Tutorial 7 – Creating a Character Class

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 create a new branch each tutorial.

In this tutorial we will be constructing a Character class. This class will enable us to draw an image on the screen and respond to input from the keyboard. You will require a new image, which will be located on Blackboard. Download 'Mario.png' and put it in to your Images folder. (There are some other useful images you may want to use for your game on Blackboard too).

1. Create a new class called 'Character' (both header and source file).
2. In the header,
    a. Include SDL, iostream and the commons file.
    b. Forward declare Texture2D.
    c. Add the following *protected* variables:

```
SDL_Renderer* m_renderer;
Vector2D m_position;
Texture2D* m_texture;
```

    d. Add the following *public* functions:

```
Character(SDL_Renderer* renderer, string imagePath, Vector2D start_position);
~Character();

virtual void Render();
virtual void Update(float deltaTime, SDL_Event e);
void SetPosition(Vector2D new_position);
Vector2D GetPosition();
```

3. Next, move into the character source file
    a. Include both the Character header and the Texture2D header.
    b. Copy over your functions from the header file and add the class name before the functions as you have previously (*Character::~Character()*).
    c. In the constructor, you will need to set up the m_renderer, m_position and the m_texture using the file path passed in. the first two are simple enough, the m_texture requires you to make a new instance. If you are at a loss for how to do this, look at the GameScreenLevel1 SetUpLevel function. You will also need to add some error catching just like what is done in the SetUpLevel function, however, it will have some minor differences:
        i. The LoadFromFile will not take a direct path, instead pass the imagePath parameter taken in by the constructor.

ii. Secondly, we do NOT need to return a value, so no return false is needed.

d. Next, the destructor. This needs only set the m_renderer to nullptr.

e. The Render function consists of calling the texture render function using the arrow notation. For now, pass m_position and SDL_FLIP_NONE. You will come back to this later and make some functional changes.

f. For Update, we need to check for user keyboard input.

i. Use a switch statement as demonstrated in tutorial 3 – Events. You will only need to deal with input that is relevant to this class. Ensure you read through the additional work of tutorial 3 if you haven't as (e.key.keysym.sym) is needed in the switch.

ii. If the left arrow is pressed, alter the x component of the position variable by -1. (for example: m_position.x -= 1;)

iii. If the right arrow is pressed, alter the x component of the position variable by 1.

Here is the list of key code values again: SDLKey (libsdl.org)

g. Finally, the Get and Set position functions. The SetPosition should simply set m_position to the new_position and the GetPosition returns the m_position.

4. Build your program, you should have no errors at this point. If you do so, do not continue until you are error free. Retrace your steps to check if you have missed anything.

Now that we have our character class, we need to create a character object in our level class and allow Updates and Rendering occur.

1. Open your GameScreenLevel1 header file and include the Character.h file.
2. Forward declare the Character class and add a private variable called my_character.

```
Character* my_character;
```

3. In the SetUpLevel function, add the following code:

```
//set up player character
my_character = new Character(m_renderer, "Images/Mario.png", Vector2D(64, 330));
```

4. Next, in the Update function, add a call to the characters update function, like so:

```
//update character
my_character->Update(deltaTime, e);
```

5. Similarly, in the render function, make a call to the characters render function (this requires no parameters).

6. Now in the destructor, delete the character and set to nullptr just as you have with the background texture.
7. Build and run the program.

At this point you should have a Mario sprite that moves left and right when the relevant key is pressed. However, you will notice that there are a couple of issues. The first being that Mario does not move from the top right of the screen or face in the direction that he is supposed to be moving, and the second, once he is unpinned from that position is that the movement does not seem as responsive as it should be. We will deal with these issues next.

1. Firstly, let's unpin Mario from the top of the screen.
    a. In Texture2D.cpp, in the render function, we need to make some small changes. First the tenderLocation states 0, 0, m_width, m_height. It is this 0, 0 that is pinning Mario in place. Change this to new_postion.x, new_position.y. Run the program and make sure the change worked.
    b. Next, in the same function, the SDL_RenderCopyEx needs a slight change. The very last parameter states SDL_FLIP_NONE. Change this to 'flip' the parameter name taken in by the function. For now, it won't make any changes, but it will shortly. Just without this change it will override what flip we ask it to do.
2. To fix the facing direction issue we need first to go to the Commons.h file and add the following enum:

```
enum FACING
{
        FACING_LEFT,
        FACING_RIGHT
};
```

3. Now, back in the Character header, add a private variable called m_facing_direction of FACING type.
4. Then, in the source file:
    a. In the constructor set m_facing_direction to face the direction that the image is. This is to the right.
    b. Then, in the update function, we need to set the facing direction dependant on the key press. So, if SDLK_LEFT is trigged, along with our movement being changed, set m_facing_direction to FACING_LEFT and the same for the right.

c. Finally, in the Render function, make the following changes:

```
if(m_facing_direction == FACING_RIGHT)
{
	m_texture->Render(m_position, SDL_FLIP_NONE);
}
else
{
	m_texture->Render(m_position, SDL_FLIP_HORIZONTAL);
}
```

Build and run again to ensure the changes have worked. The sprite should now face in the correct direction to that it is moving. Next, we will look at the seemingly unresponsive key press. The issue is that we are polling for the input, and movement only happens once the keypress message has filtered through to our Update() function. The solution to this is to start the character moving when the key is pressed and only stop it moving once the key is released. This means that our update will continually move the character without waiting for the event message.

1. In the Character header file add the following protected variables:

```
bool m_moving_left;
bool m_moving_right;
```

2. And the following protected functions:

```
virtual void MoveLeft(float deltaTime);
virtual void MoveRight(float deltaTime);
```

3. Next, in the character source file:
    a. Set both the new variables to false inside the constructor.
    b. Int the Update function, add the following code before polling for events

```
if(m_moving_left)
{
	MoveLeft(deltaTime);
}
else if(m_moving_right)
{
	MoveRight(deltaTime);
}
```

    c. Also. in the Update function, remove the code for moving the player and changing the facing direction, these will be handled by our functions MoveLeft and MoveRight.
        i. Instead, set m_moving_left to true if the arrow key is pressed and the same for the right.
        ii. Next, add a new case to the switch statement for key up events. You should currently have a case for key down followed by an internal switch statement, after the closing bracket, add the break keyword then add your

new case for key up events. This will contain another internal switch statement just like the on for key down events, however, set the variables to false.

d. Now we need to write the functions for MoveLeft and Right, declare them and complete with the following:
    i. Set the facing directions
    ii. Alter the position

4. Build and run the program. You may notice that Mario acts more like Sonic and zooms off. This is something for you to work on in the additional work section along with jumping and gravity and setting up a child class of Character with a Mario class.

Additional Work

Instead of adjusting the character position with a magic number, create a constant variable in the Constants header file called MOVEMENTSPEED and use this, play around with a speed you think works for your game, I set mine to 50.0f and changed the calculation to use deltaTime like so:

```
m_position.x += deltaTime * MOVEMENTSPEED;
```

Add a virtual function to the Character class to add gravity. This is simply a value multiplied by the delta time added to the y component of the position variable. Make sure that the minimum position on the y is the bottom of the screen, otherwise your character will continue to fall indefinitely.

```
void Character::AddGravity(float deltaTime)
```

Hint: Consider using an if else statement if the m_position + the height of Mario (64) is less than or equal to the screen height then m_position.y += GRAVITY (a constant variable for you to determine, I chose 260.0f) * deltaTime. The else will require setting up the jumping functionality first, once done set m_can_jump to true in the else statement.

Next you need to add the functionality to allow Mario to jump. You will require a couple of variables to keep track of whether he is jumping or not and what force to apply to the upward motion. Add the following to the Character header under the protected category.

```
bool m_jumping;
bool m_can_jump;
float m_jump_force;
```

Next, locate the Update function in the Character.cpp file. Before dealing with the left / right actions add the following code. Comments have been added above each line of code to explain the behaviour.

```
//deal with jumping first
if(m_jumping)
{
	//adjust position
	m_position.y -= m_jump_force * deltaTime;

	//reduce jump force
	m_jump_force -= JUMP_FORCE_DECREMENT * deltaTime;

	//is jump force 0?
	if (m_jump_force <= 0.0f)
		m_jumping = false;
}
```

You will notice there is a constant variable called JUMP_FORCE_DECREMENT. This needs to be set up in the constants file. Do this now and set it to 400.0f. Whilst you are in this file, also create a variable called INITIAL_JUMP_FORCE set to 600.0f. This will come in useful soon.

You can now call the AddGravity function and the else statement mentioned earlier now becomes relevant. This will allow the character to jump again once it has hit the bottom of the screen and stop multiple jumps.

Finally, we need to create a virtual Jump function and call it whenever a button is pressed. Create a prototype for Jump within the Character header. Now go back to the Character source file and add the following:

```
void Character::Jump()
{
	if (!m_jumping)
	{
		m_jump_force = INITIAL_JUMP_FORCE;
		m_jumping = true;
		m_can_jump = false;
	}

}
```

Notice that we set m_jump_force to the initial value, which is added to the current Y position in the Update function. This value is decreased each frame. Also, m_can_jump is set to false, stopping multiple jumps.

The final thing to do is to call the Jump function upon a key press and if m_can_jump is true. I will leave this in your capable hands to finish.

One last task remains. We have written a character class which functions correctly, and we can now move around the game world. However, what if we need a second player? Let's call this player Luigi. If Luigi is of type Character, then he will do exactly what Mario does. So, every time player 1 presses a key, both Mario and Luigi will respond. The reason for this is that all the key events are within the Character class. To solve this problem, we need to create a CharacterMario class which inherits from the Character class, and a CharacterLuigi class which also inherits from the Character class. Once we have this, we need to override the Update function in both new classes with the key events and remove this functionality from the Character base class.

To inherit functionality from another class in C++, you simply use the : followed by the class to inherit from when you name your class. Here is an example of the header and source file:

Header:

```
class CharacterMario : public Character
```

Source file:

```
CharacterMario::CharacterMario(SDL_Renderer* renderer, string imagePath, Vector2D
start_position) : Character(renderer, imagePath, start_position)
```

Don't forget to include the Character header file.

I have supplied the update function for the Mario character below, but it is up to you to create the CharacterMario class and to fix up the calls from within GameScreenLevel1 and remove the old code from Character.cpp Update function (this will just be the switch statement). When adding new code to GameScreenLevel1, most of the code is identical to how you created my_character:

- Add pointers in the header for each character
- Then add them and delete them in the source file. Keep in mind when setting them up in the SetUpLevel function to add them as a new appropriate class type. E.g. new CharacterMario.

Once you have CharacterMario functioning correctly, create yourself a CharacterLuigi class, which responds to different keys. There is a “Luigi.png” image on blackboard for you to use.

Remember to fix any issues before counting this tutorial as complete.

```
switch (e.type)
	{
	case SDL_KEYDOWN:
		switch (e.key.keysym.sym)
		{
		case SDLK_LEFT:
			m_moving_left = true;
			break;
		case SDLK_RIGHT:
			m_moving_right = true;
			break;
		case SDLK_UP:
			if (m_can_jump)
			{
				Jump();
			}
		}
		break;

	case SDL_KEYUP:
		switch (e.key.keysym.sym)
		{
		case SDLK_LEFT:
			m_moving_left = false;
			break;
		case SDLK_RIGHT:
			m_moving_right = false;
			break;
		}
		break;
	}

	Character::Update(deltaTime, e);
```