Tutorial 4 – Drawing an Image

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.

Now that our window remains open, it is time we drew something to it. This following tutorial will step you through loading an image in and drawing it to the window. Any image format will work with this implementation.

1. In your project folder (where SDL2 and Debug folders are) create a new folder called 'Images'.
2. Download the image from Blackboard called 'test.bmp' and save it to your new images folder.
3. To enable drawing, we are going to be using an SDL_Renderer. This will draw to the screen any textures associated with it, but before we use them, we need to assign them. At the top of your source file, under the g_window, add the following pointers:

```
SDL_Renderer* g_renderer = nullptr;
SDL_Texture* g_texture = nullptr;
```

4. We now need to add a few more functions, one to handle any rendering, one to handle loading textures and one to delete our texture and free memory. So, with your function prototypes, add the following:

```
void Render();
SDL_Texture* LoadTextureFromFile(string path);
void FreeTexture();
```

Firstly, lets setup our renderer. Within the InitSDL function add the code below just after where you have created your window:

```
g_renderer = SDL_CreateRenderer(g_window, -1, SDL_RENDERER_ACCELERATED);
```

SDL_CreateRenderer takes 3 parameters. The first is the window to be associated with the renderer. The second is an index of the rendering driver to initialise. –1 will initialise the first one supporting the requested flags. The third is a combination of SDL_RenderFlags.

The options for SDL_RenderFlags are:

SDL_RENDERER_SOFTWARE	The renderer is a software fallback
SDL_RENDERER_ACCELERATED	The renderer uses hardware acceleration
SDL_RENDERER_PRESENTVSYNC	Present is synchronized with the refresh rate
SDL_RENDERER_TARGETTEXTURE	The renderer supports rendering to texture

5. We then need to do some error catching and initialisation for PNG loading. This should follow where you created the renderer.

```
if(g_renderer != nullptr)
{
	//init PNG loading
	int imageFlags = IMG_INIT_PNG;
	if(!(IMG_Init(imageFlags)& imageFlags))
	{
		cout << "SDL_Image could not initialise. Error: " << IMG_GetError();
		return false;
	}
}
else
{
	cout << "Renderer could not initialise. Error: " << SDL_GetError();
	return false;
}
```

What we are stating here is that we would like to allow the loading of PNG files. IMG_Init() returns the flags that have been setup correctly. If IMG_INIT_PNG is not returned, then we output what error occurs (notice we use IMG_GetError for this). If the renderer is not created successfully, we handle whatever error we are thrown in the else and return false.

6. Now, let's work on the Render function, add the function body and then the following code within it:

```
//Clear the screen
SDL_SetRenderDrawColor(g_renderer, 0xFF, 0xFF, 0xFF, 0xFF);
SDL_RenderClear(g_renderer);
```

The above sets a colour for the renderer and clears the window. This results in the window being filled with in white, taking in as parameters a location and RGBA. Alternatively, black would be achieved using the values (g_renderer, 0x00, 0x00, 0x00, 0x00);

   a. Next, we will call SDL_Rect, this is a struct that holds positional data. We will state where we want the render location to be (top left 0,0) and to fill the size of the window. You can shrink or enlarge images by adjusting these values.

```
//set where to render the texture
SDL_Rect renderLocation = { 0,0,SCREEN_WIDTH, SCREEN_HEIGHT };
```

   b. Now we will render to the screen with the following:

```
//Render to screen
SDL_RenderCopyEx(g_renderer, g_texture, NULL, &renderLocation, 0, NULL,
SDL_FLIP_NONE);
```

There are different rendering functions in SDL, but this one allows you to set the size, an angle for rotation and a flag to set whether or not to flip the image.

The parameters are: the renderer, the texture, a source rect, a destination rect, an angle, a SDL_Point for the centre of the texture and a SDL_RendererFlip flag.

The options available for SDL_RendererFlip are:

SDL_FLIP_NONE	Don't flip the image at all
SDL_FLIP_HORIZONTAL	Flip the image horizontally
SDL_FLIP_VERTICAL	Flip the image vertically

c. For the image to be displayed in the window we need to present the renderer and that is what the following does:

```
//update the screen
SDL_RenderPresent(g_renderer);
```

7. Next up, add the function body for LoadTextureFromFile.

a. To begin with we need to clear up any memory used for a current texture in memory. We will do this via the FreeTexture() function we will create shortly, but for now just add a call to it.

```
//remove memory used for a previous texture
FreeTexture();
```

b. Make a pointer of SDL_Texture type. This is local and is what is returned from the SDL_CreateTextureFromSurface() function. Also, we require a local SDL_Surface pointer to load in the image. This will be cleaned up before we exit the function.

```
SDL_Texture* p_texture = nullptr;

//Load the image
SDL_Surface* p_surface = IMG_Load(path.c_str());
if(p_surface != nullptr)
{
```

IMG_Load allows you to load in files of numerous types. It accepts the file path as a char array so the c_str() function must be used to convert the passed in string.

c. If that has gone to plan we should have a surface to work with and create a texture from

```
//create the texture from the pixels on the surface
p_texture = SDL_CreateTextureFromSurface(g_renderer, p_surface);
if(p_texture == nullptr)
{
	cout << "Unable to create texture from surface. Error: " << SDL_GetError();
}
//remove the loaded surface now that we have a texture
SDL_FreeSurface(p_surface);
```

The SDL_CreateTextureFromSurface is self-explanatory, if something is to go wrong add some error catching. But should our texture be created successfully; we can delete the surface as it is no longer needed. This is what SDL_FreeSurface does for us. It is important to always clean up as you go so something isn't left and forgotten that could take up precious memory.

d. If setup does not go to plan, add an else to catch that scenario. We will need IMG_GetError for this instead of SDL_GetError:

```
else
{
	cout << "Unable to create texture from surface. Error: " << IMG_GetError();
}
```

e. Finally, as our function is of type SDL_Texture, we must return of type texture too. Before the very last closing bracket of the function, return the texture we created:

```
//Return the texture
return p_texture;
```

8. Now that our texture loading is complete, we need to return to the InitSDL() function and add a call to the LoadTextureFromFile() function. Remember to pass through a file name for the image. This goes in the body of code where the renderer was set up correctly. So, after your error handling for IMG_Init, add the following:

```
//Load the background texture
g_texture = LoadTextureFromFile("Images/test.bmp");
if(g_texture == nullptr)
{
	return false;
}
```

9. Now for the FreeTexture function. This will clean up any memory used. The SDL_DestroyTexture is again self-explanatory and takes in one parameter, the name of what is to be destroyed.

```
//check if texture exists before removing it
if(g_texture != nullptr)
{
	SDL_DestroyTexture(g_texture);
	g_texture = nullptr;
}
```

10. We can now call this in CloseSDL and clean up our renderer:

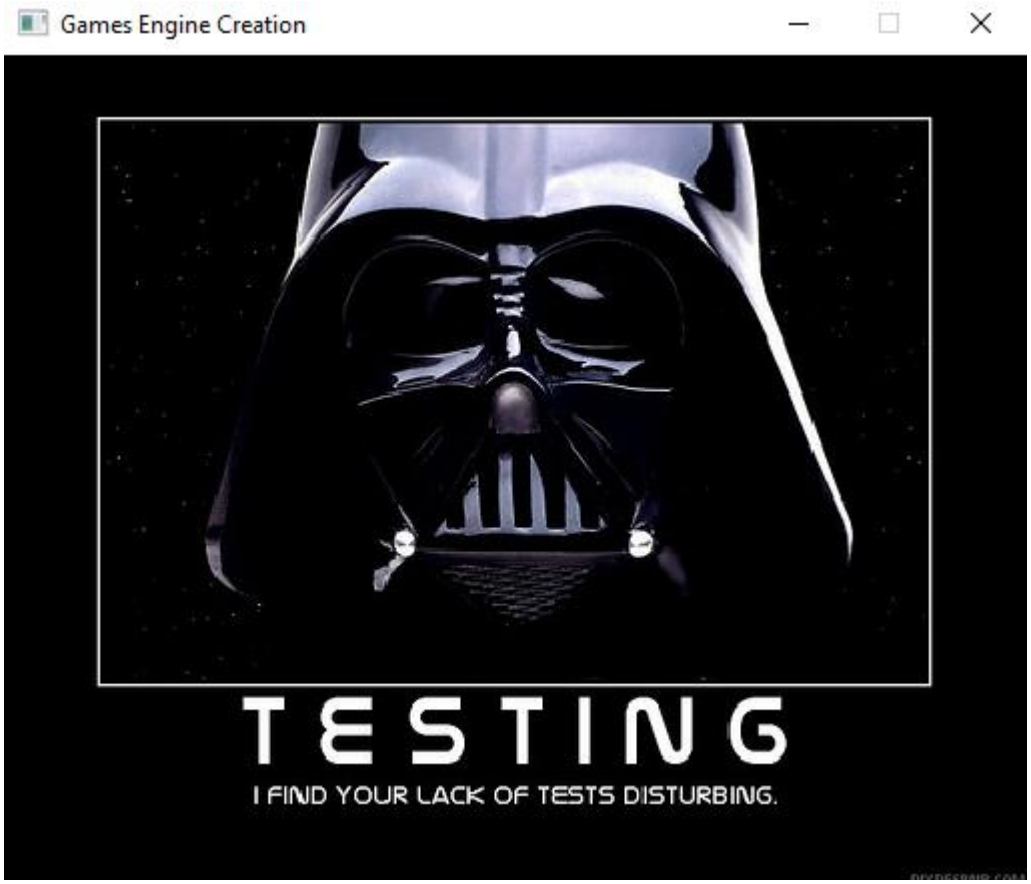
```
//clear the texture
FreeTexture();
//release the renderer
SDL_DestroyRenderer(g_renderer);
g_renderer = nullptr;
```

You may notice we reassign pointers after deleting them back to nullptr. This is to prevent double delete errors or should a condition later in the program require something to be null won't trigger and other issues. It is known as a dangling pointer, if not reassigned it will hold an invalid address and crash the program.

11. Lastly, we need make a slight alteration to our game loop in the main function and that is to simply call our Render function:

```
//Game Loop
while(!quit)
{
	Render();
	quit = Update();
}
```

Run the program and ensure everything works. You should be greeted with this:



Optional work:

Now that we have the basics set up, try to combine what you have learned from the previous four tutorials. For example, create an angle variable that increments when the user presses a key. Set the relevant parameters in the draw function and see if you can make it rotate. Try using the SDL_RendererFlip arguments to flip the image. You will use this to change the direction your character is facing in an upcoming tutorial.

This is additional work and may require you to remove what you do or change it later. So, keep that in mind. It might be worth creating a copy of your project to experiment with.