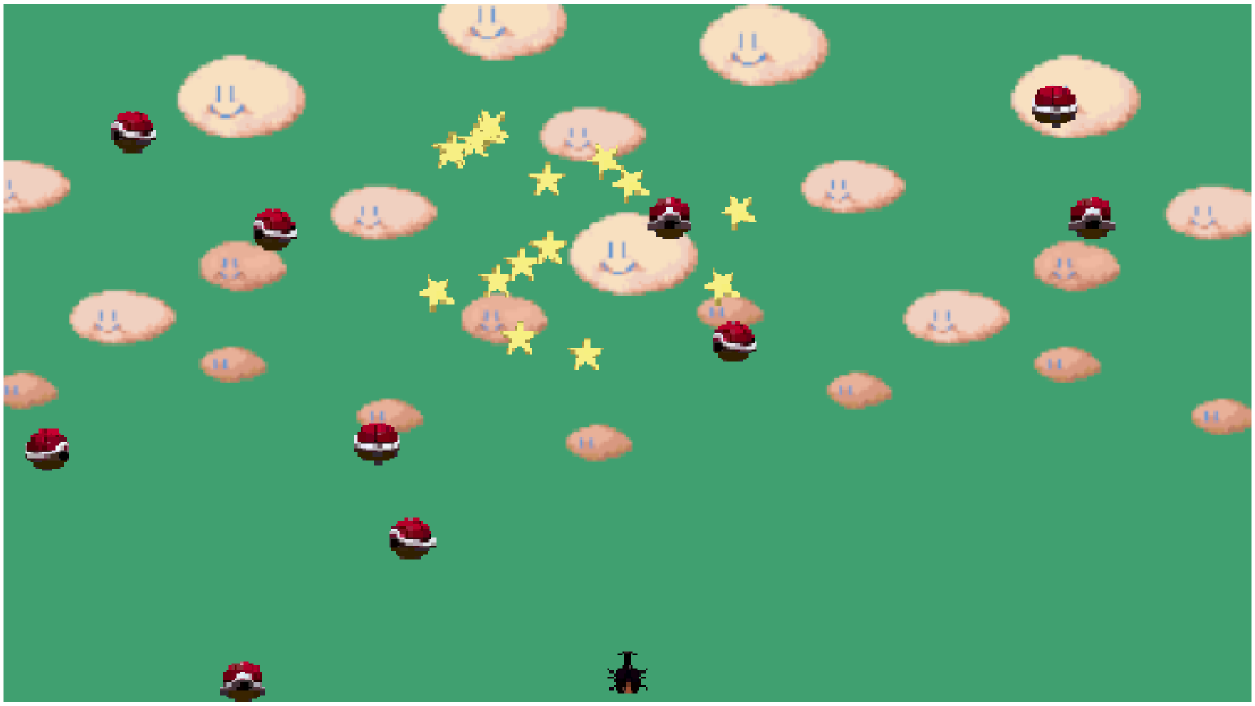
Beetle Mania: Part 3 – Finishing Touches

Lesson Goals

1. Discuss the programming patterns of States and State Machines and how they are useful for game development.
2. Learn how to create a simplified state machine. As well as create the Move and Downed states for the player.
3. Learn how to give the player temporary invincibility.
4. Create a score and score combo system.
5. Give the objects in our game sprites and animations.

Demo

1. By the end of this lesson, our game will look like:



States and State Machines

Before we start any coding let's talk about the concept of **States** and **State Machines**. For our game we are going to create a simplified version of each concept, but it will help to talk about each concept in full detail.

States

1. **States** are something that will hold only the code needed for a particular action. They are a great way to organize the code that we write into chunks.

   In the states code, there will be some action require for that state to transition into a new state.

2. For example, say we have a player. The player has the actions of idling, running and jumping.

   We could have one script that holds all the code needed for every action the player can perform. This could certainly be a method of scripting our player if our game is relatively simple. But if our game might grow in size and scope, so too might the problems with handling our player's actions this way.

   Instead of one big script we could create smaller scripts that contain the code for each action. These scripts would be the states that our player could be in.

3. A general rule of thumb in programming is we want to avoid **coupling** the code together. That is having a script serve multiple purposes or having them rely on other scripts in some way shape or form.

   We should aim to make our scripts as simple and independent as we possibly can. However sometimes scripts will be complex and will need to communicate with other scripts.

4. Just as one last example, the game we are replicating, Beetle Mania, has two states for the player: Move and Downed. So, for our game, we would expect to put all the movement code in the move state and all the downed code in the downed state.

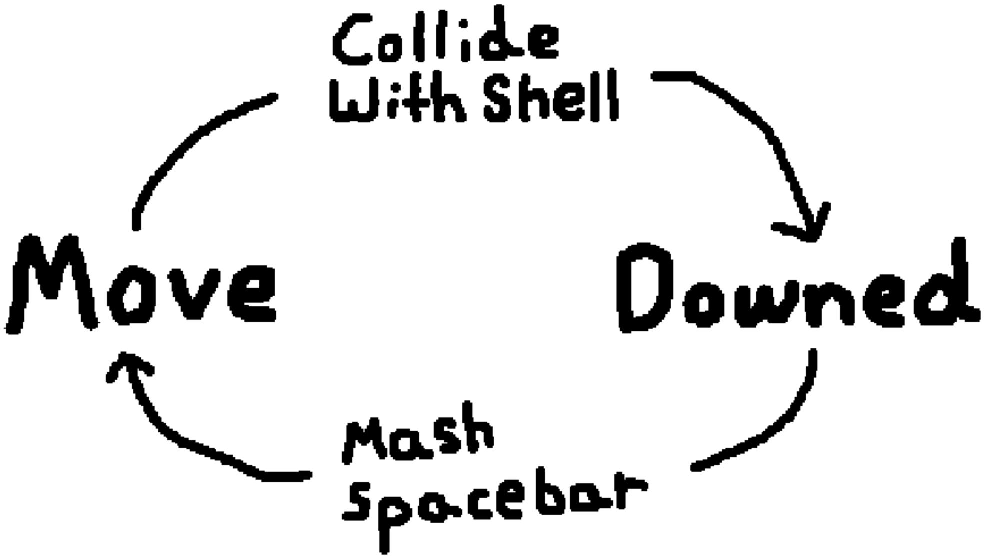
State Machines

1. In conjunction with states, we use what is known as a **State Machine**. State Machines are responsible for running each state's code as well as handling the transitions from one state to the next.

   State machines have a fixed number of states they can be in, but they can only be in **one** state at a time.

2. If we want to envision a state machine, we can think about the various states being connected by a series of arrows. Which way the arrow points determines how one state can

flow into the next.



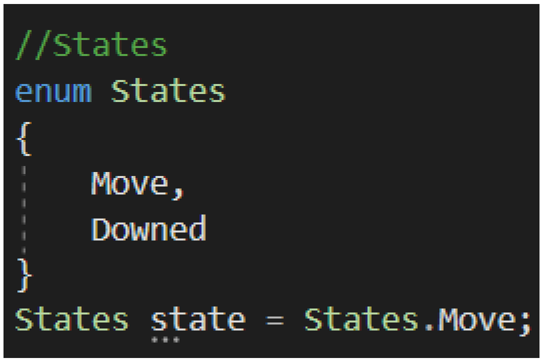
The above image represents the player from our game. The player can move to the downed state if they collide with a shell. They can them move from the downed state if they mash the space bar.

Being able to draw out a state machine can often help us plan out the code that we need to write.

As I mentioned at the top of this section, the code we will write is a simplified version of each concept. There is a lot more we can do to make proper states and state machines but what we will be doing will be fine for our game. If you want to find out more about these concepts, I highly recommend looking at Game Programming Patterns: State. This is a very good book that covers a lot of useful programming patterns for game development.

Player State Machine and Move State

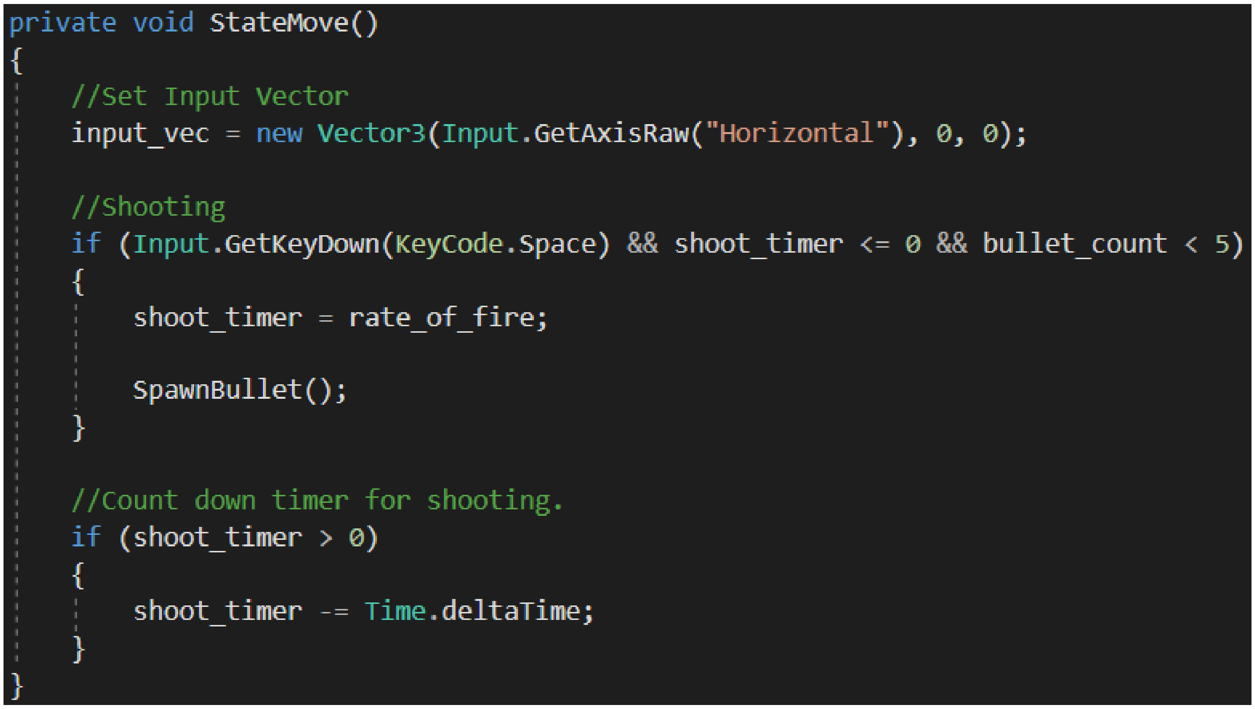
1. In order to create the states needed for the player we are going use an **Enum** variable. The enum variable type allows us to create a variable that can hold multiple values.



```
//States
enum States
{
    Move,
    Downed
}
States state = States.Move;
```

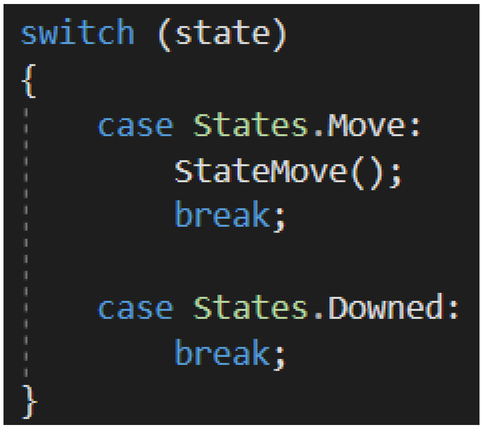
For our enum we create one called **States**. This enum has two potential values: Move and Downed. These two values will act as the different states that the player can be in.
We then create a variable called **state** that is of the type of States. This variable will be the current state that the player is in.

2. Let’s then create a new function called **StateMove()**. This function will contain only the code needed for when the player is moving. We shouldn’t need to create anything new for this function, we can just cut what is inside the **Update()** function and paste it inside the **StateMove()** function.



```
private void StateMove()
{
    //Set Input Vector
    input_vec = new Vector3(Input.GetAxisRaw("Horizontal"), 0, 0);

    //Shooting
    if (Input.GetKeyDown(KeyCode.Space) && shoot_timer <= 0 && bullet_count < 5)
    {
        shoot_timer = rate_of_fire;

        SpawnBullet();
    }

    //Count down timer for shooting.
    if (shoot_timer > 0)
    {
        shoot_timer -= Time.deltaTime;
    }
}
```

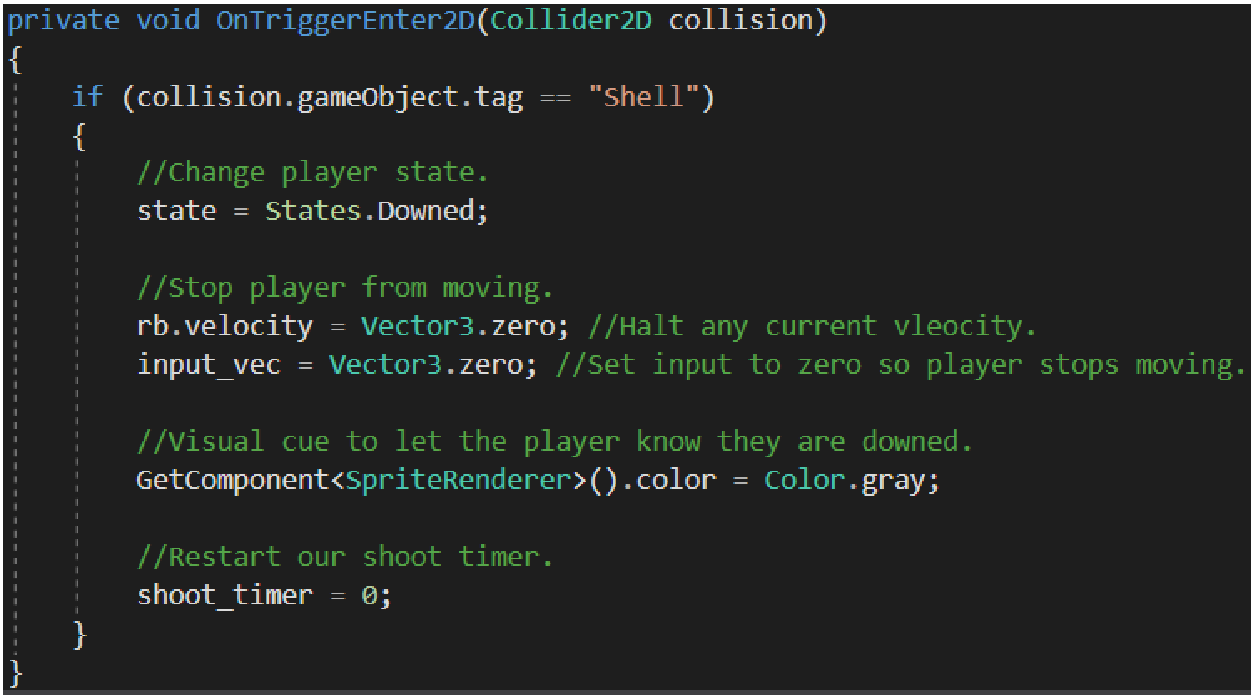
3. Then in the update function lets write the code:



```
switch (state)
{
    case States.Move:
        StateMove();
        break;

    case States.Downed:
        break;
}
```

We use a switch statement that looks at the current state of the player.
If the player’s state is States.Move, it will run the StateMove() function.
Otherwise nothing will happen.

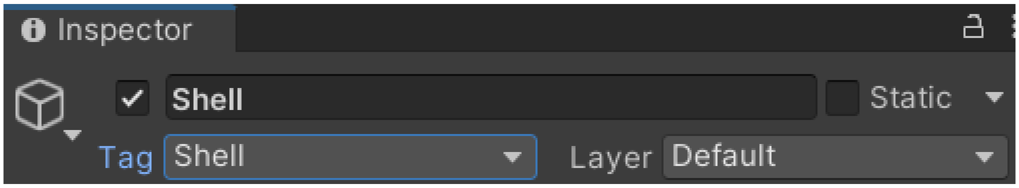
4. Then in the OnTriggerEnter2D function, lets write the code:



```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.tag == "Shell")
    {
        //Change player state.
        state = States.Downed;

        //Stop player from moving.
        rb.velocity = Vector3.zero; //Halt any current vleocity.
        input_vec = Vector3.zero; //Set input to zero so player stops moving.

        //Visual cue to let the player know they are downed.
        GetComponent<SpriteRenderer>().color = Color.gray;

        //Restart our shoot timer.
        shoot_timer = 0;
    }
}
```

We check to see if the object the player collided with was a shell.
If so, we **change** the state of the player to the States.Downed.
We stop any velocity the player had as well as force the input vector to be zero.
We set the color of the player to be gray, this will be a nice indication of what state the player is in.
Then lastly we set the shoot_time to zero. We do this so that when the player is revived they can start shooting immediately.

5. Save the script and return to Unity. We need to tag Shell as a **Shell** so the player can properly collide with it.

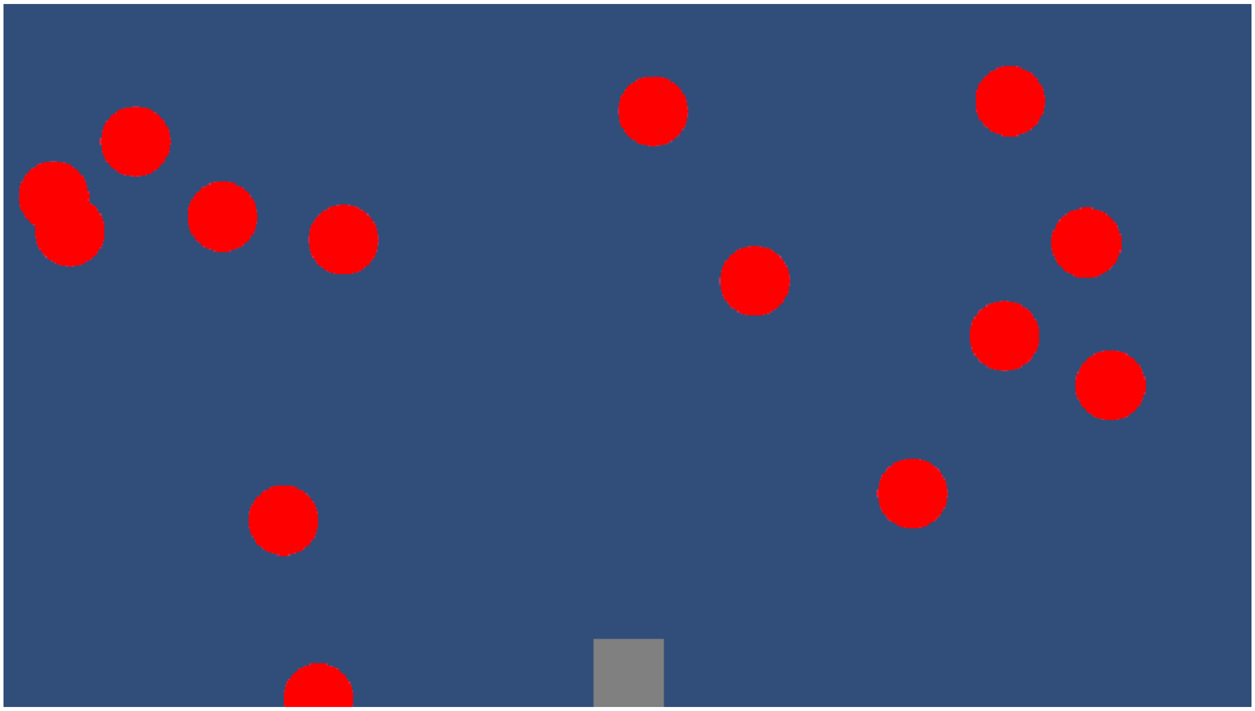


Inspector

☑ Shell □ Static

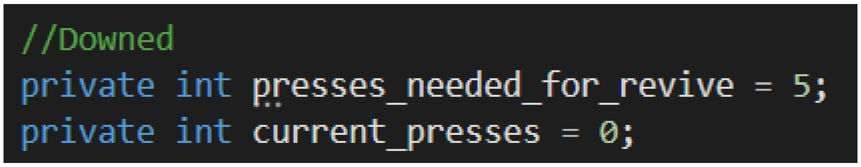
Tag Shell Layer Default

Once done, if we run the game, when the player is touched by a shell, they will turn gray and will not be able to move showing that the player has changed to the downed state.



Downed State and Reviving

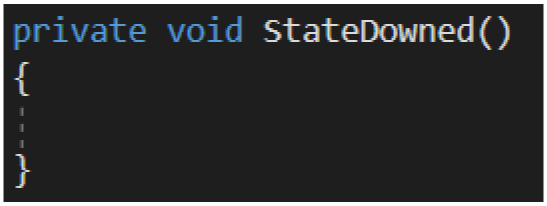
1. Next lets get to work on creating the Downed state for the player. In the **Player** script lets add the variables:



```
//Downed
private int presses_needed_for_revive = 5;
private int current_presses = 0;
```

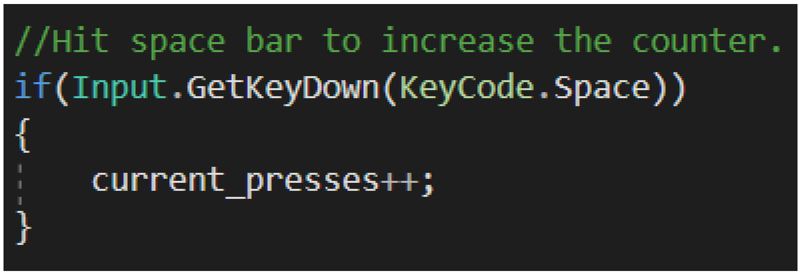
Presses_Needed_For_Revive will be the number of times the player needs to hit the space bar in order to begin moving again.
Current_Presses is the number of times the player has presses the space bar to revive.

2. Underneath the **StateMove()** function lets create a new function called **StateDown()**. This function will contain all the code for when the player is considered downed.



```
private void StateDowned()
{

}
```

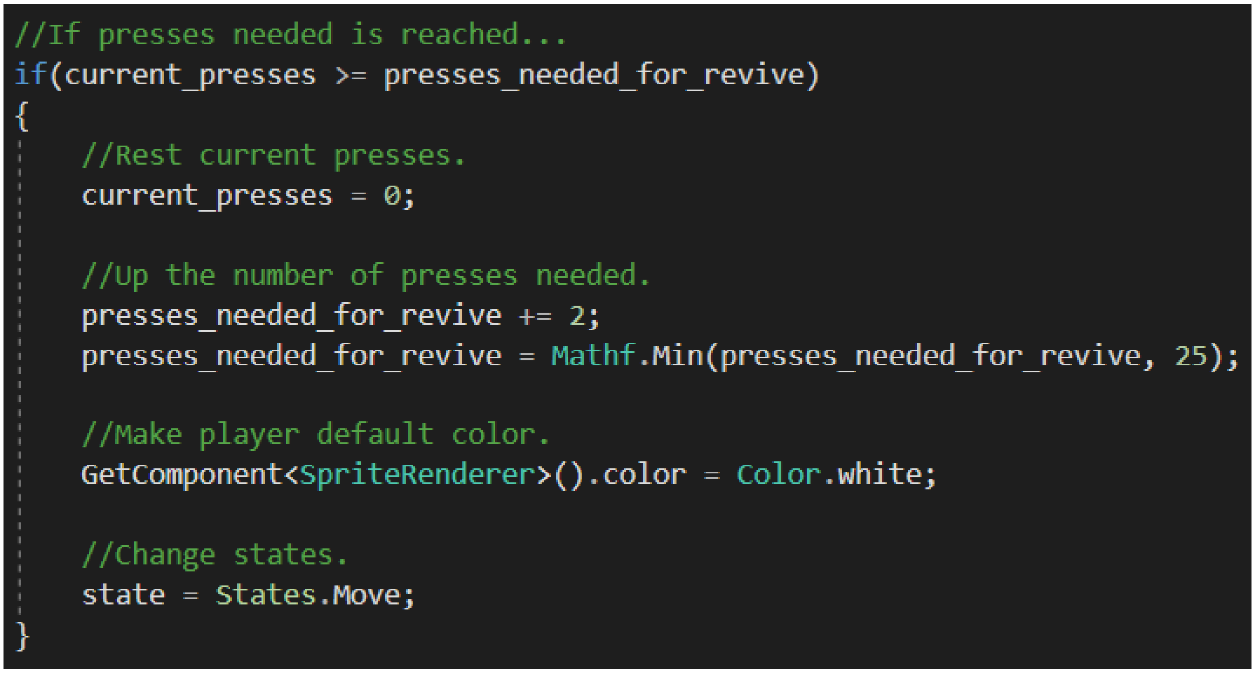
3. Inside the function let’s write the code:



```
//Hit space bar to increase the counter.
if(Input.GetKeyDown(KeyCode.Space))
{
    current_presses++;
}
```

Here we are checking to see if the spacebar is being pressed.
If it is, then we increase the value of current_presses by one.

4. Next, lets write the code:

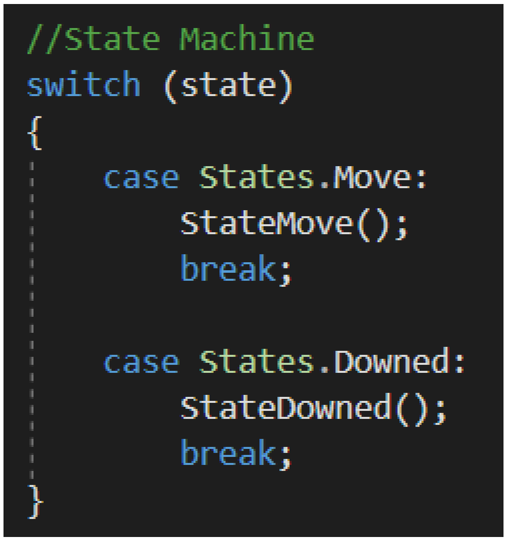


```
//If presses needed is reached...
if(current_presses >= presses_needed_for_revive)
{
    //Rest current presses.
    current_presses = 0;

    //Up the number of presses needed.
    presses_needed_for_revive += 2;
    presses_needed_for_revive = Mathf.Min(presses_needed_for_revive, 25);

    //Make player default color.
    GetComponent<SpriteRenderer>().color = Color.white;

    //Change states.
    state = States.Move;
}
```

We first write an if statement to see if the value of current_presses is greater than or equal to the presses_needed_to_revive variable.
If so, we reset the current_presses to 0.
We then increase the number of presses need to revive by two. We do this to make the process of reviving each time a little more difficult.
We also make sure to cap to presses needed to a maximum value of 25. Remember that the Min() function will take the smallest of the two supplied values.
We then change the player’s color back to white to show that the player can move normally.

We then change the player’s state back to the move state.

5. Lastly we go to the **Update()** function and tell our switch statement state machine to run the StateDowned() function when the player state is the downed state.



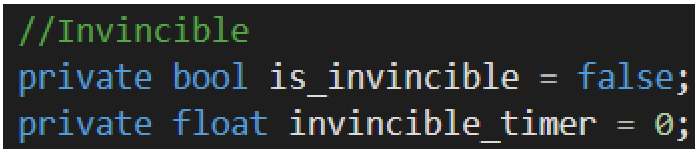
```
//State Machine
switch (state)
{
    case States.Move:
        StateMove();
        break;

    case States.Downed:
        StateDowned();
        break;
}
```

6. If you play the game now, the player will need to hit the space bar five times in order to revive. Every subsequent hit will increase the space bar presses needed by two.

Invincibility

1. Next, we should probably give the player a grace period when they revive. It is not fun to immediately revive only to then get instantly downed. To fix this issue we will make the player invincible for a short period of time.

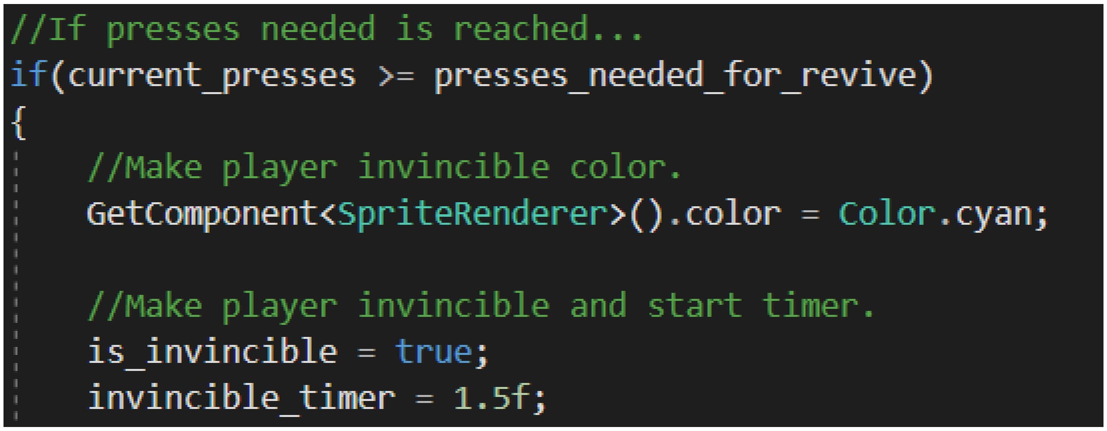
2. In the player script let’s add the variables:



```
//Invincible
private bool is_invincible = false;
private float invincible_timer = 0;
```

Is_Invincible will be used to tell the game if it is player is invincible or not.
Invincible_Timer will be a timer that will be used to keep track of how long the player will be invincible for.

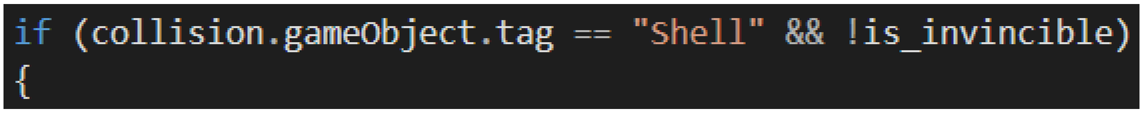
3. Next in the **StateDowned()** function lets add some code to where we are checking to see if the player has reached the presses needed for reviving.



```
//If presses needed is reached...
if(current_presses >= presses_needed_for_revive)
{
    //Make player invincible color.
    GetComponent<SpriteRenderer>().color = Color.cyan;

    //Make player invincible and start timer.
    is_invincible = true;
    invincible_timer = 1.5f;
```

   We first make the player's color cyan. We do this just to show that the player is invincible.
   We then tell that game that the player is invincible by setting is_invincible to true.
   Then we set the invincible_timer to 1.5 seconds. The player will be invincible for that amount of time.

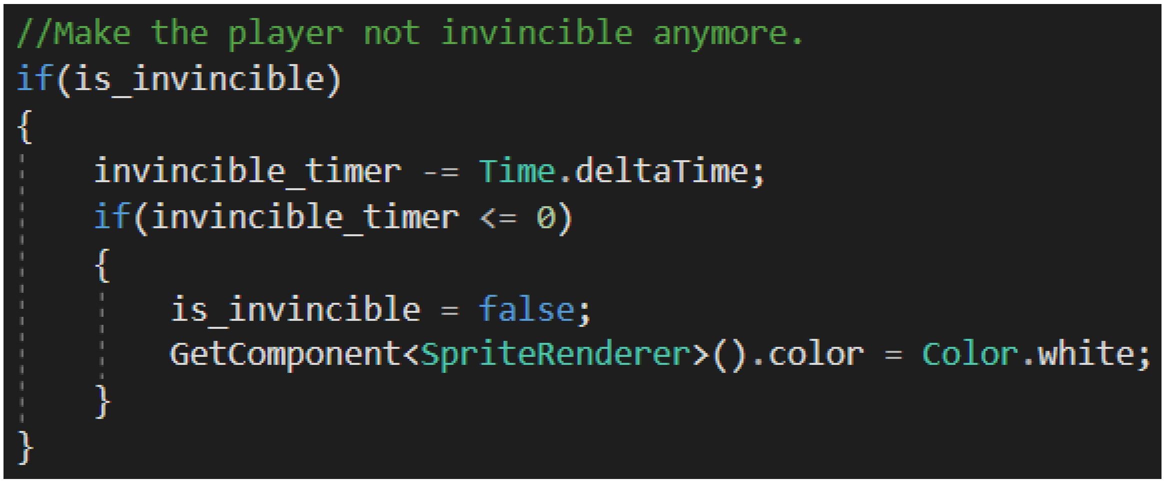
4. Next, lets hop over to the **OnTriggerenter2D()** function and change the if statement inside to read:



```
if (collision.gameObject.tag == "Shell" && !is_invincible)
{
```

   All we are doing is adding a second condition to our if statement by using the && operator.
   We are now checking to see if the player was hit by a shell AND if the player is not invincible.

5. The last thing we need to do is make the invincibility timer count down. At the bottom of the **MoveState()** function lets write the code:



```
//Make the player not invincible anymore.
if(is_invincible)
{
    invincible_timer -= Time.deltaTime;
    if(invincible_timer <= 0)
    {
        is_invincible = false;
        GetComponent<SpriteRenderer>().color = Color.white;
    }
}
```

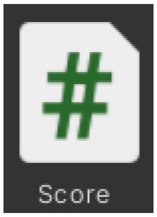
   We are first checking if the player is invincible.
   If so, we subtract Time.deltaTime from the invincible_timer.
   If the timer has reached zero or below, we tell the game the player is no longer invincible

and to change the player's color back to white.

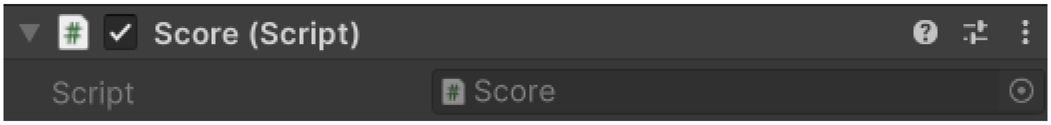
6. If we save the scripts and play the game, we should see the player turn cyan when they revive. If the player is hit by a shell while invincible, nothing will happen. The invincibility of the player will wear off after 1.5 seconds.

Score and Score Chains

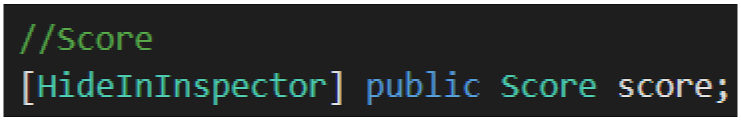
1. Next let's work on the score and score chain system that is in the game. First let's create a new script called **Score**.



Lets then attach this script to the **Game Logic** object.



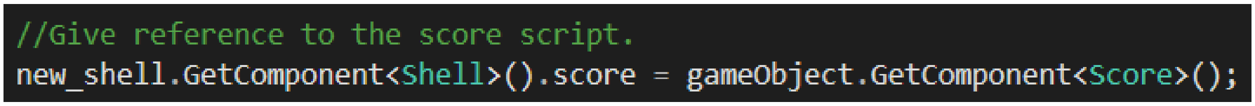
2. Before we dive into the Score script, lets first go into the **Shell** script and give it the variable:



```
//Score
[HideInInspector] public Score score;
```

Score will give the shell a reference to the score script component on the game logic object.

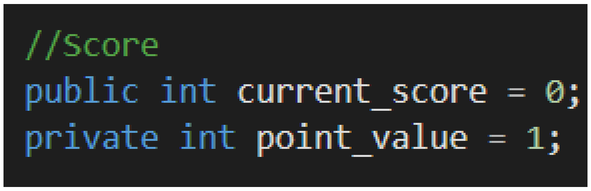
3. Next in **ShellSpawner** script, in the **SpawnShell()** function add the line:



```
//Give reference to the score script.
new_shell.GetComponent<Shell>().score = gameObject.GetComponent<Score>();
```

This will give a reference to the score script to each shell that is spawned by storing it in their score variable.

4. Let's then go into the **Score** script and add the variables:

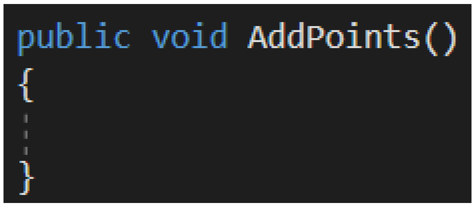


```
//Score
public int current_score = 0;
private int point_value = 1;
```

Current_Score is the total amount of points that the player has in the game.

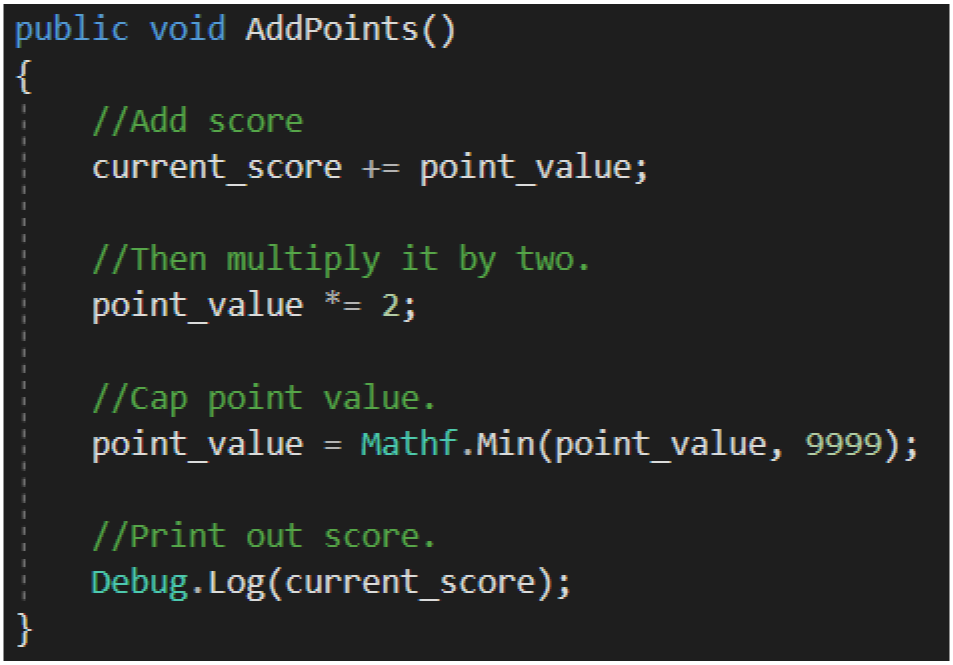
Point_Value are the points that will be added to the current score before they are modified.

5. Next let's create a function called **AddPoints()**.



```
public void AddPoints()
{

}
```

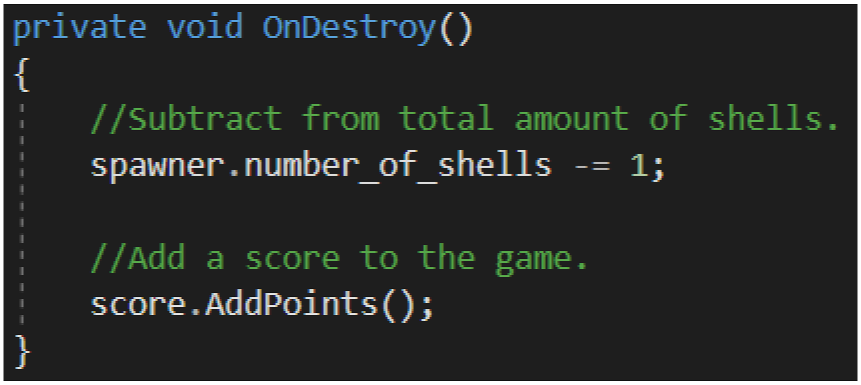
Inside, lets write the code:



```
public void AddPoints()
{
    //Add score
    current_score += point_value;

    //Then multiply it by two.
    point_value *= 2;

    //Cap point value.
    point_value = Mathf.Min(point_value, 9999);

    //Print out score.
    Debug.Log(current_score);
}
```

We first take the current point value and add it to the current score.
We then multiply the point value by 2. This way if another shell is hit, it's point value will be doubled.
We then make sure to cap the point value to 9999.
Lastly, the score is printed out to the console.

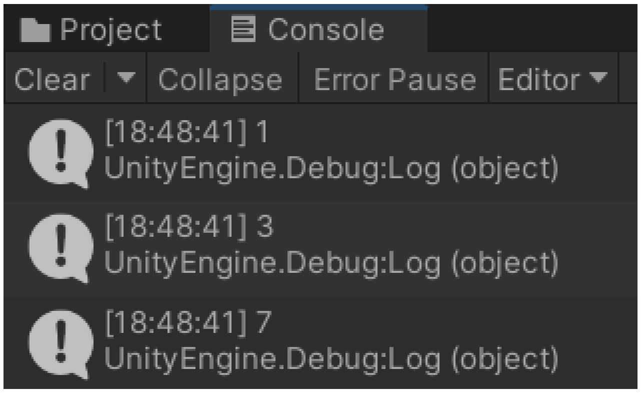
6. Next, let's go back to the Shell script and in the **OnDestroy()** function lets add the code:



```
private void OnDestroy()
{
    //Subtract from total amount of shells.
    spawner.number_of_shells -= 1;

    //Add a score to the game.
    score.AddPoints();
}
```

All we do is have the shell call the AddPoints() function. This way each time a shell is destroyed, points will be added to the score.

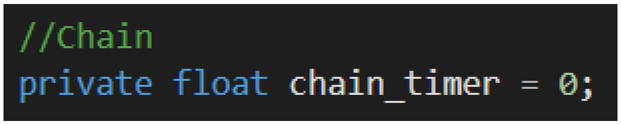
7. Saving all the scripts and running the game now, we will see the current score output in the console.



```
[18:48:41] 1
UnityEngine.Debug:Log (object)
[18:48:41] 3
UnityEngine.Debug:Log (object)
[18:48:41] 7
UnityEngine.Debug:Log (object)
```

Only thing we need to do now is make it so point value will reset once a chain has eneded.

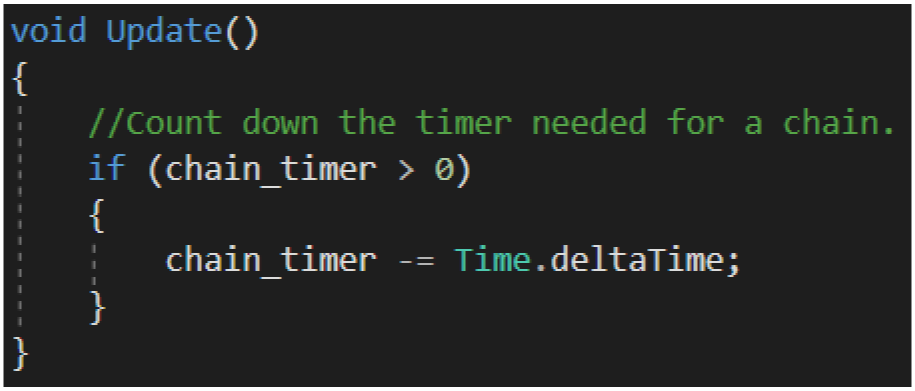
8. To do this, back in the **Score** script lets add the variable:



```
//Chain
private float chain_timer = 0;
```

Chain_Timer is a timer that will be set to a value and will then countdown. Once the timer reaches zero, the chain on the score will end.

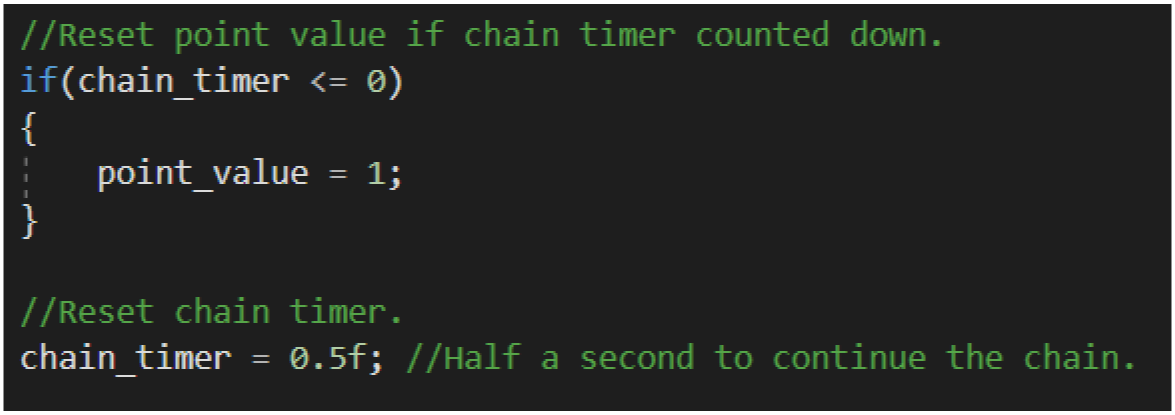
9. Next in the **Update()** function lets write the code:



```
void Update()
{
    //Count down the timer needed for a chain.
    if (chain_timer > 0)
    {
        chain_timer -= Time.deltaTime;
    }
}
```

Here we are checking to see if the chain_timer's value is greater than zero. If so, we will make the timer countdown by Timer.deltaTime.

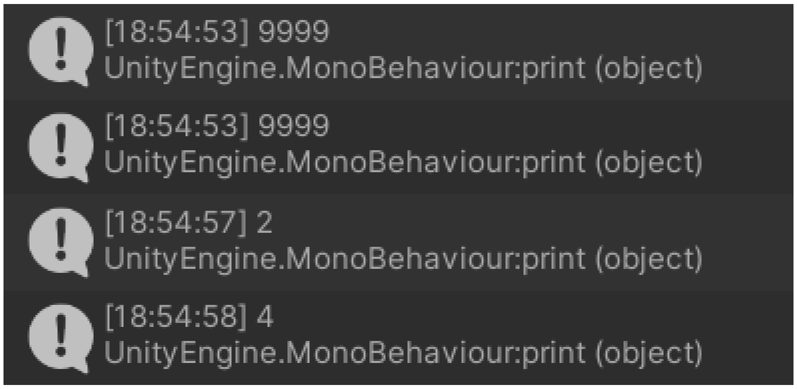
10. Next in the **AddPoints()** function lets add the code:



```
//Reset point value if chain timer counted down.
if(chain_timer <= 0)
{
    point_value = 1;
}

//Reset chain timer.
chain_timer = 0.5f; //Half a second to continue the chain.
```

First, we check to see if the chain_timer is less than or equal to zero. If so, we know that the chain has ended since the timer counted down, so we reset the point value back to one. We then set the chain_timer to 0.5. This makes it so there will be a half a second to continue the chain.

11. Save the scripts and run the game. Now the amount of points the player can gain will be reset after 0.5 seconds. Below I have output the point value to show that after a dropped chain the amount of points will reset.



```
[18:54:53] 9999
UnityEngine.MonoBehaviour:print (object)
[18:54:53] 9999
UnityEngine.MonoBehaviour:print (object)
[18:54:57] 2
UnityEngine.MonoBehaviour:print (object)
[18:54:58] 4
UnityEngine.MonoBehaviour:print (object)
```

Sprites

1. Since we have already covered how to display score in the previous game, we will not be covering it here. Instead, we will be taking a look at how to add sprites and animations into our game. Let's first look at how we create the sprites needed for our animations.

2. **Something something, distribute the sprites to the students.**
    a. I have no idea what platform is being used for student classes so I will just get back to this later.

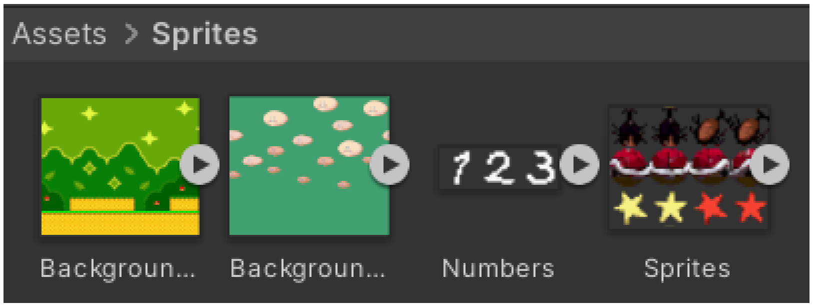
    Once you have downloaded the assets be sure to unzip the folder.

    In the folder you should see two background sprites, a sprite sheet for numbers and a sprite sheet for the beetle, shells and stars.

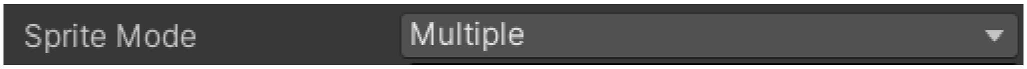
    Before we dive into importing, let's just cover some terms. A **Sprite Sheet** is a collection of images that are commonly called **Sprites**. Sprites are just images. We store multiple sprites in sprite sheets to save resources while making our game. Sprite sheets also help the game run better because the game only needs to locate one image rather than a series of multiple images.

3. Moving back into **Unity** lets create a new folder inside the **Assets** folder called **Sprites**.

    Once made, drag all the sprites from the downloaded folder and drop them into the newly created Sprites folder.

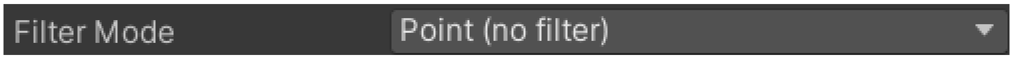


4. First, click on the **Sprites** image and in the inspector change the **Sprite Mode** property to:


Sprite Mode: Multiple

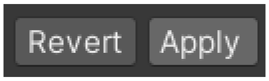
This will tell Unity that there are multiple sprites with in this image.

Let's also change the **Filter Mode** to:


Filter Mode: Point (no filter)

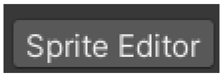
By default, Unity tries to smooth out an image by applying a filter to it. This is great for big images, but it will make small images blurry. So, changing the filter mode to **Point (no filter)** will make our small image clear to see.

Once both settings are changed, apply the changes.

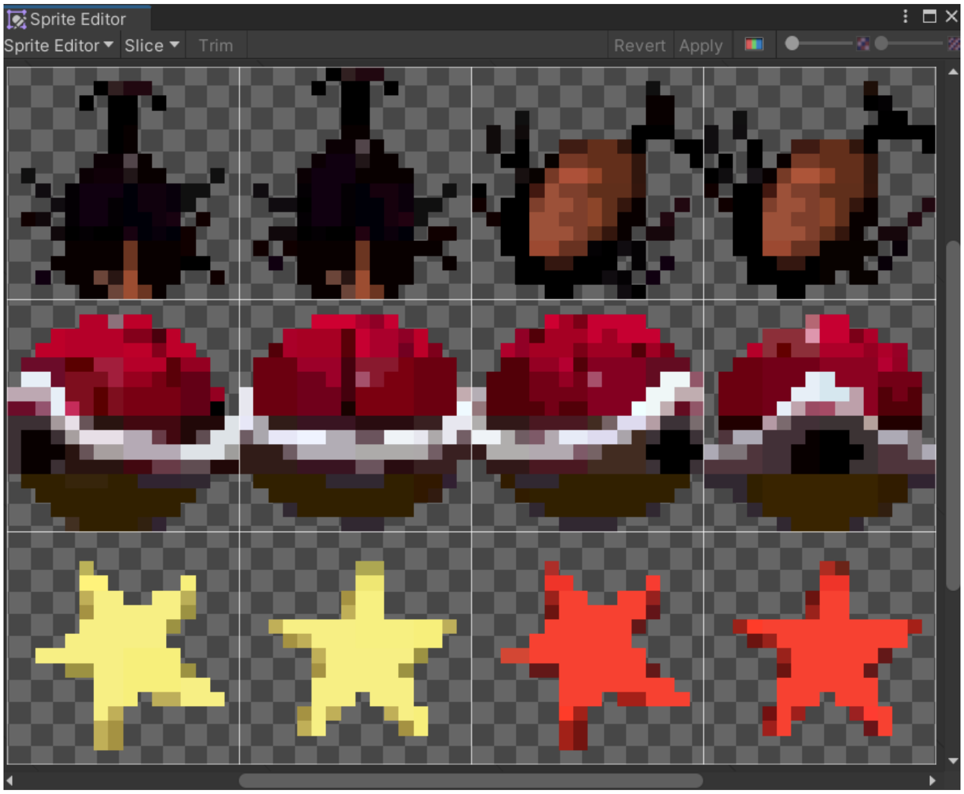


5. Next, we need to slice up our sprite sheet. We do this so we can access each of the individual sprites inside.

   First click the **Sprite Editor** button.

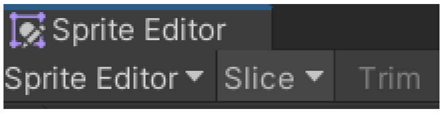


   This will open up the **Sprite Editor** window.

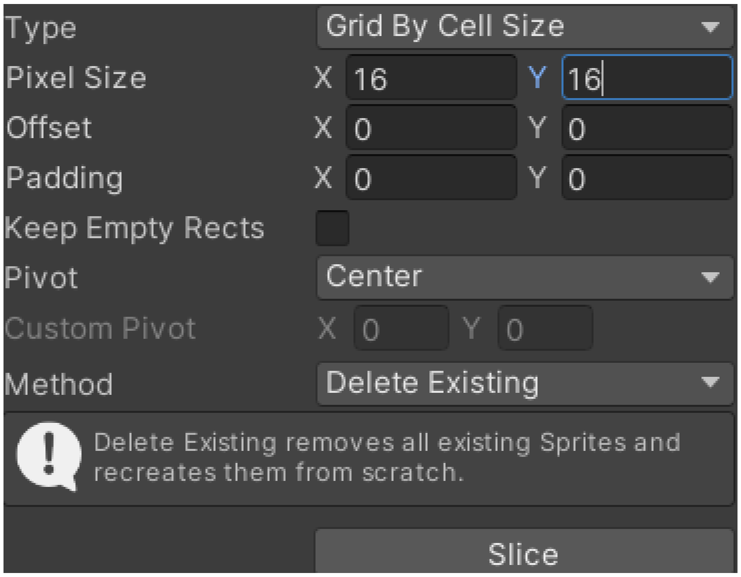


   The sprite editor allows us to define the size of each sprite.

6. At the top left-hand corner of the sprite editor select the **Slice** option.



Let’s then set the options to:

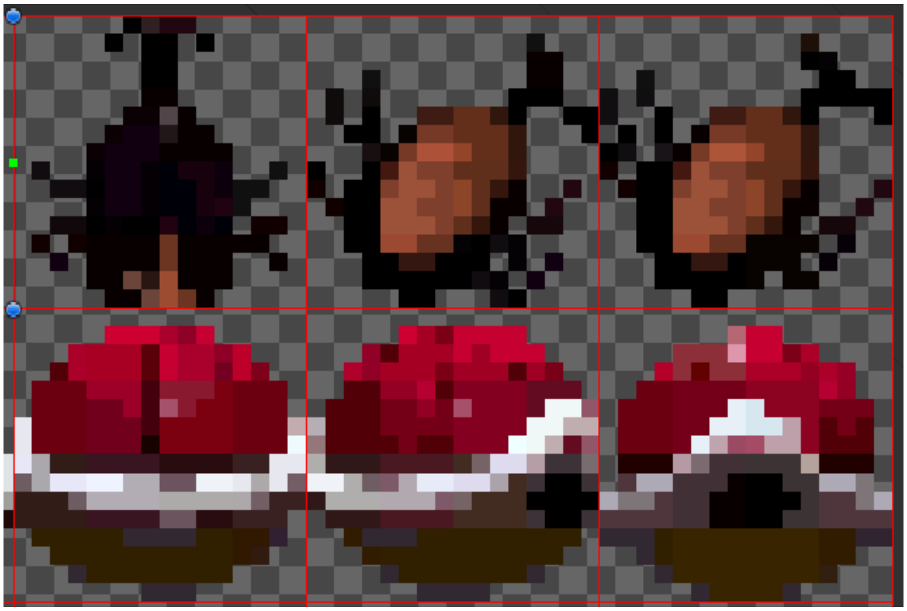


Type	Grid By Cell Size			
Pixel Size	X	16	Y	16
Offset	X	0	Y	0
Padding	X	0	Y	0
Keep Empty Rects	□			
Pivot	Center			
Custom Pivot	X	0	Y	0
Method	Delete Existing			

Delete Existing removes all existing Sprites and recreates them from scratch.

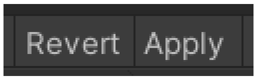
Here we are telling the sprite editor that we will define each sprite by their **cell size**. That being that each sprite is **16 pixels wide and 16 pixels tall**.

We should then see a red grid appear around each sprite. If the grid perfectly encapsulates a whole sprite, then the pixel size was set to the correct value.

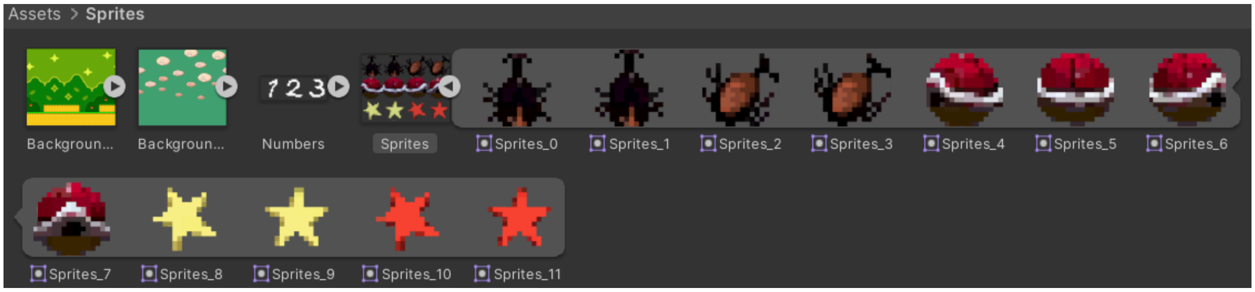


If everything looks good, we can click the **Slice** button.

Then in the sprite editor window, at the top-right corner hit the **Apply** button.



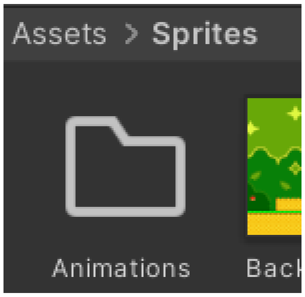
7. Close the sprite editor and look at the Sprites image. There should now be an arrow on top of it. Clicking the arrow will show how the image has been broken up into multiple sprites.



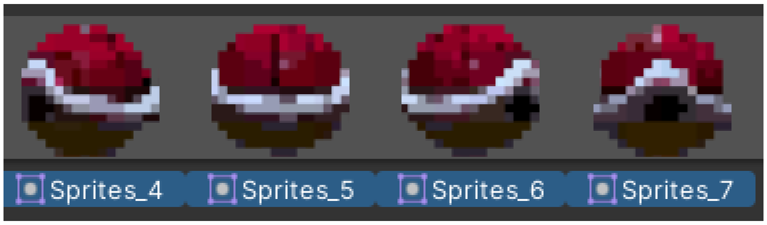
Shell Animations

1. Now that we have all the sprites needed, lets create the animation for the shells.

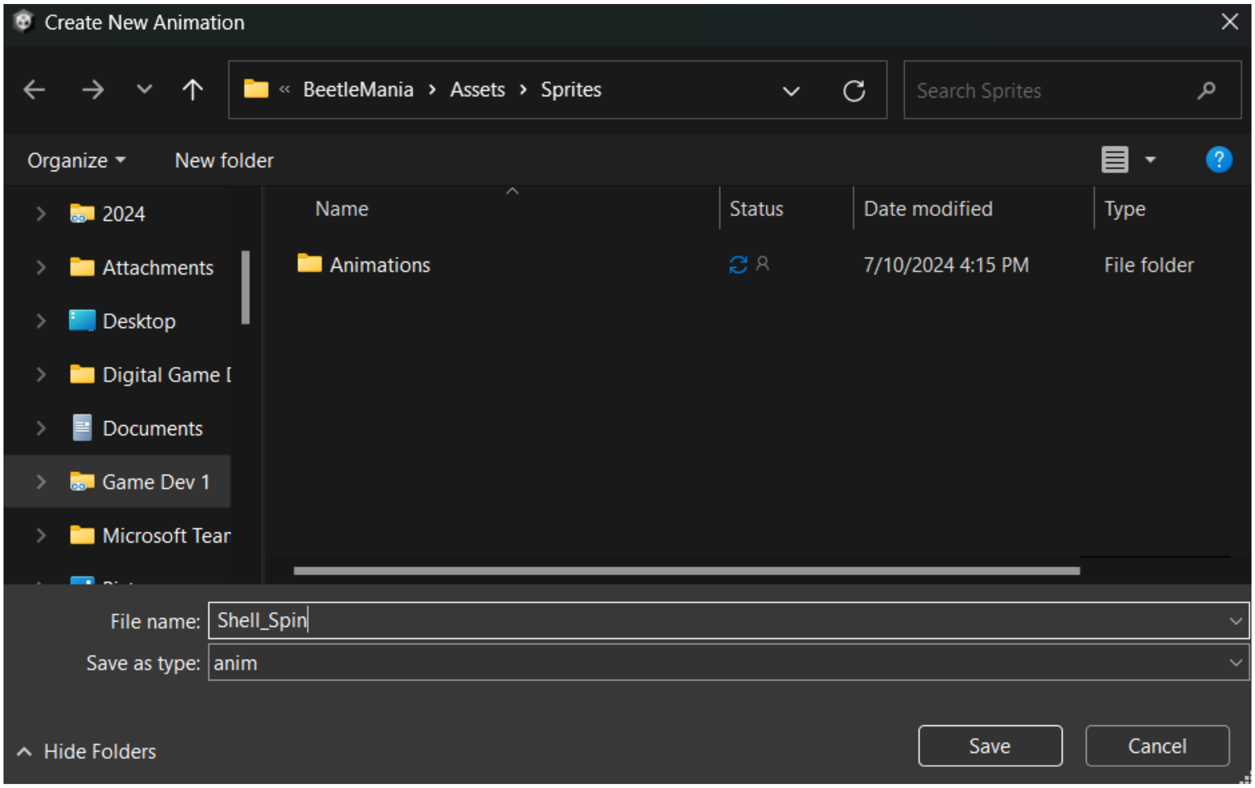
2. In the **Sprites** folder create a folder called **Animation**.



3. To create an animation, select all the sprites that make up the shell's animation. Then drag them into the scene.



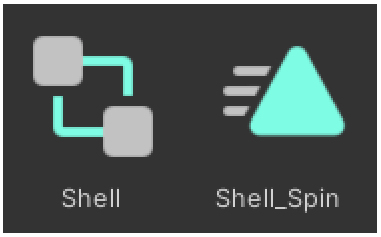
This will open a new window prompting you to create a new animation. Name the animation **Shell_Spin**.



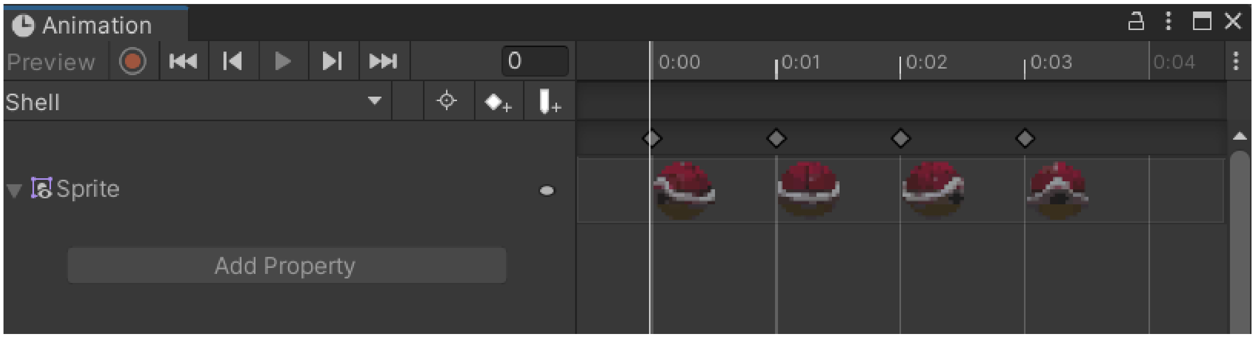
File name: Shell_Spin

Save as type: anim

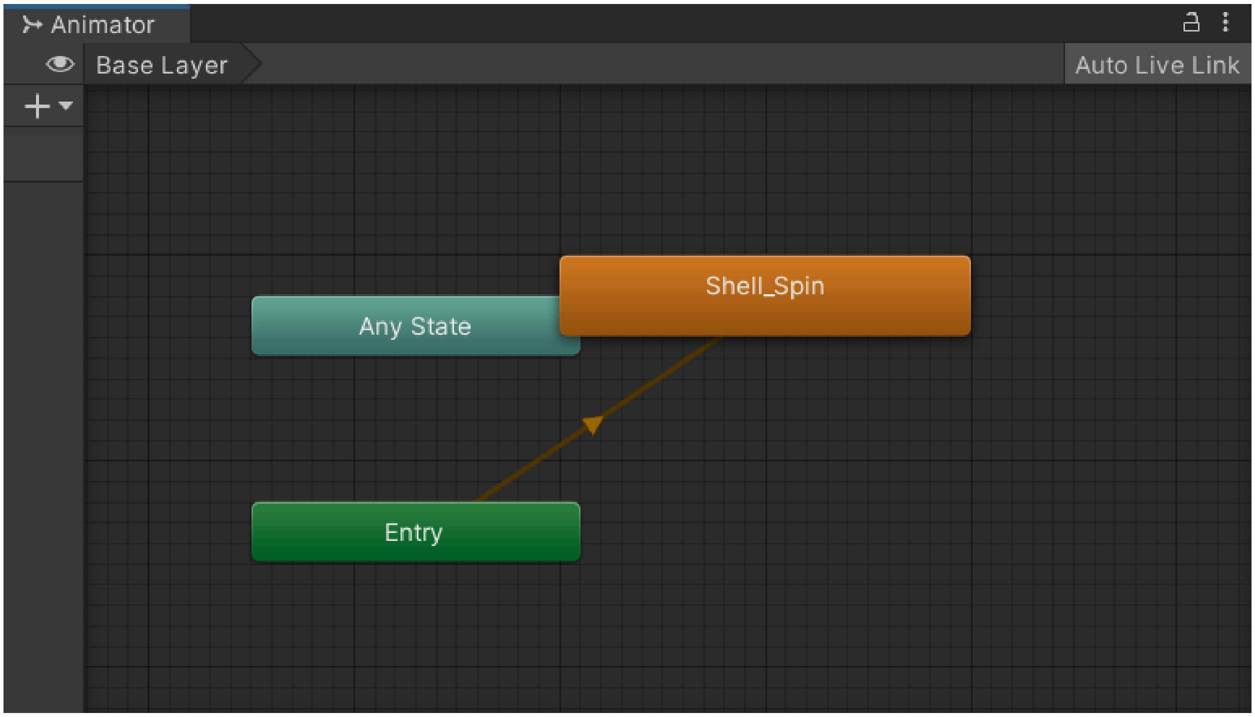
Save the animation in the **Animations** folder that we just created. Once saved you will see two objects in the folder.



The object named Shell_Spin is an **Animation Clip**. An animation clip contains the properties of an object that is being animated. If we double click on the clip, we will see a timeline. We can see that the property we are animation is the sprite of the shell.

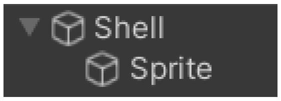


The other object is an **Animator Controller.** I've renamed to **Shell.** Double clicking on the object will open up the **Animator** tab. To keep things simple, the animator allows us to control what animations will be able to flow into each other. This may look familiar. Unity's Animator is also a **State Machine**, where each state is just a different animation.

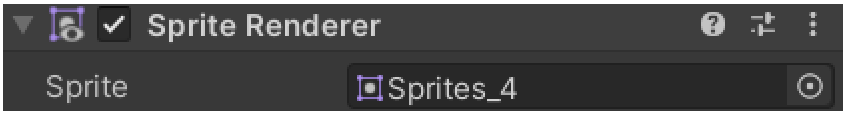


4. **Important:** Since we drag the frames of the animation into the scene, there will be a small shell sprite in our project. Let's delete this object from our game, since we just needed to create the animation.

5. Next, go into the **Shell prefab** and **remove** the **Sprite Renderer** from the shell. Then add a **2D Objects > Sprites > Circle** as child of the Shell object.

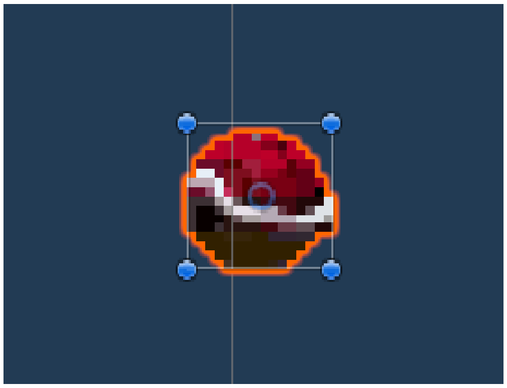


This effectively gives us our circle sprite back. However, we are going to replace the sprite by dragging the first sprite of the turtle shell into the sprite renderer's **Sprite** property.

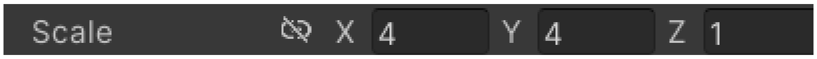


Sprite: Sprites_4

We should now see that the circle has been replaced with the turtle shell sprite.



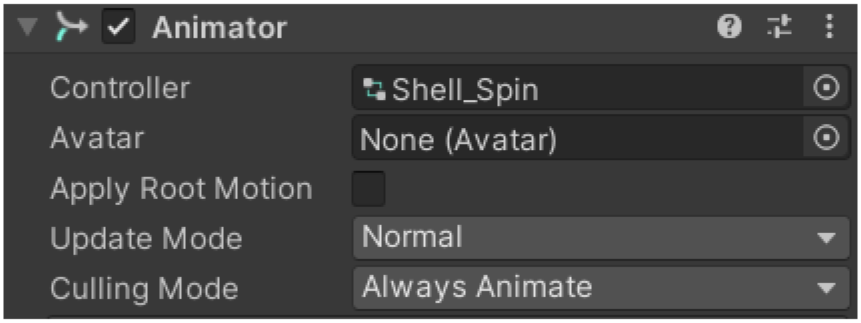
6. The shell is very tiny however, so let’s scale it up.



Scale X 4 Y 4 Z 1

Only scale the sprite and not the parent object.

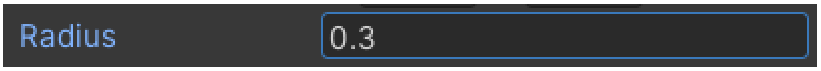
7. With the **sprite object** still **selected** drag the **Animator** on to the inspector. This will now give the sprite the animation.



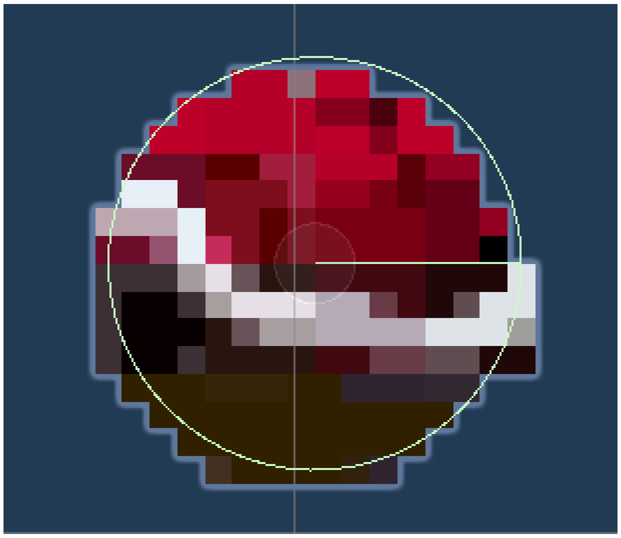
☑ Animator

Controller	Shell_Spin
Avatar	None (Avatar)
Apply Root Motion	□
Update Mode	Normal
Culling Mode	Always Animate

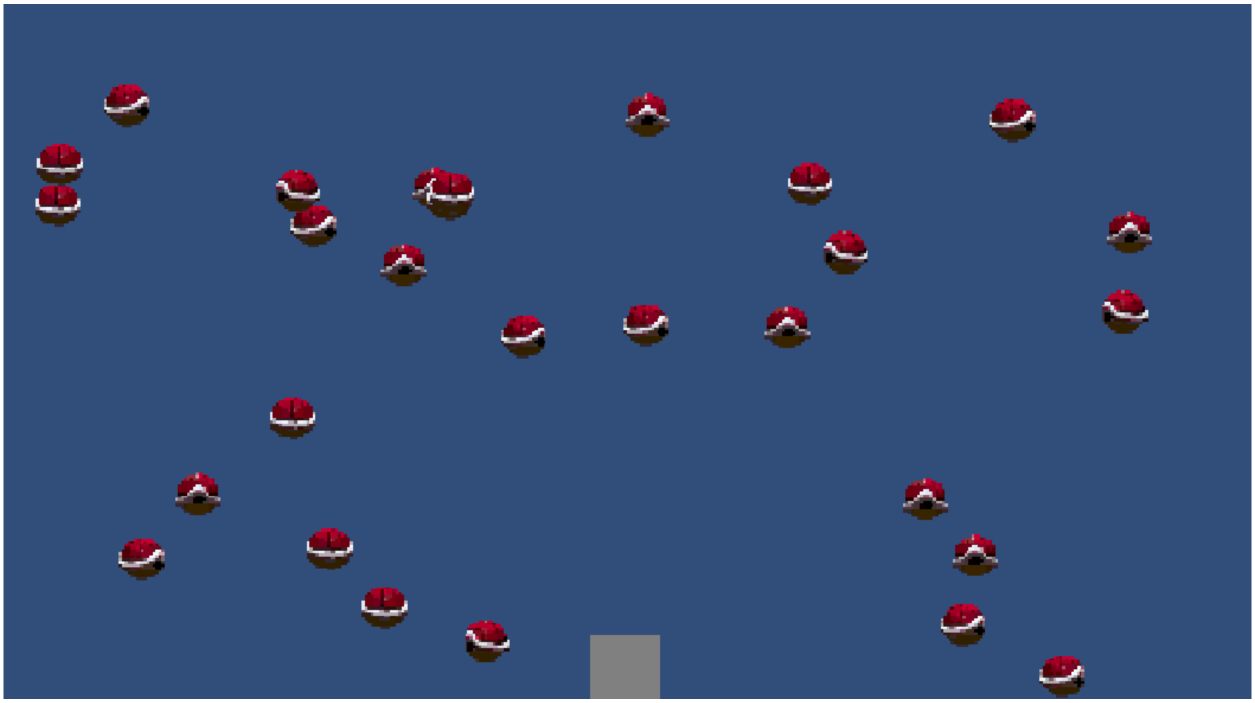
8. Lastly, select the parent shell object and adjust the size of the collider to match the size of the shell. Set both **Radius** properties of the **Circle Collider 2Ds** to **0.3**.



Radius 0.3



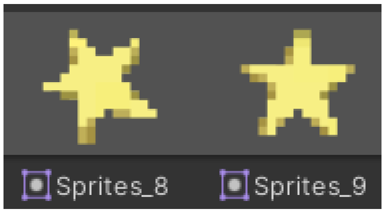
9. If we run the game now, are shells will have a spinning animation to them.



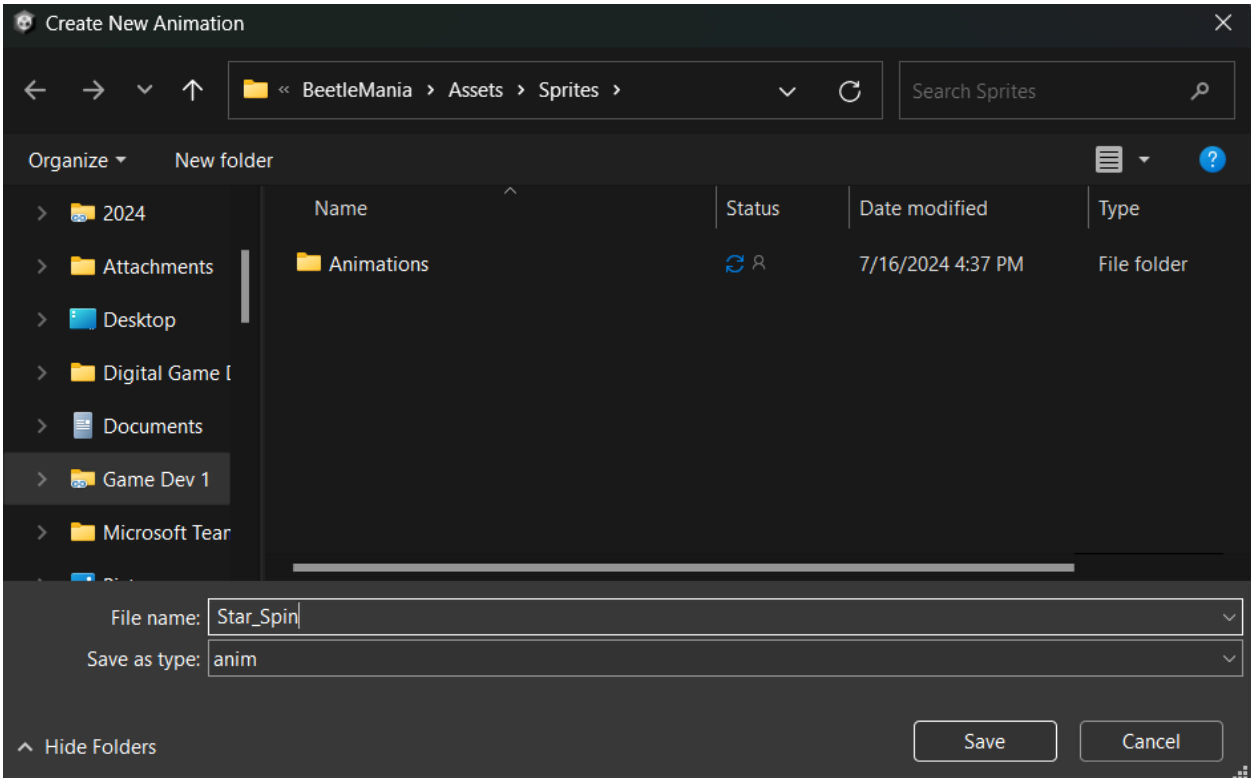
Bullet Animations

1. Let’s next work on getting the animations for the Bullet.

2. Select both the yellow start sprites and drag them into the main scene.



This will prompt you with a window asking you to create a new animation. Name the animation **Star_Spin** and save it to the animations folder.

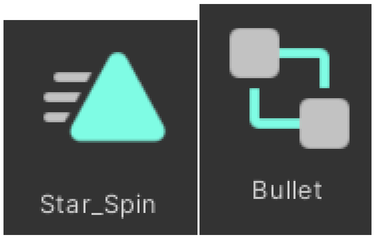


Name	Status	Date modified	Type
Animations		7/16/2024 4:37 PM	File folder

File name: Star_Spin

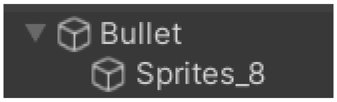
Save as type: anim

Once done you will see the new animation and the animation controller. Name the controller **Bullet**.



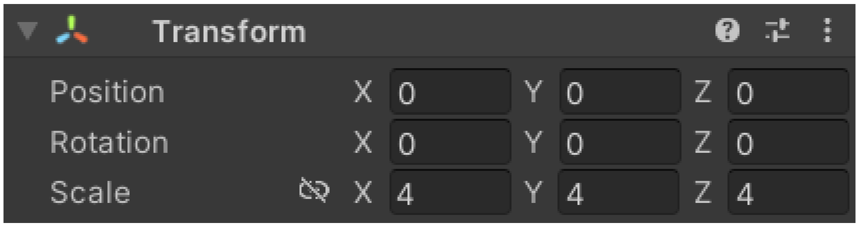
3. Open **bullet prefab** scene. Like before we are going to swap out the sprite of the prefab.

First **delete sprite render** from bullet and then click and drag **the first yellow star sprite** into the scene. This will add it as a child of the main bullet object.



Bullet
Sprites_8

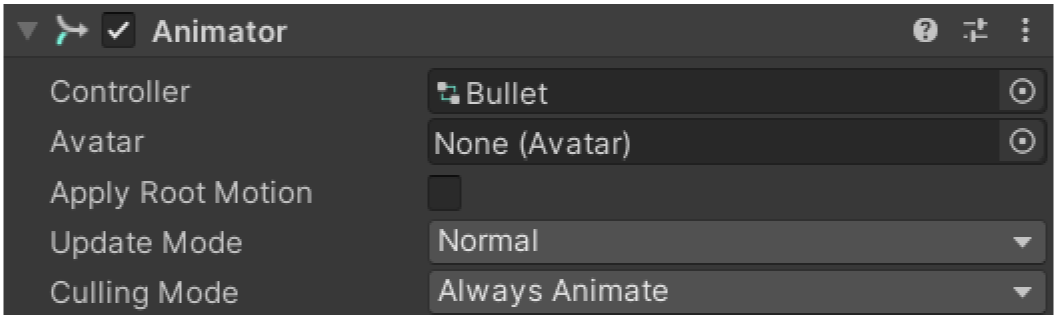
Zero out the sprite’s position and set its scale to 4.



Transform

Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 4 Y 4 Z 4

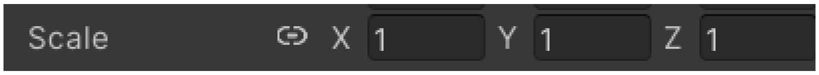
4. Next drag the **Bullet Animator** and drop in on the sprite’s inspector.



☑ Animator

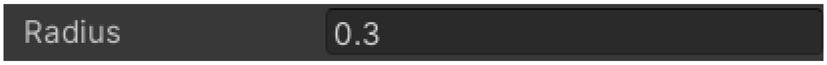
Controller: Bullet
Avatar: None (Avatar)
Apply Root Motion: □
Update Mode: Normal
Culling Mode: Always Animate

5. Select the bullet object and reset scale of bullet to 1. This will ensure the only the sprite has its scale changed.

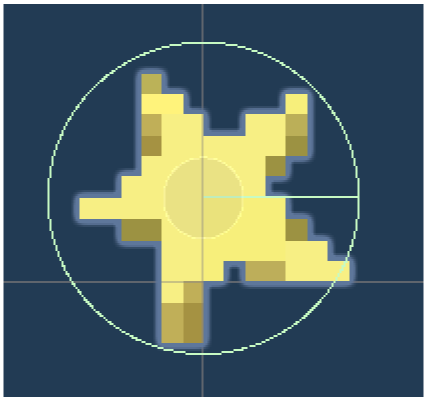


Scale X 1 Y 1 Z 1

6. Lastly adjusted **CircleCollider2D’s radius** on the bullet to match the shape of the sprite.

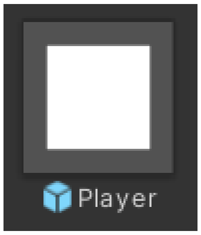


Radius 0.3

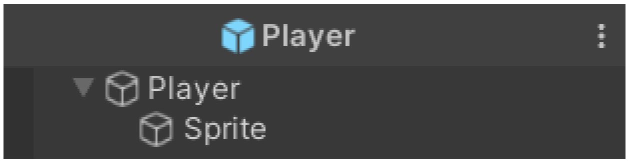


Player Animations

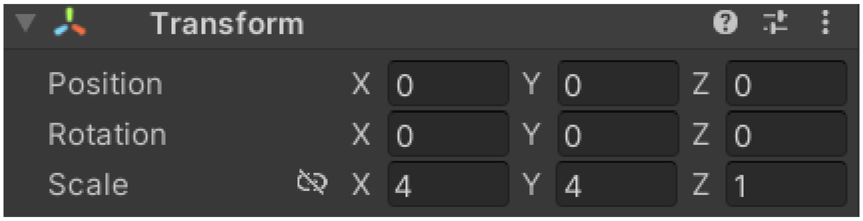
1. Let’s then add animations to the player. First drag the player into the Assets folder to make it a prefab.



2. Double click the player to open the player prefab scene. Remove the sprite renderer on the player and then add in a child sprite to the player object.

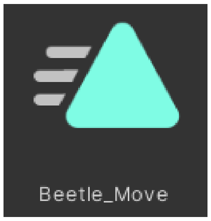


Make sure the sprite’s position is zeroed out and scale is four.

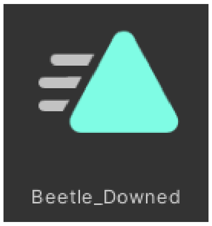


Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	4	Y	4	Z	1

3. Drag in the first two sprites of the beetle. Name the animation **Beetle_Move**.

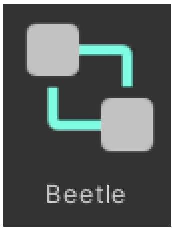


Repeat the process for the remaining two beetle sprites. Name this animation **Beetle_Downed**.



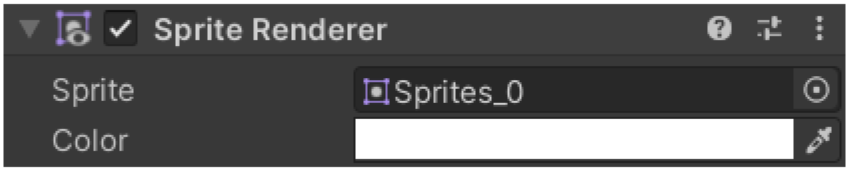
Remove the two small beetle sprites that were created when making the animations.

4. Delete **one** of the beetle **Animator Controllers**. Rename the remaining one to **Beetle**.



We only need one animation controller for the beetle since we can give the controller multiple animations to transition between.

5. Drag in one of the beetle sprites and place in the **Sprite** property of the sprite render on the child sprite object.

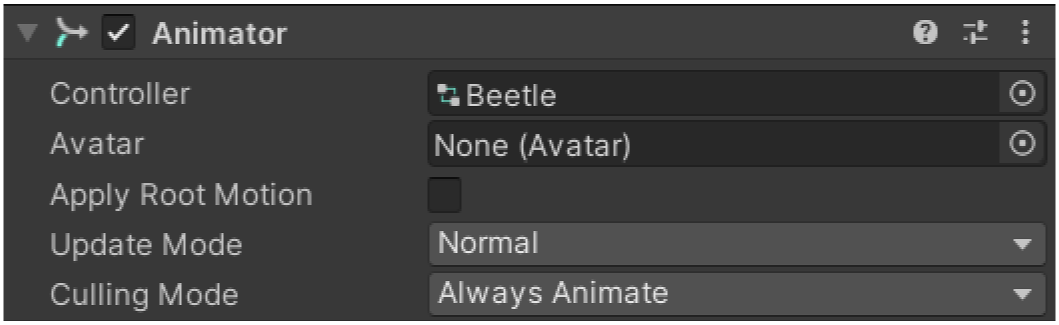


☑ Sprite Renderer

Sprite: Sprites_0

Color:

Next add the animator. Make sure the **Controller** property is **Bettle_Animations**.



☑ Animator

Controller: Beetle

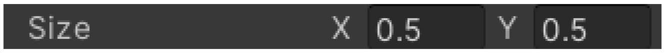
Avatar: None (Avatar)

Apply Root Motion: □

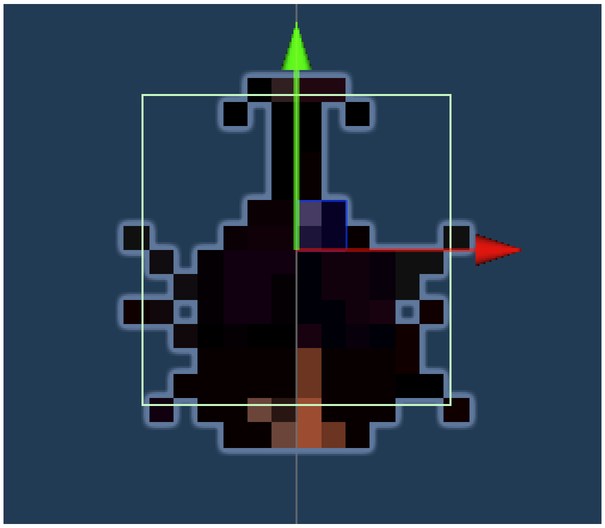
Update Mode: Normal

Culling Mode: Always Animate

6. Select the player object and adjust the size of the player’s **BoxCollider2D**.

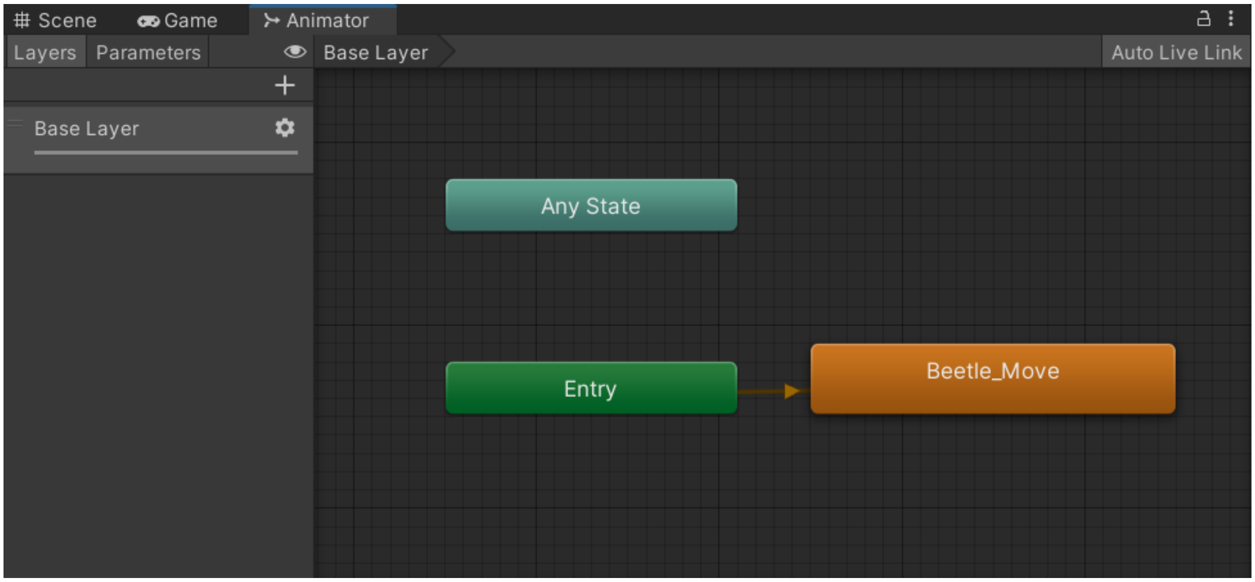


Size X 0.5 Y 0.5

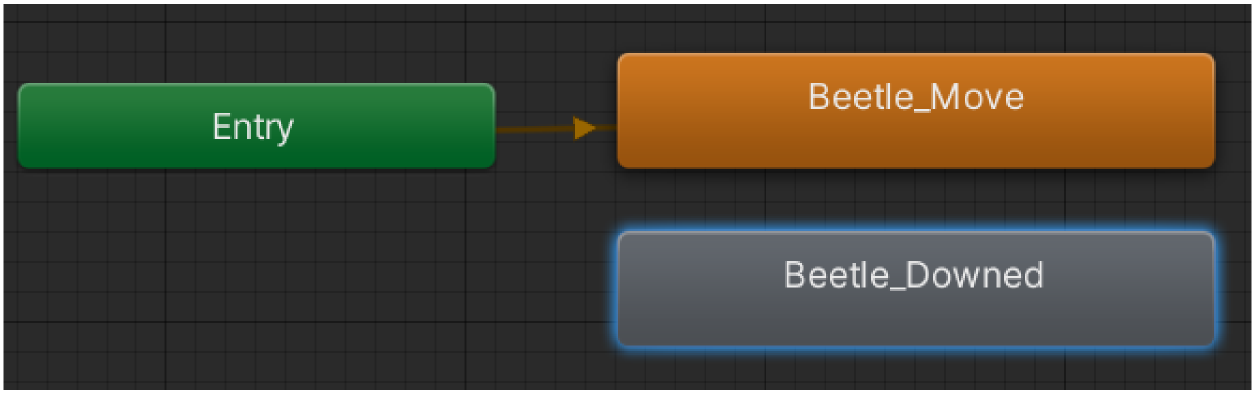


7. Double click on the **Beetle animation controller**, this will open up a window that we will use to control which animation plays for our character.

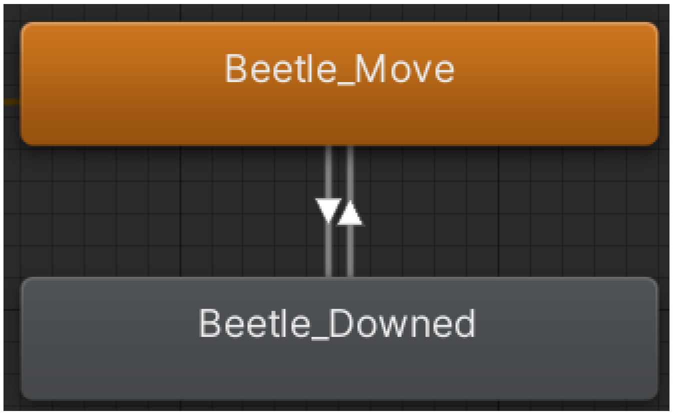
   By default the **Entry** state will be connected to an animation. Whichever animation the Entry is connected to will be the animation that is played at the start of the game.



8. Drag in the **Beetle_Downed animation clip**. This will create a new animation state called **Beetle_Downed**. Place the state right below the Beetle_Move state.

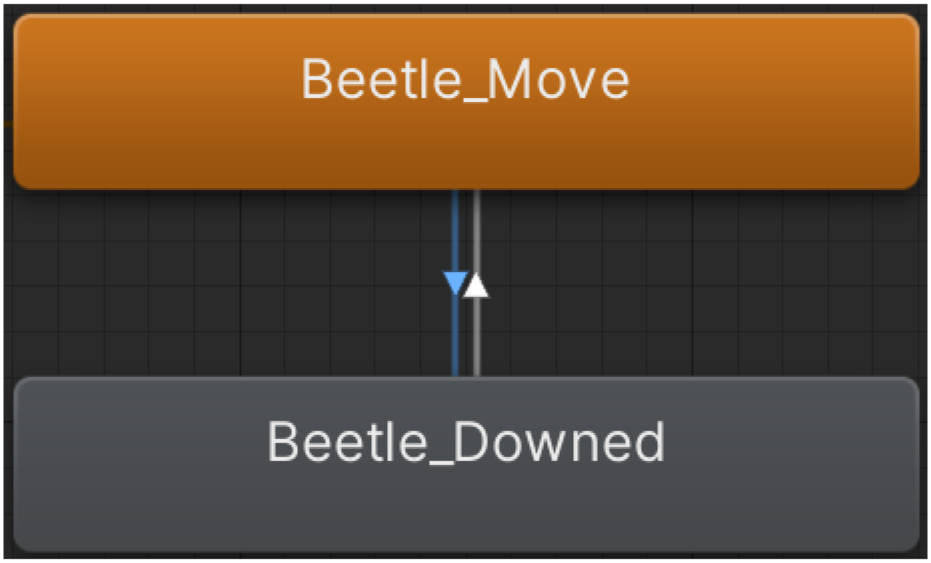


9. Right click on the Beetle_Move state and create a transition. Connect the transition to Beetle_Downed state. Do the same but connect Beetle_Downed to Beetle_Move state.

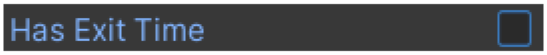


Transitions allow the animations states to transition into each other. Since they are connected Beetle_Move can transition into Beetle_Downed and vice versa. A state must have a transition in order to move into that state.

10. Click on transition from Beetle_Move to Beetle_Downed.



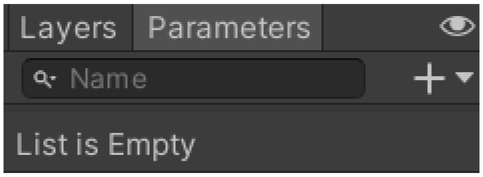
In the Inspector uncheck **Has Exit Time**.



Has Exit Time □

All this does it make it so our animations will instantly jump to the next state. By default, animations will try their best to blend together. This is great for 3D animations by really not needed for 2D animations.

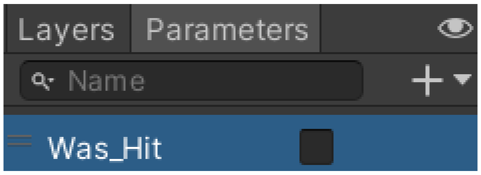
Unity will warn us that we need a condition for the transition to work. At the left side of the Animator tab, find the **Parameters** tab and click it.



List is Empty

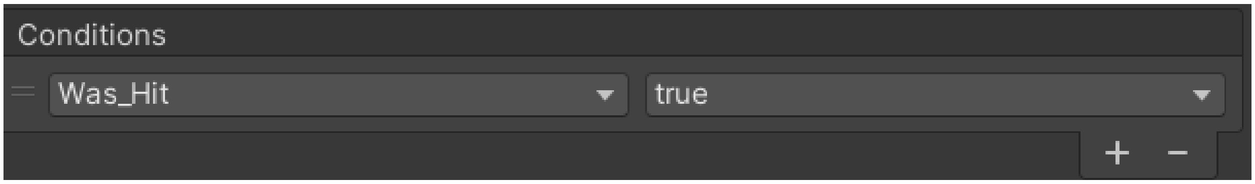
A parameter will allow us to determine when an animation will transition into another.

Hit the plus sign and create a new **Bool** parameter named **Was_Hit**.



Was_Hit □

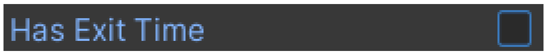
Go back to the transition, and add the condition:



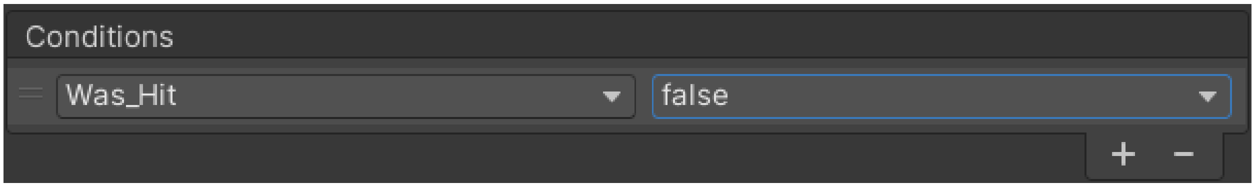
Conditions

Was_Hit true

Click on the other transition, **uncheck has exit time** and make the condition:



Has Exit Time □

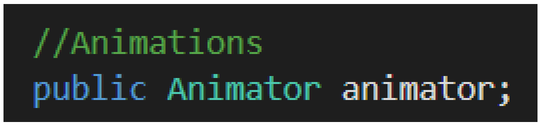


Conditions

Was_Hit false

In summation, we made it so the animation controller will transition from the Beetle_Move animation to the Beetle_Downed animation when the Was_Hit condition is true. And then it will transition back when the Was_Hit condition is false.

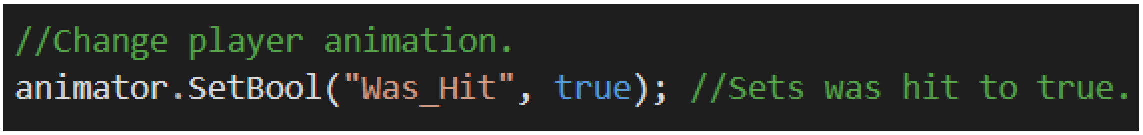
11. All we need to do now is set when the Was_Hit condition will be true or false in our code. In **Player** script, we add the variable:



```
//Animations
public Animator animator;
```

Animator is the animator component on the **sprite child object**.

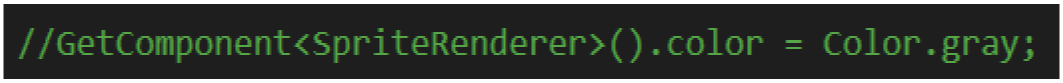
12. In the **OnTriggerEnter2D()** function add the code:



```
//Change player animation.
animator.SetBool("Was_Hit", true); //Sets was hit to true.
```

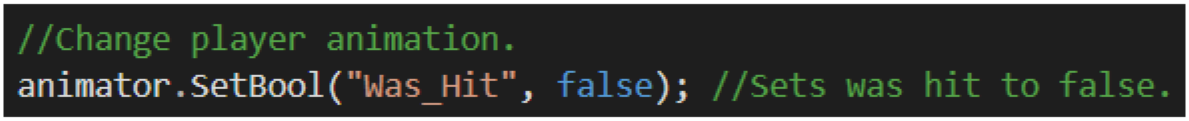
Since we know the player got hit by a shell, we can set the **Was_Hit** contition to **True**.

Let's also remove the color that will color the player gray.



```
//GetComponent<SpriteRenderer>().color = Color.gray;
```

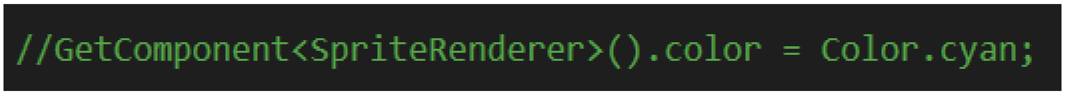
13. Then in the **StateDowned()** function find the code where the player revives and write the line:



```
//Change player animation.
animator.SetBool("Was_Hit", false); //Sets was hit to false.
```

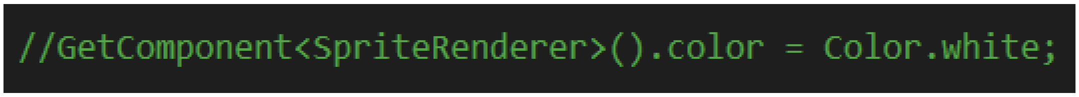
We know the player was able to hit the space bar enough, so we can tell the animator that the beetle should no longer be counted as being hit.

Let's also remove the color that will color the player cyan.



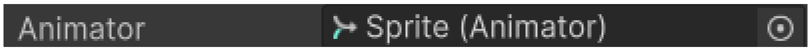
```
//GetComponent<SpriteRenderer>().color = Color.cyan;
```

14. Then in the **Update()** function we will remove the code that turns the color of our player to white.



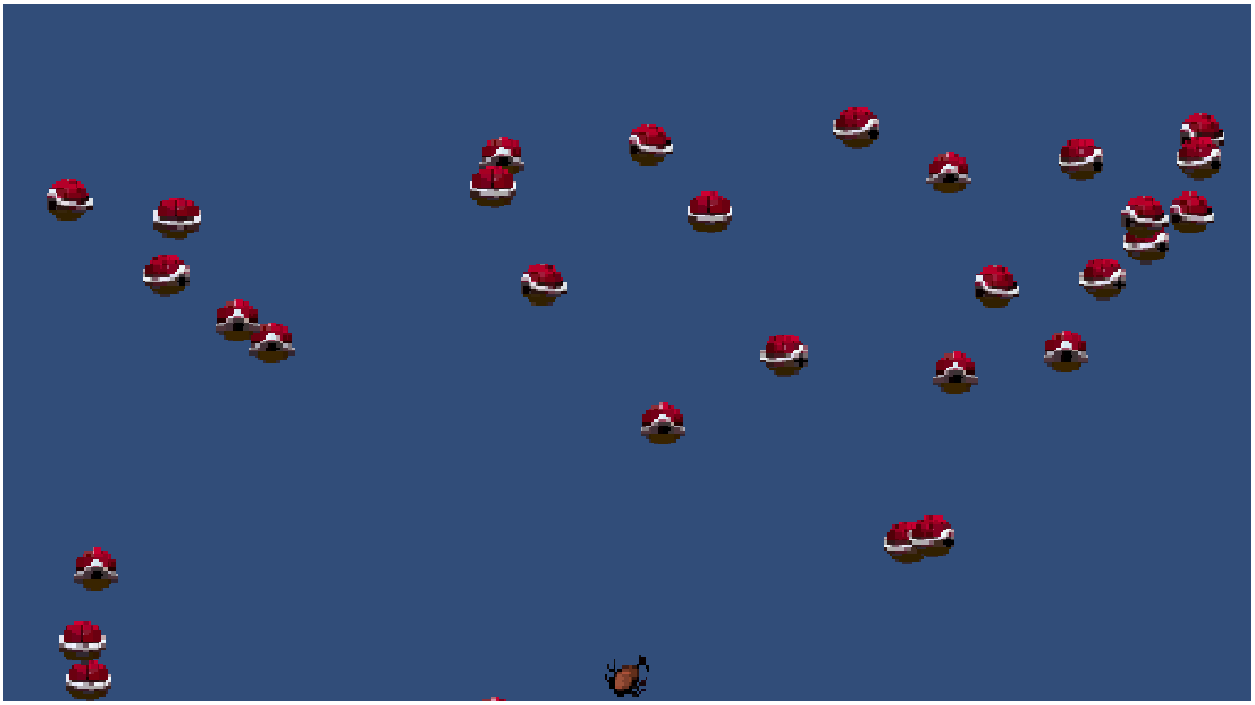
```
//GetComponent<SpriteRenderer>().color = Color.white;
```

15. Save the scripts, return to Unity. Hook up animator from the sprite to the player:



Animator | Sprite (Animator) ⊙

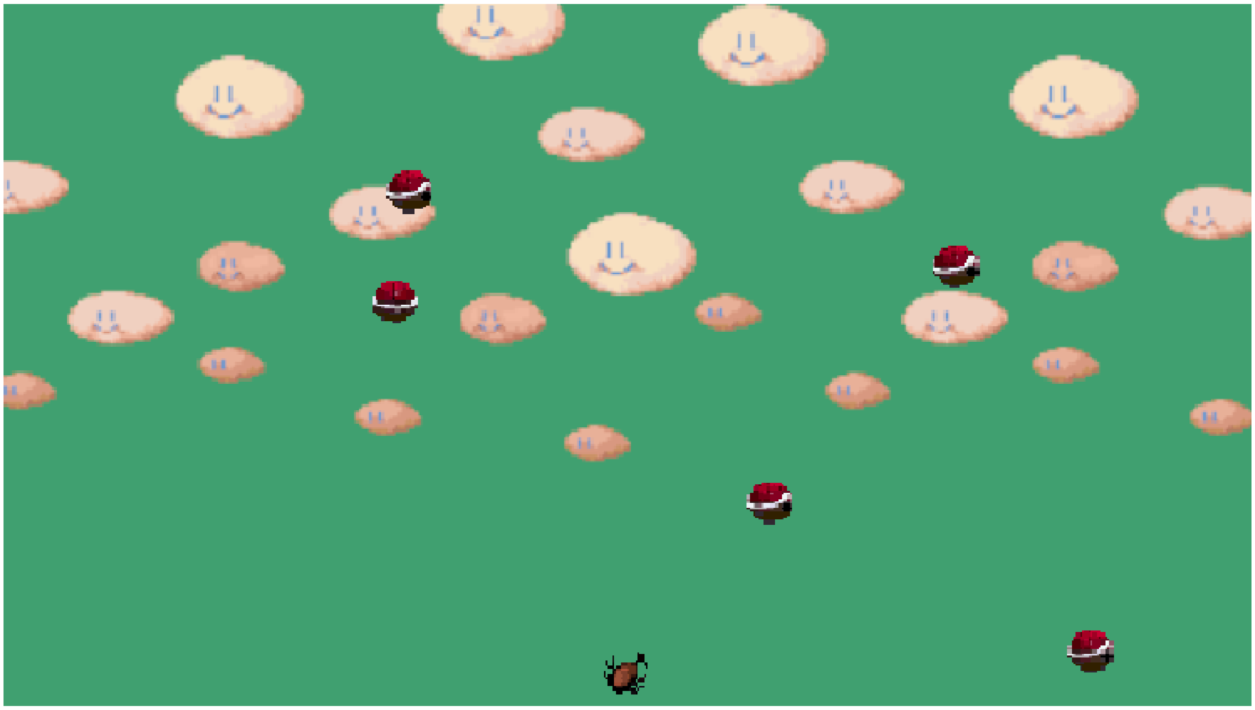
If we run the game now the player will have a moving animation but if the player gets hit, they will play the downed animation.



Background

1. Last but not least, let's add in a background to the game. All we need to do is drag in one of the background sprites and resize so that it takes up the full view of the camera. You may need to add additional background sprites so that you can fit it to the whole view of the

camera.



GitHub

1. Push the project to the GitHub repository.