Tutorial 10 The POW Block

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 adding the POW block. This brings together a few of the concepts already covered, such as level map and collisions, whilst introducing new concepts like sprite sheets and screen shakes. As ever, ensure that you have completed the previous tutorials and have no errors otherwise this tutorial will add more confusion.

To begin, you will need to download the “PowBlock.png” file from Blackboard. Next, we need to create two new files for the POW block. You should know how to do this by now, so go ahead and create yourself a PowBlock class. The PowBlock does not inherit from any other class.

The PowBlock header file will require the following public functions, while adding them it should become apparent which includes and forward declarations are needed:

1. A Constructor which takes a pointer to the SDL_Renderer and a pointer to the level map.
2. A Destructor.
3. A Render() function.
4. A GetCollisionBox() function that returns a Rect2D.
5. A function called TakeHit() which returns void.
6. And a function called IsAvailable() which returns a boolean.

You will then need to add the following private variables:

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

float m_single_sprite_w;
float m_single_sprite_h;
int m_num_hits_left;
```

Still in the header we can modify the IsAvailable and GetCollisionBox prototypes to be inline functions. The IsAvailable will look like such:

```
bool IsAvailable() { return m_num_hits_left > 0; }
```

If the m_num_hits_left is greater than 0, this function will return true, meaning the POW block is available; otherwise, it will return false.

Complete the inline function for GetCollisionBox. Remember, this returns a Rect2D expecting x and y positions, the width of the sprite and the height of the texture. If lost on how to pass the height of the texture look at the below snippet at the m_single_sprite_h.

Now open the cpp file and create stubs (function bodies) for each that requires them. The only one to not require a stub are the two you made inline.

1. In the constructor, add the following:

```
std::string imagePath = "Images/PowBlock.png";
m_texture = new Texture2D(renderer);
if(!m_texture->LoadFromFile(imagePath.c_str()))
{
       std::cout << "Failed to load texture." << std::endl;
       return;
}

m_level_map = map;
m_single_sprite_w = m_texture->GetWidth() / 3; //there are three images in this sprite sheet in a row
m_single_sprite_h = m_texture->GetHeight();
m_num_hits_left = 3;
m_position = Vector2D((SCREEN_WIDTH * 0.5f) - m_single_sprite_w * 0.5f, 260);
```

If you look at the image you will notice that there are 3 images. This is different from the other sprites you have used so far, which all had single images in each file. The PowBlock.png is a spritesheet. Granted it’s a small one, but it’s a spritesheet nonetheless. Therefore, we now need to have variables, which tell us how big an individual sprite is.

Notice in the code snippet above that we calculate the width of a single image by dividing the length of the spritesheet by three. We have three images so that makes sense. The height of a single sprite however remains the same as the height of the spritesheet. This is because we only have one row of sprites. If we were to add another row of sprites this would need to be adapted.

2. The destructor requires the following:
   a. M_renderer being set to nullptr.

   b. Delete the m_texture and set the pointer to null.
   c. Set m_level_map to nullptr.

The reason we don't delete the memory assigned for mRenderer and mLevelMap is that other classes are using this data. If we were to delete them the program would crash at the point, we remove the PowBlock. mTexture is specific to our class and nothing else accesses it. IT is safe to delete the memory used here.

3. In the TakHit function we need the following:
   a. Decrease the value of m_num_hits_left by 1.
   b. Check if the m_num_hits_left is less than or equal to 0. If so, then do the following:

```
m_num_hits_left = 0;
m_level_map->ChangeTileAt(8, 7, 0);
m_level_map->ChangeTileAt(8, 8, 0);
```

Here we are ensuring that m_num_hits_left gets set to zero, but the interesting part is that we are altering the level map. We call the ChangeTileAt() function and pass through the location where the POW block is located and make it so the space can be passed through. Mario should now be able to jump through this location.

If you did not manage to complete the additional work form the last tutorial, this is to help you understand what is needed. The function as per the prototype you were given, takes three parameters, a row, column, and a new value. You can see above we pass in 8,7,0. The first two are the row and column indexes and the last the new value, we are setting what was 1 (which took collisions) to zero, which our character can pass through. All our function need do is take the array indexes passed, m_map[row][column] and set them to the new value given.

4. The final function for us to look at is the Render() function. This is a little different from those we have written before and requires some changes to the Texture2D class as well.
   a. The first thing to do is to check if m_num_hits_left is greater than 0. If not, we don't want to draw anything.

b. So now we know that there are hits remaining, we need to determine which sprite on the sprite sheet to draw. Add the following code, which is well commented.

```
//get the portion of the sheet we want to draw
       int left = m_single_sprite_w * (m_num_hits_left - 1);

       //                          xPos,  yPos, sprite sheet width,sprite sheet height
       SDL_Rect portion_of_sprite = { left, 0, m_single_sprite_w, m_single_sprite_h };

       //determine where to draw it
       SDL_Rect dest_rect = {
              static_cast<int>(m_position.x), static_cast<int>(m_position.y),
              m_single_sprite_w, m_single_sprite_h
       };

       //draw the sprite
       m_texture->Render(portion_of_sprite, dest_rect, SDL_FLIP_NONE);
```

This is how sprite sheets work. We use a variable that keeps track of which frame we wish to draw and then use that variable to get the correct portion of the sprite sheet. In our case m_num_hits_left is not only a count for how many hits the block can take, but also our frame to draw. You will notice the static casts performed on the x and y position above too. You could do this like so (int)(m_position.x) but thought best to show you the full way first.

You will have errors here in the Render texture call, don't panic, we'll sort those now. As you have probably noticed, Texture2D does not have a Render function which takes two Rect2D types as parameters. Open Texture2D.h and add the following prototype:

```
void Render(SDL_Rect src_rect, SDL_Rect src_dest, SDL_RendererFlip flip, double angle
= 0.0);
```

Now, over in the Texture2D source file, add the stub for the new function and populate with the following:

```
SDL_RenderCopyEx(m_renderer, m_texture, &src_rect, &src_dest, angle, nullptr, flip);
```

You should now have two Render functions, don't worry about the wrong one being called, the complier will now from the parameters passed.

If you run your program at this point it should compile. Unfortunately, the PowBlock is not showing. This is because we are yet to create a variable of PowBlock type in GameScreenLevel1 and render it. We will do this next.

1. Open GameScreenLevel1.h and forward declare PowBlock.

2. Create a public function called UpdatePOWBlock which returns void and takes no parameters.
3. Add a private pointer of PowBlock type called m_pow_block.
4. Open find the GameScreenLevel.cpp and add the stub for the new function, don't forget to include PowBlock.h.
5. In the SetUpLevel function add the following to create the Pow Block:

   ```
   m_pow_block = new PowBlock(m_renderer, m_level_map);
   ```

6. Next, in the destructor delete the memory used and set nullptr.
7. In the Render function you need to call the Pow Blocks own Render function just like how you have with Mario.
8. Back to the new stub for UpdatePOWBlock
   a. Check if there is a collision instance between this and the Mario character. You will need to get each objects collision box as the Box collision only accepts two Rect2D parameters.
   b. Within this if statement, do an additional check to see if the Pow block is available.
   c. If there is a collision and the Pow block is available, then the following code needs to run:

      ```
      //collided while jumping
      if(mario->IsJumping())
      {
              DoScreenShake();
              m_pow_block->TakeHit();
              mario->CancelJump();
      }
      ```

      *Note: As you can see there are a couple of functions here that do not currently exist. The DoScreenshake function will be explained below, but the two character functions: (IsJumping and CancelJump); you are expected to code yourself. These should be public functions within the Character class, these could be inline functions as one is simple return and the other sets a variable to false.
9. The final job remaining for the PowBlock is for us to call UpdatePOWBlock from the GameScreenLevel1 Update function. Do this now.

The last thing to do is to sort the screen shake functionality. For the moment comment out the call to DoScreenshake and compile the program. It should build without errors. Move your character around. See if you can jump onto the POW block, and if you can jump up and hit it from below. You should see the POW block decrease in size with each hit until it is completely gone, at which point you should no longer be able to land on the space where the POW block used to be.

Ok, let's put a screen shake in. Open the GameScreenLevel1.h file and do the following:

1. Private variables

   ```
   bool m_screenshake;
   float m_shake_time;
   float m_wobble;
   float m_background_yPos;
   ```

2. Add a private prototype for DoScreenshake(), which returns void and takes no parameters.
3. Over in the GameScreenLevel1.cpp add the stub for DoScreenshake and populate with the following:

   ```
   m_screenshake = true;
   m_shake_time = SHAKE_DURATION;
   m_wobble = 0.0f;
   ```

   I have set SHAKE_DURATION to 0.25f in the Constants.h, but go with what works for your game.
4. Locate the SetUpLevel function and just below where you create the Pow Block:
   a. Set m_screenshake to false.
   b. Set m_background_yPos to 0,0f.
5. Then, in the Update, add the following code to the top of the function:

   ```
   /*
    * do the screen shake if required
    */
   if(m_screenshake)
   {
   	m_shake_time -= deltaTime;
   	m_wobble++;
   	m_background_yPos = sin(m_wobble);
   	m_background_yPos *= 3.0f;

   	//end shake after duration
   	if(m_shake_time <= 0.0f)
   	{
   		m_shake_time = false;
   		m_background_yPos = 0.0f;
   	}
   }
   ```

   In the event of m_screenshake being set to true this portion of code will run. It uses a sin() function to create the wobble. Each frame we increment the m_wobble variable, which keeps the value returned from sin() continually going along the wave. This value is used to move the y position of the background up / down. After the duration has passed the screen shake will be cancelled and normal gameplay resumes.

6. Now adapt the Render function. When drawing the background, we need to use the new m_background_yPos variable.

```
//draw the background
m_background_texture->Render(Vector2D(0, m_background_yPos), SDL_FLIP_NONE);
```

7. Finally, build and run your game to test everything has been implemented properly. (Ensure you uncommented the DoScreenShake call in UpdatePOWBlock).

In your GameScreenLevel1.h file, we are going to add a new method and variable to be able to use this LevelMap class. This will allow us to use the LevelMap data for Level 1 to control our characters movement. Add the following:

1. A new private method:

   ```
   void SetLevelMap();
   ```

2. A new private variable:

   ```
   LevelMap* m_level_map;
   ```

   Don't forget your includes.

3. You then will need the set the variable above to nullptr in the GameScreenLevel1 constructor.

Now you should implement the new SetLevelMap() method in your GameScreenLevel1.cpp file. Ensure you understand this method as you implement it. You should then call SetLevelMap() in your set up code for GameScreenLevel1, this will be the same place where you initialise your character. Ensure this is copied correctly as mistyping this map leads to common errors:

```
int map[MAP_HEIGHT][MAP_WIDTH] = {  { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
                                     { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
                                     { 1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1 },
                                     { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
                                     { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
                                     { 0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0 },
                                     { 1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1 },
                                     { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
                                     { 0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0 },
                                     { 1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1 },
                                     { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
                                     { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
                                     { 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 } };

//clear any old maps
if(m_level_map != nullptr)
{
	delete m_level_map;
}

//set the new one
m_level_map = new LevelMap(map);
```

When you look at the 0 and 1 values in the collision map above, you will see that there are a couple of 1s 5 rows up from the bottom, in the centre. These are for the POW block that we will add in a future tutorial. For now, you can either leave them as 1s meaning that your character will collided with that area of space, or set them to 0s and allow your character to pass through that area.

Next, we need to edit our character class, so it knows about this new LevelMap. Alter the constructor of your Character class to also accept a LevelMap pointer as a parameter.

```
public:
	Character(SDL_Renderer* renderer, string imagePath, Vector2D start_position, 
LevelMap* map);
```

You will also need to adjust the Character classes you have made to take in the map too, adjust the parameters in the header and the cpp files for them.

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

1. Now create a new private variable to store this LevelMap pointer inside the Character header:

   ```
   LevelMap* m_current_level_map;
   ```

2. Then, set the pointer passed into the constructor to this new m_current_level_map pointer, this should be done in the constructor in Character.cpp.

   ```
   m_current_level_map = map;
   ```

3. You will also have to pass the Map through to this new Character constructor from your GameScreenLevel1 set up code. It is probably showing as an error at the moment.

   ```
   SetLevelMap();

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

With this in place, we should now be able to get our character to collide with the floors but for this you need to have implemented gravity as instructed in a previous tutorial.

For those that haven't set up gravity and jumping, return to additional work from tutorial 7. You should have something like this:

```
void Character::AddGravity(float deltaTime)
{
	if (m_position.y + 64 <= SCREEN_HEIGHT)
	{
		m_position.y += GRAVITY * deltaTime;
	}
	else
	{
		m_can_jump = true;
	}
}

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

In the Character cpp file, locate the Update function and adapt it as below. This states if gravity should be applied or not:

1. We first find the tile that our character is on. TILE_WIDTH and TILE_HEIGHT is the number of pixels of the tile. Both should be set to 32 in the Constants.h file.

   ```
   //collision position variables
   int centralX_position = (int)(m_position.x + (m_texture->GetWidth() * 0.5) )/
   TILE_WIDTH;
   int foot_position = (int)(m_position.y + m_texture->GetHeight()) / TILE_HEIGHT;
   ```

2. Then we check if this is an empty tile. If it is then we add gravity, otherwise we have collided and we allow the character to jump again. Ensure you only have one call to AddGravity, remove the existing call as it will give unexpected behaviour.

   ```
   //deal with gravity
   if(m_current_level_map->GetTileAt(foot_position,centralX_position) == 0)
   {
   	AddGravity(deltaTime);
   }
   else
   {
   	//collided with ground so we can jump again
   	m_can_jump = true;
   }
   ```

With this in place, run your application. You should have a character that falls down the screen and hits the floor!

Additional Work

As stated earlier there are collisions happening where the POW block will go. To be able to set this area to be a non-collision space after the POW block has been destroyed, we need a function in the LevelMap that will allow us to change any value on the map to something else.

This is the prototype you will require for your LevelMap header file. It should have public access so that other classes, such as the Character or GameScreenLevel1, can access it.

```
void ChangeTileAt(unsigned int row, unsigned int column, unsigned int new_value);
```

It is up to you to write the code required to implement this in the LevelMap.cpp file. We be using this in the next tutorial.