Tutorial 8 – Collision Detection

Before starting this tutorial, you must complete the last one and have it working. Failure to do so will result in not being able to complete these steps. It is your choice but is recommended to crate a new branch each tutorial.

In this tutorial we will be constructing a Collision Detection singleton class. This class will enable us to test if two objects have collided. There are several techniques that can be used, Bounding Box, Bounding Circle and Per-Pixel. This tutorial will detail Circle collision. Marks will be awarded in your assignment for implementing Bounding Box collision and Per-Pixel collision detection.

Before we continue, it's important you understand what a singleton class is. Some of you may have skipped over the term, some may know what it is already, and others may think, "what on earth is a singleton class?". A singleton class is a class that provides one unique instance across a system. So, when we create an instance in the program it will return the same instance created earlier, this limits the number of objects that can be created. These are constructed differently form your standard classes. They require:

- A public static function
- A private static pointer
- And a private singleton constructor and copy constructor

1. Firstly, create a new class called 'Collisions'
2. In the header add a forward declaration to the Character class and include the Commons.h file, we will need to edit that shortly.
3. Add the following public functions:

```
~Collisions();

static Collisions* Instance();

bool Circle(Character* character1, Character* character2);
bool Box(Rect2D rect1, Rect2D rect2);
```

You'll most likely have errors on the Rect2D types, don't worry about them just know we fix them we add them to the Commons file.

4. As mentioned before the constructor is NOT public so add it to the private section along with the following private static variable:

```
Collisions();

static Collisions* m_instance;
```

   The static keyword states that only one of these variables will be created regardless of how many times you instantiate the Collisions class.

5. Before we move on to the Collisions source file, jump over to the Commons file and we will complete the Rect2D struct so we can use it in the Box collision code. The struct will need 4 floats to begin with: x, y, width, and height. Next, add a constructor to the struct like so:

```
Rect2D(float x_pos, float y_pos, float width, float height)
{
	x = x_pos;
	y = y_pos;
	this->width = width;
	this->height = height;
}
```

You may be wondering what the keyword 'this' is for and why it uses the arrow pointer notation? If you remove the 'this->' from width, you will see a colour change in the variable and both width's will become grey. The compiler cannot distinguish between the two different uses of the same name. There are numerous uses for the 'this' keyword but in our case, we are using it as member variable initialisation. More information on 'this' can be found here: This pointer.

1. Now move to the Collision.cpp file and include the Character header and the Collisions header if it isn't automatically added. Directly below these add the following:

```
//initialise instance to nullptr
Collisions* Collisions::m_instance = nullptr;
```

2. Go ahead and add the constructor and destructor. The constructor can be left empty, the destructor though, will simply need to set m_instance to nullptr.
3. Next, add the method body for the Instance function. This method checks if the static pointer (m_instance) has been set up. If not, complete the set up and then simply return it.

```
if(!m_instance)
{
	m_instance = new Collisions;
}

return m_instance;
```

4. All that remains is to complete the collision detection functions. These will return a Boolean value indicating whether a collision occurred. First, we will look at Circle collision, the implementation required for this will be detailed below.
    a. Firstly, we calculate the vector that separates the two objects

```
Vector2D vec = Vector2D((character1->GetPosition().x - character2->GetPosition().x),
       (character1->GetPosition().y - character2->GetPosition().y));
```

    b. Next, calculate the length of the vector. By creating a variable of type double that is equal to the square root of the positions; two x positions times together, plus the two y positions times together.

```
double distance = sqrt((vec.x * vec.x) + (vec.y * vec.y));
```

    c. We then need to get the collision radius of each character and accumulate them.

```
double combined_distance = (character1->GetCollisionRadius() + character2-
>GetCollisionRadius());
```

    Don't worry about the errors, we will construct this function shortly.

    d. Finally, check whether the distance is shorter than the accumulated collision radii.

```
return distance < combined_distance;
```

5. At this point you should only have 2 errors regarding the GetCollisionRadius functions. So, open the Character.h file and add the following:

    protected variable:

```
float m_collision_radius;
```

    public function:

```
float GetCollisionRadius();
```

6. In the Character source file, in the constructor, set the size of the collision radius. I have set this to 15, which seems a good fit for the Mario image. Feel free to play with the size to see what you think fits best.

```
m_collision_radius = 15.0f;
```

    Next, add the body for the GetCollisionRadius, this needs only to return m_collision_radius.

7. Build your program and it should be error free.

Before making use of the new class, if you haven't already, add a second player (Luigi) to your game. For this you just need to replicate what you did when creating Mario and ensure to set up different controls such as WASD.

To use the new functionality of your new class all you need so is add the following code to the GameScreenLevel1 Update function, don't forget your include at the top of the file. This should be adapted to use your character names as the passed parameters and the relevant collision function you wish to call. It is a good idea to test out both functions once compete. Call them both and cout "box hit" or "circle hit" when a collision is encountered and see how they differ in the console window.

```
if(Collisions::Instance()->Circle(mario, luigi))
{
	cout << "Circle hit!" << endl;
}
```

Next, we will work on the box collision and make use of the Rect2D struct we made earlier. Return to the Collision.cpp file and in the Box function, add the following complex conditioned if statement:

```
if(rect1.x + (rect1.width/2) > rect2.x &&
	rect1.x + (rect1.width/2) < rect2.x + rect2.width &&
	rect1.y + (rect1.height/2) > rect2.y &&
	rect1.y + (rect1.height/2) < rect2.y + rect2.height)
{
	return true;
}
return false;
```

This may seem complex; however, it is merely checking if either box overlaps the other.

For this to be of any use to us we will need to go back to our Character and add the following public function. As this function is just returning a newly created Rect2D struct we can create this function as an inline function. This means the functionality is added in the header. You should only do this for very small functions otherwise compile times will lengthen. Notice how the Rect2D is given the character position and the textures width and height. You will need to include Texture2D in the header if not already done so and remove it from the source file.

```
Rect2D GetCollisionBox(){ return Rect2D(m_position.x, m_position.y,
	m_texture->GetWidth(), m_texture->GetHeight()); }
```

Similarly, to the Circle call used earlier, to use Box collision adapt the following call. This should put within your GameScreenLevel1 Update function to determine if collisions between characters has occurred.

Additional Work

You have been given two approaches to collision detection. Both have their benefits. Do some research to determine which is most appropriate in different circumstances.

Improvements for Circle collision

Notice how Circle takes in two-character functions? This limits its usability as you can only check for circle collisions between Characters, not with any other objects. As an improvement it would benefit from taking two positions and two radii. Maybe a new struct would be the way to go here.

Adapt the Circle function so that it can be used objects other than Characters and create the relevant functions within Character to facilitate this. Follow the example of how Box collision is constructed.