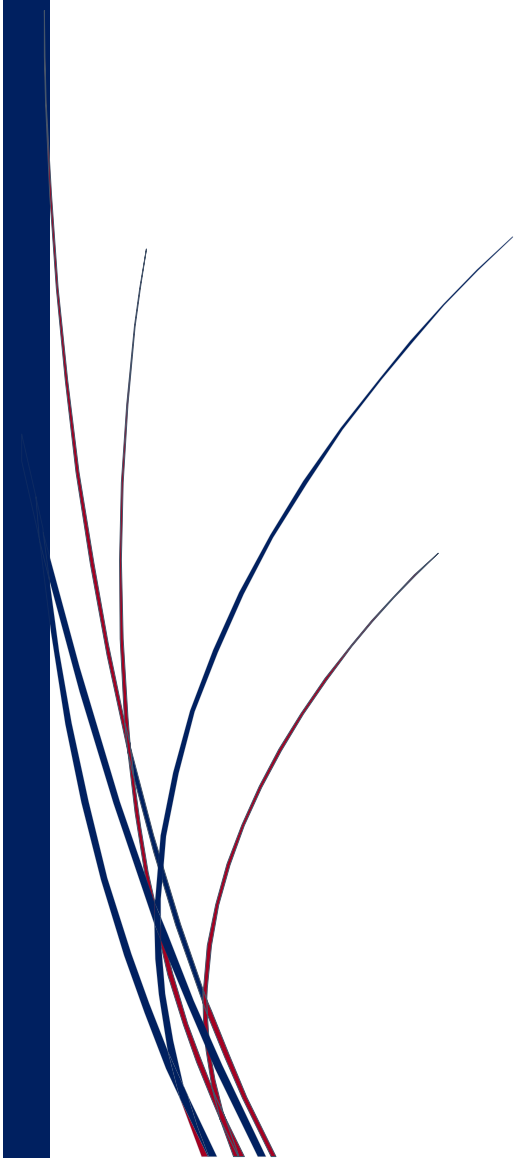
Media Specialist Practice

Game Engine Architecture



Gordon Johnson
K1451760

Table of Contents

Introduction

As with many software products that we use daily on our computers, we take many simple tasks for granted, and this is true of Game Engines such as Unity and Unreal. Behind the scenes Game engines are vast in size, the purpose of this project is to improve understanding and to create a crude implementation of a game engine framework. Initially, the focus will be rendering objects to the screen and manipulating them, this can then be extended to cater for textures and fonts.

To achieve this, I will first investigate what a Game Engine involves by looking at the component of what is typically running behind the scenes and the technologies used. Following this, I will analyse what is feasible within the time constraints and make decisions on what technology to use and which components to implement.

If the technology being used, is new to me, I will utilise books, videos and official documentation to understand, learn and then implement the proposed Game Engine Architecture, into a working model.

Portfolio: http://gordon.johnson7.co.uk/

Box Link: https://kingston.box.com/s/4kcwuyb5v5ooywkor0p0e3wkjsrvso39

GitHub: https://github.com/LordGee/gge

YouTube: https://youtu.be/C3fwU6IsSTA

Game Engine and its Core Components

A Game Engine is the software that is primarily used to develop games but can be used for the development of other artefacts that require visual rendering. The game engine provides tools to assist in this development, which can be utilised either via coding or a GUI environment. Most are developed for a specific type of game genre, which will produce the envisioned end-product, other engines such as Unity or Unreal, are more generic and can be applied to many different genres. As seen in (Figure 1), a game engine can consist of many components.

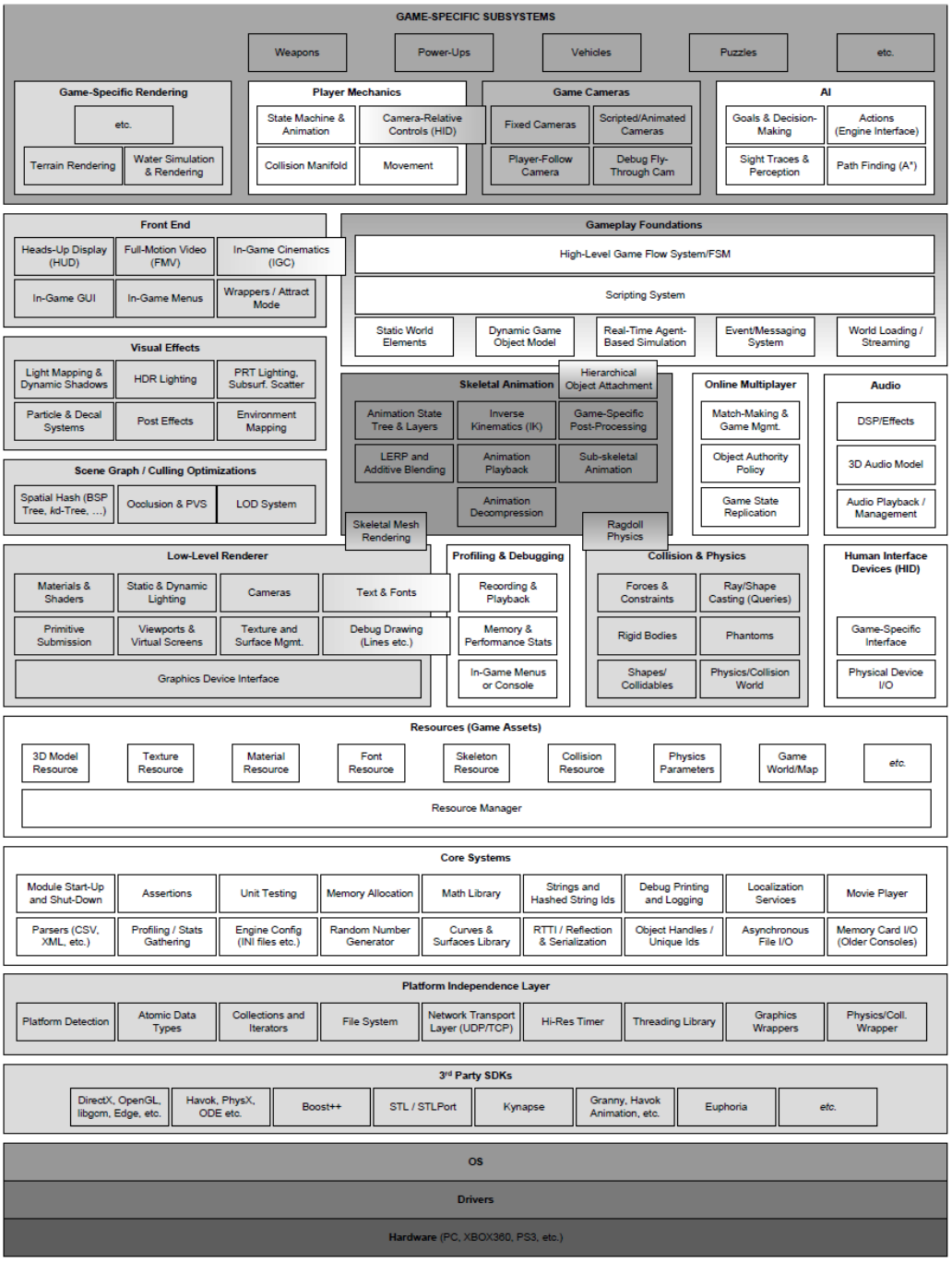


Figure 1 - Runtime game engine architecture. (Gregory, 2014)

Rendering Component

The low-level rendering component of a Game Engine, is a key component and is responsible for the communication between the engine and the graphics processing unit. An engine can specialise in 2D and or 3D graphics rendering, this is a decision determined early in its development lifecycle. When rendering objects, this component should handle Shaders, Lighting, Textures, Fonts and Camera Perspectives. “At this level, the design is focused on rendering a collection of geometric primitives, as quickly and richly as possible” (Gregory, 2014). Everything revolves around the rendering component and without it nothing will be displayed on the screen which would result in a poor game.

Physics & Collisions Component

The physics component, is responsible for providing realism to the objects within the game, by simulating behaviour such as gravity, movement and colliding with other objects. Depending on the object, if a collision is detected, the on-looker would expect to see a cause and effect, where an object may bounce or provide force to an opposing object. A practical example would be when a car starts to brake, or acceleration is released, with a physics component the car would slow down, but this would also depend on the mass / weight, dimensions of the car, as well as any external conditions such as friction, wind and depending on the game the condition of the car.

Animation Component

The animation component is responsible for bringing objects to life. There are two main types of animation that can be implemented into this component, Sprite and Skeletal animation. The sprite animation type, would deal with 2D animation, this would typically be read from a sprite sheet, where each texture on an image, has been pre-drawn for each frame of the animation. The skeletal animation, is more advanced and can be used for both 2D and 3D objects, by simulating the animation based on bone and joint positions and allowing the mesh to move with them.

Audio Component

The audio component is responsible for handling the loading and manipulation of sound. Like the other three components, the audio component adds another dimension to the engine. Having background noise / music, plus sound effects such as footsteps, adds a layer of immersive experience and additional realism to the game.

There are many other components not mentioned, these include but not limited to, Interfaces, Networking, Inputs, Scenes, Events, Scripting, AI and many more.

Design Decisions

A Game Engine consists of many components, this project will focus on the rendering component. This is probably the key aspect of the entire engine, as other components tend to complement the rendering process. I have opted to use the OpenGL specification, as it is well documented, cross platform and has an extensive community following, alternatives to this could have been DirectX or Vulkan. To use OpenGL, I will utilise GLEW and GLFW, to create the window and rendering components. The project will predominantly be coded in the C++ language.

Some of the third-party libraries I will utilise are:

Free Image – will be used to load in images that can be used as texture and added to sprites within the game. This library is also used by Unity Game Engine.

Free Type – will be used to load in fonts and assist in the positioning, so the text is correctly rendered. This library, is used by many operating systems including Android and iOS.

Gorilla Audio – will be used to load in audio clips and send the clips out of the appropriate audio hardware.

Aims & Objectives

The aims and objectives are broken down into MoSCoW tables, to provide a priority list. This will be a working document and will change over the course of the development lifespan. These are as follows:

Must Have

Features	Description
Window	Create a class that handles the window creation and size.
2D Rendering	Create a class for rendering 2D objects and a class to manage the renderable object, whether a sprite, text or other.
Maths Library	Create a Maths class to handle reusable mathematic features, plus hand structures such as Vector2, Vector3, Vector4 and Matrix 4x4.
Game Loop	Create a Loop class, that handles the main game loop and exposes functions such as Start and Update to the game development area.
Sprites	Create a sprite class, that can include a texture or an RGBA colour, as a renderable object. This class should also contain position and size.
Textures	Create a textures class, using a third-party library (FreeImage, 2015) to load in the images correctly from file.

Should Have

Features	Description
Text / Fonts	Using a third-party library (FreeType, 2018), create a font manager class to add new fonts to a collection, which can then be accessed and rendered to the screen. The font class, handles the font itself and is a type of renderable, this object can then be passed to the renderer to display the text.
Audio	Using a third-party library (Gorilla Audio, 2014), create audio manager class to add new audio samples to a collection, which can then be accessed and returns a type of audio. Create audio class, which handles the audio itself, by containing functions such as play, pause, resume and stop.

Could Have

Features	Description
LUA	Expose the C++ classes to LUA, as a high-level scripting language. All the low-level tools implemented in C++ will then be accessible to the game developer in LUA.
Interaction and Interface	Create input class, to allow interaction with interface widgets allowing selections e.g. buttons, sliders etc.
Scene Manager	Create a scene class, to manage different levels, menu screens etc.
3D Rendering	Create a class for rendering 3D objects and a class to manage the 3D specific renderable object, such as meshes.
Physics / Collision	Use a third -party library, such as (ODE, 2014), to manage physics and collisions.

Won't Have

- Animation
- Networking
- Artificial Intelligence
- Particle System
- Event System
- GUI Interface

Implementation

Coding

The following implementation section, highlights some of the main functions and classes within the engine.

Main Game Loop in the GGEngine Class

The main game loop is responsible for ensuring that the game remains running, this loop will continue until the game window has been closed. During this loop, certain pure functions are called that are override in the primarily Game class, however this can include other classes that inherit the 'GGEngine' class.

```
void Run() {
	/* Set new Timer and initialise other variables*/
	m_Timer = new Timer();
	m_Time = 0.0f;
	float updateTimer = 0.0f;
	const float updateTick = 1.0f / 60.0f;
	unsigned int frames = 0, updates = 0;
	/* Main Game Loop until Window is closed */
	while (!m_Window->IsClosed()) {
		/* Reset Window content */
		m_Window->WindowClear();
		/* If 60 times per second */
		if (m_Timer->Elapsed() - updateTimer > updateTick) {
			/* Execute Update functions */
			Update();
			updateTimer = m_Timer->Elapsed();
			updates++;
		}
		/* Execute the Fast Update function */
		FastUpdate();
		/* Execute the Render functions */
		Render();
		frames++;
		/* Perform the Window update */
		m_Window->WindowUpdate();
		/* If once per second */
		if (m_Timer->Elapsed() - m_Time > 1.0f) {
			m_Time = m_Timer->Elapsed();
			m_FramesPerSecond = frames;
			m_UpdatesPerSecond = updates;
			frames = updates = 0;
			/* Execute the Tick functions */
			Tick();
		}
	}
	/* Deconstruct the Window */
	m_Window->WindowDestroy();
}
```

Game Class

The game class is the template in which to develop your game, the structure of the header file is as follows:

```
#define GAME_NAME "My Game"
#define WINDOW_WIDTH 1920
#define WINDOW_HEIGHT 1080

class Game : public GGEngine {
private:
	Window *	m_Window;

public:
	Game() {
		m_Window = CreateWindow(
			GAME_NAME,
			WINDOW_WIDTH,
			WINDOW_HEIGHT
		);
	}
	~Game() { }
	void Start() override;
	void Tick() override;
	void Update() override;
	void FastUpdate() override;
	void Render() override;
};
```

This provides an almost blank scenario, with only the Window being created for you in the constructor. The five override functions can be implemented in the CPP file:

```
#include "Game.h"

/* Start() executes only at the start of the game */
void ::Game::Start() { }

/* Tick() executes once per second */
void ::Game::Tick() { }

/* Update() executes 60 times a second */
void ::Game::Update() { }

/* FastUpdate() executes as fast as the Render method */
void Game::FastUpdate() { }

/* Render() executes as fast as possible */
void ::Game::Render() { }
```

To give functions a meaning, I have included a comment to indicate how and when, each function will be executed. Examples for the game class can be found in the technical documentation or at (Johnson, 2018).

Initialise function in the Window Class

After creating a new instance of the Window class, by passing through the name, width, height or optionally declare it full screen, the initialiser is then executed.

```
bool Window::Init() {
	// Intialise GLFW
	// Create Window
	// Check Window created successfully
	// Set Window to the current Thread
	// Setup call backs
	// Initialise GLEW
	// Enable blend functionality
	// Log and return
}
```

The main part of this function, is to create a new GLFW Window, from the get started code found on (GLFW, 2018). I've modified this, to allow for the possibility of creating the window full screen, by passing a Boolean value in the constructor.

```
if (m_FullScreen) {
	/* Gets the primary monitor used by the user */
	GLFWmonitor* m_Monitor = glfwGetPrimaryMonitor();
	const GLFWvidmode* mode = glfwGetVideoMode(m_Monitor);
	/* resets the width and height to the size of the monitor */
	m_Width = mode->width;
	m_Height = mode->height;
	/* Generate window based on parameters */
	m_Window = glfwCreateWindow(m_Width, m_Height, m_Title, m_Monitor, nullptr);
} else {
	/* Generate window based on parameters */
	m_Window = glfwCreateWindow(m_Width, m_Height, m_Title, nullptr, nullptr);
}
```

Batch Renderer Class

The Batch Renderer class, is responsible for the Initialise, Begin, Submit, End and Flush processes, which are executed from the container class that holds the renderable items. The Batch Renderer, Renderer, Shader and Index Buffer class, were implemented with the guidance of (Cherno, 2018), (Wolff, 2013) and (Hussain, 2017) OpenGL tutorials. The Vertex Data, is initialised as a Map Buffer, which "maps the entire data store of a specified buffer object into the client's address space." (Khronos, 2018). This makes the rendering process much faster, compared to rendering one object at a time.

Initialise the Batch Renderer

```
void BatchRenderer::Init() {
	/* Returns vertex array object names */
	glGenVertexArrays(1, &m_VertexArrayObject);
	/* Returns buffer object names */
	glGenBuffers(1, &m_VertexBufferObject);
	/* Binds the vertex array object with name */
	glBindVertexArray(m_VertexArrayObject);
	/* Binds a buffer object to the specified buffer
	 * binding point */
	glBindBuffer(GL_ARRAY_BUFFER, m_VertexBufferObject);
	/* Create a new data store for a buffer object */
	glBufferData(GL_ARRAY_BUFFER, BUFFER_SIZE, NULL,
		GL_DYNAMIC_DRAW);
	/* Enable the generic vertex attribute array specified
	 * by index */
	glEnableVertexAttribArray(ATTR_VERTEX_INDEX);
	glEnableVertexAttribArray(ATTR_COLOUR_INDEX);
	glEnableVertexAttribArray(ATTR_UV_INDEX);
	glEnableVertexAttribArray(ATTR_TEXID_INDEX);
	/* Specify the location and data format of the
	 * array of generic vertex attributes at index to use
	 * when rendering */
	glVertexAttribPointer(ATTR_VERTEX_INDEX, 3, GL_FLOAT,
		GL_FALSE, VERTEX_SIZE, (const GLvoid*)0);
	glVertexAttribPointer(ATTR_COLOUR_INDEX, 4, GL_UNSIGNED_BYTE,
		GL_TRUE, VERTEX_SIZE,
		(const GLvoid*)(offsetof(VertexData, VertexData::colour)));
	glVertexAttribPointer(ATTR_UV_INDEX, 2, GL_FLOAT,
		GL_FALSE, VERTEX_SIZE,
		(const GLvoid*)(offsetof(VertexData, VertexData::textureCoord)));
	glVertexAttribPointer(ATTR_TEXID_INDEX, 1, GL_FLOAT,
		GL_FALSE, VERTEX_SIZE,
		(const GLvoid*)(offsetof(VertexData, VertexData::textureID)));
	/* Unbinds a buffer object */
	glBindBuffer(GL_ARRAY_BUFFER, 0);
	/* Generates the indicies values that will draw
	 * two triagles to make a square */
	GLuint* indicies = new GLuint[INDICES_SIZE];
	int offset = 0;
	for (int i = 0; i < INDICES_SIZE; i += 6) {
		indicies[i] = offset + 0;
		indicies[i + 1] = offset + 1;
		indicies[i + 2] = offset + 2;
		indicies[i + 3] = offset + 2;
		indicies[i + 4] = offset + 3;
		indicies[i + 5] = offset + 0;
		offset += 4;
	}
	/* Create instance of the IndexBuffer pasiing over the indicies
	 * data and size */
	m_IndexBufferObject = new IndexBuffer(indicies, INDICES_SIZE);
	/* Unbinds the vertex array object */
	glBindVertexArray(0);
}
```

Begin in the Batch Renderer Class

The begin function, binds the Vertex Buffer object and sets an instance of the Vertex Data as a Map Buffer array.

```
void BatchRenderer::Begin() {
	/* Binds a buffer object to the specified buffer binding point */
	glBindBuffer(GL_ARRAY_BUFFER, m_VertexBufferObject);
	/* Map the entire data store of a specified buffer object */
	m_Buffer = (VertexData*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
}
```

Submit in the Batch Renderer Class

The submit function, has been broken down into three sections. The first is initialising the variable that will be allocated to the Vertex Data map. The second is to get the texture ID and the third is to set the final Vertex Data values.

```
void BatchRenderer::Submit(const Renderable* renderable) {
	// Set values based on renderable object
	// Identify Texture ID
		// if not found add to array
		// if max textures, then end, flush and begin
	// Set Index Buffer values
}
```

Submit Set Values

The values to be initialised, are obtained from the renderable object. These are stored as constant values, to prevent accidental changes. They should not be changed at this point.

```
// Set values based on renderable object
const maths::Vector3& position = renderable->GetPosition();
const maths::Vector2& size = renderable->GetSize();
const unsigned int colour = renderable->GetColour();
const std::vector<maths::Vector2>& uv = renderable->GetUV();
const GLuint textureID = renderable->GetTextureID();
```

Submit Get Texture ID

If a texture ID has been set, a check is performed to see if it already exists in the texture slots array. This will ensure that if the texture required is already in the array, it can be re-used. If it's not been found, a check is done to see if the texture slot array has reached its maximum value. If it has, everything already stored will be processed, this will clear out everything already added, allowing room for additional textures to be added. Either way the new texture ID is added to the array.

```
// Identify Texture ID
float textureSlot = 0.0f;
if (textureID > 0) {
	bool found = false;
	for (int i = 0; i < m_TextureSlots.size(); i++) {
		if (m_TextureSlots[i] == textureID) {
			textureSlot = (float)(i + 1);
			found = true;
			break;
		}
	}
	// if not found add to array
	if (!found) {
		// if max textures, then end, flush and begin
		if (m_TextureSlots.size() >= MAXIMUM_TEXTURES) {
			End();
			Flush();
			Begin();
		}
		m_TextureSlots.push_back(textureID);
		textureSlot = (float)m_TextureSlots.size();
	}
}
```

Submit Set Index Buffer Values

Each point is added to the Vertex Data map.

```
// Set Vertex Data values
m_Buffer->vertex = *m_TransformationBack * position;
m_Buffer->textureCoord = uv[0];
m_Buffer->textureID = textureSlot;
m_Buffer->colour = colour;
m_Buffer++;
m_Buffer->vertex = *m_TransformationBack * maths::Vector3(position.x, position.y +
size.y, position.z);
m_Buffer->textureCoord = uv[1];
m_Buffer->textureID = textureSlot;
m_Buffer->colour = colour;
m_Buffer++;
m_Buffer->vertex = *m_TransformationBack * maths::Vector3(position.x + size.x,
position.y + size.y, position.z);
m_Buffer->textureCoord = uv[2];
m_Buffer->textureID = textureSlot;
m_Buffer->colour = colour;
m_Buffer++;
m_Buffer->vertex = *m_TransformationBack * maths::Vector3(position.x + size.x,
position.y, position.z);
m_Buffer->textureCoord = uv[3];
m_Buffer->textureID = textureSlot;
m_Buffer->colour = colour;
m_Buffer++;
m_IndexCount += 6;
```

End in the Batch Renderer Class

Unbinds the Map Buffer used by the Vertex Data and unbinds the Vertex Buffer object.

```
void BatchRenderer::End() {
	/* Unmaps the buffer array */
	glUnmapBuffer(GL_ARRAY_BUFFER);
	/* Unbinds the buffer array */
	glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Flush in the Batch Renderer Class

The final step of the renderer, is to flush out everything stored, by applying the textures and drawing the object.

```
void BatchRenderer::Flush() {
	/* Binds and Activates all Textures */
	for (int i = 0; i < m_TextureSlots.size(); i++) {
		glActiveTexture(GL_TEXTURE0 + i);
		glBindTexture(GL_TEXTURE_2D, m_TextureSlots[i]);
	}
	/* Binds the Vertex array object */
	glBindVertexArray(m_VertexArrayObject);
	/* Bind the index buffer */
	m_IndexBufferObject->BindIndexBuffer();
	/* Draw all elemets */
	glDrawElements(GL_TRIANGLES, m_IndexCount, GL_UNSIGNED_INT, NULL);
	/* Unbind Index Buffer */
	m_IndexBufferObject->UnbindIndexBuffer();
	/* Unbind Vertex Array */
	glBindVertexArray(0);
	/* Reset index count to 0 ready for next frame */
	m_IndexCount = 0;
	/* Clear all texture slots */
	m_TextureSlots.clear();
}
```

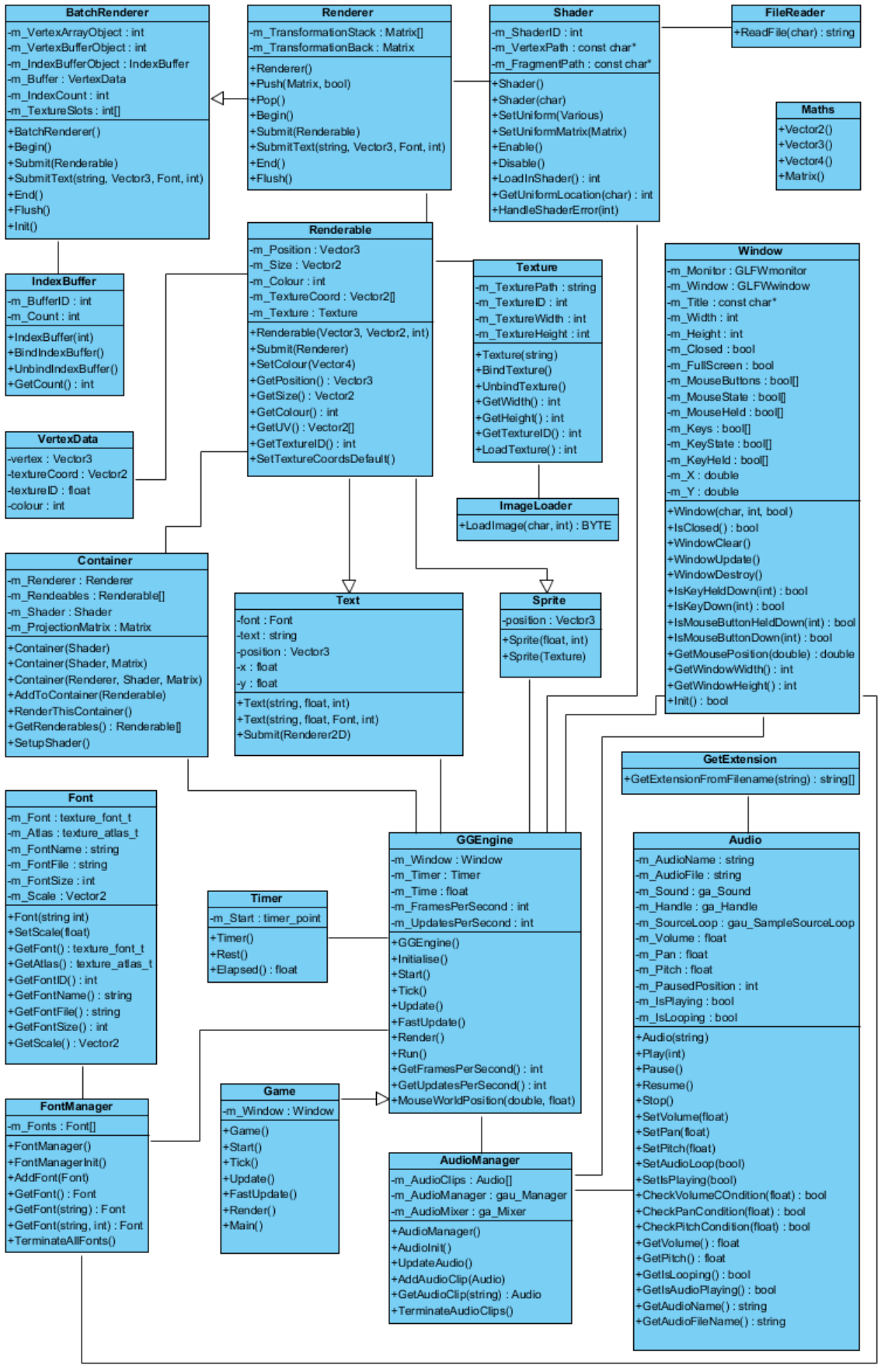
Shader Class

The shader class, is responsible for setting up the program object, by reading the fragment and vertex shaders. These shaders, are assigned, compiled and checked for any errors, before being attached to the shader program.

Load in Shader Function

```
GLuint Shader::LoadInShader() {
	/* Creates an empty program object and returns a
	 * non-zero value by which it can be referenced */
	GLuint program = glCreateProgram();
	/* Creates an empty shader object and returns a
	 * non-zero value by which it can be referenced */
	GLuint vertex = glCreateShader(GL_VERTEX_SHADER);
	GLuint fragment = glCreateShader(GL_FRAGMENT_SHADER);
	/* Loads in the shader and stored in a string variable */
	std::string vertSourceString = ReadFile(m_VertexPath);
	std::string fragSourceString = ReadFile(m_FragmentPath);
	/* Convert string to a const char* */
	const char* vertexSource = vertSourceString.c_str();
	const char* fragmentSource = fragSourceString.c_str();
	/* Sets the source code in shader to the vertex shader and sets
	 * the source code to the value the from the vertex file */
	glShaderSource(vertex, 1, &vertexSource, NULL);
	/* Compiles the source code that have been stored in the
	 * vertex shader object */
	glCompileShader(vertex);
	GLint result;
	/* Gets the vertex shaders, compile status which returns
	 * GL_TRUE if the last compile operation on shader was
	 * successful, and GL_FALSE otherwise */
	glGetShaderiv(vertex, GL_COMPILE_STATUS, &result);
	if (result == GL_FALSE) {
		HandleShaderError(vertex);
		return 0;
	}
	/* Sets the source code in shader to the fragment shader and sets
	* the source code to the value the from the fragment file */
	glShaderSource(fragment, 1, &fragmentSource, NULL);
	/* Compiles the source code that have been stored in the
	* fragment shader object */
	glCompileShader(fragment);
	result = NULL;
	/* Gets the fragment shaders, compile status which returns
	* GL_TRUE if the last compile operation on shader was
	* successful, and GL_FALSE otherwise */
	glGetShaderiv(fragment, GL_COMPILE_STATUS, &result);
	if (result == GL_FALSE) {
		HandleShaderError(fragment);
		return 0;
	}
	/* Attaches the shader to the specified program object */
	glAttachShader(program, vertex);
	glAttachShader(program, fragment);
	/* Links the program object specified */
	glLinkProgram(program);
	/* Checks to see whether the executables contained in program
	 * can execute given the current OpenGL state */
	glValidateProgram(program);
	/* Deletes the no longer need shader IDs */
	glDeleteShader(vertex);
	glDeleteShader(fragment);
	return program;
}
```

Class Diagram



Technical Documentation

Window Class

Constructor

```
Window(const char* name, int width, int height, bool fullScreen = false);
```

Example code:

- Define Resolution:

```
Window* window = CreateWindow("My Game", 1920, 1080);
```

- Full Screen:

```
Window* window = CreateWindow("My Game", 1920, 1080, true);
```

Methods

Is Closed

Returns a Boolean, depending on the state of the window. For example, returns false if the Window has closed.

```
bool IsClosed();
```

Example code:

```
Window* window = CreateWindow("GG Engine", 1920, 1080);
while (!window->IsClosed()) {
	// Do Something
}
```

Window Clear

Clears the window ready for the next update.

```
void WindowClear();
```

Example code:

```
Window* window = CreateWindow("GG Engine", 1920, 1080);
while (!window->IsClosed()) {
	window->WindowClear();
}
```

Window Update

Updates the window with any new draw calls.

```
void WindowUpdate();
```

Example code:

```
Window* window = CreateWindow("GG Engine", 1920, 1080);
while (!window->IsClosed()) {
	window->WindowClear();
	window->WindowUpdate();
}
```

Window Destroy

Destroys the window appropriately after use.

```
void WindowDestroy();
```

Example code:

```
Window* window = CreateWindow("GG Engine", 1920, 1080);
while (!window->IsClosed()) {
	window->WindowClear();
	window->WindowUpdate();
}
window->WindowDestroy();
```

Is Key Held Down

Checks if the key is held down.

```
bool IsKeyHeldDown(unsigned int keycode);
```

Example code:

```
if (window->IsKeyHeldDown(GLFW_KEY_SPACE)) {
	std::cout << "Space Bar Pressed" << std::endl;
}
```

Is Key Down

Checks if the key is pressed, only returns once per key down action.

```
bool IsKeyDown(unsigned int keycode);
```

Example code:

```
if (window->IsKeyDown(GLFW_KEY_SPACE)) {
	std::cout << "Space Bar Pressed" << std::endl;
}
```

Is Mouse Button Held Down

Checks if the mouse button is held down.

```
bool IsMouseButtonHeldDown(unsigned int button);
```

Example code:

```
if (window->IsMouseButtonHeldDown(GLFW_MOUSE_BUTTON_1)) {
	std::cout << "Left mouse button down" << std::endl;
}
```

Is Mouse Button Down

Checks if the mouse button is held down, only returns once per mouse button down action.

```
bool IsMouseButtonDown(unsigned int button);
```

Example code:

```
if (window->IsMouseButtonDown(GLFW_MOUSE_BUTTON_1)) {
	std::cout << "Left mouse button down" << std::endl;
}
```

Get Mouse Position

Updates X and Y values passed in argument, with the current mouse position.

```
void GetMousePos(double& xpos, double& ypos);
```

Example code:

```
double mouse_X, mouse_Y;
window->GetMousePos(mouse_X, mouse_Y);
std::cout << "X Pos: " << mouse_X << " Y Pos: " << mouse_Y << std::endl;
```

Shader Class

Constructor

```
Shader();
Shader(const char* vertexPath, const char* fragmentPath);
```

Example code:

- Default shaders allocated:

```
Shader* shader = new Shader();
```

- Defined shaders allocated:

```
Shader* shader = new Shader("src/shaders/basic.vert", "src/shaders/basic.frag");
```

Methods

Set Uniform Values

Choose appropriate method, depending on the uniform data type, that requires to be set.

```
void SetUniform1f(const GLchar* name, float value);
void SetUniform1i(const GLchar* name, int value);
void SetUniform1fv(const GLchar* name, int count, float* value);
void SetUniform1iv(const GLchar* name, int count, int* value);
void SetUniform2f(const GLchar* name, const maths::Vector2& vec2);
void SetUniform3f(const GLchar* name, const maths::Vector3& vec3);
void SetUniform4f(const GLchar* name, const maths::Vector4& vec4);
void SetUniformMatrix(const GLchar* name, const maths::Matrix& matrix);
```

Enable / Disable

The shader needs to be enabled, before setting a new uniform value and should be disabled afterwards.

```
void Enable() const;
void Disable() const;
```

Texture Class

Textures, can be allocated to a Sprite renderable.

Constructor

```
Texture(const std::string filepath);
```

Example code:

```
Texture* image = new Texture("img/test.png");
```

Sprite Class

Constructor

```
Sprite(float x, float y, float width, float height, unsigned int colour);
Sprite(float x, float y, float width, float height, Texture* texture);
```

Example code:

- Assign with colour.

```
Sprite* sprite = new Sprite(0, 0, 2, 2, 0xffffffff);
```

- Assign with a texture.

```
Texture* image = new Texture("img/test.png");
Sprite* sprite = new Sprite(0, 0, 2, 2, image);
```

Container Class

Constructor

```
Container(Shader* shader);
Container(Shader* shader, maths::Matrix projectionMatrix);
Container(Renderer* renderer, Shader* shader, maths::Matrix projectionMatrix);
```

Example code:

- Default renderer and projection matrix allocated.

```
Shader* shader = new Shader();
Container* bucket = new Container(shader);
```

- Default renderer allocated.

```
Shader* shader = new Shader();
Container* bucket = new Container(shader,
	Matrix::orthographic(-16.0f, 16.0f, -9.0f, 9.0f, -1.0f, 1.0f));
```

- Defined renderer and projection matrix.

```
Shader* shader = new Shader();
Container* bucket = new Container(new BatchRenderer(), shader,
	Matrix::orthographic(-16.0f, 16.0f, -9.0f, 9.0f, -1.0f, 1.0f));
```

Methods

Add to this Container

Add a renderable object to the container array.

```
virtual void AddToContainer(Renderable* renderable);
```

Example code:

```
Shader* shader = new Shader();
Container* bucket = new Container(shader);
Texture* image = new Texture("img/test.png");
Sprite* sprite = new Sprite(0, 0, 2, 2, image);
bucket->AddToContainer(sprite);
```

Render this Container

Each container that rendered, should call this method in the Render loop.

```
virtual void RenderThisContainer();
```

Example code:

```
void Example::Render() {
	bucket->RenderThisContainer();
}
```

Font Manager Class

Add new fonts to the font manager class.

Methods

Add Font

Add a new font to the collection.

```
static void AddFont(Font* font);
```

Example code:

```
FontManager::AddFont(new Font("MyFont", "font/Muli-Bold.ttf", 32));
```

Text Class

Once a font has been added to the Font Manager class, then either that one or the default font, can be used to render text to the display. The text class is also a type of renderable, so any Text objects can be added to an existing container for rendering.

Constructor

```
Text(std::string text, float x, float y, unsigned int colour);
Text(std::string text, float x, float y, Font* font, unsigned int colour);
Text(std::string text, float x, float y, const std::string& font,
       unsigned int colour);
Text(std::string text, float x, float y, const std::string& font,
       unsigned int size, unsigned int colour);
```

Example code:

```
Shader* shader = new Shader();
Container* bucket = new Container(shader);
FontManager::AddFont(new Font("MyFont", "font/Muli-Bold.ttf", 32));
Text* label = new Text("Hello", -13.5f, 6.5f, "MyFont", GGE_COL_BLACK);
bucket->AddToContainer(label);
```

Audio Manager Class

Add new audio clips to the audio manager class and get audio clips when required.

Methods

Add Audio Clip

Add a new audio clip to the collection.

```
static void AddAudioClip(Audio* audio);
```

Example code:

```
AudioManager::AddAudioClip(new Audio("SFX", "audio/BeBop.wav"));
```

Get Audio Clip

Get an existing audio clip from the collection.

```
static Audio* GetAudioClip(const std::string& name);
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
```

Audio Class

Methods

Play Audio Clip

Start playing an audio clip.

```
void Play(unsigned int override = 0);
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
sfx->Play();
```

Pause Audio Clip

Pause an audio clip currently playing.

```
void Pause();
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
sfx->Play();
sfx->Pause();
```

Resume Audio Clip

Resume an already paused audio clip.

```
void Resume();
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
sfx->Play();
sfx->Pause();
sfx->Resume();
```

Stop Audio Clip

Stop an audio clip that is currently playing.

```
void Stop();
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
sfx->Play();
sfx->Pause();
sfx->Resume();
sfx->Stop();
```

Set Volume for Audio Clip

Adjust the volume level for the audio clip, this can be set between 0.0 (quiet) and 1.0 (load).

```
void SetVolume(float volume);
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
sfx->Play();
sfx->SetVolume(0.5f);
sfx->Stop();
```

Set Pan for Audio Clip

Adjust the pan for the audio clip, this can be set between -1.0 (left) and 1.0 (right). Set to 0.0 for centre.

```
void SetPan(float pan);
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
sfx->Play();
sfx->SetPan(-0.9f);
sfx->Stop();
```

Set Pitch for Audio Clip

Adjust the pitch for the audio clip.

```
void SetPitch(float pitch);
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
sfx->Play();
sfx->SetPitch(8.0f);
sfx->Stop();
```

Set Audio Clip on a Loop

Set whether the audio clip will loop again after completion. Setting to true will enable this, setting to false will disable the looping process.

```
void SetAudioLoop(bool loop);
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
sfx->Play();
sfx->SetAudioLoop(true);
```

Is Audio Clip Playing

Return whether the audio clip is currently playing.

```
inline const bool IsAudioPlaying() const { return m_IsPlaying; }
```

Example code:

```
Audio* sfx = AudioManager::GetAudioClip("SFX");
if (!sfx->IsAudioPlaying()) {
	sfx->Play();
}
```

Evaluation

Ultimately this project has provided me with two key outcomes. The first, is a deeper understanding of how a game engine is structured and how some of the pieces fit together. The second, has been improved C++ language skills, as well as working on previous weaknesses using more inheritance and polymorphism. Overall, the experience has been beneficial to my personal development.

At the beginning of the project, it was apparent that this would be a huge task to implement, so it was important to scale down and provide realistic objectives. While some of the original objectives were not achieved, a significant proportion were and included the fundamental aim to provide a rendering component. The artefact, allows a developer to create a context window, render sprites, text and textures, group them in a container with a unique shader and play audio clips. An example project (Appendix One), is included with the source files, that implement all these items, with the ability to control an object on the screen using basic inputs. This example project, can be disabled in the 'game.h' by setting the following define to 0:

```
#define EXECUTE_EXAMPLE_PROJECT 0
```

One of the elements that did not get implemented was physics and collisions. Although this would have been a great addition to the engine, it was not possible due to time constraints. Another feature that I didn't get to implement, was the integration of LUA. This would not have been overly beneficial to the project, but it's a language skill that a lot of game development employers find desirable. However, this project will continue through the summer, and these two missing components are next on my list.

UPDATE: I have had to remove the Gorilla Audio component, as through additional testing, I found that this did not want to work on other computers and refused to compile. The code has been commented out for the time being, until I can fix the issue.

BONUS UPDATE: I made a little racing car game with the GGEngine, a build of this can be found in the Box link provided at the start of this report, it is also now included as the example project in the source code.

References

Resources Used

FreeImage (2015) *FreeImage The Productivity Booster*. Available at: http://freeimage.sourceforge.net/ (Accessed: 06 April 2018)

FreeType (2018) *The FreeType Project*. Available at: https://www.freetype.org/ (Accessed: 07 April 2018)

GLEW (2017) *The OpenGL Extension Wrangler Library*. Available at: http://glew.sourceforge.net/ (Accessed: 03 April 2018)

GLFW (2018) *GLFW Documentation*. Available at: http://www.glfw.org/documentation.html (Accessed: 03 April 2018)

Gorilla Audio (2014) *Gorilla Audio 0.3.1 Cross-platform C audio mixer library for games*. Available at: http://www.finalformgames.com/gorilla/html/index.html (Accessed: 24 April 2018)

Leap Motion (2015) *leapmotion/FreeImage*. Available at: https://github.com/leapmotion/FreeImage/blob/master/Examples/OpenGL/TextureManager/TextureManager.cpp (Accessed: 06 April 2018)

ODE (2014) *Open Dynamics Engine*. Available at: http://ode.org/ (Accessed 29 April 2018)

Learning Materials

Cherno (2018) *The Cherno Project*. Available at: https://www.youtube.com/user/TheChernoProject/featured (Accessed 28 April 2018)

Gregory, J. (2014) *Game Engine Architecture*. 2nd edn. Boca Raton: CRC Press.

Hussain, F. (2017) *Modern OpenGL C++ 3D Game Tutorial Series & 3D Rendering*. Available at: https://www.udemy.com/opengl-tutorials/ (Accessed 01 April 2018)

Khronos (2018) *OpenGL 4.5 Reference Pages*. Available at: https://www.khronos.org/registry/OpenGL-Refpages/gl4/ (Accessed 30 April 2018)

Millington, I. (2010) *Game Physics – Engine Development*. 2nd edn. Burlington: Elsevier.

Rautenbach, P. (2008) *3D Game Programming Using DirectX 10 and OpenGL*. London: Cengage Learning EMEA.

Wolff, D. (2013) *OpenGL 4 Shading Language Cookbook*. 2nd edn. Birmingham: Packt Publishing.

Self-Reference

Johnson, G. (2018) *GGe Documentation. A Good Game Engine.* Available at: http://gordon.johnson7.co.uk/documentation.php?doc=gge/doc.php (Accessed: 01 May 2018)

Appendix One – Example Project

Header File

```
#pragma once

#include <GGEngine.h>

#define EX_GAME_NAME "My Example Project"
#define EX_WINDOW_WIDTH 1920
#define EX_WINDOW_HEIGHT 1080
/* Setting the full screen option to 1 will void width and height */
#define EX_FULL_SCREEN 0

class Example : public GGEngine {

private:
	Window *				window;
	Shader*					shader;
	std::vector<Texture*>			images;
	Container*				bucket;
	Sprite*					square;
	Sprite*					fpsBackground;
	Text*					fpsHeading;
	Text*					fpsCounter;
	Audio*					sfxAudio;
	Audio*					mainAudio;
	const float				speed = 0.5f;

public:
	Example() {
		window = CreateWindow(EX_GAME_NAME, EX_WINDOW_WIDTH, EX_WINDOW_HEIGHT);
		FontManager::GetFont()->SetScale(window->GetWindowWidth() / 32,
			window->GetWindowHeight() / 18);   }
	~Example() {
		for (int i = 0; i < images.size(); i++) {
			delete images[i];
		}
	}
	void Start() override;
	void Tick() override;
	void Update() override;
	void FastUpdate() override;
	void Render() override;
};
```

CPP File

```
#include "Example.h"
```

Start

```
void Example::Start() {
	// Create Default shader
	shader = new Shader();
	// Create a container to hold all assets
	bucket = new Container(shader);
	// Add a couple of images to our texture array
	images = {
		new Texture("img/test.png"),
		new Texture("img/test4.png")
	};
	// Cover the background with these images at random
	srand(0);
	for (float y = -9.0f; y < 9.0f; y += 0.5f) {
		for (float x = -16.0f; x < 16.0f; x += 0.5f) {
			bucket->AddToContainer(new Sprite(x, y, 0.4f, 0.4f,
				images[rand() % 2]));
		}
	}
	// Create a blue square sprite
	square = new Sprite(-2.0f, -2.0f, 2.0f, 2.0f, GGE_COL_BLUE);
	// Add the blue square sprite to the container
	bucket->AddToContainer(square);
	// Add a new font to the font manager
	FontManager::AddFont(new Font("MyFont", "font/Muli-Bold.ttf", 32));
	FontManager::GetFont("MyFont")->SetScale(window->GetWindowWidth() / 32,
		window->GetWindowHeight() / 18);
	// Add an fps counter
	// Create a slightly transparent background
	fpsBackground = new Sprite(-15.0f, 6.0f, 4, 2, 0xc8ffffff);
	// Create two new text objects one with default font and one with new
	fpsHeading = new Text("FPS Counter", -14.5, 7.2, GGE_COL_BLACK);
	fpsCounter = new Text("", -13.5f, 6.5f, "MyFont", GGE_COL_BLACK);
	// Add all fps objects to the container
	bucket->AddToContainer(fpsBackground);
	bucket->AddToContainer(fpsHeading);
	bucket->AddToContainer(fpsCounter);
	// Add a new sound effect to the audio manager
	AudioManager::AddAudioClip(new Audio("SFX", "audio/BeBop.wav"));
	// Get the new sound to use locally
	sfxAudio = AudioManager::GetAudioClip("SFX");
	// Play the default music file on a loop
	mainAudio = AudioManager::GetAudioClip(GGE_DEFAULT_STRING);
	mainAudio->SetAudioLoop(true);
	mainAudio->Play();
}
```

Tick

```
void Example::Tick() {
	// Update the fps counter every 1 second
	fpsCounter->text = std::to_string(GetFramesPerSecond());
	// Log out to console information
	std::cout << GetUpdatesPerSecond() << " UPS, " << GetFramesPerSecond()
	<< " FPS." << std::endl;
}
```

Update

```
void Example::Update() {
	// Add a player controller to move the square object around the screen
	if (window->IsKeyHeldDown(GLFW_KEY_LEFT)) {
		square->position.x -= speed;
		sfxAudio->Play();
	} else if (window->IsKeyHeldDown(GLFW_KEY_RIGHT)) {
		square->position.x += speed;
		sfxAudio->Play();
	}
	if (window->IsKeyHeldDown(GLFW_KEY_UP)) {
		square->position.y += speed;
		sfxAudio->Play();
	} else if (window->IsKeyHeldDown(GLFW_KEY_DOWN)) {
		square->position.y -= speed;
		sfxAudio->Play();
	}
}
```

Fast Update

```
void Example::FastUpdate() {
	// Pause and resume the main audio playback,
	if (window->IsKeyDown(GLFW_KEY_SPACE)) {
		if (mainAudio->IsAudioPlaying()) {
			mainAudio->Pause();
		} else {
			mainAudio->Resume();
		}
	}
}
```

Render

```
void Example::Render() {
	// Render all objects held in the container
	bucket->RenderThisContainer();
}
```