SPRITES, ANIMATIONS & TEXT

Overview

- The *Handmade Game Engine* is perfect to create small and playable **2D** games. For this it supports basic ***Sprites***, ***Animations*** and ***Text***
- Before creating and using sprites, the images themselves first need to be loaded into memory, so that there is something to render
- To load any images, we make use of the *Texture Manager*, and for that we need the following header file :

```
#include "TextureManager.h"
```

Overview

- The *LoadTextureFromFile()* function is used to load an image into memory and store it for later use :

```
TheTexture::Instance()->LoadTextureFromFile(name_of_file, tag_name);
```

- The arguments passed are the name of the image file, which should be situated somewhere in the *Assets* folder, and the tag name, which will label the loaded image with a name
- To unload images, simply use the *Texture Manager's UnloadFromMemory()* function :

```
//remove all texture images from memory
TheTexture::Instance()->UnloadFromMemory(TextureManager::TEXTURE_DATA, TextureManager::ALL_DATA);

//remove a specific image from memory
TheTexture::Instance()->UnloadFromMemory(TextureManager::TEXTURE_DATA, TextureManager::CUSTOM_DATA, "MY_TEXTURE");
```

Sprites

- These are the main (large or small) images that will be rendered on screen to make our games more colorful and exciting
- They can include anything like the background image, objects on screen, inventory items, menu buttons, or the HUD
- To create and use sprites, we need to use the *Sprite* class and create an object of this type. Make sure the following header is included :

```
#include "Sprite.h"
```

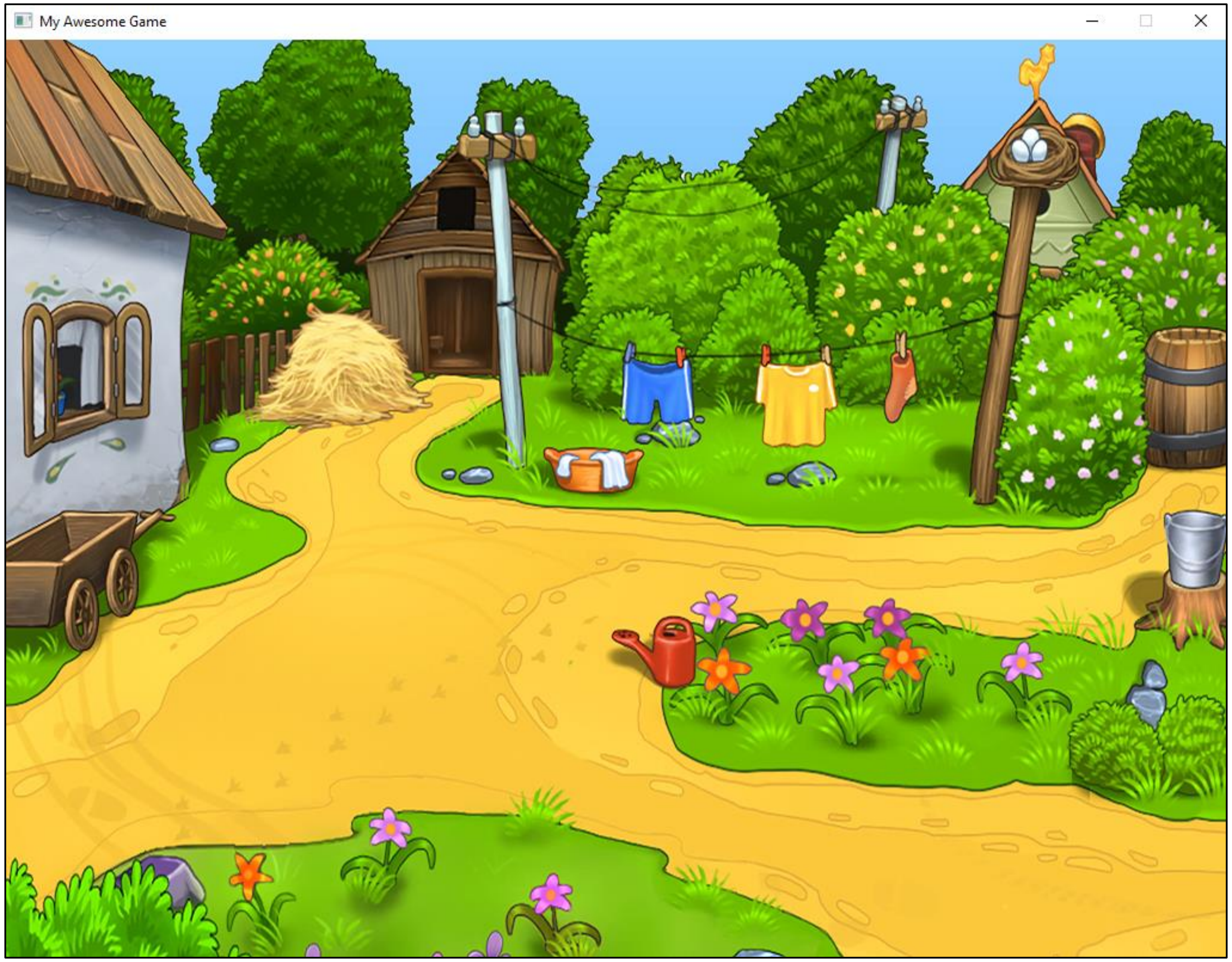
- Now we can create a *Sprite* object like so :

```
Sprite mySprite;
```

Sprites

- Take a look once again at the *Background* class and note how we use an internal *Sprite*
- In the constructor, the sprite image is loaded into memory and the sprite properties are set up accordingly
- In the *Background* object's *Draw()* function, we render the sprite, and in the destructor we remove the sprite image from memory again
- Try load in a nice and colourful backdrop image instead.

Sprites



Sprites

- Let's try create our own game object that uses a *Sprite* and displays a *door* in the scene
- We first need to create a *Door* game object and make sure it derives from *GameObject* properly

```
class Door : public GameObject
{

public:

    Door();
    virtual ~Door();

public:

    virtual void Update() {}
    virtual bool Draw();

private:

    Sprite m_image;

};
```

Sprites

- Now we can make use of any image that looks like a door, so first load that from the *Sprites* sub-folder within the *Assets* folder :

```
TheTexture::Instance()->LoadTextureFromFile("Assets\\Sprites\\Door.png", "DOOR");
```

- We now have to link our sprite object with the texture image and set the various properties of the sprite :

```
m_image.SetTexture("DOOR");
m_image.SetSpriteDimension(250, 320);
m_image.SetTextureDimension(1, 1, 650, 720);
```

Sprites

- The *SetTexture()* function links the loaded image with the sprite object. Now the sprite knows which image it needs to render
- The *SetSpriteDimension()* sets the *width* and *height* of the sprite image, just as it should appear on screen
- The *SetTextureDimension()* sets the properties of the texture image, in this case it's **1x1** because it's a single image and the *width* and *height* is **650x720**
- ***Note : The dimensions of the door image is an example. Whatever size your door sprite is, set it accordingly!***

Sprites

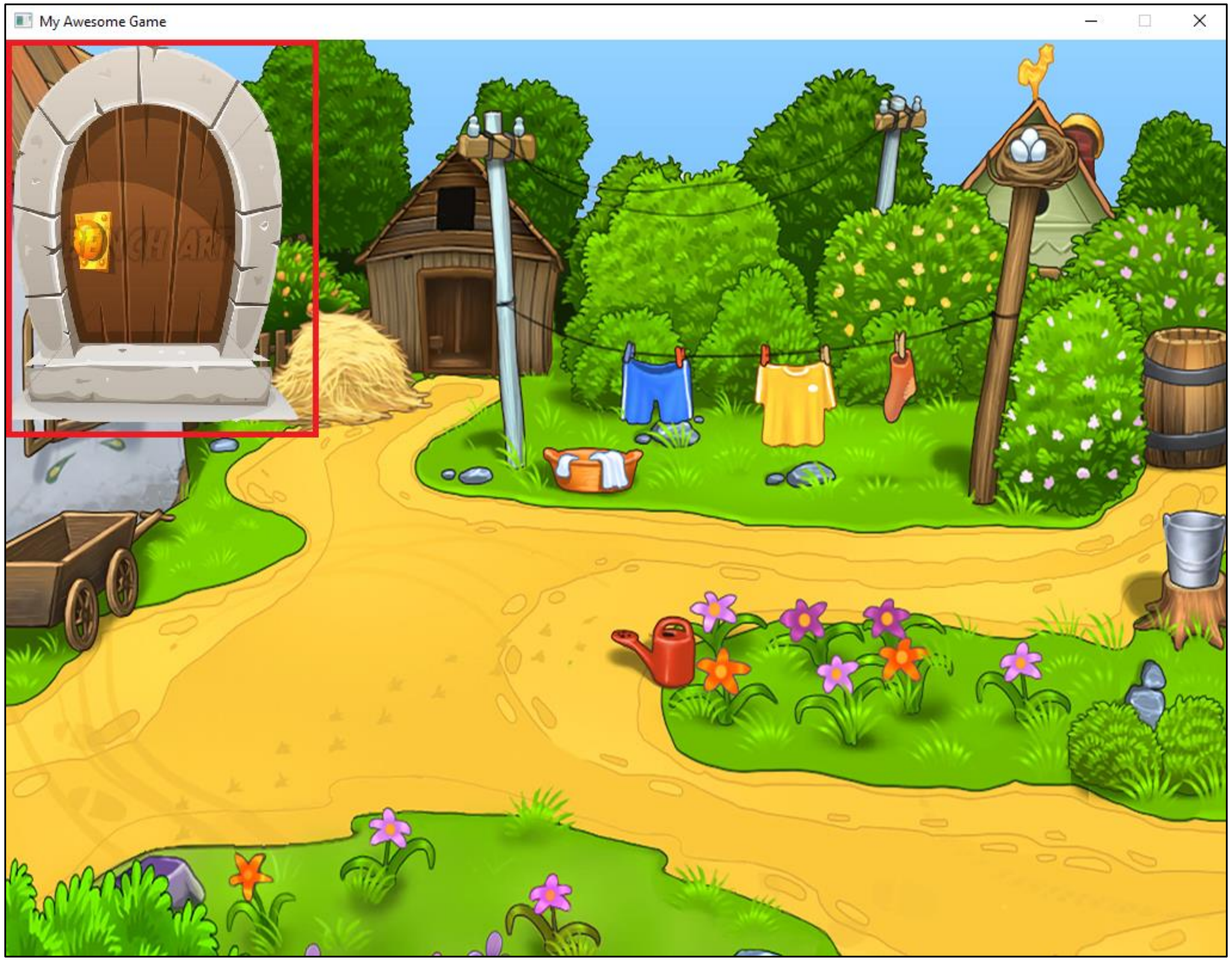
- Now, all that's left to do is render the image on screen, and for that there is a *Draw()* function :

```
m_image.Draw();
```

- We can place the above function in the *Door* object's overridden *Draw()* function
- Before anything renders, we need to first create the *Door* game object in the *Play* state and make sure to invoke the *Draw()* call
- To avoid calling the *Draw()* routine, simply add it to the vector of game objects :

```
m_gameObjects.push_back(new Door);
```

Sprites



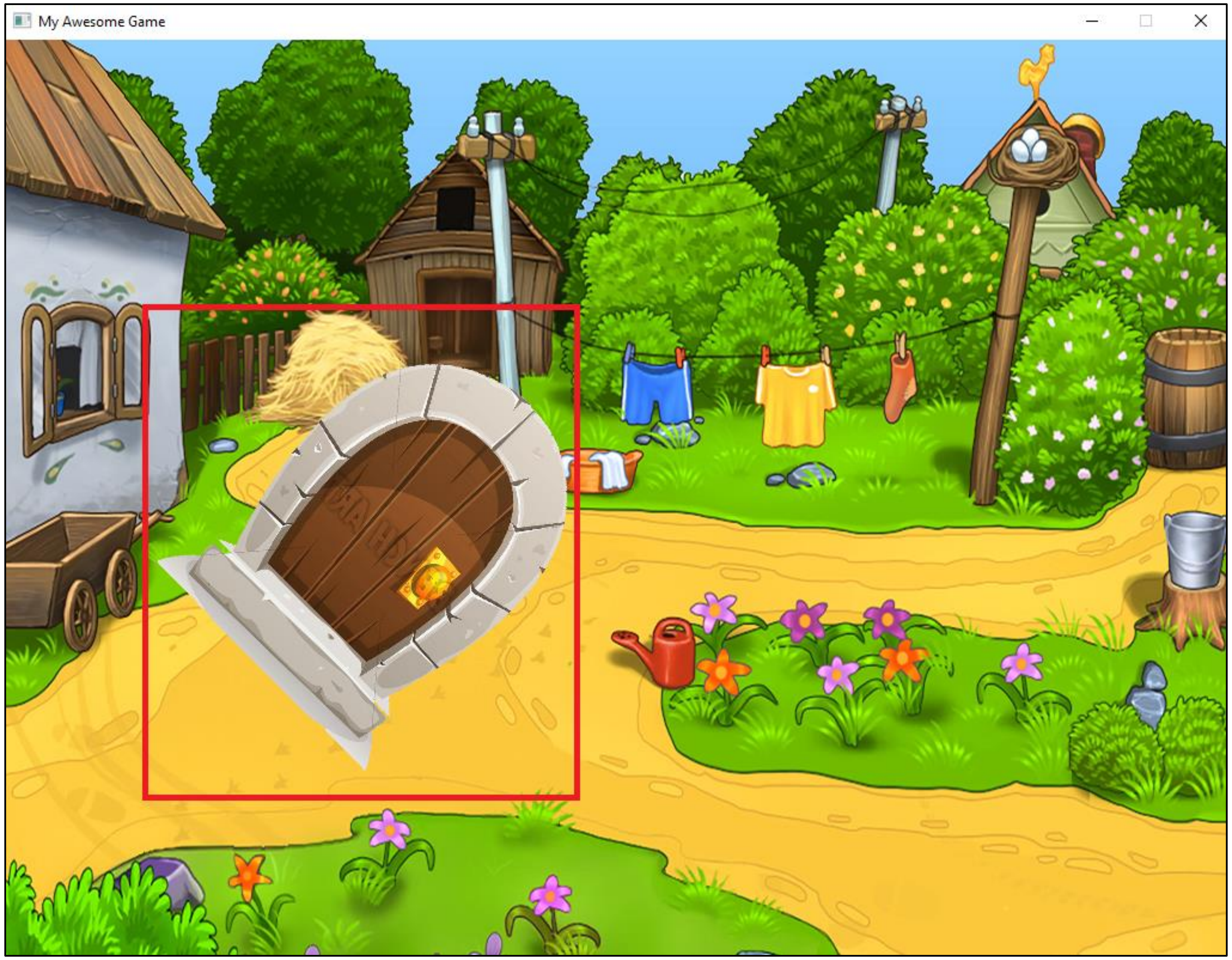
Sprites

- Of course, you may wish to pass in some arguments such as the position on screen where to render the sprite, the angle of rotation and if the image should be flipped :

```
background.Draw(200, 250, 45, Sprite::HORIZONTAL);
```

- The above function will render the image at position **200** in x and **250** in y, at a **45** degree angle, with the image flipped *horizontally*
- The image may also be flipped vertically or not at all. For these we have a **VERTICAL** and **NO_FLIP** flag respectively
- ***Note : The position on the screen where the image is rendered relates to the top left corner of the image***

Sprites



Sprites

- We can also load images from a texture sheet that contains multiple images
- For this create a sprite sheet of your own, and add **3** images into it, each cell having equal dimensions
- Load that into memory :

```
TheTexture::Instance()->LoadTextureFromFile("Assets\\Sprites\\Objects.png", "OBJECTS");
```

- Each image will be linked to its own separate *Sprite* object, so for that we need **3** sprites, which we will use within a *Collectible* game object :

```
Sprite m_barrel;
Sprite m_box;
Sprite m_rock;
```

Sprites

- Each sprite will "cut-out" its own image from the texture file. For this, we need to specify which texture cell the sprite requires
- For our intents and purposes, the barrel will use image **1**, the box will use image **2** and the rock will use the last image :

```
m_barrel.SetTextureCell(0, 0);
m_box.SetTextureCell(1, 0);
m_rock.SetTextureCell(2, 0);
```

- The *SetTextureCell()* function takes in a *column* and *row* value and will use that to search for the correct image in the texture.
- ***Note : All images in multiple texture sheets are zero based and run along the texture file from left to right and top to bottom***

Sprites

- The full code for all **3** sprite objects should look something like this :

```
m_barrel.SetTexture("OBJECTS");
m_barrel.SetSpriteDimension(100, 100);
m_barrel.SetTextureDimension(3, 1, 200, 200);
m_barrel.SetTextureCell(0, 0);

m_box.SetTexture("OBJECTS");
m_box.SetSpriteDimension(100, 100);
m_box.SetTextureDimension(3, 1, 200, 200);
m_box.SetTextureCell(1, 0);

m_rock.SetTexture("OBJECTS");
m_rock.SetSpriteDimension(80, 80);
m_rock.SetTextureDimension(3, 1, 200, 200);
m_rock.SetTextureCell(2, 0);
```

Sprites

- Looking at the above code, note that the sprite dimension can be set to any size you desire. Beware of making sizes larger than the size they are in the sprite sheet – this will lead to image *tearing*
- With texture sheets that contain multiple images, the *SetTextureDimension()* function needs to be set correctly, like so :

```
m_barrel.SetTextureDimension(3, 1, 200, 200);
```

- The first **2** arguments are the amount of columns and rows in the texture sheet, respectively
- The last **2** arguments are the size of each texture cell
- In our case we have an image file consisting of **3** columns and **1** row, and each image cell is **200x200** pixels in dimension

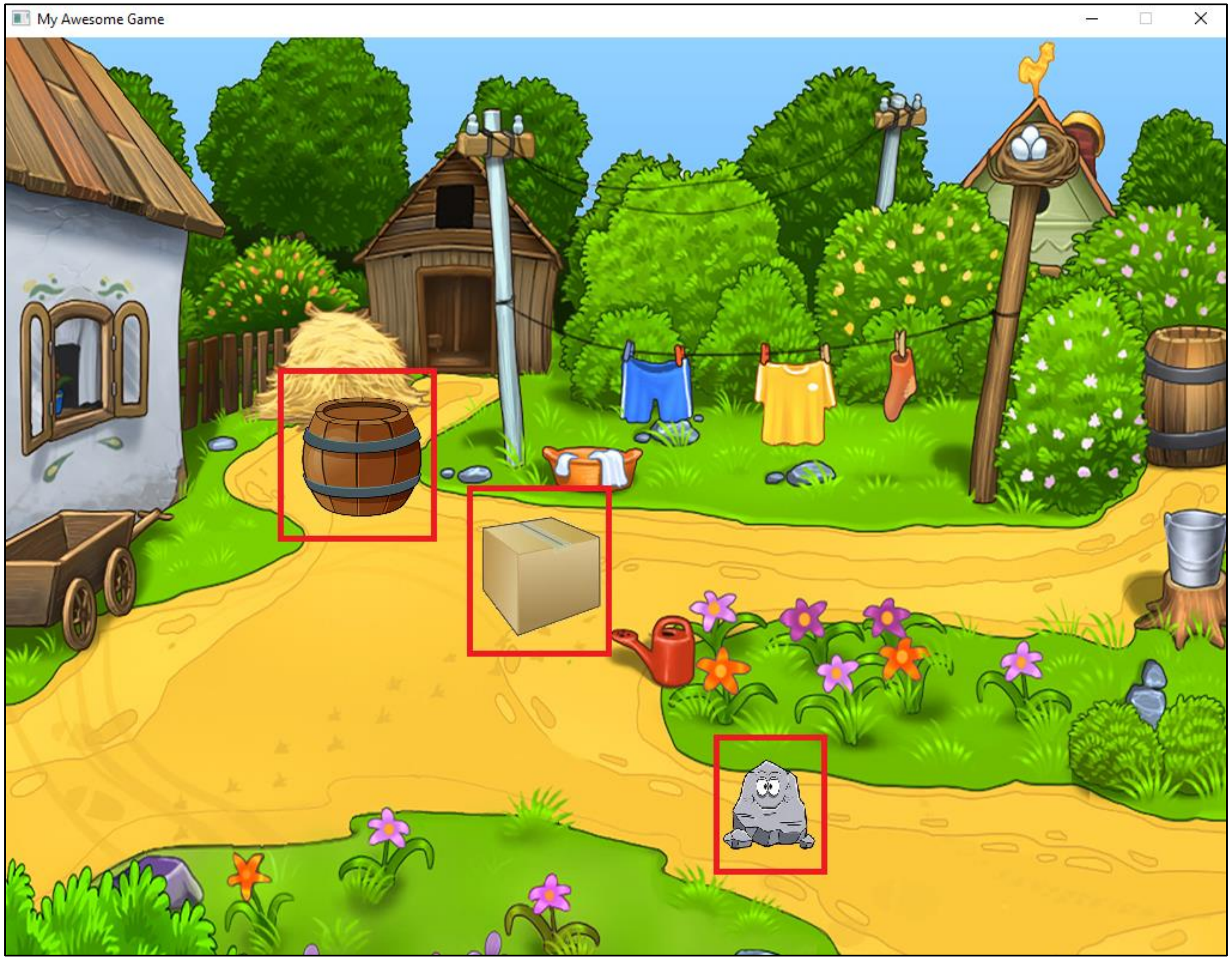
Sprites

- Lastly, we simply call the *Sprite* object's *Draw()* method for each of our objects, specifying the position where to place the image on screen :

```
m_barrel.Draw(250, 300);
m_box.Draw(400, 400);
m_rock.Draw(600, 600);
```

- ***Note : Don't forget to instantiate the Collectible game object in the Play state as well!***
- ***Note : Ideally we will want each object to be encapsulated within its own game object, however the above is purely for demonstration purposes***

Sprites



Animations

- Animations are dynamic sprites, nothing more than moving images, that you can add to the game to make objects more life-like
- These include things like character animations, such as a walking or jumping cycle, or an explosion
- To create and use animations, we need to use the *Animation* class and create an object of this type. Make sure the following header is included :

```
#include "Animation.h"
```

- Now we can create a *Animation* object like so :

```
Animation myAnimation;
```

Animations

- The *Animation* class will loop through a sprite sheet of images and "cut out" the correct texture cell based on a certain speed
- Animations may loop only once, such as an explosion, or loop repeatedly, like a character's walk cycle
- We will demonstrate both animation options, but first we need to create **2** additional game objects, namely *Explosion* and *Player*
- Both game objects will have an *Animation* component within, just as we did with the previous game objects and their *Sprite* component

Animations

- For the *Explosion* game object, we need to load in the animation sprite sheet, so find one online and load it in :

```
TheTexture::Instance()->LoadTextureFromFile("Assets\\Sprites\\Explosion.png", "EXPLOSION");
```

- We then set all the properties of the animation component, like so :

```
m_anim.SetTexture("EXPLOSION");
m_anim.SetAnimationVelocity(15.0f);
m_anim.SetSpriteDimension(100, 100);
m_anim.SetTextureDimension(8, 6, 128, 128);
m_anim.IsAnimationLooping() = false;
```

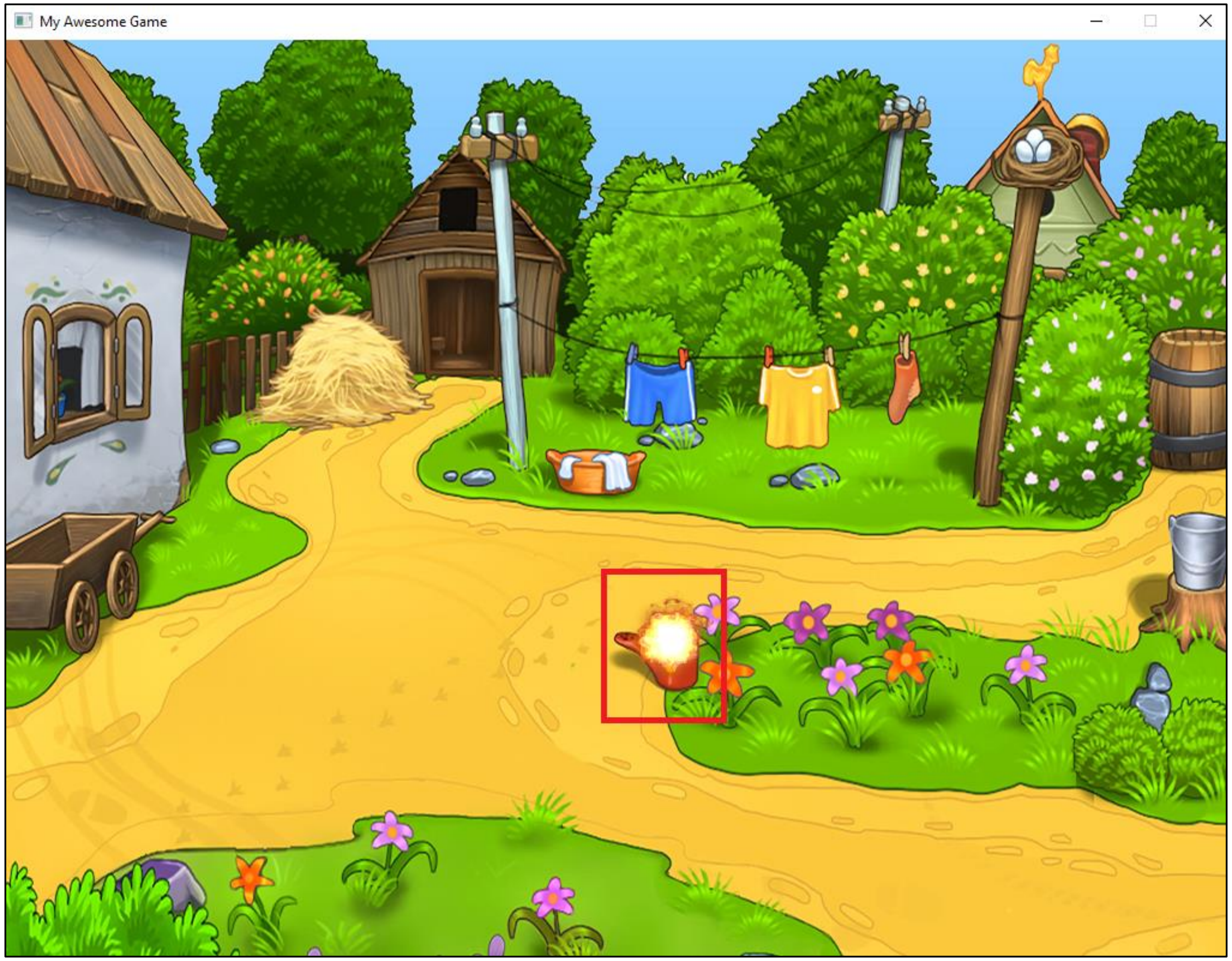
Animations

- The *SetAnimationVelocity()* will set the speed at which the animation should run at
- The *IsAnimationLooping()* will toggle whether the animation should loop or not. We only want **1** explosion, so we keep it ***false***
- All that's left is to render the animation on screen, and for that there is a *Draw()* call :

```
m_anim.Draw(500, 435);
```

- Now, after instantiating the *Explosion* object in the *Play* state, we invoke its *Draw()* call and will be able to see an exploding watering can

Animations



Animations

- For the *Player* object, we can use the following code to set up the animation component :

```
TheTexture::Instance()->LoadTextureFromFile("Assets\\Sprites\\Hero.png", "HERO");

m_anim.SetTexture("HERO");
m_anim.SetAnimationVelocity(20.0f);
m_anim.SetSpriteDimension(125, 250);
m_anim.SetTextureDimension(8, 1, 125, 250);
m_anim.IsAnimationLooping() = true;
```

- This time we want the hero's walk cycle to loop, so we set the *IsAnimationLooping()* to ***true***
- Now, after creating the *Player* object in the *Play* state, we can render it on screen

Animations

- To have the hero sprite move, we need to update him, but first we need to set his initial position :

```
m_position = glm::vec2(500, 435);
```

- In the *Player* class' overridden *Update()* function, we can adjust his x position :

```
m_position.x -= 2;
```

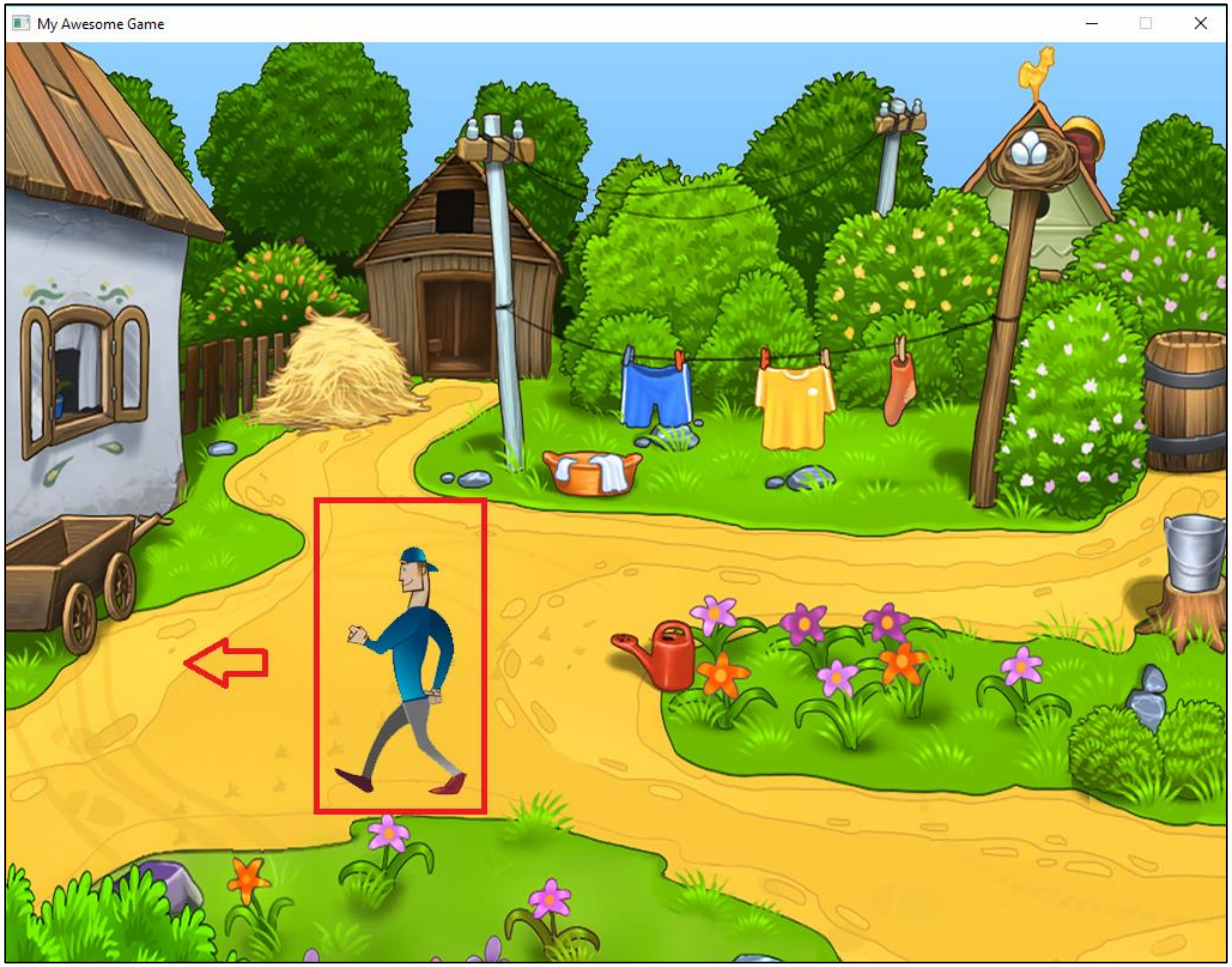
- Now just draw the player at the position he is assigned to, like so :

```
m_anim.Draw(m_position.x, m_position.y);
```

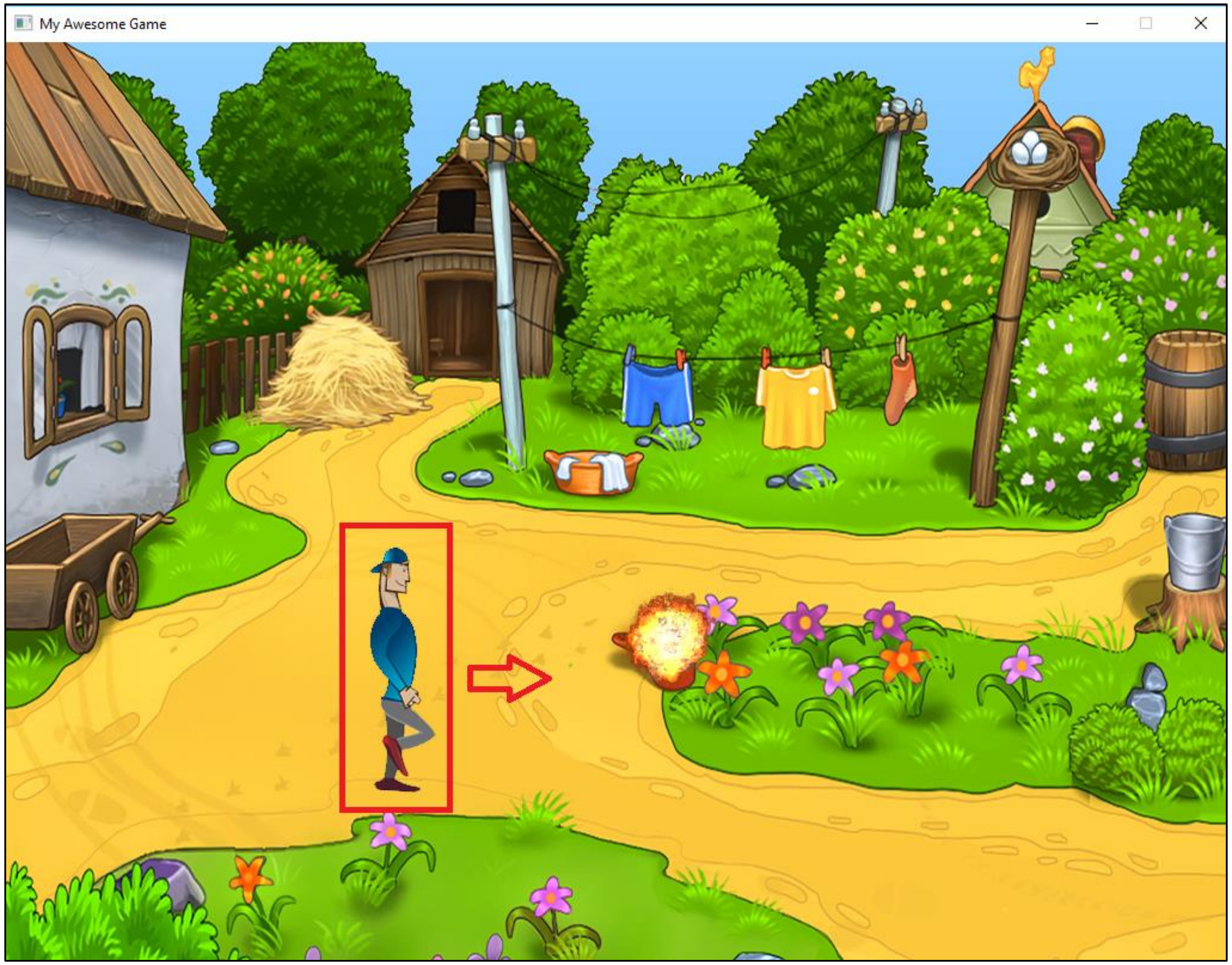
- If you want the hero to "walk the other way", simply increment the x position and flip the animation object horizontally :

```
m_anim.Draw(m_position.x, m_position.y, 0, Sprite::HORIZONTAL);
```

Animations



Animations



Online Sources

- A good place to look for free sprites and character animations is *Google Images*. There are of course other places too :
- ***Widget Worx :*** http://www.widgetworx.com/projects/sl.html
- ***Open Game Art :*** http://opengameart.org/
- ***Game Art 2D :*** http://www.gameart2d.com/
- ***Kenney's Website :*** http://kenney.nl/
- ***Open Game Graphics :*** https://opengamegraphics.com/
- ***Unity Asset Store :*** https://www.assetstore.unity3d.com/en/

Text

- You may also want to display some text on screen and for this the *Handmade Engine* supports basic *text display*
- Before creating and using text, the fonts that will be used to render the text need to be loaded into memory
- To load fonts, we again make use of the *Texture Manager* :

```
#include "TextureManager.h"
```

Text

- The *LoadFontFromFile()* function is used to load a **TTF** font file into memory and store it for later use :

```
TheTexture::Instance()->LoadFontFromFile(name_of_file, size, tag_name);
```

- The arguments passed are the name of the font file, which should be situated somewhere in the *Assets* folder, the size you wish to load the font in as, and the tag name, which will label the font with a name
- To unload images, simply use the *Texture Manager's UnloadFromMemory()* function :

```
//remove all fonts from memory
TheTexture::Instance()->UnloadFromMemory(TextureManager::FONT_DATA, TextureManager::ALL_DATA);

//remove a specific font from memory
TheTexture::Instance()->UnloadFromMemory(TextureManager::FONT_DATA, TextureManager::CUSTOM_DATA, "MY_FONT");
```

Text

- Sometimes it would be nice to display some feedback to the player or offer menu choices, and this, and more, can be achieved by displaying text on screen
- To create and use text, we need to use the *Text* class and create an object of this type. Make sure the following header is included :

```
#include "Text.h"
```

- Now we can create a *Text* object like so :

```
Text myText;
```

Text

- We'll demonstrate the use of the Text component by creating a *HUD* class that will display the elapsed time passed
- We will make use of the *Comic.ttf* font file, so first load that into memory (find it on 1001Fonts.com) :

```
TheTexture::Instance()->LoadFontFromFile("Assets\\Fonts\\Comic.ttf", 100, "FONT");
```

- We now have to link our text object with the font and set the various properties of the text :

```
m_display.SetFont("FONT");
m_display.SetColor(255, 235, 0);
m_display.SetSize(300, 80);
m_display.SetText("Time elapsed :");
```

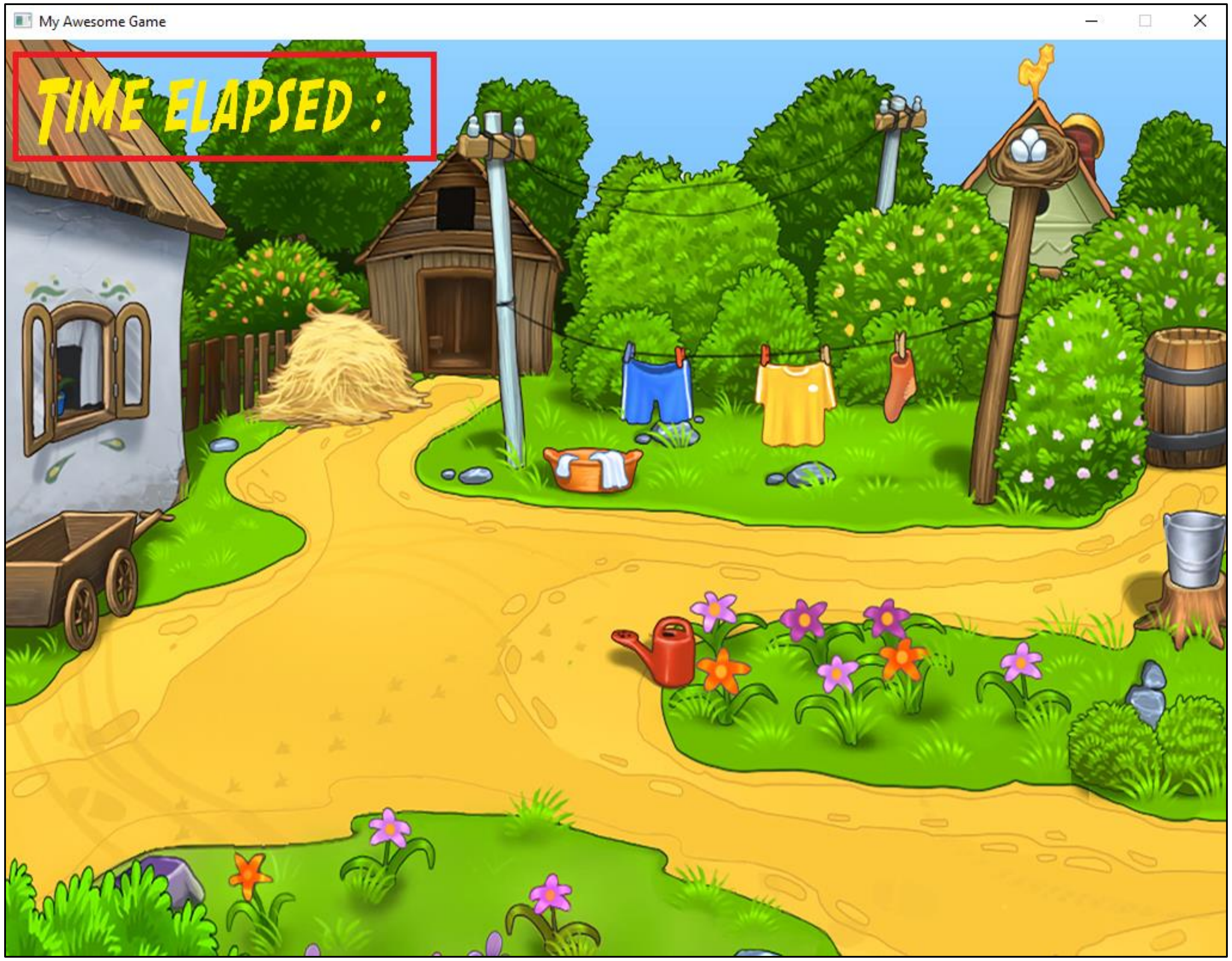
Text

- The *SetFont()* function links the loaded font with the text object.
- The *SetSize()* sets the *width* and *height* of the text object, just as it should appear on screen
- The *SetText()* and *SetColor()* routines set the text string to display and its color, respectively
- Now, all that's left to do is render the text on screen, at a specific position :

```
m_display.Draw(20, 15);
```

- Create the *HUD* in the *Play* state and call its *Draw()* function to see the *HUD* displayed
- ***Note : If you set the text size too high and the text starts to look pixelated, consider loading the font file in a higher size***

Text



Text

- In order to change the text so that we can display some dynamic text, such as the elapsed time, we need to add code to the *HUD* object's *Update()* function :

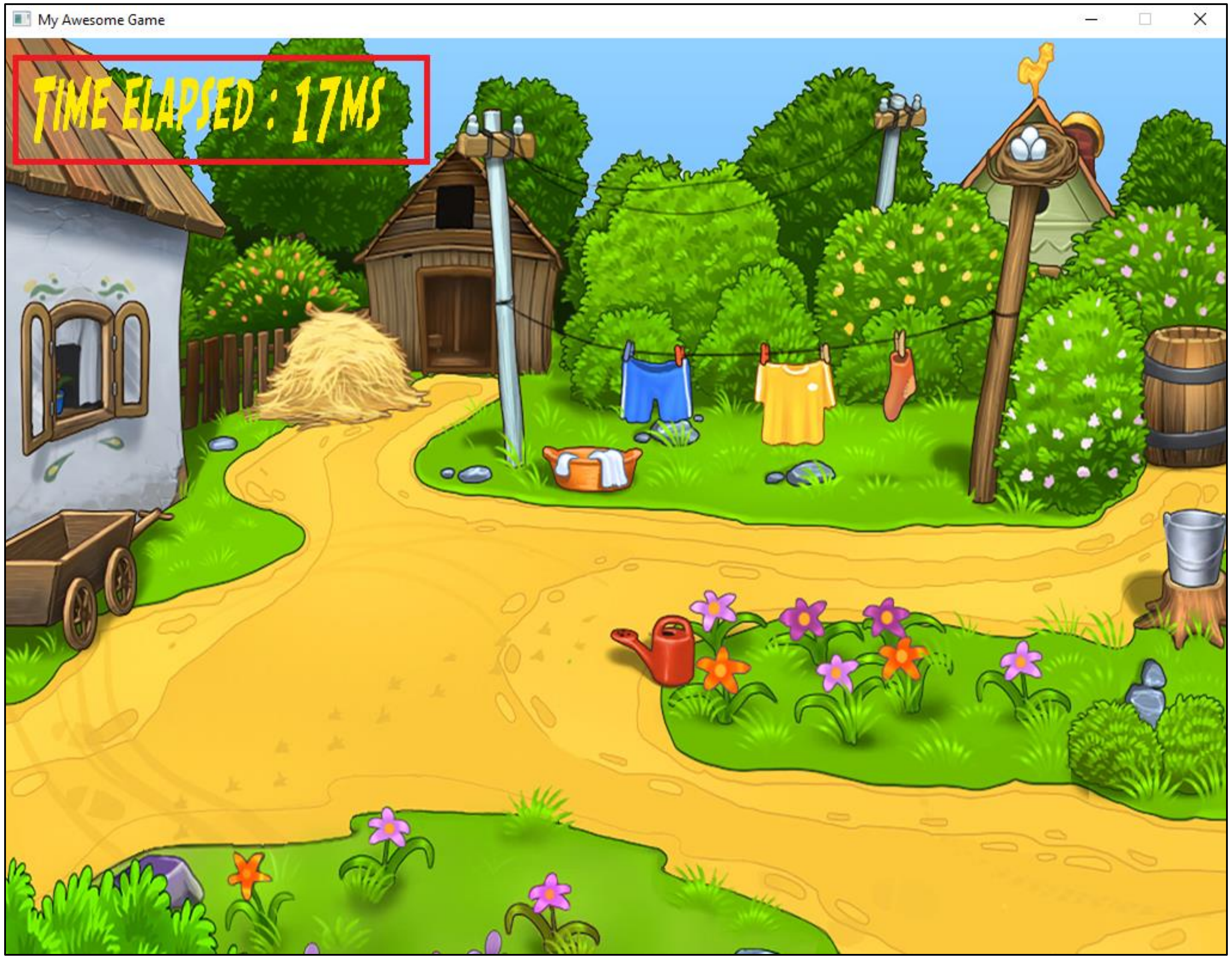
```
void HUD::Update()
{

    std::string s = std::to_string(TheGame::Instance()->GetElapsedTime());
    m_display.SetText("Time elapsed : " + s + "ms");

}
```

- The above code takes the elapsed time between each frame and will convert it into a *string* object, before assigning that string text to the *Text* object

Text

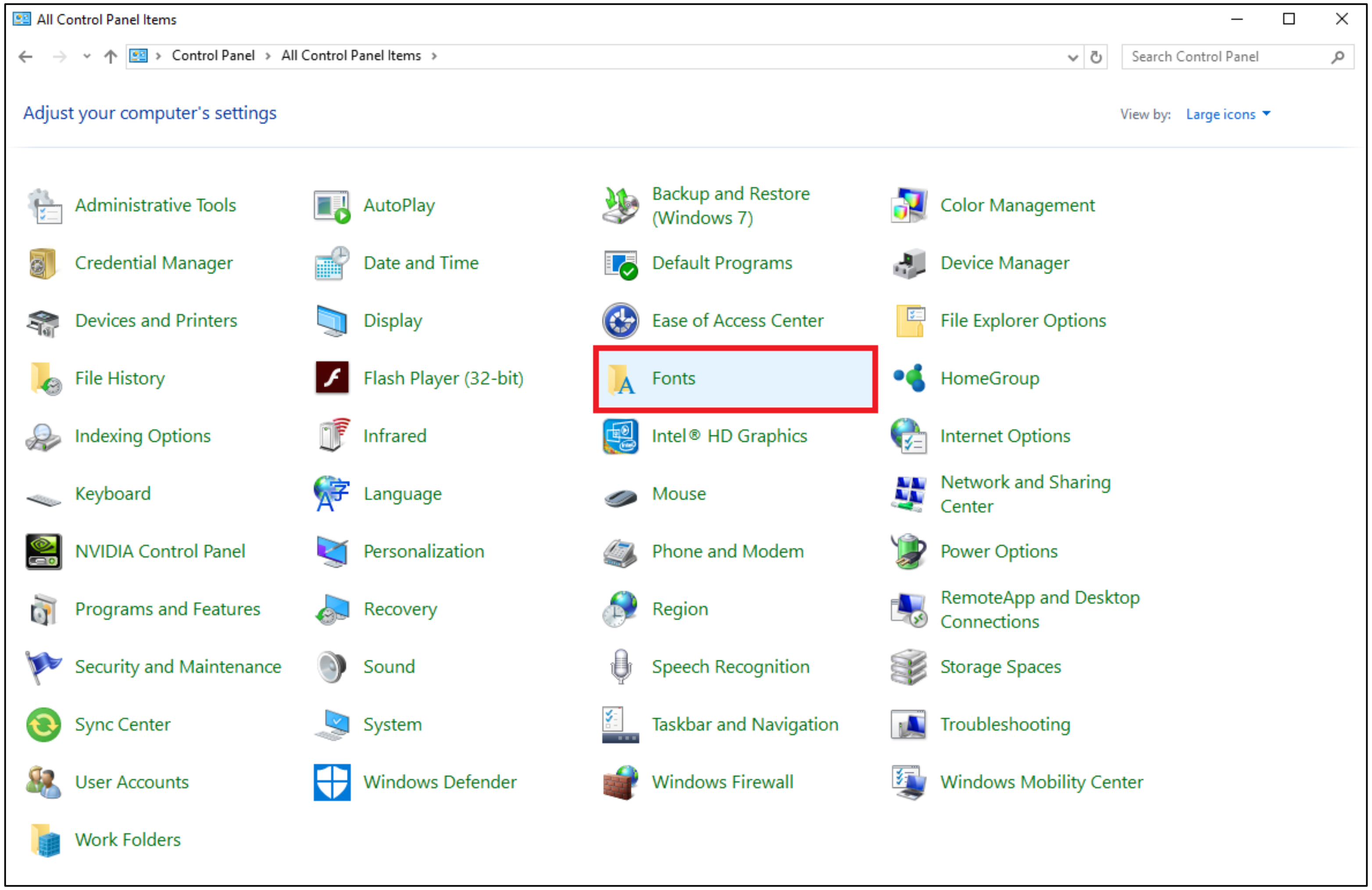


Text

- Take another closer look at the *MainMenu* class and see how the *Text* components within have been used
- Regarding **TTF** font files, you can use any of the pre-installed fonts on your Windows system.
- Simply head over to the *Control Panel*, select the *Fonts* folder and copy the **TTF** files into your *Assets* folder
- Alternatively, you can also download font files from an abundant selection on the site ***1001 Fonts*** :

  http://www.1001fonts.com/

Text



All Control Panel Items

Control Panel › All Control Panel Items ›

Search Control Panel

Adjust your computer's settings

View by: Large icons

Administrative Tools	AutoPlay	Backup and Restore (Windows 7)	Color Management
Credential Manager	Date and Time	Default Programs	Device Manager
Devices and Printers	Display	Ease of Access Center	File Explorer Options
File History	Flash Player (32-bit)	Fonts	HomeGroup
Indexing Options	Infrared	Intel® HD Graphics	Internet Options
Keyboard	Language	Mouse	Network and Sharing Center
NVIDIA Control Panel	Personalization	Phone and Modem	Power Options
Programs and Features	Recovery	Region	RemoteApp and Desktop Connections
Security and Maintenance	Sound	Speech Recognition	Storage Spaces
Sync Center	System	Taskbar and Navigation	Troubleshooting
User Accounts	Windows Defender	Windows Firewall	Windows Mobility Center
Work Folders			