[] spritePairs)
{
    foreach (SpritePair s in spritePairs)
    {
        if (source.sprite == s.Key)
        {
            output.sprite = s.Value;
            return;
        }
    }
}
```

You can now use this method as many times as you want with  as many outputs as you want. Simply add the following line into a **LateUpdate()** method, and  define new **rendOutput**X and **pairs** variables as needed:

```
ChangeSpriteColor(rendSource, rendOutput, pairs);
```

Try your game out. You can change the color of your enemy by changing the **Color** variable in the **Color 1** renderer.

You can also set your enemy to change colors from the Blueprint, eliminating the need to make multiple prefabs simply to have different colors of enemies. Override the **ChangeColorScheme(Color[] colors)** method with the following:

```
public override void ChangeColorScheme(Color[] colors)
{
    if (colors.Length >= 1)
    {
        spriteOutput1.color = colors[0];
        spriteOutput2.color = colors[1];
        spriteOutput3.color = colors[2];
        ...
    }
}
```

Change the **colors.Length** and add or remove spriteOutputs accordingly.

**Modified Drops:**

To modify the drops of an enemy, you need to simply override the **Kill(bool makeItem,  bool makeBolt)** method, add the item drop part in yourself, then kill the enemy normally without any drops.

Override the method:

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

}
```

The code for dropping a random item is the following:

```
    GameObject item = Item.GetObjectFromItem(Item.GetRandomItem(1, 1, 1, 1, 1, 1, 1, 1));
    if (item)
    {
        item = Instantiate(item);
        item.transform.position = transform.position + center;
        item.transform.rotation = transform.rotation;
        item.transform.localScale = transform.localScale;
    }
```

To drop a bolt instead, replace the first line with the following:

```
    GameObject item = Item.GetObjectFromItem(Item.GetRandomBolt(1, 1, 1, 1));
```

You can also add some bounce to the item dropped by  adding the following line after Instantiating the item:

```
item.GetComponent<Rigidbody2D>().AddForce(transform.up * Random.Range(-15000, 15000f) +
transform.right * Random.Range(0, 8000f));
```

Finish by adding the following line, to call the original Kill method and destroy the enemy:

```
base.Kill(false, false);
```