The End of Asteroids
Detailed Instructions

Overview

In this project (the final increment of our Asteroids game development), you're making bullets split larger asteroids into smaller asteroids and making bullets destroy asteroids that are small enough.

After you finish the Week 4 material in the course, you might want to also add a timer and sound effects to the game.

Step 1: Make it so bullets split asteroids into smaller asteroids

For this step, you're making it so a collision with a bullet splits an asteroid into two smaller asteroids instead of destroying the asteroid. This is how the original Asteroids game actually worked.

1. Open the `Asteroid` script in Visual Studio.
2. In the `OnCollisionEnter2D` method, delete the code that destroys the asteroid.
3. Add code that cuts the scale of the asteroid in half in both `x` and `y`. Remember, you'll have to copy the `localScale` into a `Vector3` variable, make the changes, then copy your variable back into `localScale` to do this.
4. We also actually need to scale the collider to make sure screen wrapping works for smaller asteroids. Cut the radius of the Circle Collider 2D attached to the asteroid in half.
5. At this point, shooting asteroids should make them shrink. That's nice, but we actually want them to split into two asteroids with different velocities. Instantiate the `gameObject` twice, then destroy the `gameObject`. When you remove the original asteroid from the scene by destroying it, you'll be left with the two new (smaller) asteroids.
6. Unfortunately, the two new (smaller) asteroids aren't actually moving! That's because we get asteroids moving by calling the `Initialize` method. Unfortunately, we can't use that method because we're only allowed to provide a `Direction` value as the first argument when we call that method, and we want our new asteroids to move in a random direction rather than just left, right, up, and down.
7. Pull the block of code that applies the impulse force to the asteroid out of the `Initialize` method into a new public `StartMoving` method. The new method should have a single float argument for the angle at which to move the asteroid. Call the new method from the `Initialize` method to get the initial asteroids moving.
8. Now add code after you instantiate each of the smaller asteroids to call the `StartMoving` method with a random angle between `0` and `2 * Mathf.PI`. You'll need to change the code that instantiates the smaller asteroids to save the resulting game objects into variables so you can use the `GetComponent` method to get references to the `Asteroid` scripts attached to them. At this point, shooting an asteroid should yield two smaller asteroids moving in random directions.

9. It might look strange that the smaller asteroids can be a different color from the larger asteroid they're splitting from, but I like it that way, so I left that behavior in the game!

When you run your game, asteroids should split into two smaller asteroids with random velocities. Of course, they just keep getting smaller and smaller forever.

By the way, testing this was where I got really tired of my bullets going so slowly! I changed the force to get the bullets moving to 10 and reduced their life to 1 second to make it more fun for me. Feel free to do the same if you'd like.

Step 2: Make it so bullets destroy asteroids that are smaller than half the original size

For this step, you're making it so a collision with a bullet actually does destroy an asteroid if the asteroid is smaller than half the original size. Because we divide the scale of the asteroid in half for each collision, that means each full-size asteroid can only be split twice before it's destroyed by a bullet.

1. Open the `Asteroid` script in Visual Studio.
2. In the `OnCollisionEnter2D` method, add an if statement after you destroy the bullet. The if clause for your new if statement should destroy the asteroid if the `x` component of its `localScale` is less than 0.5; otherwise, you should split the asteroid into two smaller asteroids as you did in the previous step.

When you run your game, you should be able to still split larger asteroids but destroy asteroids that have been split to 1/4 their original size.

After Week 4 Material

Step 3: Add game timer

For this step, you're adding a timer that shows how long the player has been playing.

1. Add a Canvas to the Hierarchy window and rename the canvas HUD.
2. Change the UI Scale Mode in the Canvas Scaler component of the canvas to Scale With Screen Size using a 1280 by 720 Reference Resolution.
3. Add a Text - TextMeshPro component to the canvas and rename it ScoreText.
4. Change the characteristics of the Text component so that it's reasonably large, white text centered horizontally near the top of the screen.
5. Create a new HUD script and attach it to the HUD canvas.
6. Open the `HUD` script in Visual Studio.
7. Add a documentation comment at the top of the script.
8. Declare a field to hold the `TextMeshProUGUI` component, marking it with `[SerializeField]` so you can populate it in the Inspector. Remember, the `TextMeshProUGUI` class is in the `TMPro` namespace. Go populate the field in the Inspector.

9. In the `Start` method, set the `text` property of your `TextMeshProUGUI` field to `"0"`. When you run the game, 0 should be displayed near the top center of the screen. Of course, that doesn't change at this point!
10. Although you might think we should add a Timer component to the HUD, that doesn't work very well for a couple of reasons. First, our current Timer component counts down, not up. That's of course fixable, but we also need to know the current value of the game timer to display it in the game. We could also change our `Timer` script to provide that information, but it will actually be easier for us to handle the simple game timer ourselves in the HUD.
11. Declare a float field to store elapsed seconds, initializing the field to 0.
12. In the `Update` method, add `Time.deltaTime` to your elapsed seconds field. Set the `text` property of your `Text` field to the value of your elapsed seconds field, cast to an integer and converted to a string using the `ToString` method.

When you run the game, the timer should move up by 1 approximately every second.

Step 4: Stop game timer when ship is destroyed

Our game timer should really stop when the ship is destroyed since at that point the player is no longer playing the game. That's what you're adding in this step.

1. Open the `HUD` script in Visual Studio.
2. Add a boolean field to tell whether or not the game timer is running and initialize the field to true.
3. Change the code in the `Update` method to only update the timer and the timer text if the game timer is running.
4. Create a new public method named `StopGameTimer`. In the method body, change the field that tells whether or not the game timer is running to false.
5. In the `Ship` class, declare a field to hold the `GameObject` for the HUD, marking it with `[SerializeField]` so you can populate it in the Inspector. Go populate the field in the Inspector. It's actually pretty ugly having our ship know about the HUD. Don't worry, though, you'll learn a much better way to do this in the next course in the specialization!
6. In the `Ship` `OnCollisionEnter2D` method, add code that retrieves the `HUD` component attached to your HUD field and call its `StopGameTimer` method before destroying the ship.

When you run the game, the timer should stop when the ship is destroyed.

Step 5: Add sound effects

For this step, you're adding sound effects to the game. You'll be adding sound effects for shooting bullets, for when a bullet collides with an asteroid, and for when the ship is destroyed.

It's actually amazing how quickly sound effects get complicated. For example, we want to play a sound effect when a bullet hits an asteroid. Unfortunately, both of those objects are immediately destroyed, so even if we tell one of them to play the sound effect we won't hear it in the game.

I've solved this problem by providing you with an `AudioManager` class you can use to play the sound effects in the game.

1. Copy the AudioClipName.cs, AudioManager.cs, and GameAudioSource.cs files from the zip file into the Scripts folder for your project.
2. Add a new Resources folder to the Project window. This name has to be exactly that, starting with a capital R, or the code I provided to you won't work.
3. Copy 3 audio files for the given sound effects into that folder. Those files have to be named shoot, hit, and die or the code I provided to you won't work.
4. Next, you'll add the AudioSource that actually plays all the sound effects in the game. Right click in the Hierarchy window and select Create Empty. Change the name of the new game object to GameAudioSource. In the Inspector, click the Add Component button and select Audio > Audio Source. There's no need to assign a clip to the Audio Source because our audio manager will select the appropriate clips to play as the game runs. Finally, add the GameAudioSource script to the GameAudioSource game object.
5. Add code to the `Ship Update` method to call the `AudioManager Play` method when you shoot a bullet. You'll need to use the `AudioClipName` enumeration I provided to you for the argument in the method call.
6. Add code to the `Asteroid OnCollisionEnter2D` method at the beginning of the if body (where you know the asteroid collided with a bullet) to call the `AudioManager Play` method with the appropriate `AudioClipName`. You should now hear that sound effect when you hit the asteroids with a bullet.
7. Add code to the `Ship OnCollisionEnter2D` method to call the `AudioManager Play` method just before you destroy the ship.

When you run your game, you should have full gameplay, with all three sound effects.

Congratulations, you're done with this project and the game!