Week 3

Managing Resources

Instructor: Hooman Salamat

2019-01-21

Objectives

- Learn the motivation behind external resources
- Explore the classes for resource handling and manipulation in SFML
- Examine a typical use case in games
- Apply the need to manage resources in a simplified way

Resources

- Resources can be defined as an external component that the game loads during runtime
  - Also known as an asset
  - Most often multimedia components such as images, music or fonts
  - Generally large in size - occupy a lot of memory

Resources (cont'd.)

- Can also be scripts that describe world content, i.e. Metadata
- Also configuration files
- Whatever the case, resources are loaded from files on the hard drive
- The RAM or the Network

Resources (cont'd.)

- To load a resource in SFML, typically we use:

```
bool loadFromFile(const std::string& filename);
```

  - mTexture.loadFromFile("Media/Textures/Eagle.png“);
  - mFont.loadFromFile("Media/Sansation.ttf");

- Like most languages, the boolean return type holds whether or not the load was successful
- Checking for a successful resource load is critical in any program

Resources (cont'd.)

- loadFromStream() loads a resource using a custom sf::InputStream instance.
  - Allows the user to exactly specify the loading process
  - Use cases of user-defined streams are encrypted and/or compressed file archives.
- loadFromMemory() loads a resource from RAM
  - Useful to load resources that are directly embedded into the executable

loadFromMemory

- const char my_file[] =
  {
      ... lots of digits pairs ...
  }
- image.loadFromMemory(my_file, sizeof(my_file));

Load from http address

```
sf::Texture Game::dlpicture(std::string
host, std::string uri)
{
sf::Http http(host);
sf::Http::Request request(uri);
auto response = http.sendRequest(request);
auto data = response.getBody();

sf::Texture text;
text.loadFromMemory(data.data(),
data.length());
return text;
}
```

loadfromStream

sf::FileInputStream provides the read-only data stream of a file, while sf::MemoryInputStream serves the read-only stream from memory. Both are derived from sf::InputStream and can thus be used polymorphic.

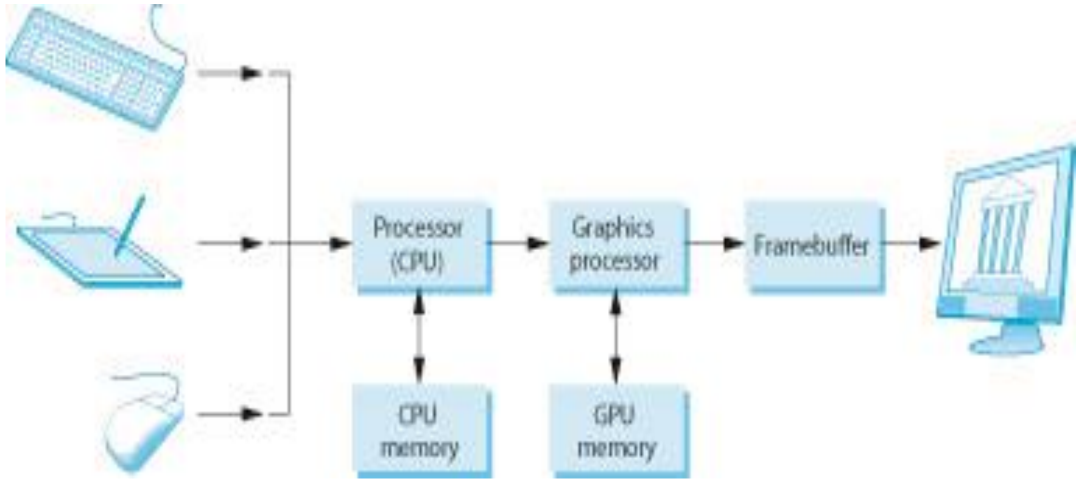
```
sf::FileStream stream;
stream.open("image.png");
sf::Texture texture;
texture.loadFromStream(stream);
```

Sf::InputStream

```
// custom stream class that reads from inside a zip file
class ZipStream : public sf::InputStream
{
public:
ZipStream(std::string archive);
bool open(std::string filename);
Int64 read(void* data, Int64 size);
Int64 seek(Int64 position);
Int64 tell();
Int64 getSize();
private:
...
};
// now you can load textures...
sf::Texture texture;
ZipStream stream("resources.zip");
stream.open("images/img.png");
texture.loadFromStream(stream);
// musics...
sf::Music music;
ZipStream stream("resources.zip");
stream.open("musics/msc.ogg");
music.openFromStream(stream);
// etc.
```

A Graphics System

- The image we see on the output device is an array (**the raster**) of picture elements, or pixels produced by graphics system
- All modern graphics systems are raster based
- Every pixel corresponds to a location in the image
- Collectively, the pixels are stored in a part of memory called the **framebuffer**.
- frame buffer depth or precision
  - 1-bit-deep => Two colors
  - 8-bit deep => 256 colors
  - Full color => 24 bit or more



Sprite

- Even modern 2D games are made using vertices.
- The 2D sprites are made up of two triangles to form a square.
- This is often referred to as a quad.
- The quad is given a texture and it becomes a sprite.
- Two-dimensional games use a special projection transformation that ignores all the 3D data in the vertices, as it's not required for a 2D game.
- Sprites are 2D bitmaps that are drawn directly to a render target without using the pipeline for transformations, lighting or effects. Sprites are commonly used to display information such as health bars, number of lives, or text such as scores. Some games, especially older games, are composed entirely of sprites.

Textures

- Textures in SFML are represented as graphical images in the `sf::Texture` class
  - Stored as an array of pixels in the graphics card
  - Each pixel is a 32 bit RGBA value
  - Can be drawn to the screen with `sf::Sprite`

- A sprite can be part of a texture, so we consider it lightweight
- A texture is the source image for a sprite

Images

- A container for pixel values using the `sf::Image` class
  - Stores its pixels in RAM instead of video memory
  - Capable of saving the stored image back to a file.
  - So we can manipulate single pixels
  - sf::Texture loads the data using an intermediate sf::Image

- Some restrictions
  - To display a sf::Image, First have to convert it to `sf::Texture` and then create a `sf::Sprite` that refers to it
  - If you don't need per pixel access, use texture

Fonts

- To load a character font for use and manipulation, use the `sf:Font` class
- Supports many formats including the common true type fonts (TTF) and open type fonts (OTF)

- To display text on screen, use `sf::Text`
- Like the texture-sprite relationship, the font is the source for many text instances

Open GL

- Every computer has special graphics hardware that controls what you see on the screen.
- OpenGL tells this hardware what to do.
- The Open Graphics Library is one of the oldest, most popular graphics libraries game creators have.
- It was developed in 1992 by Silicon Graphics Inc. (SGI) and used for GLQuake in 1997.
- The GameCube,Wii, PlayStation, and the iPhone all use OpenGL.
- Cross-platform standard: OpenGL today is supported on all platforms

Vertex

- The basic unit in OpenGL is the vertex. A vertex is a point in space.
- Extra information can be attached to these points—how it maps to a texture, if it has a certain weight or color—but the most important piece of information is its position.

- Games spend a lot of their time sending OpenGL vertices or telling OpenGL to move vertices in certain ways.
- The game may first tell OpenGL that all the vertices it's going to send are to be made into triangles. In this case, for every three vertices OpenGL receives, it will attach them together with lines to create a polygon, and it may then fill in the surface with a texture or color.

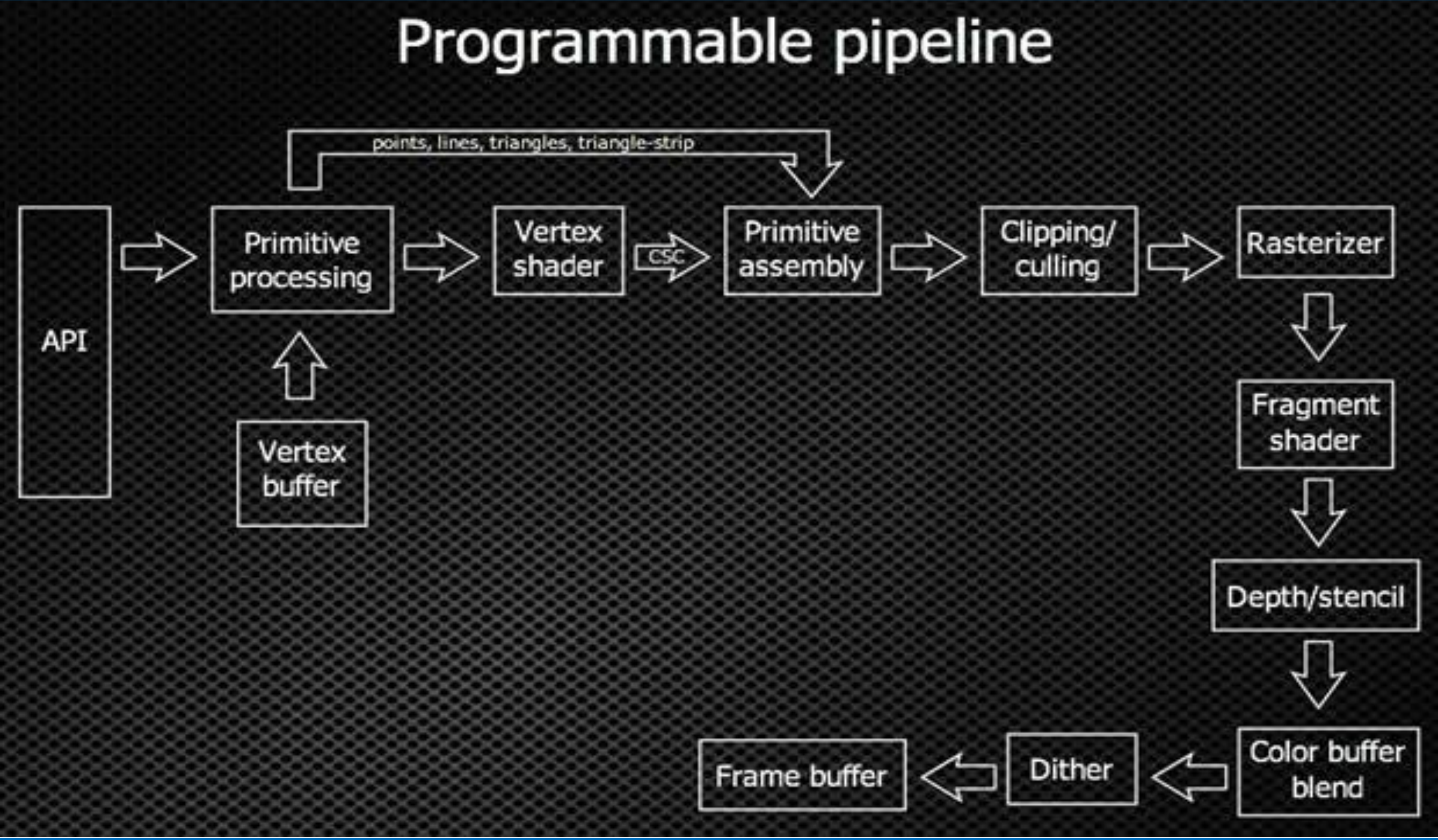
Pipeline

- Modern graphics hardware is very good at processing vast sums of vertices, making polygons from them and rendering them to the screen.
- This process of going from vertex to screen is called the pipeline.
- The pipeline is responsible for positioning and lighting the vertices, as well as the projection transformation.

Pipeline

- The pipeline has become programmable.
- Programs can be uploaded to the graphics card.
- There are few steps in the pipeline.
- All the steps could be applied in parallel.
- Each vertex can pass through a particular stage at the same time, provided the hardware supports it.
- This is what makes graphics cards so much faster than CPU processing.

Pipeline


Programmable pipeline
points, lines, triangles, triangle-strip
API
Primitive processing
Vertex buffer
Vertex shader
CSC
Primitive assembly
Clipping/ culling
Rasterizer
Fragment shader
Depth/stencil
Color buffer blend
Dither
Frame buffer

Shaders

- A program that operates on the graphics card and applies effects to rendered assets
- SFML builds upon OpenGL, so it uses GLSL (OpenGL Shading Language)
  - SFML supports Vertex shaders and fragment/pixel shaders
    - Vertex Shaders: affects geometry of objects in the scence
    - Fragment shaders: manipulate pixel of the scene
- An SFML shader or `sf::Shader` can be created from a string that contains the GLSL code of the source shader

Sounds

- A sample is a 16 bits signed integer that defines the amplitude of the sound at a given time. The sound is then reconstituted by playing these samples at a high rate (for example, 44100 samples per second is the standard rate used for playing CDs)
- audio samples are like texture pixels, and a sf::SoundBuffer is similar to a sf::Texture.
- A sound buffer can be loaded from a file (see loadFromFile() for the complete list of supported formats), from memory, from a custom stream (see sf::InputStream) or directly from an array of samples. It can also be saved back to a file.

Sounds

- SFML contains a `sf::SoundBuffer` class used to store sound effects
  - 16 bit audio samples
- `sf::Sound` is the class that actually plays audio from the sound buffer
  - Can be played, paused and stopped and have their volume and pitch modified

Sounds

- SFML contains a `sf::SoundBuffer` class used to store sound effects
  - 16 bit audio samples
- `sf::Sound` is the class that actually plays audio from the sound buffer
  - Can be played, paused and stopped and have their volume and pitch modified

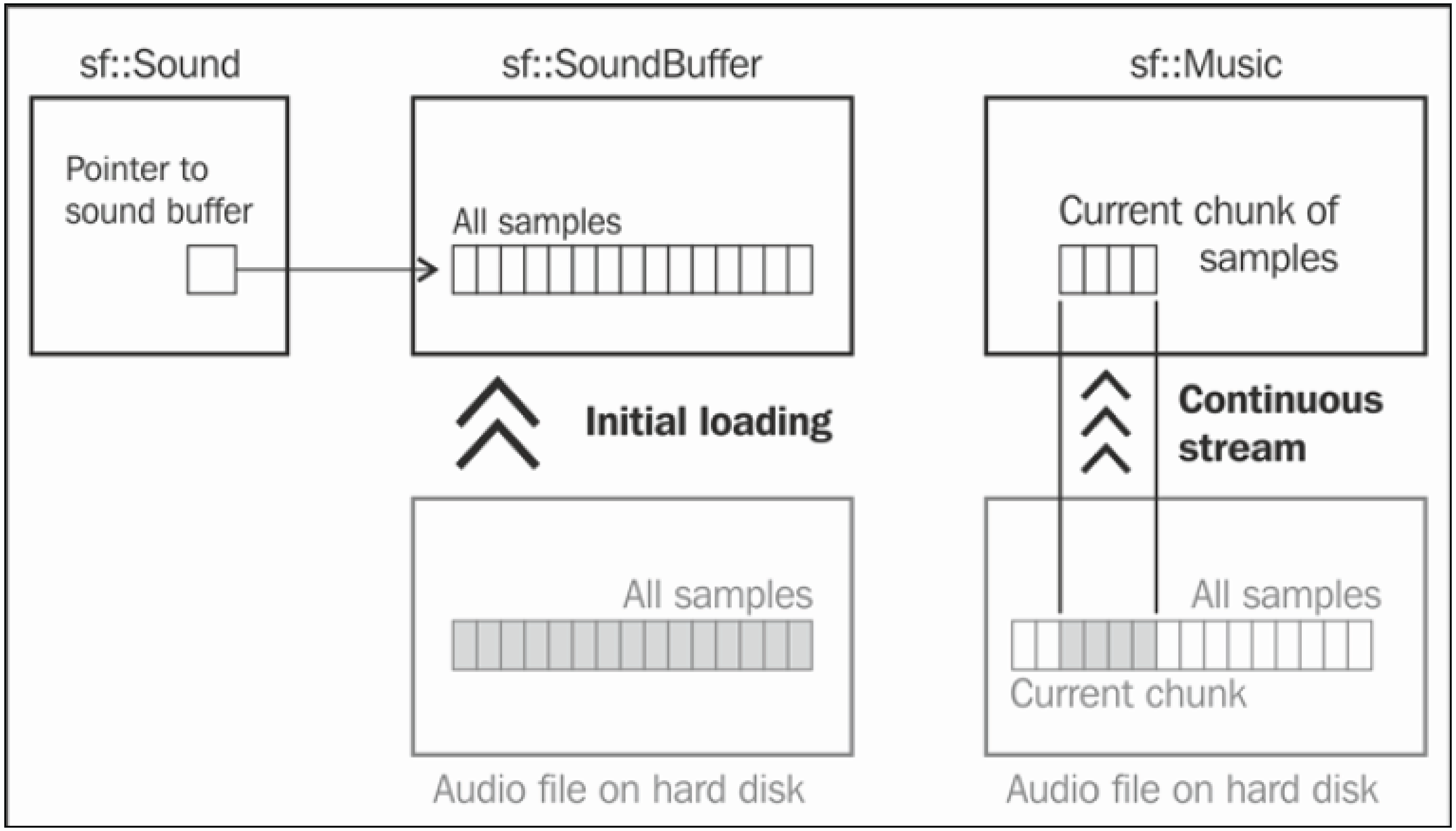
Sound (cont'd.)

- Valid formats are WAV, OGG, AIFF and more
  - See http://www.sfml-dev.org/faq.php#audio-formats for the full list
- **SFML does NOT support MP3s because of their restrictive license**

Sound (cont'd.)

- Music in SFML is played by the sf::Music class
- Does not use the sound buffer class
- Streams the music, i.e., it loads small chunks continuously
- The difference between the sound buffer and music is shown on the next slide

Sound (cont'd.)


sf::Sound
Pointer to sound buffer
sf::SoundBuffer
All samples
Initial loading
All samples
Audio file on hard disk
sf::Music
Current chunk of samples
Continuous stream
All samples
Current chunk
Audio file on hard disk

Using Resources

- Game entities like the player and enemies are represented by sprites and text
  - They access the source files but do not own them
- All resources must remain in scope for however long they are needed in the game
- Game entities are separated from any sounds that are associated with them
  - For example, we want to play the enemy death sound even if the enemy object has been removed

Using Resources (cont'd.)

- Typically, you want to load the resource before you use it – for example, upon the start of a game or new level
  - So the game performance is not affected
- Similarly, release resources at the end of a level or game
- You will have a class that manages that functionality
  - RAII or Resource Acquisition Is Initialization
  - http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

Using Resources (cont'd.)

- Choosing the right data structure to hold the resources is important as well
  - We're going to use a map (`std::map`) for our example as we're not going to change its size
  - We want to avoid reallocating the structure
  - We're also going to be using an enumeration to act as our key type for our textures
- Let's start creating our containers

Working with Textures

```
class TextureHolder
{
   public:
   void load(Textures::ID id, const std::string& filename);

   private:
   std::map<Textures::ID,std::unique_ptr<sf::Texture>> mTextureMap;
};

void TextureHolder::load(Textures::ID id, const std::string& filename)
{
   std::unique_ptr<sf::Texture> texture(new sf::Texture());
   texture->loadFromFile(filename);

   mTextureMap.insert(std::make_pair(id, std::move(texture)));
}
```

Type Inference

- Woah, hold on there… what was that `auto` type?
  - New C++ 11 feature
  - The `auto` keyword deduces what the variable should be based on the left side of the assignment
  - Articles below explain it more:
    - http://www.cprogramming.com/c++11/c++11-auto-decltype-return-value-after-function.html
    - http://en.wikipedia.org/wiki/C%2B%2B11

Accessing the Textures

- Now we want to grant access to our textures via a few accessors/getters:

```
sf::Texture& get(Textures::ID id);

const sf::Texture& get(Textures::ID id) const;

sf::Texture& TextureHolder::get(Textures::ID id)
{
  auto found = mTextureMap.find(id);
  return *found->second;
}
```

A const method can be called on a const object

```
class CL2 {
 public: void const_method() const;
void method();
private: int x;
 };

const CL2 co;
CL2 o;
co.const_method(); // legal
co.method(); // illegal, can't call regular method on const object
o.const_method(); // legal, can call const method on a regulard object
o.method(); // legal
```

New TextureHolder

- Our `TextureHolder` class now looks like this:

```
class TextureHolder
{
    public:
    void load(Textures::ID id, const std::string& filename);
    sf::Texture& get(Textures::ID id);
    const sf::Texture& get(Textures::ID id) const;

    private:
    std::map<Textures::ID,std::unique_ptr<sf::Texture>> mTextureMap;
};
```

- And can be used in the following manner:

```
TextureHolder textures;
textures.load(Textures::Airplane, "Media/Textures/Airplane.png");

sf::Sprite playerPlane;
playerPlane.setTexture(textures.get(Textures::Airplane));
```

Error Handling

- The program could encounter many errors in the program, and it is important to account for them
- So let's add some of the more common approaches

```
if (!texture->loadFromFile(filename))
  throw std::runtime_error("TextureHolder::load -
  Failed to load "+ filename);
```

Error Handling (cont'd.)

- Let's have a look at the full `load` method:

```
void TextureHolder::load(Textures::ID id, const std::string&
   filename)
{
   std::unique_ptr<sf::Texture> texture(new sf::Texture());

   if (!texture->loadFromFile(filename))
        throw std::runtime_error("TextureHolder::load - Failed to
        load " + filename);

   auto inserted = TextureMap.insert(std::make_pair(id,
   std::move(texture)));

   assert(inserted.second);
}
```

Error Handling (cont'd.)

- We can add an `assert` to the `get` method too:

```
sf::Texture& TextureHolder::get(Textures::ID id)
{
  auto found = mTextureMap.find(id);
  assert(found != mTextureMap.end());
  return *found->second;
}
```

Error Handling (cont'd.)

- We can add an `assert` to the `get` method too:

```
sf::Texture& TextureHolder::get(Textures::ID id)
{
  auto found = mTextureMap.find(id);
  assert(found != mTextureMap.end());
  return *found->second;
}
```

Generalized Template

- Our `TextureHolder` is great, but can we create others for different resources?
  - Yes, but we can do something even better!
  - Alter `TextureHolder` and make it a general class

  - We can call it `ResourceHolder`
  - It will be a template class with two template parameters
    - The type of resource and an ID type for resource access

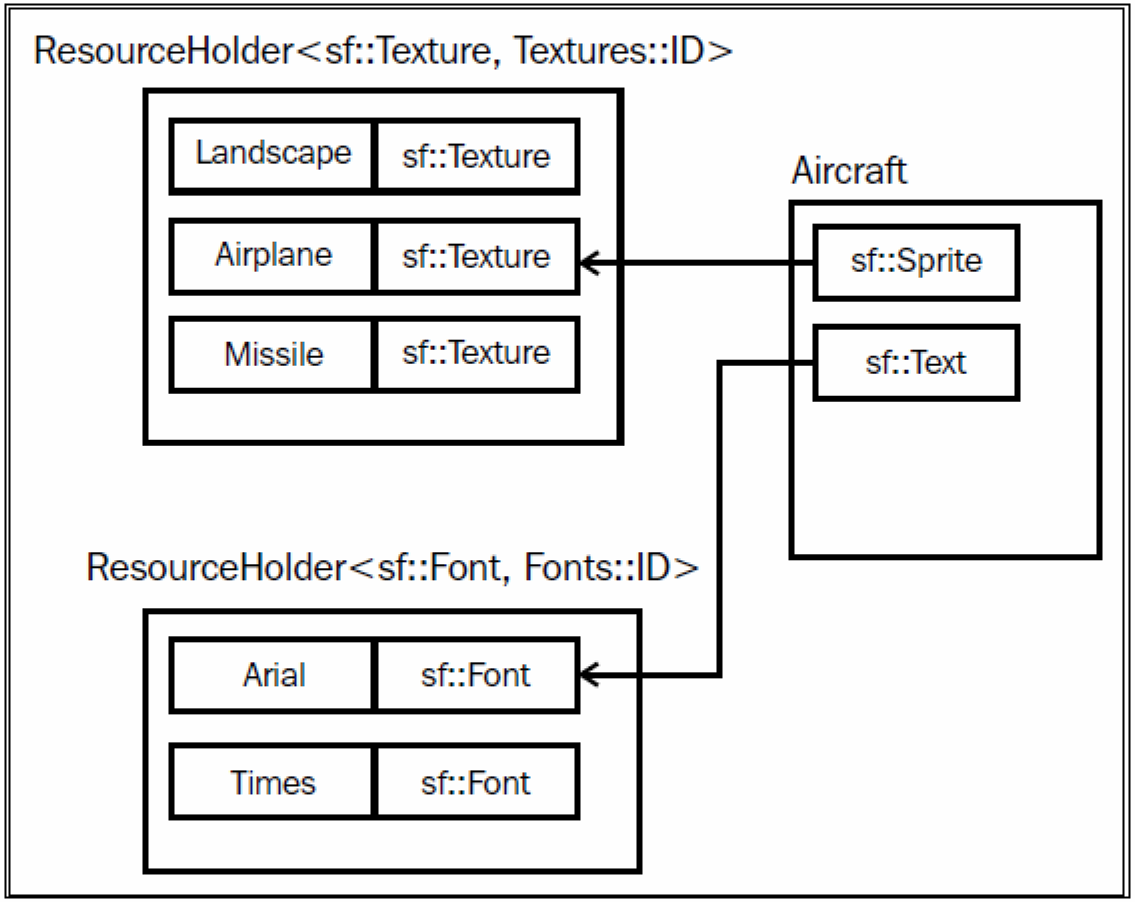
ResourceHolder Class

```
template <typename Resource, typename Identifier>
class ResourceHolder
{
  public:
    void load(Identifier id, const std::string& filename);
    Resource& get(Identifier id);
    const Resource& get(Identifier id) const;
  private:
    std::map<Identifier,
    std::unique_ptr<Resource>> mResourceMap;
};
```

load Method

```
template <typename Resource, typename Identifier>
void ResourceHolder<Resource, Identifier>::load(Identifier id,
const std::string& filename)
{
   std::unique_ptr<Resource> resource(new Resource());
   if (!resource->loadFromFile(filename))
       throw std::runtime_error("ResourceHolder::load - Failed to
       load " + filename);
   auto inserted = mResourceMap.insert(std::make_pair(id,
   std::move(resource)));
   assert(inserted.second);
}
```

Visualizing ResourceHolder



Appendix A

- Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
 T result;
 result = (a>b)? a : b;
 return (result);
}
int main () {
 int i=5, j=6, k;
 long l=10, m=5, n;
 k=GetMax<int>(i,j);
 n=GetMax<long>(l,m);
 cout << k << endl;
 cout << n << endl;
 return 0;
}
```