Universidad Jaume I de Castellón
Design and Development of Videogames Degree
Final Project Report

Design and Development of a 2D Game Engine

Student: Álvaro Ruiz Gallego
Tutor: Miguel Chover Selles

Abstract

In the past years videogame development has been strongly boosted thanks to the emergence of game engines, such as Unity, Unreal Engine or CryEngine, which make video game production much more easier. To simplify and facilitate the development of video games, these engines hide a lot of the complexities below, allowing us to work at a higher level without having to worry about more technical problems like rendering, collision detection, physics simulation, etc. However, due to this, the users of these engines usually don't go deeper into these technical complexities and end up working in a superficial way, without really knowing in detail the mechanisms that allow games to work.

But knowing those mechanisms not only allows us to better use the tools we have, which means we could make better games, but is also a highly demanded knowledge in the video game industry, especially in the triple A games industry. That's why I personally think that every videogame programmer should know how games are built from scratch.

With the development of this project, what I am looking for is to delve into the technical and low-level complexities hidden in the game engines by designing and implementing a 2D game engine from scratch, so as to expand my knowledge on the subject. This engine will include rendering, audio and physics modules. It will be implemented in the C++ language, using OpenGL and OpenAL as APIs for rendering graphics and audio respectively. In addition, libraries will be used to simplify some of the main tasks (window creation, communication with input devices, image loading, font loading, etc). Each of the modules will be implemented from scratch, with the help of those libraries.

Keywords: *Rendering, audio, physics, collision detection, libraries, engine*

Index

Index of figures

1. Introduction

1.1 Project objectives and motivation

The objective of the project is the creation of a 2D game engine from scratch. This includes the implementation of three main modules: rendering, audio and physics/collision. This engine is aimed to be both easy to use and efficient at the same time and it will allow the user to create scenes easily, render sprites, animated sprites and text, play 2D audio and simulate realistic physics.

There are already plenty of good engines out there, why reinvent the wheel? Well, with this project I'm not trying to reinvent the wheel, what I'm trying to do is to understand how wheels work and to learn how to build them, so some day, if I need it, I will be able to make a wheel that fits my needs better than a regular one. In other words, the "real" objective behind this project is not the creation of an engine that will be used to make commercial games, what I want to accomplish by doing this project is to increase my knowledge about how the internal mechanisms of a game engine works, to know the low level processes that make a game run, so, in a future, I can use this knowledge to make a better use of the already created engines, or even to create an engine that fits my needs better than Unity, Unreal, or any other engine.

Furthermore, this knowledge is required to work at almost any AAA game studio and, in fact, even the majority of the most successful indie games of the last years have been developed using an "in house" engine. *That Game Company*, the creators of one of the most successful indie games of the last years, *Journey*, ask job applicants to write a demo in C/C++ or WebGL instead of using an engine, to prove that they have the required technical skills: *"To showcase your technical abilities, we ask that you write your demo in C or C++ or Javascript/WebGL rather than using an off-the-shelf engine."* I believe that part of the success of this studios is due to the fact that they used a custom engine, which fitted better to their needs and allowed them to have a better workflow, and also gave them a technical advantage against other studios that were stuck using engines like Unity or Unreal.

1.2 Features

Below is a summary of the main features of the engine:

1- Scene management: Users are able to create independent scenes and navigate through them with ease. Resources are unloaded automatically when the scenes change, so the user doesn't need to worry about it.

2- Game Objects: The base entities of the games developed using the engine are the game objects. Similarly to other engines, like Unity, game objects have a serie of components attached that define their behaviours.

3- Simple workflow: Users only need to create components, attach them to game objects and add those game objects to a scene to get their game working. The game loop, window creation, event handling, initialization, etc, are managed automatically by the engine.

3- Input: Input is fully handled by the engine and includes keyboard, mouse and controller input. Up to 16 controllers can be connected at the same time.

4- Rendering: The engine handles the rendering automatically. Users only need to attach components (sprite renderer) to game objects to display images. Animation, parallax, text rendering, transparency and sprite sheets are also fully supported.

5- Audio: Loading and playing audio is as easy as displaying an image, users just need to add an audio source to a game object. Also, audio sources allow spatial audio, this means that audio sources will be attenuated and panned depending on their position in the game world.

6- Physics and collision detection: The engine includes a physics system that allow the users to simulate realistic physics (force, torque, impulses, friction, bouncing...) and that handles collision detection and resolution of convex polygons.

7- Easy configuration: Engine parameters are handled using a .txt file, so parameters like window size, display mode, physics precision, master volume, etc, can be modified without recompiling the code.

Aside from the engine, the project will also include a serie of demos displaying the features previously described.

1.3 Environment and project specifications

The engine will be delivered as a dynamic library, and it has been developed in the C++ language using Visual Studio Community 2015 under Windows 8.1 operating system.

The engine makes use of the following libraries and APIs. These libraries have been chosen because they are cross-platform, lightweight and provide just the required features:

-OpenGL (Open Graphics Library): A cross-platform application programming interface (API) for rendering 2D and 3D graphics. It has been used for the rendering system of the engine.

-OpenAL (Open Audio Library): A cross-platform audio API designed for rendering of 3D positional audio. It has been used for the audio system of the engine.

-GLM (OpenGL Mathematics): A header-only mathematics library for graphics programming. It has been used as the main mathematics library of the engine, for matrix transformations, vector operations, etc.

-GLEW (OpenGL Extension Wrangler Library): A cross-platform extension loading library that provides run-time mechanisms for determining which OpenGL extensions are supported on the target platform. Nowadays is almost a must have in any OpenGL project.

-GLFW: A cross platform library for OpenGL for creating windows, contexts, receiving input, timing, etc. It is a lightweight and modern alternative for the SDL library, which was too large for my needs.

-FreeImage: A cross-platform library for image loading and handling. It has been used to load images from disk.

-FreeType: A cross-platform library for text rasterization. It has been used to load font files and convert them into bitmaps, so the engine can use them.

Even though it has been developed using cross-platform libraries, for simplicity and due to the fact that I can only test and build the engine in a Windows system, it will only be available for Windows systems. But in case in a future I wanted to deploy the engine to other operating systems, it will be quite straightforward to do so.

1.4 Planning

The tasks performed in this project are the following ones:

- Project configuration
  - Library installation
  - Project settings

- Core functionality of the engine
  - Game class
  - Scenes
  - GameObjects & Component

- Input
  - Keyboard and mouse input
  - Controller input

- Rendering
  - Basic configuration (window creation, OpenGL initialization, data structures)
  - Camera
  - Texture loading
  - Sprite rendering
  - Text rendering
  - Line rendering
  - Others: Transparency, parallax effect, etc

- Audio
  - Sound system configuration (OpenAL initialization, data structures)
  - Audio loading
  - Spatial audio
  - AudioSource & AudioListener components

- Physics
  - Rigid body dynamics
  - Collision detection
  - Collision resolution
  - Others: Raycast, collision events, etc

- Demos & Testing

- Final project report

The following chart shows the amount of hours dedicated to the realization of each of the tasks compared to the estimated hours that were planned at the start of the project.

Task	**Total estimated hours**	**Total hours**
Project configuration	10h	10h
Core functionality	35h	40h
Input	10h	15h
Rendering	90h	75h
Audio	25h	25h
Physics	50h	110h
Demos & Testing	30h	70h
Final project report	25h	60h

2. Development

2.1 Project creation and configuration

Project configuration is a key aspect of a good workflow environment. Since I spent a considerable amount of time configuring the project and setting up everything, I have decided to include a description of the steps that were necessary for its correct configuration.

2.1.1 What is a dynamic library?

To understand some of the steps we first need to understand the concept of dynamic library. A library is a collection of functions that can be shared amongst different applications. There are two different type of libraries: static and dynamic. Static libraries are directly linked into the program at compile time whilst dynamic libraries are referenced at runtime. This means that even if the code of a dynamic library is changed, you don't need to recompile the application that uses it, whilst with static libraries you do need to recompile. Thus, using a dynamic library speeds up the workflow and makes easier to modify the engine.

2.1.2 Solution setup

Due to the fact that the game engine is built as a dynamic library (.dll file in Windows), it cannot be executed by itself. For this reason, during the development of the engine I have used two different projects: the first one is the engine itself, and the second one is a project for testing the features of the engine. Thus, the first step was creating a Visual Studio solution with both projects in it: *Engine* and *TestingProject*.

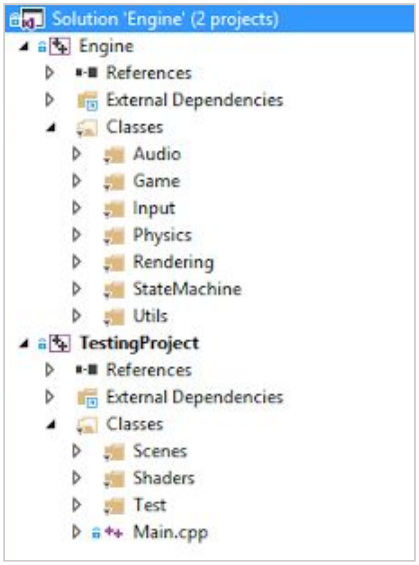

Figure 1: Solution setup

2.1.3 Project properties configuration

The next step was configuring the properties of each project. The *Engine* project has to configure the additional include directories, library directories and dependences, so it can use the libraries needed. It also has to be configured to build as a dynamic library.

General	
Target Platform	Windows
Target Platform Version	**8.1**
Output Directory	**$(ProjectDir)$(Configuration)\$(Platform)**
Intermediate Directory	**$(ProjectDir)$(Configuration)\$(Platform)\Temp**
Target Name	$(ProjectName)
Target Extension	**.dll**
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.obj.enc;*.ilk;*.ipdb;*.iobj;*.resources;*.tlb;*.tli;*.tlh;*.tmp;*.rsp;*.pgc;*.p
Build Log File	$(IntDir)$(MSBuildProjectName).log
Platform Toolset	**Visual Studio 2015 (v140)**
Enable Managed Incremental Build	No
Project Defaults	
Configuration Type	**Dynamic Library (.dll)**
Use of MFC	Use Standard Windows Libraries
Character Set	**Use Unicode Character Set**
Common Language Runtime Support	No Common Language Runtime Support
.NET Target Framework Version	
Whole Program Optimization	<different options>
Windows Store App Support	No

Figure 2: Engine general project configuration

Additionally, since the .dll file obtained needs to be in the same directory as the *TestingProject* executable, a post-build event was configured to copy the file into the desired folder. Furthermore, for this to work the *Engine* project has to build before the *TestingProject* project. This is done by changing the Solution dependencies and build order.

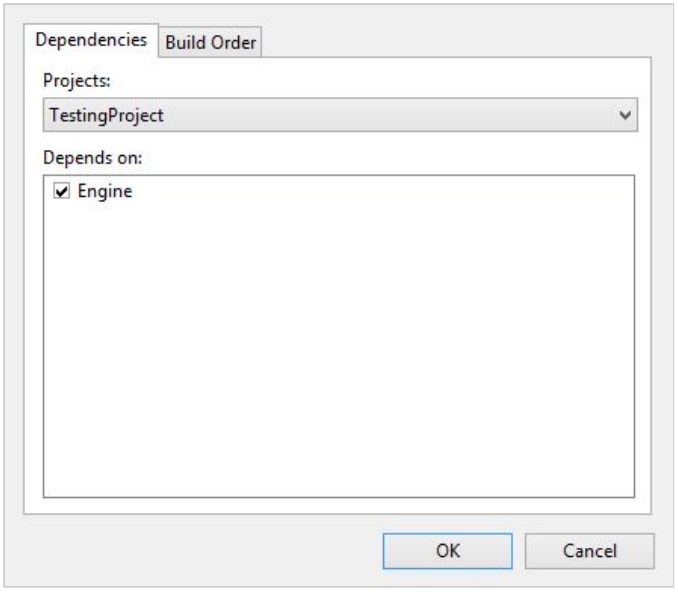

Figure 3: Configuring the dependencies and build order

The *TestingProject* project has to be configured to link to the engine library in the same way the *Engine* project was configured to link to its libraries. Thus, it is achieved by configuring the additional include directories, library directories and dependences of the project.

Finally, the optimization flags are configured in both projects to assure the best performance possible.

Optimization	**Full Optimization (/Ox)**
Inline Function Expansion	Default
Enable Intrinsic Functions	**Yes (/Oi)**
Favor Size Or Speed	**Favor fast code (/Ot)**
Omit Frame Pointers	No (/Oy-)
Enable Fiber-Safe Optimizations	No
Whole Program Optimization	Yes (/GL)

Figure 4: Configuring the optimization flags

2.2 Engine structure

The main elements of the engine are the Game class, the physics, rendering, audio and input modules, the scenes, game objects and components. Since the Game class is a complex class that acts as a hub between the rest of classes and modules, first it is necessary to understand each one of this classes to fully understand the Game class. That is why, for now, the Game class will be treated as a black box.

The elements that form the base structure of a game developed using the engine are the scenes (class Scene), game objects (class GameObject) and components (class Component). A Scene is formed of a list of GameObjects. Each GameObject, in turn, has a position, orientation and scale, and a list of Components, which contain the logic that control the GameObjects and define their behaviour.

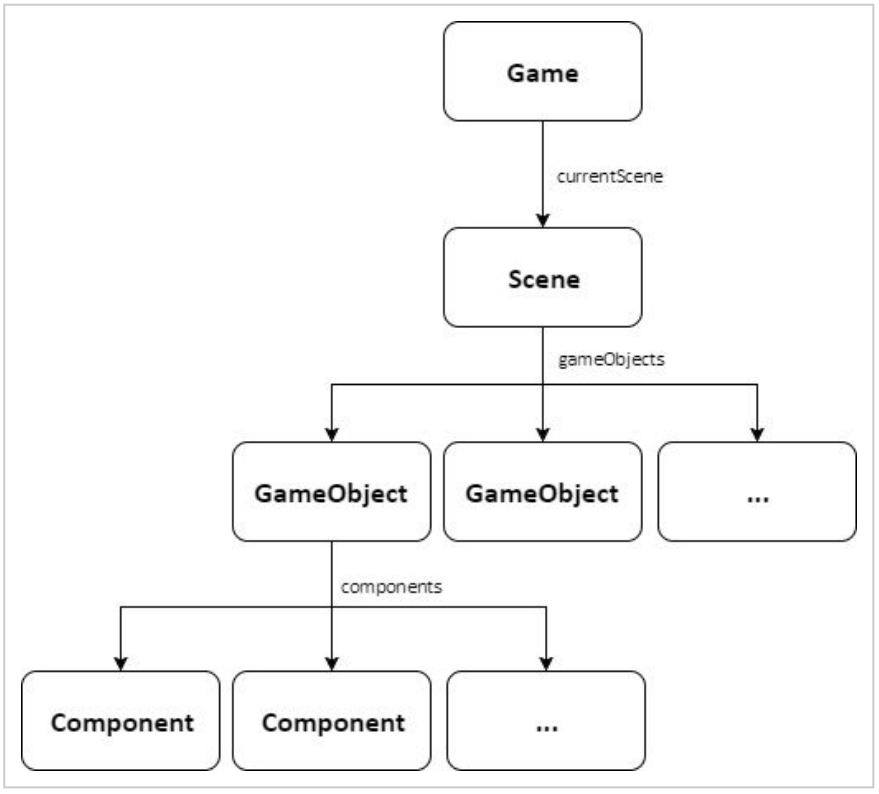

Figure 5: Base structure of the engine

The Scene, GameObject and Component classes have a serie of methods (called event methods) that are called when an event occurs. This events are input events, start and end of a scene, update and physics update. The methods are called in sequence: first, the game calls a method in the current scene, this scene calls the same method on every game object added to it, and each one of those game objects calls the same method on every component added to it. The events that trigger these methods are the following ones:

OnInitialize
For the scene it is called when the scene is started and for the GameObjects it is called when they are added to the scene.

OnTerminate
For the scene it is called when the scene ends and for the GameObjects it is called when they are destroyed or the scene ends.

Update
It is called every game loop. It receives the delta time (time elapsed since the previous update call) as a parameter.

FixedUpdate
It is called every physics step. It receives the fixed timestep (physics time step) as a parameter.

OnButtonDown
It is called every time a button from the keyboard is pressed. It receives the id of the button pressed as a parameter.

OnButtonUp
It is called every time a button from the keyboard is released. It receives the id of the button pressed as a parameter.

OnControllerButtonDown
It is called every time a button from a controller is pressed. It receives the id of the button pressed and the id of the controller as parameters.

OnControllerButtonUp
It is called every time a button from a controller is released. It receives the id of the button pressed and the id of the controller as parameters.

OnMouseButtonDown
It is called every time a button from the mouse is pressed. It receives the id of the button pressed and the screen and world position of the mouse at the moment of the event as parameters.

OnMouseButtonUp
It is called every time a button from the mouse is released. It receives the id of the button pressed and the screen and world position of the mouse at the moment of the event as parameters.

OnMouseMove
It is called every time the mouse moves. It receives the screen and world position of the mouse at the moment of the event as parameters.

2.2.1 Component

The Component is a base class meant to be overridden by the user to define its event methods. They can be added to GameObjects to define their behaviour. This class is composed of the event methods plus the following elements:

Private variables:

GameObject _gameObject*
The parent GameObject of the component.

bool _unique
A boolean that indicates if the component is unique, which means that there can only be one Component of that type per GameObject.

Each one of these variables has a public getter.

Other event methods:

void OnAddedToGameObject()
Method that is called directly from the parent GameObject when the Component is added to it.

void OnRemovedFromGameObject()
Method that is called directly from the parent GameObject when the Component is removed from it.

void OnCollisionEnter(ContactInfo contactInfo)
Method that is called from the parent GameObject when it has a Collider Component added and it has collided with something.

void OnCollisionExit(ContactInfo contactInfo)
Method that is called from the parent GameObject when it has a Collider Component added and it has stopped colliding with something.

void OnTriggerEnter(ContactInfo contactInfo)
Method that is called from the parent GameObject when it has a Collider Component added, it has collided with something and it is marked as trigger.

void OnTriggerEnter(ContactInfo contactInfo)
Method that is called from the parent GameObject when it has a Collider Component added, it has stopped colliding with something and it is marked as trigger.

2.2.1.1 Example of Component

The following is an example of a Component defined by the user. It is a Component that simply prints a text to the console whenever there is an Update, OnAddedToGameObject or OnInitialize event:

```
class TestComponent: public Component
{
    public:
        void Update(double deltaTime) override;
        void OnAddedToGameObject() override;
        void OnInitialize() override;
};
```

Figure 6: TestComponent declaration

```
void TestComponent::Update(double deltaTime)
{
    printf("\nUPDATE");
}

void TestComponent::OnAddedToGameObject()
{
    printf("\nON ADDED TO GAME OBJECT");
}

void TestComponent::OnInitialize()
{
    printf("\nON INITIALIZE");
}
```

Figure 7: TestComponent definition

2.2.2 GameObject

GameObjects are the “physical” elements of the game world, they exist in the game world, they have a position, orientation and scale. They can have multiple Components attached, and the combination of the behaviours of all of those Components is the general behaviour of the GameObject. They can also have others GameObjects as children, so the parent GameObject’s transform affects every one of its children. Apart from the event methods, the main elements of the GameObject class are the following:

Private variables:

Scene* _parentScene
The Scene which the GameObject is added.

GameObject* _parentGameObject
The parent GameObject (if there is one).

vector<GameObject*> _childGameObjects
A list of it’s children GameObjects.

vector<Component*> _components
A list of the Components added to the GameObject.

vec3 _position
It’s world position. It is a 3D vector since the z values are used for determining the z order of the sprites. It has several public setters and a getter.

vec2 _scale
The scale of the GameObject. It has several public setters and a getter.

float _orientation
The orientation of the GameObject in radians and clockwise order. It has a public setter and getter.

mat4 _transform
The transformation matrix of the GameObject. It is updated only when the position, scale or orientation has changed. It has a public getter.

bool _isTransformDirty
A boolean that indicates if the transformation matrix needs to be updated. It has a public getter.

vector<Component*> _pendingComponentsToAdd
A list of Components that are pending to be added. When a Component is added to the GameObject, it is stored in this list. At the start of the next frame, each Component in this list is added to the GameObject and removed from this list. This is done to avoid modifying the *_components* list while it is being looped.

vector<Component*> _pendingComponentsToRemove
A list that acts the same way than *_pendingComponentsToAdd* list but with the GameObjects that are pending to be removed.

Private methods:

void addPendingComponents()
Adds the Components that are in the *_pendingComponentsToAdd* list. It is called at the start of the public method *Update.*

void removePendingCompoenents()
Removes the Components that are in the *_pendingComponentsToRemove* list. It is called at the start of the public method *Update.*

template<typename T> bool canAddComponent(Component component)*
A template methodthat checks if a component can be added. Components cannot be added when they are marked as *unique* and there is already one Component of that type in the GameObject

Other event methods:

void OnCollisionEnter(ContactInfo contactInfo)
Method that is called from the Physics Engine when it has a Collider Component added and it has collided with something. It calls to the same method on every Component added to it.

void OnCollisionExit(ContactInfo contactInfo)
Method that is called from the Physics Engine when it has a Collider Component added and it has stopped colliding with something. It calls to the same method on every Component added to it.

void OnTriggerEnter(ContactInfo contactInfo)
Method that is called from the Physics Engine when it has a Collider Component added, it has collided with something and it is marked as trigger. It calls to the same method on every Component added to it.

void OnTriggerEnter(ContactInfo contactInfo)
Method that is called from the Physics Engine when it has a Collider Component added, it has stopped colliding with something and it is marked as trigger. It calls to the same method on every Component added to it.

Public methods:

void RemoveComponent(Component component)*
Adds to the *_pendingComponentsToRemove* list a Component.

void AddChildGameObject(GameObject gameObject)*
Adds a GameObject as child.

void RemoveChildGameObject(GameObject gameObject)*
Removes a child GameObject.

template<typename T> T AddComponent(T* component)*
A template method that checks if a Component can be added using *canAddComponent* method, then adds it to the *pendingComponentsToAdd* list if it can be added and returns it.

template<typename T> T GetComponent()*
A template method that returns the first Component of type T found in the GameObject components, or nullptr if the GameObject doesn't contain a Component of that type.

template<typename T> T GetComponentInChildren()*
A template method that returns the first Component of type T found in the GameObject's children components, or nullptr if the children don't contain a Component of that type.

template<typename T> std::vector<T> GetComponents()*
A template method that returns a list with all the Components of type T found in the GameObject components.

template<typename T> std::vector<T> GetComponentsInChildren()*
A template method that returns a list with all the Components of type T found in the GameObject's children components.

2.2.3 Scene

In order to define a scene, the user only needs to create a class that inherits from the base class Scene. Then the user can override the event methods to define the behaviour of the scene and to add and create GameObjects. Apart from the event methods, the main elements of the Scene class are the following:

Private variables:

vector<GameObject> _gameObjects:*
A list that contains all the GameObjects added to the scene.

vector<GameObject> _pendingGameObjectsToAdd*
A list of GameObjects that are pending to be added. When a GameObject is added to the scene, it is stored in this list. At the start of the next frame, each GameObject in this list is added to the scene and removed from this list. This is done to avoid modifying the *_gameObjects* list while it is being looped.

vector<GameObject> _pendingGameObjectsToDestroy*
A list that acts the same way than *_pendingGameObjectsToAdd* list but with the GameObjects that are pending to be destroyed.

Private methods:

void addPendingGameObjects()
Adds the GameObjects that are in the *_pendingGameObjectsToAdd* list. It is called at the start of the public method *Update.*

void destroyPendingGameObjects()
Destroys the GameObjects that are in the *_pendingGameObjectsToDestroy* list. It is called at the start of the public method *Update.*

Public methods:

GameObject CreateGameObject()*
It creates, adds to the *_pendingGameObjectsToAdd* list and returns a GameObject.

void DestroyGameObject(GameObject gameObject)*
It adds to the *_pendingGameObjectsToDestroy* list a GameObject.

void ChangeToScene(string name)
It calls a method of the Game class that unloads the current scene and loads the scene identified by the *name* string.

void TerminateGame()
It calls a method of the Game class that terminates the game.

int GetWindowWidth()
It returns the window width.

int GetWindowHeight()
It returns the window height.

int GetReferenceWindowWidth()
It returns the reference window height.

int GetReferenceWindowHeight()
It returns the reference window height.

2.2.3.1 SceneManager

The SceneManager class is a singleton which acts as a factory of Scenes. A factory is a creational design pattern that deals with the problem of creating objects without having to specify the exact class of the object that will be created. In this case, it receives a string (name) as input and returns the Scene corresponding to that string.

For this to work, the scenes must register themselves to the SceneManager. To do so, the SceneManager defines a macro that has to be included at the end of the .h file of every scene.

```
#define REGISTER_SCENE(T) bool sceneAux##T = SceneManager::GetInstance()->RegisterScene<T>(#T);
```

Figure 8: Register scene macro

This macro calls the *RegisterScene* template function with the name of the scene (the class name) as parameter, then this function adds to a map an instance of the scene registered with the name of the scene as key.

```
template<typename T> Scene* RegisterScene(std::string  name)
{
    _sceneMap.emplace(std::string(name), new T());
    return GetScene(name);
}
```

Figure 9: Register scene function

To retrieve a scene, which is done automatically from the Game class, the *GetScene* method is called.

```
Scene * SceneManager::GetScene(std::string  name)
{
    auto it = _sceneMap.find(std::string(name));
    if (it != _sceneMap.end())
    {
        return _sceneMap.at(std::string(name));
    }

    printf("\nWARNING: SceneManager::GetScene -> Scene not added");
    return nullptr;
}
```

Figure 10: Get scene function

2.2.3.1 Scene example

The following Scene is a simple scene that just register itself to the SceneManager, creates a GameObject and adds a TestComponent to it.

```
#ifndef TEST_SCENE_H
#define TEST_SCENE_H

#include "Scene.h"
#include "SceneManager.h"

class TestScene: public Scene
{
    protected:
        virtual void onInitialize() override;
};

REGISTER_SCENE(TestScene);

#endif
```

Figure 11: TestScene.h

```cpp
#include "TestScene.h"

#include "TestComponent.h"

void TestScene::onInitialize()
{
    CreateGameObject()->AddComponent(new TestComponent());
}
```

Figure 12: TestScene.cpp

2.3 Input

One of the most essential subsystems of a game engine is the Input module, since there cannot be a game without input. Most of the input of the game engine is handled via events, which are invoked from the _InputManager singleton class. The underscore at the start of the name of the class means that it is a engine-only class, which means that the user of the engine can't access to it. In the other hand, to retrieve some special inputs (the axis of the controllers) the user has to use a specific singleton class called ControllerInput, which can be freely accessed, and acts as a bridge between the _InputManager and the user.

The _InputManager class has a serie of callbacks that are called every time an input is received. This callbacks are set by the Game class, so, every time an input occurs, a method of the Game class will be called. To set the callback it uses the following methods:

```
void _InputManager::SetButtonCallback(ButtonCallback buttonCallback)
{
    _buttonCallback = buttonCallback;
}

void _InputManager::SetControllerButtonCallback(ControllerButtonCallback controllerButtonCallback)
{
    _controllerButtonCallback = controllerButtonCallback;
    _controllerButtonCallbackSet = true;
}

void _InputManager::SetMouseButtonCallback(MouseButtonCallback mouseButtonCallback)
{
    _mouseButtonCallback = mouseButtonCallback;
}

void _InputManager::SetMouseMoveCallback(MouseMoveCallback mouseMoveCallback)
{
    _mouseMoveCallback = mouseMoveCallback;
}
```

Figure 13: _InputManager callback setters

To receive the inputs, the InputManager class makes use of the GLFW library, and it has to be configured before it can start receiving them. First, the Game class sets the _Window (this class is a wrapper of the GLFWwindow class, and it will be explained in more detail later) which is going to receive the input using the following method:

```
void _InputManager::SetWindow(_Window * window)
{
    _window = window;

    glfwSetKeyCallback(_window->GetGLFWwindow(), &_InputManager::keyCallbackGLFW);
    glfwSetMouseButtonCallback(_window->GetGLFWwindow(), &_InputManager::mouseButtonCallbackGLFW);
}
```

Figure 14: _InputManager SetScene method

This method stores the _Window and sets the GLFW key and mouse button callbacks to call *keyCallbackGLFW* and *mouseButtonCallbackGLFW* methods respectively. The method *keyCallbackGLFW* will be called each time a key is pressed, released or held and it will store in a list the key id and the action (pressed, released or held). The method *mouseButtonCallbackGLFW* will be called each time a mouse button is pressed or released and it will store in a list the button id, the action (pressed or released) and the current position of the mouse.

```
void _InputManager::keyCallbackGLFW(GLFWwindow * window, int key, int scancode, int action, int mods)
{
    _triggeredButtons.push(ButtonTrigger(key, action));
}

void _InputManager::mouseButtonCallbackGLFW(GLFWwindow * window, int button, int action, int mods)
{
    _triggeredMouseButtons.push(MouseButtonTrigger(button, action, _mouseX, _mouseY));
}
```

Figure 15: _InputManager keyCallbackGLFW and mouseButtonCallbackGLFW methods

Every update tick the _InputManager will call the callbacks set by the Game class using the information stored in these lists, and then it will clear them. It will also update the mouse position and the controller input.

```
void _InputManager::Update()
{
    updateMousePosition();
    updateControllerInput();

    while (_triggeredButtons.size() > 0)
    {
        ButtonTrigger buttonTrigger = _triggeredButtons.front();
        _buttonCallback(buttonTrigger.button, buttonTrigger.action);
        _triggeredButtons.pop();
    }

    while (_triggeredMouseButtons.size() > 0)
    {
        MouseButtonTrigger mouseButtonTrigger = _triggeredMouseButtons.front();
        _mouseButtonCallback(mouseButtonTrigger.button, mouseButtonTrigger.action,
            mouseButtonTrigger.x, mouseButtonTrigger.y);
        _triggeredMouseButtons.pop();
    }
}
```

Figure 16: _InputManager Update method

The *updateMousePosition* method updates the mouse position and calls the mouse move callback if needed:

```
void _InputManager::updateMousePosition()
{
    double currentMouseX, currentMouseY;
    bool mouseMoved = false;

    glfwGetCursorPos(_window->GetGLFWwindow(), &currentMouseX, &currentMouseY);

    if (currentMouseX != _mouseX)
    {
        _mouseX = currentMouseX;
        mouseMoved = true;
    }
    if (currentMouseY != _mouseY)
    {
        _mouseY = currentMouseY;
        mouseMoved = true;
    }

    if (mouseMoved)
    {
        _mouseMoveCallback(_mouseX, _mouseY);
    }
}
```

Figure 17: _InputManager updateMousePosition method

The *updateControllerInput* method checks if a controller button has been pressed or released and calls the controller button callback if needed:

```
void _InputManager::updateControllerInput()
{
    for (int controllerIndex = 0; controllerIndex < MAX_CONTROLLERS; controllerIndex++)
    {
        if (_controllerStatus[controllerIndex] == GLFW_CONNECTED)
        {
            int count;
            const unsigned char* buttons =
                glfwGetJoystickButtons(GLFW_JOYSTICK_1 + controllerIndex, &count);

            for (int buttonIndex = 0; buttonIndex < count; buttonIndex++)
            {
                if (buttonIndex >= MAX_CONTROLLER_BUTTONS) break;

                if (_controllerPressedButtons[controllerIndex][buttonIndex] != buttons[buttonIndex])
                {
                    _controllerButtonCallback(CONTROLLER_1 + controllerIndex,
                        INPUT_CONTROLLER_BUTTON_1 + buttonIndex, buttons[buttonIndex]);
                    _controllerPressedButtons[controllerIndex][buttonIndex] = buttons[buttonIndex];
                }
            }
        }
    }
}
```

Figure 18: _InputManager updateControllerInput method

Aside from the controller buttons, the user may also need to know the position of the axis (joysticks and triggers) and if a controller is connected or not. The _InputManager implements two functions for this:

```
float _InputManager::GetAxis(int controller, int axis)
{
    int controllerIndex = controller - CONTROLLER_1;
    int axisIndex = axis - INPUT_CONTROLLER_AXIS_1;

    if (_controllerStatus[controllerIndex] == GLFW_CONNECTED)
    {
        int count;
        const float* axes = glfwGetJoystickAxes(GLFW_JOYSTICK_1 + controllerIndex, &count);

        if (count > axisIndex)
        {
            return axes[axisIndex];
        }
    }

    return 0;
}

bool _InputManager::IsControllerConnected(int controller)
{
    int controllerIndex = controller - CONTROLLER_1;

    return _controllerStatus[controllerIndex] == GLFW_CONNECTED;
}
```

Figure 19: _InputManagerGetAxis and IsControllerConnected methods

But, as already stated, the user can't access directly the _InputManager class, he/she need to use the ControllerInput singleton class to do so. This class defines the following methods:

```
float ControllerInput::GetAxis(int controller, int axis)
{
    return _InputManager::GetInstance()->GetAxis(controller, axis);
}

bool ControllerInput::IsControllerConnected(int controller)
{
    return _InputManager::GetInstance()->IsControllerConnected(controller);
}
```

Figure 20: ControllerInput GetAxis and IsControllerConnected methods

2.4 Display

The display is another essential element of a game engine. It refers not only to the rendering but also to the window creation and handling.

2.4.1 Window

The creation of the game window is handled using the GLFW library, which allows to create a window in a system independent way. The _Window class is a wrapper around the GLFWwindow class that hides the creation of the window and that simplifies the communication with it. The creation of the game window takes place in the constructor of the _Window class, as well as the setup of its parameters.

```
_Window::_Window(int width, int height, std::string  title, WindowMode mode)
{
    glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
    glfwWindowHint(GLFW_SAMPLES, 8);

    switch (mode)
    {
        case WINDOWED:
            _glfwWindow = glfwCreateWindow(width, height, title.c_str(), NULL, NULL);
            break;

        case FULLSCREEN:
            _glfwWindow = glfwCreateWindow(width, height, title.c_str(), glfwGetPrimaryMonitor(), NULL);
            break;
    }

    if (!_glfwWindow)
    {
        glfwTerminate();
        printf("\nERROR: _Window::_Window -> Window or OpenGL context creation failed");
        fflush(stdout);
        exit(EXIT_FAILURE);
    }
}
```

Figure 21: _Window constructor

2.4.2 Sprites

2.4.2.1 Quad

Even though the engine renders 2D sprites, the rendering is done using OpenGL, a 3D graphics rendering API. This means that we need to find a way to render 2D sprites in a 3D environment. The solution to this is to render a quad (two triangles that form a square) for

each sprite, with the sprite image as texture and with the transform specified by the sprite. The quads have to be perpendicular to the camera and the rendering has to be done using an orthographic projection. This way, even if the z coordinates of the quads are different, the sizes will remain the same. The quad can be seen as the base skeleton that is common to every sprite. The basic information of the quad is stored in the struct _Quad.

```
struct _Quad
{
    const GLfloat vertices[12] =
    {
        -0.5, -0.5, 0,
         0.5, -0.5, 0,
         0.5,  0.5, 0,
        -0.5,  0.5, 0
    };

    const GLubyte indices[6] =
    {
        0,1,2, 2,3,0
    };

    const GLenum renderMode = GL_TRIANGLES;

    const GLenum vertexDataType = GL_FLOAT;
    const GLsizei vertexSize = sizeof(GLfloat) * 3;

    const GLenum indexDataType = GL_UNSIGNED_BYTE;
    const GLsizei indexCount = 6;
};
```

Figure 22: _Quad struct

This struct stores the information that the rendering engine needs to know in order to render a quad. It stores the vertices and indices, the render mode, type of data, and data size.

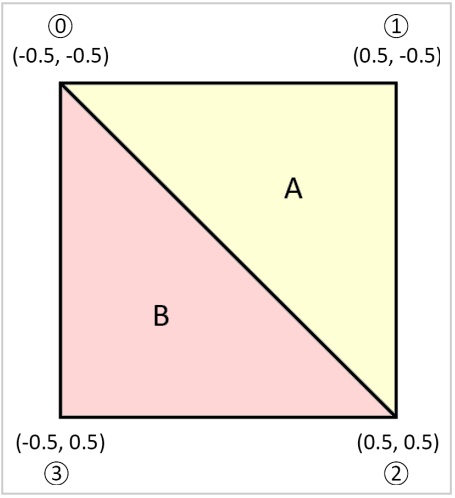

Figure 23: The quad represented by the _Quad struct

The information related to the texture and transform isn't stored here because it isn't common to every sprite. Each sprite has its own transformation, texture and UV coordinates.

2.4.2.2 Sprite

The sprite is represented in the engine by the _Sprite class. As we already state, the information stored in a sprite is the texture, UV coordinates and the transform. It also stores the size sprite (defined by the size of the texture and the UV coordinates) and it can also set a color, so the sprite will be tinted of that color when rendered. The sprite also has a parent GameObject, and its transforms is combined with the transform of the parent to get the final transform. The size of the sprite is also applied to the transform as a scale transformation, so the size of the quad is adjusted to its size. For instance, if the size of the the sprite is 200x300, since the quad is a 1x1 square, applying an horizontal scale of 200 and a vertical scale of 300 will transform the base quad to a 200x300 quad, which is what we need to correctly render the sprite.

2.4.2.2.1 Parallax

Another parameter of the _Sprite class is the parallax value. The parallax effect is a way of fake depth in a 2D environment that is done by reducing the displacement of objects that are further away. For instance, a sprite that has a parallax value of 0.5 will move at half the speed (when the camera moves or when a velocity is applied to its GameObject) that a sprite with a parallax value of 1. This will create the illusion that the first object is further away than the second one. Note that the parallax effect only affects the position, not the size, scale or orientation.

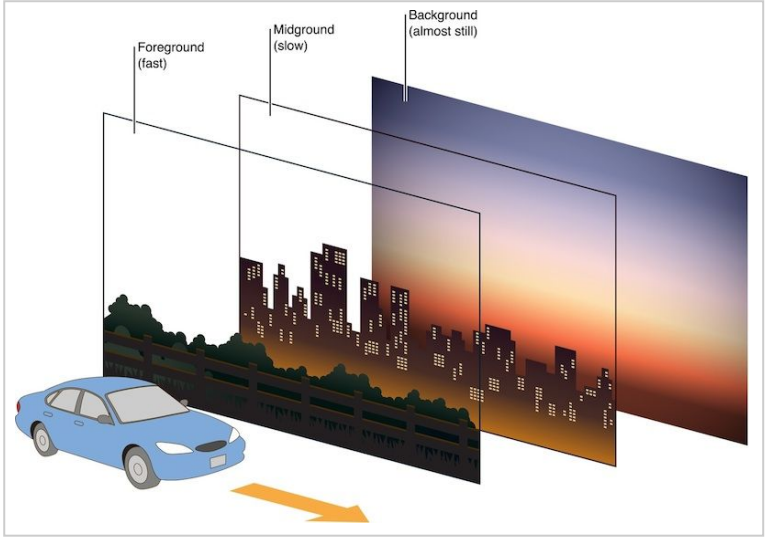

Figure 24: Parallax effect. Image from here

2.4.2.2.2 Camera space

The _Sprite can also be set to render in camera space. This means that the transformation of the camera won't affect the sprite. This is useful when the user wants to display HUD elements.

2.4.2.3 Textures

The class that stores the information of a texture is the _Texture class. It stores an array of pixels, the width and height of the texture and the image path. The FreeImage library is used to load the texture. This is done in the private *readImageFile* method of the _Texture class, that is called from the constructor:

```
void _Texture::readImageFile(std::string  imagePath)
{
    FIBITMAP* baseImage = FreeImage_Load(FreeImage_GetFileType(imagePath.c_str(), 0), imagePath.c_str());
    FIBITMAP* image = FreeImage_ConvertTo32Bits(baseImage);

    _width = FreeImage_GetWidth(image);
    _height = FreeImage_GetHeight(image);

    if (_width == 0 || _height == 0)
    {
        printf("\nWARNING: _Texture::readImageFile -> Image path (%s) not valid", imagePath.c_str());
        _empty = true;
    }

    _pixels = new GLubyte[4 * _width * _height]();
    GLubyte* tempPixels = (GLubyte*)FreeImage_GetBits(image);

    //Swap red and blue colors
    int posPixels;
    for (int i = _height - 1; i >= 0; --i)
    {
        for (int j = _width - 1; j >= 0; --j)
        {
            posPixels = (i * _width + j) * 4;

            _pixels[posPixels + 0] = tempPixels[posPixels + 2];
            _pixels[posPixels + 1] = tempPixels[posPixels + 1];
            _pixels[posPixels + 2] = tempPixels[posPixels + 0];
            _pixels[posPixels + 3] = tempPixels[posPixels + 3];
        }
    }

    //[0,0] -> bottom left

    FreeImage_Unload(baseImage);
    FreeImage_Unload(image);
}
```

Figure 25: _Texture readImageFile method

The array of pixels retrieved using FreeImage is stored in BGRA format, so, in order to convert it to RGBA (the format used by the engine), every red and blue component of each pixel is swapped. If the textured failed to load it will be marked as empty, so the renderer won't even try to render it.

2.4.2.4 Sprite batching

Having a different texture for each sprite is not only a waste of resources but it also affects the performance of the game. That's why the engine handles this problem by batching together sprites that use the same texture and rendering them together in one pass. To do so, the engine uses the class _SpriteBatch. This class stores a _Texture and a list of all the _Sprites that use it. The _SpriteBatch class also loads the texture to the rendering engine, so it can be used for rendering.

2.4.2.5 Sprite sheets

Sprites with the same texture don't necessarily have to display the same image, since sprites can have different UV coordinates, which indicate what part of the texture is shown in the sprite. For instance, the following texture could be used to display two different characters, depending of the UV coordinates of the sprite using it:

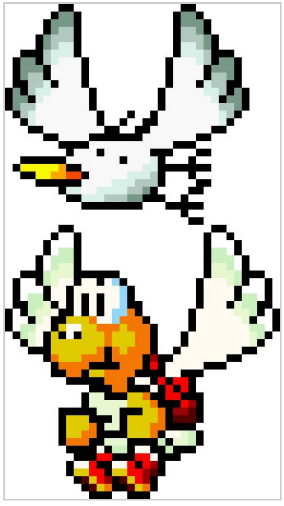

Figure 26: Sprite sheet texture example. Sprites from "Yoshi's Island".

The SpriteSheet is a class whose function is to help with the use of sprite sheets. It is implemented to work with sprite sheets created in TexturePacker, which are exported in json format. To create a sprite sheet the user just needs to pass the path of the json file to the SpriteSheet constructor. Once the SpriteSheet is created, the user can retrieve the UV

coordinates of each individual image by calling the method *GetUVs* with the name of the image. It can also return a list of UV coordinates for animation using the function *GetAnimationUVs*. In this case, the parameter needed is the base name of all the images of that animation.

2.4.2.5.1 Sprite sheet manager

The SpriteSheetManager is a singleton class created to facilitate the creation of SpriteSheets. To create a sprite sheet, the user just needs to call its *GetSpriteSheet* method with the sprite sheet file path. This method will check if this SpriteSheet has already been created and will create a new one if it hasn't.

2.4.2.6 Sprite manager

The _SpriteManager class is a singleton that is used as a hub between the different classes that participate in the sprite rendering pipeline. It has a list of all the sprites and sprite batches and several public functions.

2.4.2.6.1 Sprite creation

When a sprite is created, it has to check if there is already a sprite batch of sprites with the same texture and if not, a new sprite batch has to be created. This is done through the _SpriteManager. In the _Sprite constructor, its _Texture is retrieved using the _SpriteManager *GetTexture* method. This method receives the image path as parameter and checks if there is a batch that has a texture with the same image path. If it doesn't it creates a new _SpriteBatch and then returns the _Texture, which has been created at the _SpriteBatch constructor. The method in charge of checking if a batch exists and creating it if it doesn't is the following one:

```
_SpriteBatch * _SpriteManager::getSpriteBatch(std::string  imagePath)
{
    for (_SpriteBatch* spriteBatch : _spriteBatches)
    {
        if (spriteBatch->GetTexture()->GetImagePath() == imagePath)
        {
            return spriteBatch;
        }
    }

    _SpriteBatch * spriteBatch = new _SpriteBatch(imagePath);

    if (spriteBatch->GetTexture()->IsEmpty())
    {
        delete spriteBatch;
        return nullptr;
    }

    _spriteBatches.push_back(spriteBatch);

    return spriteBatch;
}

_SpriteBatch * _SpriteManager::getSpriteBatch(_Sprite * sprite)
{
    return getSpriteBatch(sprite->_texture);
}
```

Figure 27: _SpriteManager getSpriteBatch method

2.4.2.6.2 Adding sprites

Once created, the _Sprite must be added to the _SpriteManager in order to be rendered. This is done using the _SpriteManager *AddSprite* method. This method uses the *getSpriteBatch* method to retrieve the batch in which the sprite should be added and adds it if it hasn't been added already.

```
void _SpriteManager::AddSprite(_Sprite * sprite)
{
    if (std::find(_sprites.begin(), _sprites.end(), sprite) == _sprites.end())
    {
        _SpriteBatch* spriteBatch = getSpriteBatch(sprite);
        if (spriteBatch != nullptr)
        {
            spriteBatch->AddSprite(sprite);
            _sprites.push_back(sprite);
        }
    }
    else
    {
        printf("\nWARNING: _SpriteManager::AddSprite -> Sprite already added");
    }
}
```

Figure 28: _SpriteManager AddSprite method

The list of batches will be retrieved by the rendering engine and it will render every sprite of every batch. That's why, in order to render a sprite, it has to be added through the _SpriteManager.

2.4.2.7 Sprite renderer

The SpriteRenderer is the class in charge of creating and adding a sprite to the _SpriteManager. It inherits from the base class Component. To display a _Sprite in a game, the user has to create a GameObject and then add a SpriteRenderer component to it. The constructor of the SpriteRenderer takes the image path, a boolean indicating if it is displayed in camera space and the UV coordinates as parameters. The SpriteRenderer will create the sprite in its constructor and will add it to the _SpriteManager in its *OnInitialize* method (that, remember, is called when the parent GameObject is added to the scene).

2.4.2.8 Animated sprite renderer

The AnimatedSpriteRenderer class inherits from the SpriteRenderer class and it is used to display animations. For the AnimationSpriteRenderer to work, the different frames that form the animation must be in the same sprite sheet. Its constructor takes a SpriteSheet and the base name of the sprites that are used in the animation. If the sprites are named "character1", "character2", etc, the base name will be "character". It also takes the frame duration, the starting frame and a boolean indicating if it is displayed in camera space.

The animation process is done in the *Update* method. Every update, it checks if the elapsed time is greater than the frame duration. If it is, it calls the *advanceFrames* method with the number of frames to advance. This method calculates the index of the next animation frame and gets the UVs of the corresponding frame from the *_animationUVs* list, which is the list of UV coordinates retrieved from the SpriteSheet. Then it sets the _Sprite UV coordinates to the new ones.

```
void AnimatedSpriteRenderer::Update(double deltaTime)
{
    SpriteRenderer::Update(deltaTime);

    _elapsedTime += deltaTime;

    if (_elapsedTime > _frameDuration)
    {
        int framesToAdvance = int(_elapsedTime / _frameDuration);
        advanceFrames(framesToAdvance);
        _elapsedTime -= _frameDuration * framesToAdvance;
    }
}

void AnimatedSpriteRenderer::advanceFrames(int nFrames)
{
    if (_animationUVs.size() == 0) return;

    _frameIndex += nFrames;

    while (_frameIndex > _animationUVs.size() - 1)
    {
        _frameIndex -= _animationUVs.size();
    }

    _sprite->SetUVs(_animationUVs[_frameIndex]);
}
```

Figure 29: AnimatedSpriteRenderer Update and advanceFrames methods

2.4.3 Rendering

By now, we have seen most of the elements that take place in the rendering process. But the actual rendering hasn't been presented yet. The class in charge of this is the _RenderingEngine. It is quite a complex class so it will be presented step by step.

2.4.3.1 Initializing the rendering engine

The _RederingEngine is a singleton that is initialized from the Game class using the method *Initialize*.

```
void _RenderingEngine::Initialize(_Window* window, GLint textureFilteringMode)
{
    if (_initialized) return;
    _initialized = true;

    _window = window;
    _window->MakeContextCurrent();
    _textureFilteringMode = textureFilteringMode;

    initializeGL();
    installShaders();
    prepareGL();

    setCamera(_defaultCamera);
}
```

Figure 30: _RenderingEngine Initialize method

This method takes the game window as a parameter, and also the texture filtering mode. It calls the *MakeContextCurrent* method of the window, that in turn calls the same method on the GLFW library. This method is in charge of creating an OpenGL context and making it current to the calling thread. Anter that, *initializeGL*, *installShaders* and *prepareGL* are called. Finally, the camera is set to the default one (that has been created on the constructor).

The *initializeGL* method initializes the GLEW library and sets some flags and parameters.

```
void _RenderingEngine::initializeGL()
{
    glewExperimental = true;
    if (glewInit() != GLEW_OK)
    {
        printf("\nERROR: _RenderingEngine::initializeGL -> Failed to initialize GLEW");
        fflush(stdout);
        exit(EXIT_FAILURE);
    }

    glClearColor(0, 0, 0, 1);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_MULTISAMPLE);//Antialiasing

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    glfwSwapInterval(0);
}
```

Figure 31: _RenderingEngine initializeGL method

The *installShaders* method calls the *installShader* method for every program object of the _RenderingEngine.

```
void _RenderingEngine::installShaders()
{
    _programID = installShader("Default.vx", "Default.fg");
    _translucencyProgramID = installShader("Default.vx", "Translucent.fg");
    _linesProgramID = installShader("Lines.vx", "Lines.fg");
}
```

Figure 32: _RenderingEngine installShaders method

The *installShader* method reads the text from the shader files, uses it to create and compile the shaders and creates and link a program object.

```
GLuint _RenderingEngine::installShader(std::string  vertexShaderFile, std::string  fragmentShaderFile)
{
    GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    std::string vertexShaderCodeStr = readShaderCode(vertexShaderFile);
    std::string fragmentShaderCodeStr = readShaderCode(fragmentShaderFile);

    const char*  vertexShaderCode = vertexShaderCodeStr.c_str();
    const char*  fragmentShaderCode = fragmentShaderCodeStr.c_str();

    glShaderSource(vertexShaderID, 1, &vertexShaderCode, 0);
    glShaderSource(fragmentShaderID, 1, &fragmentShaderCode, 0);

    glCompileShader(vertexShaderID);
    glCompileShader(fragmentShaderID);

    if (!checkShaderCompilationStatus(vertexShaderID) || !checkShaderCompilationStatus(fragmentShaderID))
    {
        return -1;
    }

    GLuint programID = glCreateProgram();
    glAttachShader(programID, vertexShaderID);
    glAttachShader(programID, fragmentShaderID);
    glLinkProgram(programID);

    if (!checkProgramLinkStatus(programID))
    {
        return -1;
    }

    return programID;
}
```

Figure 33: _RenderingEngine installShader method

The engine uses three different programs: one for rendering the opaque parts of the sprites (Default), another for rendering only the translucent parts of the sprites (Translucent) and another to render lines, which can be used for debug purposes (Lines shader).

The *prepareGL* method is in charge of generating and setting up the buffers that will be used in the rendering. To understand the code of this method we first need to understand the rendering method that has been used: instanced rendering. This method consist in drawing many instances of a geometry (that can have his own UV coordinates, transform matrix, color, etc) in a single render call. Since the geometry of all the sprites is the same (a quad), the engine makes use of this method to highly increase its performance.

The first step is to generate the vertex buffer and load it with the quad geometry. This is the only buffer that is loaded with data in the *prepareGL* method, since the other buffers will update their data every frame.

```
//Vertex buffer
glGenBuffers(1, &_vertexBufferID);

glBindBuffer(GL_ARRAY_BUFFER, _vertexBufferID);
glBufferData(GL_ARRAY_BUFFER, sizeof(_quad.vertices), _quad.vertices, GL_STATIC_DRAW);
```

Figure 34: _RenderingEngine prepareGL method (vertex Buffer)

Then, the UV coordinates buffer is generated, and the attribute pointers and divisors are set.

```
//UV buffer
glGenBuffers(1, &_uvBufferID);
glBindBuffer(GL_ARRAY_BUFFER, _uvBufferID);

glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);
glEnableVertexAttribArray(3);
glEnableVertexAttribArray(4);

glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 2 * 4, (void*)(0));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 2 * 4, (void*)(sizeof(GLfloat) * 2));
glVertexAttribPointer(3, 2, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 2 * 4, (void*)(sizeof(GLfloat) * 4));
glVertexAttribPointer(4, 2, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 2 * 4, (void*)(sizeof(GLfloat) * 6));

glVertexAttribDivisor(1, 1);
glVertexAttribDivisor(2, 1);
glVertexAttribDivisor(3, 1);
glVertexAttribDivisor(4, 1);
```

Figure 35: _RenderingEngine prepareGL method (uv buffer)

The previous code looks quite complex but all we need to know is that it is specifying that the attribute at location 1 (which will be used in the vertex shader) is an array of four UV coordinates (2D vectors). It also specifies (using the attribute divisor) that this array will be the same for each vertex in the instance (each quad will have his own array of UV coordinates, but this array will be shared between all the vertices of the quad). Each vertex will access to its UV coordinate using the index of the vertex. The first vertex will use the first UV coordinate of the array, the second vertex the second one, etc.

The next step is to do the same with the transform matrix. But is this case, instead of an array of four 2D vectors (a 4x2 matrix), it will be a 4x4 matrix.

```
//Transforms
glGenBuffers(1, &_transformBufferID);
glBindBuffer(GL_ARRAY_BUFFER, _transformBufferID);

glEnableVertexAttribArray(5);
glEnableVertexAttribArray(6);
glEnableVertexAttribArray(7);
glEnableVertexAttribArray(8);

glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 4 * 4, (void*)(0));
glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 4 * 4, (void*)(sizeof(GLfloat) * 4));
glVertexAttribPointer(7, 4, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 4 * 4, (void*)(sizeof(GLfloat) * 8));
glVertexAttribPointer(8, 4, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 4 * 4, (void*)(sizeof(GLfloat) * 12));

glVertexAttribDivisor(5, 1);
glVertexAttribDivisor(6, 1);
glVertexAttribDivisor(7, 1);
glVertexAttribDivisor(8, 1);
```

Figure 36: _RenderingEngine prepareGL method (transform buffer)

The next buffer is the color buffer. Again, we do the same that with the previous buffers, but with a 4d vector.

```
//Color buffer
glGenBuffers(1, &_colorBufferID);

glBindBuffer(GL_ARRAY_BUFFER, _colorBufferID);

glEnableVertexAttribArray(9);
glVertexAttribPointer(9, 4, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 4, 0);

glVertexAttribDivisor(9, 1);
```

Figure 37: _RenderingEngine prepareGL method (color buffer)

The last buffer is the index buffer. It is generated and loaded with the quad index data.

```
//Indices
GLuint indexBufferID;

glGenBuffers(1, &indexBufferID);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(_quad.indices), _quad.indices, GL_STATIC_DRAW);
```

Figure 38: _RenderingEngine prepareGL method (index buffer)

An extra buffer is also generated for storing vertices of lines, which will be renderer using a different shader that the sprites. This lines can be generated using the Drawer class (which will be explained later) for debugging purposes.

```
//LINES
//Vertex buffer
glGenBuffers(1, &_linesVertexBufferID);
```

Figure 39: _RenderingEngine prepareGL method (lines vertex buffer)

Finally, the texture unit that is going to be used is set and the shader uniforms are updated consequently.

```
//Textures
glActiveTexture(GL_TEXTURE0);

glProgramUniform1i(_programID, glGetUniformLocation(_programID, "tex"), 0);
glProgramUniform1i(_translucencyProgramID, glGetUniformLocation(_translucencyProgramID, "tex"), 0);
```

Figure 40: _RenderingEngine prepareGL method (texture)

2.4.3.2 Shaders

The Default and Translucent programs use the same vertex shader, *Default.vx.* This shader receives the position, UV coordinates, transform and color of the vertex. It applies the transform to the vertex position and sets the texCoord and fragColor variables with the UV coordinate and color of the vertex respectively. Both variables will be interpolated for every fragment.

```
#version 450

in layout(location=0) vec3 position;
in layout(location=1) vec2 uvs[4];
in layout(location=5) mat4 transform;
in layout(location=9) vec4 color;


out vec2 texCoord;
out vec4 fragColor;

void main()
{
    gl_Position = transform * vec4(position, 1.0);
    texCoord = uvs[gl_VertexID];
    fragColor = color;
}
```

Figure 41: Default vertex shader

The default fragment shader *Default.fg* receives the interpolated texCoord and fragColor values. It also receives via uniform variables the sampler2D of the texture it has to use, and the screenRegionMin and screenRegionMax variables. This two variables define the visible area of the screen, which is defined depending on some parameters that will be explained later. This shader also defines the method *outsideScreenRegion* whose function is to check if the pixel is inside the screen region or not.

If the alpha of the texel (texture pixel at the coordinate textCoord) or the fragColor alpha is less than one or the pixel is outside the screen region, it is discarded. Else, the pixel color is set to be the texel color multiplied by fragColor.

The translucent fragment shader *Translucent.fg* is exactly the same except it discards the pixel if the texel alpha is one and the alpha of the fragColor is also one, or if any of these two alpha values is 0. So, basically, it discards any pixel that isn't translucid (pixels that are fully opaque or fully transparent).

```
#version 450

out layout(location = 0) vec4 outColor;

in vec2 texCoord;
in vec4 fragColor;

uniform sampler2D tex;
uniform vec2 screenRegionMin;
uniform vec2 screenRegionMax;

bool outsideScreenRegion()
{
    return gl_FragCoord.x < screenRegionMin.x ||
           gl_FragCoord.y < screenRegionMin.y ||
           gl_FragCoord.x > screenRegionMax.x ||
           gl_FragCoord.y > screenRegionMax.y;
}

void main()
{
    vec4 texel = texture(tex, texCoord);
    if(texel.a < 1 || fragColor.w < 1 || outsideScreenRegion())
    {
        discard;
    }

    outColor = texel * fragColor;
}
```

Figure 42: Default fragment shader

```
#version 450

out layout(location = 0) vec4 outColor;

in vec2 texCoord;
in vec4 fragColor;

uniform sampler2D tex;
uniform vec2 screenRegionMin;
uniform vec2 screenRegionMax;

bool outsideScreenRegion()
{
    return gl_FragCoord.x < screenRegionMin.x ||
           gl_FragCoord.y < screenRegionMin.y ||
           gl_FragCoord.x > screenRegionMax.x ||
           gl_FragCoord.y > screenRegionMax.y;
}

void main()
{
    vec4 texel = texture(tex, texCoord);
    if((texel.a == 1 && fragColor.w == 1) || (texel.a == 0 || fragColor.w == 0) || outsideScreenRegion())
    {
        discard;
    }

    outColor = texel * fragColor;
}
```

Figure 43: Translucent fragment shader

The lines vertex shader *Lines.vx* and fragment shader *Lines.fg* are simplified versions of the corresponding default shaders. They only use the fragColor, since lines aren't rendered using textures, and the fragment shader only discards the fragment if it's outside the screen region, it doesn't check any type of translucency.

```
#version 450

in layout(location=0) vec3 position;
in layout(location=9) vec4 color;


out vec4 fragColor;

void main()
{
    gl_Position = vec4(position, 1.0);
    fragColor = color;
}
```

Figure 44: Lines vertex shader

```
#version 450

out layout(location = 0) vec4 outColor;

in vec4 fragColor;

uniform vec2 screenRegionMin;
uniform vec2 screenRegionMax;

bool outsideScreenRegion()
{
    return gl_FragCoord.x < screenRegionMin.x ||
           gl_FragCoord.y < screenRegionMin.y ||
           gl_FragCoord.x > screenRegionMax.x ||
           gl_FragCoord.y > screenRegionMax.y;
}

void main()
{
    if(outsideScreenRegion())
    {
        discard;
    }
    outColor = fragColor;
}
```

Figure 45: Lines fragment shader

2.4.3.3 Render update

Each game loop, the Game class calls the *Update* method of the _RenderingEngine. This method clears the screen, updates the data needed for rendering, sets the necessary flags and renders the scene.

```
void _RenderingEngine::Update()
{
    glEnable(GL_DEPTH_TEST);
    glDepthMask(GL_TRUE);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //SPRITES
    updateRenderData();

    //Default (opaque)
    glUseProgram(_programID);
    render();

    //Translucent
    glDepthMask(GL_FALSE);
    glUseProgram(_translucencyProgramID);
    render();

    //LINES
    updateLinesRenderData();

    glDisable(GL_DEPTH_TEST);
    glUseProgram(_linesProgramID);
    linesRender();

    Drawer::GetInstance()->Clear();

    //Swap buffers
    _window->SwapBuffers();
}
```

Figure 46: _RenderingEngine Update method

At the start of the method the depth test is enabled, the depth mask flag is set to true, which means that everything that is renderer will also update the depth buffer (which is used for doing the depth test) and the color and depth buffer are cleared. Next the data needed for rendering sprites is updated and loaded to its corresponding buffers using the method *updateRenderData*.

Now that the data has been loaded the sprites are rendered. To do so, the default program (which only renders paque pixels and discard the rest) is set as the active program and then the *render* method is called. After that the depth mask flag is set to false and the translucent program is set as the active one. Then the *render* method is called again. The depth mask is set to false because you can see through translucent fragments, so fragments of sprites that are more far away should alse be rendered even if they are behind a translucid fragment.

After this, the data needed to render lines is loaded with the *updateLinesRenderData* method, the depth test is disabled (since lines are drawn in front of everything else), the

lines program is set to be the active program and the lines are rendered with the method *linesRender*. The the *Clear* method of the Drawer class is called. Finally the window buffers are swapped using the *SwapBuffers* method of the _Window class.

The *updateRenderData* is a complex method so it will be explained step by step. First, the method sets the location and data format of the attribute 0.

```
//Vertices
glBindBuffer(GL_ARRAY_BUFFER, _vertexBufferID);
glVertexAttribPointer(0, 3, _quad.vertexDataType, GL_FALSE, _quad.vertexSize, 0);
```

Figure 47: _RenderEngine updateRenderData method (set attribute 0)

Next, three lists that contain the information of all the sprites in the scene are created and loaded: one contains the transforms, another the UV coordinates and the last one the colors of the sprites. The transform matrix of each sprite (which is the combination of the transform of the sprite and the parent game object) is combined with the camera transform matrix and it is modified to apply the parallax effect.

```
//Transforms + UVs + Colors
std::vector<glm::mat4> transforms;
transforms.reserve(_spriteManager->GetSpriteCount());

std::vector<GLfloat> uvs;
uvs.reserve(_spriteManager->GetSpriteCount()*8);

std::vector<glm::vec4> colors;
colors.reserve(_spriteManager->GetSpriteCount());

for (int i = 0; i < _spriteManager->GetSpriteBatchVector()->size(); ++i)
{
    _SpriteBatch* spriteBatch = _spriteManager->GetSpriteBatchVector()->at(i);
    std::vector<_Sprite*>* sprites = spriteBatch->GetSpriteVector(true);

    for (int j = 0; j < sprites->size(); ++j)
    {
        _Sprite* sprite = sprites->at(j);
        glm::mat4 transformMatrix = sprite->IsCameraSpace() ?
            _camera->GetCameraSpaceTransform()* sprites->at(j)->GetTransform() :
            _camera->GetTransform()* sprites->at(j)->GetTransform();

        transformMatrix[3][0] *= sprites->at(j)->GetParallax();
        transformMatrix[3][1] *= sprites->at(j)->GetParallax();

        transforms.push_back(transformMatrix);

        const std::vector<GLfloat>* spriteUVs = sprites->at(j)->GetUVs();
        uvs.insert(uvs.end(), spriteUVs->begin(), spriteUVs->end());

        colors.push_back(sprite->GetColor());
    }
}
```

Figure 48: _RenderEngine updateRenderData method (transforms, uvs and colors lists)

Finally, the data of the lists is loaded into the respective buffers.

```
if (transforms.size() > 0)
{
    glBindBuffer(GL_ARRAY_BUFFER, _transformBufferID);
    glBufferData(GL_ARRAY_BUFFER, sizeof(glm::mat4) * transforms.size(), &transforms[0], GL_DYNAMIC_DRAW);
}
if (uvs.size() > 0)
{
    glBindBuffer(GL_ARRAY_BUFFER, _uvBufferID);
    glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*uvs.size(), &uvs[0], GL_DYNAMIC_DRAW);
}
if (colors.size() > 0)
{
    glBindBuffer(GL_ARRAY_BUFFER, _colorBufferID);
    glVertexAttribDivisor(9, 1);
    glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec4)*colors.size(), &colors[0], GL_DYNAMIC_DRAW);
}
```

Figure 49: _RenderEngine updateRenderData method (load data)

The lines verion of the *updateRenderData* method is the *updateLinesRenderData.* This method is a simplified version of the previous, it simply loads the vertices and colors of the lines into their respective buffers.

```
void _RenderingEngine::updateLinesRenderData()
{
    std::vector<glm::vec4>* vertices = Drawer::GetInstance()->getVertices();
    std::vector<GLfloat> processedVertices;

    for (int i = 0; i < vertices->size(); ++i)
    {
        vertices->at(i) = _camera->GetTransform() * vertices->at(i);

        processedVertices.push_back(vertices->at(i).x);
        processedVertices.push_back(vertices->at(i).y);
        processedVertices.push_back(vertices->at(i).z);
    }

    if (processedVertices.size() > 0)
    {
        glBindBuffer(GL_ARRAY_BUFFER, _linesVertexBufferID);
        glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*processedVertices.size(), &processedVertices[0], GL_STATIC_DRAW);

        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 3, 0);
    }

    std::vector<glm::vec4>* colors = Drawer::GetInstance()->getColors();

    if (colors->size() > 0)
    {
        glBindBuffer(GL_ARRAY_BUFFER, _colorBufferID);
        glVertexAttribDivisor(9, 0);
        glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec4)*colors->size(), &colors->at(0), GL_DYNAMIC_DRAW);
    }
}
```

Figure 50: _RenderEngine updateLinesRenderData method

The *render* method is in charge of rendering the sprites. It loops through all the sprite batches, sets the active texture to the texture of the batch, and then draws all the sprites of the batch.

```
void _RenderingEngine::render()
{
    GLint baseInstance = 0;
    GLsizei spriteCount;
    for (int i = 0; i < _spriteManager->GetSpriteBatchVector()->size(); i++)
    {
        _SpriteBatch* spriteBatch = _spriteManager->GetSpriteBatchVector()->at(i);

        setTexture(spriteBatch->GetTexture());

        spriteCount = (GLsizei)spriteBatch->GetSpriteVector()->size();

        glDrawElementsInstancedBaseInstance(_quad.renderMode, _quad.indexCount,
                                            _quad.indexDataType, 0, spriteCount, baseInstance);

        baseInstance += spriteCount;
    }
}
```

Figure 51: _RenderEngine render method

The texture is set using the *setTexture* method. This method gets the texture id of the _Texture that receives as parameter and calls *glBindTexture* with that texture id.

```
void _RenderingEngine::setTexture(_Texture* texture)
{
    auto it = _textureMap.find(texture);
    if (it != _textureMap.end())
    {
        glBindTexture(GL_TEXTURE_2D, it->second);
    }
    else
    {
        printf("\nWARNING: _Renderer::setTexture -> Texture not loaded");
    }
}
```

Figure 52: _RenderEngine setTexture method

To be able to use a texture it has to be loaded before. This is done in the *LoadTexture* method, which is called in the constructor of the _SpriteBatch class. This method generates a texture, sets its properties and load its data.

```
void _RenderingEngine::LoadTexture(_Texture * texture)
{
    if (_textureMap.find(texture) != _textureMap.end()) return;

    GLuint textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, _textureFilteringMode);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, _textureFilteringMode);

    glTexImage2D(GL_TEXTURE_2D, 0, texture->GetFormat(), texture->GetWidth(), texture->GetHeight(),
                 0, texture->GetFormat(), texture->GetDataType(), texture->GetPixels());

    _textureMap.insert(std::pair<_Texture*, GLuint>(texture, textureID));
}
```

Figure 53: _RenderEngine LoadTexture method

The *linesRender* is the lines verison of the *render* method. It calls the *glDrawArrays* (the method that actually starts the rendering) multiple times, one per line width. For instance, if the user wants to render two lines, one with width one and the other with width two, this method will set the line width to one, render the first line, then change the line width to two and render the other line.

```
void _RenderingEngine::linesRender()
{
    int lineWidth, nLines;
    int baseVertex = 0;
    int size;

    for (auto it = Drawer::GetInstance()->_lineMap.begin(); it != Drawer::GetInstance()->_lineMap.end(); ++it)
    {
        lineWidth = it->first;
        nLines = it->second.size();
        size = nLines * 2;

        glLineWidth(lineWidth);
        glDrawArrays(GL_LINES, baseVertex, size);

        baseVertex = size;
    }
}
```

Figure 54: _RenderEngine linesRender method

Finally, the _RenderingEngine has a method for setting the current camera. This method is *SetCamera.* This method also updates the uniform variables of the shaders that will be used to determine if a pixel is inside the visible screen region or not.

```
void _RenderingEngine::SetCamera(Camera * camera)
{
    _camera = camera;
    Camera::setActiveCamera(_camera);

    //Screen Region
    Rectangle screenRegion = _camera->GetScreenRegion();
    GLfloat min[2] = { screenRegion.min.x, screenRegion.min.y };
    GLfloat max[2] = { screenRegion.max.x, screenRegion.max.y };

    glProgramUniform2fv(_programID, glGetUniformLocation(_programID, "screenRegionMin"), 1, &min[0]);
    glProgramUniform2fv(_programID, glGetUniformLocation(_programID, "screenRegionMax"), 1, &max[0]);
    glProgramUniform2fv(_translucencyProgramID, glGetUniformLocation(_translucencyProgramID, "screenRegionMin"), 1, &min[0]);
    glProgramUniform2fv(_translucencyProgramID, glGetUniformLocation(_translucencyProgramID, "screenRegionMax"), 1, &max[0]);
    glProgramUniform2fv(_linesProgramID, glGetUniformLocation(_linesProgramID, "screenRegionMin"), 1, &min[0]);
    glProgramUniform2fv(_linesProgramID, glGetUniformLocation(_linesProgramID, "screenRegionMax"), 1, &max[0]);
}
```

Figure 55: _RenderEngine SetCamera method

2.4.4 Drawer

The Drawer class allows the user to draw lines. It acts as an interface between the user and the _RenderingEngine. To draw a line, the user only needs to call the *DrawLine* method. This method stores the line information (vertices and color) in a map that uses the line width as key. This is done because lines with different width have to be rendered in different draw calls, and this is an easy way to get all lines with the same width.

```
void Drawer::DrawLine(glm::vec2 pointA, glm::vec2 pointB, glm::vec4 color, int width)
{
    glm::vec4 pA = glm::vec4(pointA.x, pointA.y, 0, 1);
    glm::vec4 pB = glm::vec4(pointB.x, pointB.y, 0, 1);

    _lineMap[width].push_back({ pA, pB, color });
}
```

Figure 56: Drawer DrawLine method

Every frame, the _RendererEngine renders the lines stored in the map, and, after the rendering, it clears the map by calling the *Clear* method of the Drawer class.

2.4.5 Camera

The camera is an essential element of the rendering process. The Camera class inherits from the Component class so its transform depends of its parent game object transform. For every sprite, the rendering engine gets the camera transform and combines it with the sprite transform. Depending if the sprite is rendered in camera space or not, the rendering engine will call either *GetTransform* method or *GetCameraSpaceTransform* method.

The *GetTransform* method combines the parent game object transform with the orthographic projection matrix, which is retrieved using the *getOrthoTransform* method, and the look-at matrix (a viewing matrix derived from an eye point). The *GetCamerSpaceTransform* simply return the result of calling *getOrthoTransform* with the value of zoom = 1. This value has been stored in a variable to avoid recalculating it.

```
glm::mat4 Camera::GetTransform()
{
    if (_isTransformDirty || (_gameObject != nullptr && _gameObject->IsTransformDirty()))
    {
        float orientation = 0;
        glm::vec3 position = glm::vec3();

        if (_gameObject != nullptr)
        {
            orientation = _gameObject->GetOrientation();
            position = _gameObject->GetPosition();
            position.z = 0;
        }

        glm::vec3 up = glm::vec3(glm::vec4(0.0f, 1.0f, 0.0f, 1.0f) *
                       glm::rotate(glm::mat4(), - orientation, glm::vec3(0.0f, 0.0f, -1.0f)));

        _transform = getOrthoTransform(_zoom) * glm::lookAt(position, position +
                     glm::vec3(0.0f, 0.0f, -1.0f), up);

        _isTransformDirty = false;
    }

    return _transform;
}

glm::mat4 Camera::GetCameraSpaceTransform()
{
    return _cameraSpaceTransform;
}
```

Figure 57: Camera GetTransform and GetCameraSpaceTransform methods

To understand the getOrthoTransform we first need the concept of reference window size and screen fit. The reference window size is the size of the window (or screen if it is played in fullscreen) the game is meant to be played. For instance, most games nowadays are meant to be played in a 1920x1080 window. When this games are played in a window with different size, say 1024x720, the camera transform has to be modified in some way so it shows the same (or a similar) portion of the game world than in a 1920x1080 window. The parameter that specifies how the camera has to be modified is the screen fit mode. There are four screen fit modes: expand, width fit, height fit and best fit. Expand will expand (or shrink) the game view to show exactly the same than in the reference window, deforming it if necessary. The width fit will adjust it so the horizontal borders (left and right) stay in the

corresponding horizontal window borders. The height fit will do the same but with the vertical borders. And finally the best fit will choose automatically between width fit or height fit based on which mode doesn't cut off any of the original game view. This concepts are better understood with images:

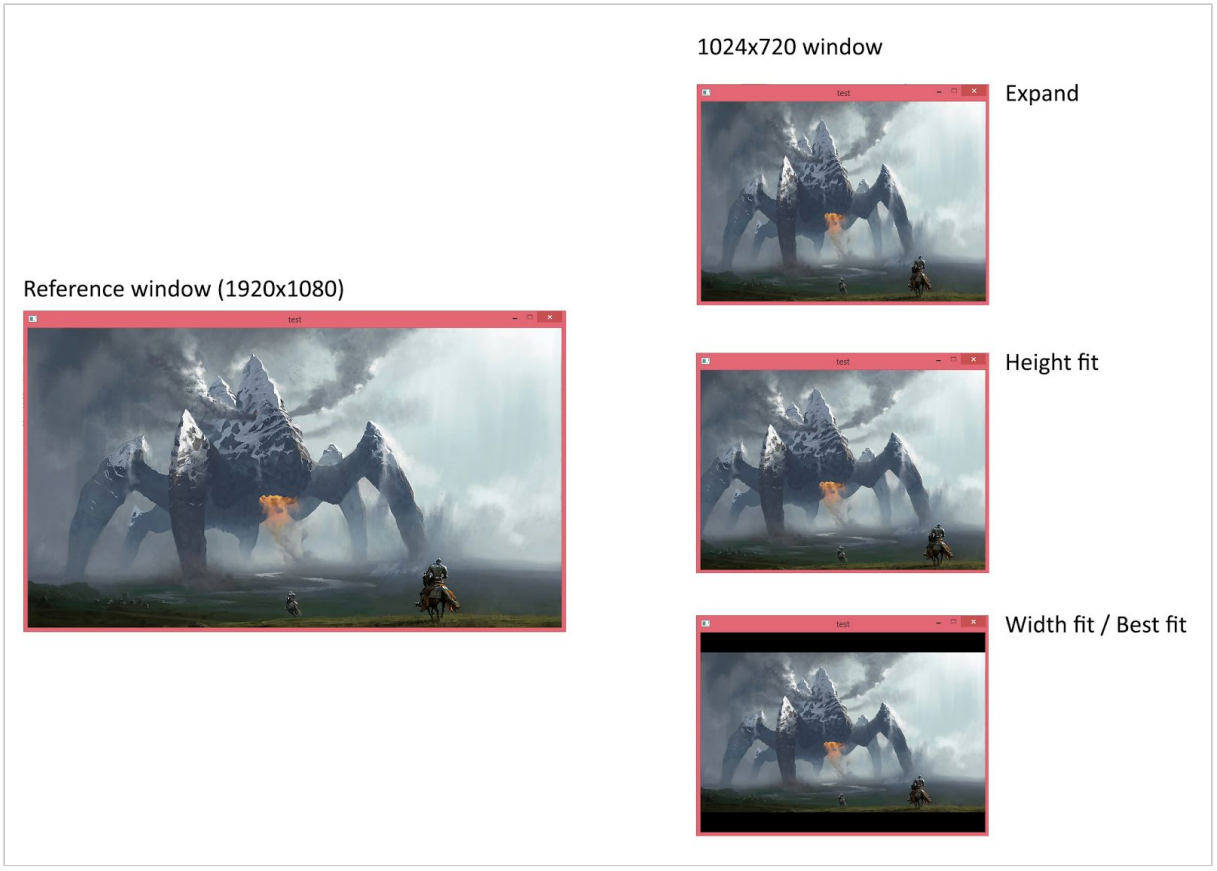

Figure 58: Screen fit modes

The *getOrthoTransform* method is the one in charge or obtaining the transform that applies the screen fit mode. This transform also moves the camera by an offset to set its pivot point at the center of the screen (by default is at the bottom left corner).

```
glm::mat4 Camera::getOrthoTransform(float zoom)
{
    float zoomOffsetW, zoomOffsetH, zoomMultiplier;

    switch (_screenFitMode)
    {
        case Camera::_ScreenFitMode::BEST_FIT:
            zoomMultiplier = glm::min(_windowWidth / _referenceWindowWidth, _windowHeight / _referenceWindowHeight);
            zoomOffsetW = (_windowWidth - (_windowWidth / (zoom * zoomMultiplier))) / 2;
            zoomOffsetH = (_windowHeight - (_windowHeight / (zoom * zoomMultiplier))) / 2;
            break;

        case Camera::_ScreenFitMode::WIDTH_FIT:
            zoomMultiplier = (float)_windowWidth / _referenceWindowWidth;
            zoomOffsetW = (_windowWidth - (_windowWidth / (zoom * zoomMultiplier))) / 2;
            zoomOffsetH = (_windowHeight - (_windowHeight / (zoom * zoomMultiplier))) / 2;
            break;

        case Camera::_ScreenFitMode::HEIGHT_FIT:
            zoomMultiplier = (float)_windowHeight / _referenceWindowHeight;
            zoomOffsetW = (_windowWidth - (_windowWidth / (zoom * zoomMultiplier))) / 2;
            zoomOffsetH = (_windowHeight - (_windowHeight / (zoom * zoomMultiplier))) / 2;
            break;

        case Camera::_ScreenFitMode::EXPAND:
            zoomOffsetW = (_windowWidth - (_windowWidth / (zoom * _windowWidth / _referenceWindowWidth))) / 2;
            zoomOffsetH = (_windowHeight - (_windowHeight / (zoom * _windowHeight / _referenceWindowHeight))) / 2;
            break;
        }

    glm::mat4 orthoTranform = glm::ortho(0.0f + zoomOffsetW,
                              _windowWidth - zoomOffsetW, 0.0f + zoomOffsetH,
                              _windowHeight - zoomOffsetH,
                              _maxDepth, -_maxDepth);

    return  orthoTranform * glm::translate(glm::mat4(), glm::vec3(_windowWidth / 2, _windowHeight / 2, 0));
}
```

Figure 59: Camera getOrthoTransform method

The Camera also has a function to get the visible screen region. Everything outside this region will be renderer black. This region is calculated using the window size, reference window size and screen fit mode. This method is used by the _RenderingEngine to update the shader uniforms that will be used to determine if a pixel is rendered or not (if it is inside the visible screen region or not). This method makes use of the *CameraToScrenPosition*, which is one of the multiple method that the camera has to transform coordinates from a space to another. This methods transform coordinates between the world, camera and screen space, in any combination.

```
Rectangle Camera::GetScreenRegion()
{
    Rectangle screenRegion;

    if (_screenFitMode == _ScreenFitMode::EXPAND)
    {
        screenRegion = Rectangle(0, 0, _windowWidth, _windowHeight);
    }
    else
    {
        if (_screenFitMode == _ScreenFitMode::WIDTH_FIT ||
           (_screenFitMode == _ScreenFitMode::BEST_FIT &&
           (float)_windowWidth / _referenceWindowWidth < (float)_windowHeight / _referenceWindowHeight))
        {
            glm::vec2 minCameraSpace =
                glm::vec2(-(float)_windowWidth * 0.5,
                          -((float)_windowWidth / (float)_referenceWindowWidth) * (float)_referenceWindowHeight * 0.5);
            glm::vec2 maxCameraSpace = -minCameraSpace;
            screenRegion = Rectangle(CameraToScreenPosition(minCameraSpace), CameraToScreenPosition(maxCameraSpace));
        }
        else
        {
            glm::vec2 minCameraSpace =
                glm::vec2(-((float)_windowHeight / (float)_referenceWindowHeight) * (float)_referenceWindowWidth * 0.5,
                          -(float)_windowHeight * 0.5);
            glm::vec2 maxCameraSpace = -minCameraSpace;
            screenRegion = Rectangle(CameraToScreenPosition(minCameraSpace), CameraToScreenPosition(maxCameraSpace));
        }
    }

    return screenRegion;
}
```

Figure 60: Camera GetScreenRegion

2.4.6 Text

Text rendering is an essential tool in any game engine since most games need to display text for dialogs, menus, hud, etc. To display text, first, a font has to be loaded and all the individual characters have to be stored in a single texture. The first part is done in the _FontLoader class and the second one in the _Font class. To do so, both classes use the FreeType library.

The main method of the _FontLoader is the *LoadFont* method. This method checks if the font has already been loaded, returns it if so, or creates a new one if it hasn't. The font is loaded as a _Font object and the base size of the font can be set by the user.

```
_Font* _FontLoader::LoadFont(std::string file)
{
    auto it = _fontMap.find(file);
    if (it != _fontMap.end())
    {
        return _fontMap.at(file);
    }

    FT_Face face;
    if (FT_New_Face(_ftLibrary, file.c_str(), 0, &face))
    {
        printf("\nWARNING: _FontLoader::LoadFont -> Could not open font\n");
        return nullptr;
    }

    FT_Set_Pixel_Sizes(face, _fontSize, _fontSize);

    _Font* font = new _Font(face);
    _fontMap[file] = font;

    return font;
}
```

Figure 61: _FontLoader LoadFont method

When the _Font object is created, it generates a texture with all the characters of the font as sprites. This way it can be used to render text in a quick and efficient way. The characters that will be loaded are stored in the string *AVAILIBLE_CHARS.*

```
const std::string _Font::AVAILABLE_CHARS = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890 ,.:!?()\n";
```

Figure 62: _Font AVAILIBLE_CHARS string

The method used to create the texture is the *createTexture* method. This method first gets the max character width and height by looping to all of the characters. This information is needed in order to create the texture with all the characters. After that, it loops again through all the available chars, gets their bitmaps (using FreeType methods) and adds them to a single texture.

```
FT_GlyphSlot glyph = _ftFace->glyph;

int maxCharacterWidth = 0;
int maxCharacterHeight = 0;

for (int i = 0; i < AVAILABLE_CHARS.size(); ++i)
{
    char character = AVAILABLE_CHARS[i];
    if (FT_Load_Char(_ftFace, character, FT_LOAD_RENDER))
    {
        printf("\nWARNING: _Font::createTexture -> Could not load character %s", character);
        continue;
    }

    int characterWidth = glyph->bitmap.width;
    int characterHeight = glyph->bitmap.rows;

    if (characterWidth > maxCharacterWidth) maxCharacterWidth = characterWidth;
    if (characterHeight > maxCharacterHeight) maxCharacterHeight = characterHeight;
}
```

Figure 63: _Font createTexture method (get max character size)

```
float sqrtCharN = glm::sqrt(AVAILABLE_CHARS.size());
int nRows, nColumns;
nRows = nColumns = (sqrtCharN - (int)sqrtCharN) > 0 ? (int)sqrtCharN + 1 : (int)sqrtCharN;
int textureWidth = nColumns * maxCharacterWidth;
int textureHeight = nRows * maxCharacterHeight;

GLubyte* texturePixels = new GLubyte[textureWidth * textureHeight * 4]();
Rectangle uvs;

for (int i = 0; i < AVAILABLE_CHARS.size(); ++i)
{
    char character = AVAILABLE_CHARS[i];
    if (FT_Load_Char(_ftFace, character, FT_LOAD_RENDER))
    {
        printf("\nWARNING: _Font::createTexture -> Could not load character %s", character);
        continue;
    }

    int characterWidth = glyph->bitmap.width;
    int characterHeight = glyph->bitmap.rows;

    int currentRow = (int)(i / nColumns);
    int currentColumn = i - currentRow * nColumns;

    uvs.min = glm::vec2((float)currentColumn * (float)maxCharacterWidth / (float)textureWidth,
            (float)currentRow * (float)maxCharacterHeight / (float)textureHeight);
    uvs.max = glm::vec2(((float)currentColumn * (float)maxCharacterWidth + (float)characterWidth) / (float)textureWidth,
             ((float)currentRow * (float)maxCharacterHeight + (float)characterHeight) / (float)textureHeight);

    blitCharacter(texturePixels, glyph->bitmap.buffer, uvs, textureWidth, textureHeight, characterWidth, characterHeight);

    _characterMap[character] = _CharacterInfo{ uvs,
                                               glm::vec2(glyph->bitmap.width, glyph->bitmap.rows),
                                               glm::vec2(glyph->bitmap_left, glyph->bitmap_top), glyph->advance.x / 64 };
}

_characterSheet = new _Texture(texturePixels, textureWidth, textureHeight);
```

Figure 64: _Font createTexture method (create texture)

The method used to add a character to the texture is the *blitCharacter* method. This method simply fills the texture at the corresponding UV coordinates (which are obtained for each character in the *createTexture* method, before calling this method) with the character texture in white (and with alpha).

```
void _Font::blitCharacter(GLubyte * texturePixels, GLubyte * characterPixels, Rectangle uvs,
    int textureWidth, int textureHeight, int characterWidth, int characterHeight)
{
    int basePosX = uvs.min.x * textureWidth;
    int basePosY = uvs.max.y * textureHeight -1;

    for (int y = 0; y < characterHeight; y++)
    {
        for (int x = 0; x < characterWidth; x++)
        {
            int texturePixelPos = ((basePosY - y) * textureWidth + basePosX + x) * 4;
            int characterPixelPos = (y * characterWidth + x);

            texturePixels[texturePixelPos + 0] = 255;
            texturePixels[texturePixelPos + 1] = 255;
            texturePixels[texturePixelPos + 2] = 255;
            texturePixels[texturePixelPos + 3] = characterPixels[characterPixelPos];
        }
    }
}
```

Figure 65: _Font blitCharacter method

The _Font class also implements some methods to get the information needed to correctly display text (sequences of characters). These are the position relative to the cursor, how much the cursor has to advance after adding a character, the size of the character and the line height. Those are called the glyph metrics.

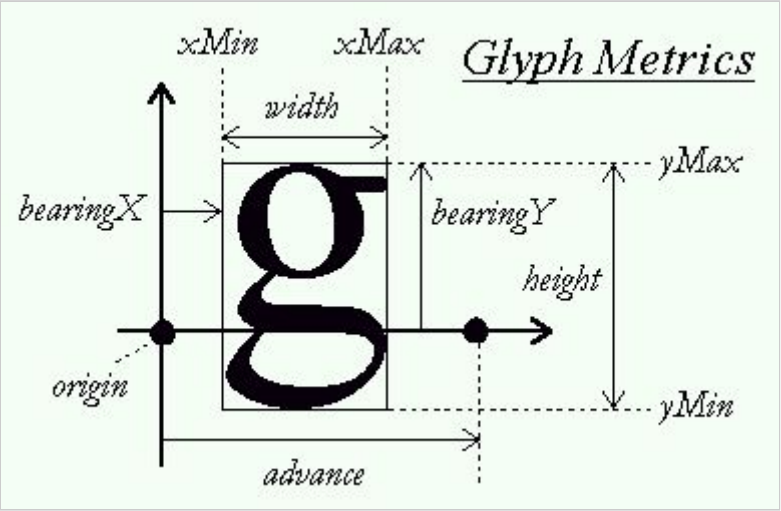

Figure 66: Glyph metrics. Image from here.

Finally, the class in charge of displaying text is the TextLabel. It inherits from the Component class, so it can be added to a game object as any other Component. This class simply adds a sprite to the scene for every character of the text string. For instance, if the user wants to render the text “Hello”, a sprite for each character will be added to the scene. Also, it can align the text horizontally (left, right or centered) and vertically (top, bottom or centered). To do so, it uses the *alignSpritesHorizontally* and *alignSpritesVertically* methods.

```
void TextLabel::alignSpritesHorizontally(int spriteIndex, int lineWidth)
{
    float alignmentScale = _horizontalAlignment == TextHorizontalAlignment::H_LEFT ?
                    0 : _horizontalAlignment == TextHorizontalAlignment::H_CENTER ?
                    0.5 : 1;

    for (int i = spriteIndex; i < _characterSprites.size(); ++i)
    {
        _Sprite* sprite = _characterSprites[i];
        sprite->SetPosition(sprite->GetPosition().x - (float)lineWidth * alignmentScale, sprite->GetPosition().y);
    }
}

void TextLabel::alignSpritesVertically(int top, int bottom)
{
    int height = top - bottom;

    float alignmentScale = _verticalAlignment == TextVerticalAlignment::V_TOP ?
                    0 : _verticalAlignment == TextVerticalAlignment::V_CENTER ?
                    0.5 : 1;

    for (int i = 0; i < _characterSprites.size(); ++i)
    {
        _Sprite* sprite = _characterSprites[i];
        sprite->SetPosition(sprite->GetPosition().x, sprite->GetPosition().y - top + height * alignmentScale);
    }
}
```

Figure 67: TextLabel alignSpritesHorizontally and alignSpritesVertically methods

2.5 Audio

The next essential subsystem of the game engine is the audio system. The engine allows the user to play spatial audio, which means that the audio properties (panning and volume) will change automatically depending of where is the audio source located in respect to the listener. The API used to play audio is OpenAL.

2.5.1 Audio engine

The class in charge of managing the low level aspects of the audio (loading, playing, spatial audio, etc) is the _AudioEngine. Before playing audio, the audio engine has to be initialized. This is done in the class constructor.

```
_AudioEngine::_AudioEngine()
{
    _nBuffersUsed = 0;

    initializeAL();
    createBuffers();
    createSources();
    initializeListener();
}
```

Figure 68: _AudioEngine constructor

First, the *initializeAL* method is called, this method creates the OpenAL device and context and also sets the OpenAL distance model to none. This means that OpenAL will disable automatic 3D spatial audio attenuation. This is done because the attenuation will be managed by the engine since it has some special features.

After that, a fixed number of sources and buffers is created. The buffers are the "containers" of the audio data and the sources are the audio players. Multiple sources can play the audio of a single buffer, but a source can only play audio from one buffer at a time. Also, the listener is initialized. The listener is the "virtual ears", the panning and attenuation of the sources will be obtained in reference to the listener. There can only be one listener per scene.

Once initialized, some extra parameters can be se calling the *SetParameters* method. This method sets the master gain and the distances used in the spatial audio calculation. This distances are the min and max panning distance and the min and max attenuation distance. The min panning distance is the horizontal distance of an audio source to the listener at

which the audio output is at the center. If the audio source is at less distance the panning won't be affected, but if it increases its distance to the listener, the audio will start panning to the left or right depending on the direction. The max panning distance is the distance at which the audio of a source will play completely on the left or completely on the right. Increasing the distance won't affect the panning, but decreasing it will. The attenuation distances work in the same way, but they affect the gain of the audio, not the panning. They also consider the Y coordinate. When the distances are set, the listener Z position is set relative to the panning distances. This is done because the panning "area" depends on the Z position of the listener relative to the sources Z position (which is always 0).

```
void _AudioEngine::SetParameters(float masterGain, float minAttenuationDistance,
    float maxAttenuationDistance, float minPanningDistance, float maxPanningDistance)
{
    SetMasterGain(masterGain);
    SetDistances(minAttenuationDistance, maxAttenuationDistance, minPanningDistance, maxPanningDistance);
}

void _AudioEngine::SetMasterGain(float gain)
{
    alListenerf(AL_GAIN, gain);
}

void _AudioEngine::SetDistances(float minAttenuationDistance, float maxAttenuationDIstance,
    float minPanningDistance, float maxPanningDistance)
{
    _minAttenuationDistance = minAttenuationDistance;
    _maxAttenuationDistance = maxAttenuationDIstance;
    _minPanningDistance = minPanningDistance;
    _maxPanningDistance = maxPanningDistance;

    _listenerZPosition = (_maxPanningDistance - _minPanningDistance) * 2;

    float listenerPos[3];
    alGetListenerfv(AL_POSITION, &listenerPos[0]);
    listenerPos[2] = _listenerZPosition;
    alListenerfv(AL_POSITION, listenerPos);
}
```

Figure 69: _AudioEngine SetParameters, SetMasterGain and SetDistances methods

Before playing an audio file, it must be loaded before. This is done in the *LoadAudio* mehod. This method checks if the audio has already been loaded. If not, it checks if there are free buffers and if so, it loads the audio into a buffer using the _AudioLoader class. The _AudioLoader is a class whose only function is to load audio files. Since OpenAL doesn't have any functionality to load audio, this process has been implemented from scratch. To simplify things a bit, at this moment the engine only works with .wav files, since the wav format is easy to understand and wav files aren't compressed.

The _AudioLoader *LoadWAV* is the method in charge of loading wav files . A WAV file is composed of differents fields which contain specific information about the file, like the number of channels, the byte rate and, of course, the raw audio data. This method simply reads field by field, discarding the ones that aren't needed, and storing the information of the ones that are. It returns an array of bytes containing the raw audio and also outputs (using reference variables) the format, size and frequency of the audio.

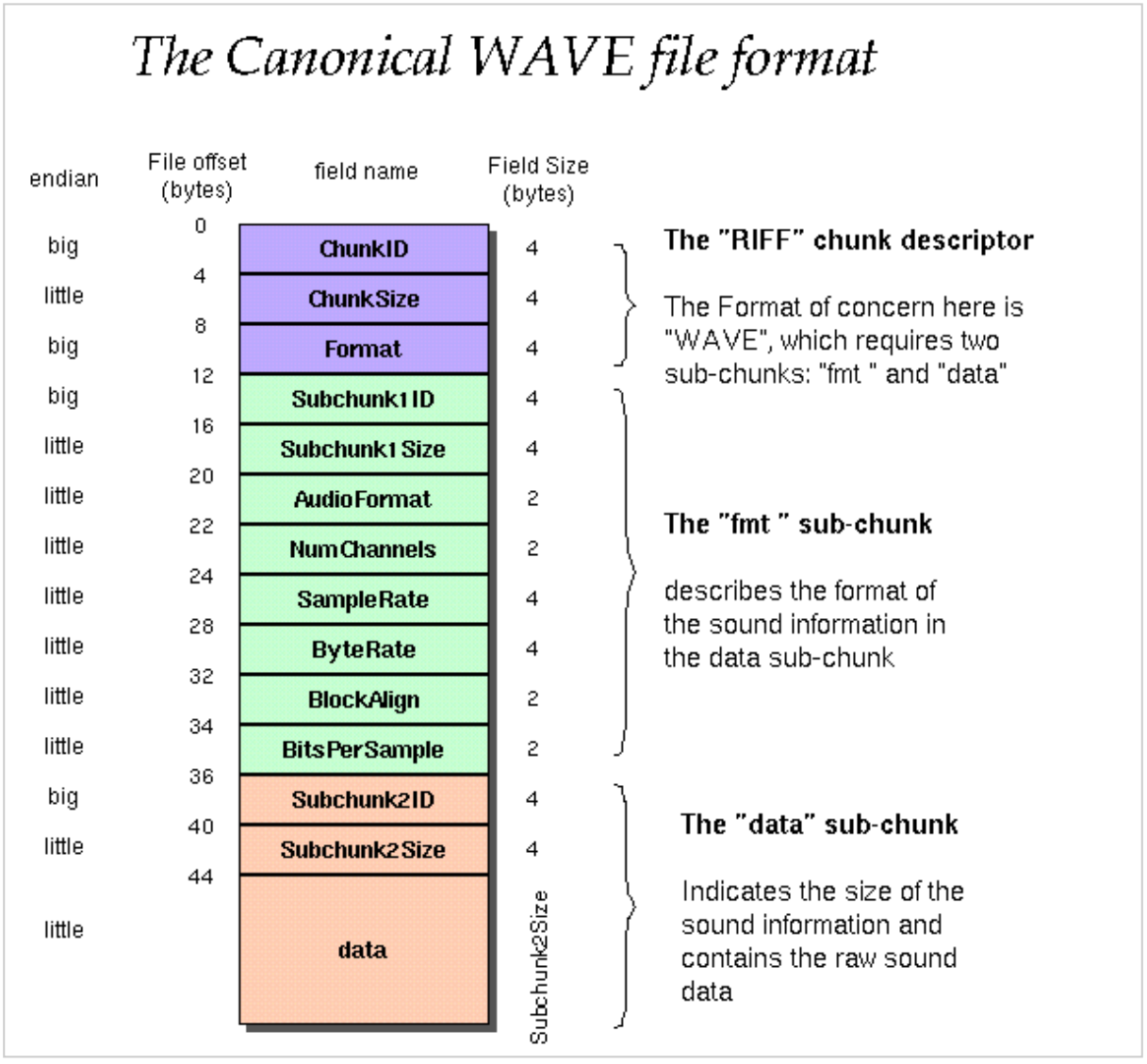

Figure 70: The WAV file format. Image from here.

Once an audio has been loaded into a buffer, the method *PlayAudio* can be used to play it. This method uses the method *getFreeSource* to get a free source (a source that is not playing or paused), sets its properties with *SetAudioProperties* method and plays it calling *PlayAudioFromSource* with the source id. The *PlayAudioFromSource* method simply tells OpenAL to play the source.

```
int _AudioEngine::PlayAudio(unsigned int bufferID,
    glm::vec2 position, bool looping, bool is3d, float gain, float pitch, float attenuationFactor)
{
    int sourceID = getFreeSource();

    if (sourceID != -1)
    {
        SetAudioProperties(sourceID, bufferID, position, looping, is3d, gain, pitch, attenuationFactor);
        PlayAudioFromSource(sourceID);
    }

    return sourceID;
}

void _AudioEngine::PlayAudioFromSource(unsigned int sourceID)
{
    if (sourceID != -1)
    {
        alSourcePlay(sourceID);
    }
}
```

Figure 71: _AudioEngine PlayAudio and PlayAudioFromSource methods

The *PlayAudio* method also returns the source id, so the class that calls this method can also call the pause and stop method, which need the source id of the source that is going to be paused or stopped. These method just call their respective OpenAL methods.

```
void _AudioEngine::StopAudio(unsigned int sourceID)
{
    if (sourceID != -1)
    {
        alSourceStop(sourceID);
    }
}

void _AudioEngine::PauseAudio(unsigned int sourceID)
{
    if (sourceID != -1)
    {
        alSourcePause(sourceID);
    }
}
```

Figure 72: _AudioEngine StopAudio and PauseAudio methods

The *SetAudioProperties* sets the source properties (buffer id, pitch and if it is looping or not) and also calls the *setSource3DProperties* method. This method applies the attenuation effect modifying the gain value depending on the source distance to the listener. It also sets the position of the source so when its distance to the listener is equal or less than the min panning distance, its distance relative to the listener is 0, and when it is larger than the max panning distance, its distance relative to the listener is the max panning distance.

```
void _AudioEngine::setSource3dProperties(unsigned int sourceID, _Source3dProperties source3dProperties)
{
    float listenerPos[3];
    alGetListenerfv(AL_POSITION, &listenerPos[0]);
    glm::vec2 listenerPosition = glm::vec2(listenerPos[0], listenerPos[1]);

    if (source3dProperties.is3d)
    {
        //Attenuation
        if (source3dProperties.attenuationFactor == 0)
        {
            alSourcef(sourceID, AL_GAIN, source3dProperties.gain);
        }
        else
        {
            float attenuationDistance = (_maxAttenuationDistance - _minAttenuationDistance) /
                                        source3dProperties.attenuationFactor;
            float distance = glm::clamp(glm::distance(source3dProperties.position, listenerPosition),
                            _minAttenuationDistance, _minAttenuationDistance + attenuationDistance);

            alSourcef(sourceID, AL_GAIN,
                source3dProperties.gain * (1 - (distance - _minAttenuationDistance) / (attenuationDistance)));
        }

        //Panning
        float positionX = source3dProperties.position.x - listenerPosition.x;

        if (positionX < 0)
        {
            positionX = glm::clamp(positionX, -_maxPanningDistance, -_minPanningDistance);
            positionX += _minPanningDistance;
        }
        else
        {
            positionX = glm::clamp(positionX, _minPanningDistance, _maxPanningDistance);
            positionX -= _minPanningDistance;
        }

        alSource3f(sourceID, AL_POSITION, listenerPosition.x + positionX, source3dProperties.position.y, 0);
    }
    else
    {
        alSourcef(sourceID, AL_GAIN, source3dProperties.gain);
        alSource3f(sourceID, AL_POSITION, listenerPosition.x, listenerPosition.y, 0);
    }

    //Update map
    _source3dPropertiesMap[sourceID] = source3dProperties;
}
```

Figure 73: _AudioEngine setSource3DProperties method

Finally, the listener properties can also be set calling the method *SetListenerProperties*. The properties of the listener that can change are the position and the orientation. Every time the listener properties are changed, the *setSource3DProperties* method is called again for each active source to update their panning and attenuation.

2.5.2 Audio source

The AudioSource class is a Component whose function is to play audio. Since it is a Component, it has to be attached to a game object, so the position of the game object will affect the sound (if the spatial audio is enabled). The AudioSource allow the user to fade in or out the audio, so it starts or stops playing in a smoother way. Whenever an audio is played, stopped or paused, a flag is set to indicate if the audio is fading and if so, if it is fading in or fading out, and if it is fading out if after the fade it has to stop or pause. The fading speed is also set depending on the time the user wants the fade to last. Then, in the update method, the gain is adjusted slightly every frame until it is completely faded.

```
void AudioSource::Update(double deltaTime)
{
    if (_fadeState != FadeState::NONE)
    {
        if (_fadeState == FadeState::FADING_IN_PLAY)
        {
            _gain += _fadeSpeed * deltaTime;

            if (_gain > _baseGain)
            {
                _gain = _baseGain;
                _fadeState = FadeState::NONE;
            }
        }
        else
        {
            _gain -= _fadeSpeed * deltaTime;

            if (_gain < 0)
            {
                _gain = 0;

                if (_fadeState == FadeState::FADING_OUT_STOP)
                {
                    stop();
                }
                else
                {
                    pause();
                }

                _fadeState = FadeState::NONE;
            }
        }
    }

    updateAudioProperties();
}
```

Figure 74: AudioSource Update method

Every AudioSource has his unique source id. When the audio source plays for the first time or after being stopped, it calls the *PlayAudio* method from the _AudioEngine and stores the source id returned. The, it uses it to pause and stop the audio. After pausing, instead of calling *PlayAudio* again, it calls *PlayAudioFromSource* since the source hasn't changed.

2.5.3 Audio listener

The AudioListener is another Component whose only function is to update the listener properties of the _AudioEngine. This is done in the *Update* method.

```
void AudioListener::Update(double deltaTime)
{
    if (_activeListener)
    {
        if (_previousPosition.x != _gameObject->GetPosition().x ||
            _previousPosition.y != _gameObject->GetPosition().y ||
            _previousOrientation != _gameObject->GetOrientation())
        {
            _AudioEngine::GetInstance()->SetListenerProperties(glm::vec2(_gameObject->GetPosition()),
                                                               _gameObject->GetOrientation());

            _previousPosition = glm::vec2(_gameObject->GetPosition());
            _previousOrientation = _gameObject->GetOrientation();
        }
    }
}
```

Figure 75: AudioListener Update method

There can only be one active listener per scene, so the *OnInitialiize* method check if there is already a listener (checking the static variable *_existsListener*), and if so, sets the *_activeListener* variable to false. If there isn't another listener, it sets both *_activeListener* and *_existsListener* to true.

```
void AudioListener::OnInitialize()
{
    _previousPosition = glm::vec2();
    _previousOrientation = 0;

    if (_existsListener)
    {
        _activeListener = false;

        printf("\nWARNING: AudioListener::OnInitialize -> Already exists a listener");
    }
    else
    {
        _existsListener = true;
        _activeListener = true;
    }
}
```

Figure 76: AudioListener OnInitialize method

2.6 Physics

The last big module of the engine is the physics and collision module. This module is in charge of detecting and resolving collisions, simulating physics (forces, impulses, friction, restitution, etc) and generating collision queries (raycast, collision events, etc). The two base elements of the physics module are the Colliders and the Rigidbodies.

2.6.1 Collider

The Collider class inherits from the Component class and it defines a convex collider that will be used in collision detection. It also implements some methods to get extra information from the collider.

The Collider stores two list of the vertices that form the collider shape and two list of the normals. One of the lists contains the vertices/normals in local space and the other contains them in world space. It also stores the vertices transform and the normal transform. This is done to avoid recalculating the world space vertices/normals each time. It also stores a boolean that indicates if it is trigger or not, which means that the Collider will contribute to the collision resolution (if it isn't a trigger) or not. A trigger collider will still "trigger" collision events. The Collider also has a material, which stores its restitution and friction, and a mask, which are used to determine if a collision can happen between two colliders. It also stores the containing circle (the circle with center at the center of the collider with the smaller radius that surrounds the entire collider shape) and a reference to the Rigidbody added to its parent game object (if it exists).

```
std::vector<vec2> _vertices; //Clockwise order
std::vector<vec2> _transformedVertices;
std::vector<vec2> _normals;
std::vector<vec2> _transformedNormals;

bool _isTrigger;
PhysicMaterial _material;
Mask _mask;

Circle _containingCircle;
mat4 _verticesTransform, _normalsTransform;

Rigidbody* _rigidbody = nullptr;
```

Figure 77: Collider variables

The collider implements a method to determine if a point is inside it. This is the *ContainsPoint* method. This method simply checks for each normal the angle between it and the vector from the first vertex of the side to the point. If it is smaller than 90 for any of the normals, the point is outside the collider.

```
bool Collider::ContainsPoint(vec2 point)
{
    std::vector<vec2>* vertices = GetVerticesWorldSpace();
    std::vector<vec2>* normals = GetNormalsWorldSpace();

    for (int i = 0; i < vertices->size(); ++i)
    {
        vec2 normal = normals->at(i);
        vec2 vertex = vertices->at(i);

        vec2 vertex2Point = point - vertex;

        if (dot(normal, vertex2Point) > 0)
        {
            return false;
        }
    }

    return true;
}
```

Figure 78: Collider ContainsPoint method

It also implements a method for obtaining the support point, the *getSupportPoint* method. This point is the vertex that is farther away in a certain direction. To do so, it simply checks the projection of every vertex with the direction vector, the one with a larger projection is the support point.

```
vec2 Collider::getSupportPoint(vec2 direction, bool worldSpace)
{
    vec2 supportPoint;
    float maxDistance = -std::numeric_limits<float>::max();

    std::vector<vec2>* vertices = worldSpace ? GetVerticesWorldSpace() : &_vertices;

    for (int i = 0; i < vertices->size(); i++)
    {
        vec2 vertex = vertices->at(i);
        float distance = dot(direction, vertex);
        if (distance > maxDistance)
        {
            maxDistance = distance;
            supportPoint = vertex;
        }
    }

    return supportPoint;
}
```

Figure 79: Collider getSupportPoint method

Lastly, it implements a method for obtaining the the projection of the collider in a certain axis. This is done by projecting all the vertices and storing the max and min projection distance.

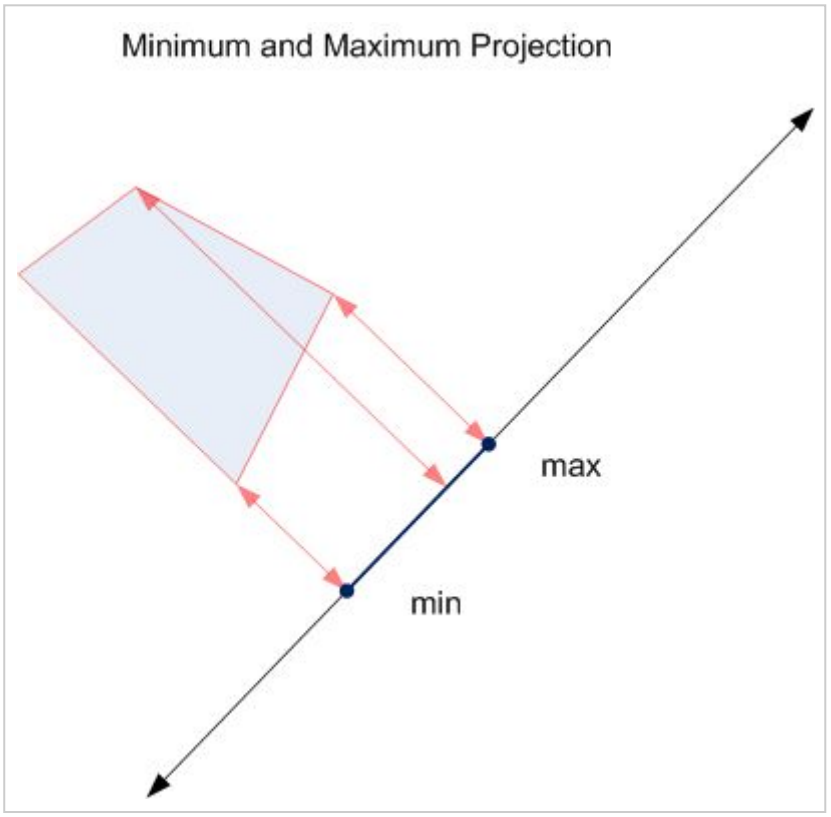

Figure 80: Collider projection. Image from here.

2.6.2 Rigidbody

Without a Rigidbody, a Collider can only be static. If we want to allow the Colliders to interact between them and to behave dynamically we need to use the Rigidbody component. Once a Rigidbody is added to a GameObject, forces and impulses can be applied to it.

The main parameters of the Rigidbody class are the gravity scale, which indicates how much the body is affected by gravity and the mass and moment of inertia (or “rotational mass”). It also has the freeze rotation and freeze position flags, which indicate is the Rigidbody can rotate or can move in a certain axis respectively. Furthermore, the Rigidbody stores its current linear velocity and acceleration, the net force and net impulse that is being applied to it and the angular speed, angular acceleration, net torque and net angular impulse. It also has a boolean that indicates if the parent game object has one or more colliders attached.

```
//Gravity
float _gravityScale;

//Linear
float _mass; //Mass of 0 -> Infinite mass
vec2 _linearVelocity;
vec2 _linearAcceleration;
vec2 _netForce;
vec2 _netLinearImpulse;

//Angular
float _momentOfInertia;
float _angularSpeed;
float _angularAcceleration;
float _netTorque;
float _netAngularImpulse;

bool _hasCollider;

bool _freezeRotation;
bool _freezePositionX;
bool _freezePositionY;
```

Figure 81: Rigidbody variables

The Rigidbody class has several methods in charge of applying forces and impulses. Those are the *ApplyForce*, *ApplyTorque* and *ApplyForceAtPoint* methods for the forces and *ApplyLinearImpulse*, *ApplyAngularImpulse* and *ApplyImpulseAtPoint* methods for the impulses. Those methods simply add the force/torque or linear impulse/angular impulse to the respective variables (_netForce, _netTorque, _netLinearImpulse or _netAngularImpulse).

The *ApplyForceAtPoint* applies both force and torque based on the position where the force is applied. Same with the *ApplyImpulseAtPoint*, except this time it applies linear and angulatr impulse.

```
void Rigidbody::ApplyForce(vec2 force)
{
    if (_freezePositionX) force.x = 0;
    if (_freezePositionY) force.y = 0;

    _netForce += force;
}

void Rigidbody::ApplyTorque(float torque)
{
    if (!_freezeRotation)
    {
        _netTorque += torque;
    }
}

void Rigidbody::ApplyForceAtPoint(vec2 force, vec2 point)
{
    vec3 r = vec3(point.x - _gameObject->GetPosition().x, point.y - _gameObject->GetPosition().y, 0);
    vec3 force3 = vec3(force.x, force.y, 0);

    ApplyTorque(-cross(r, force3).z);
    ApplyForce(force);
}
```

Figure 82: Rigidbody ApplyForce, ApplyTorque and ApplyForceAtPoint methods

```
void Rigidbody::ApplyLinearImpulse(vec2 impulse)
{
    if (_freezePositionX) impulse.x = 0;
    if (_freezePositionY) impulse.y = 0;

    _netLinearImpulse += impulse;
}

void Rigidbody::ApplyAngularImpulse(float impulse)
{
    if (!_freezeRotation)
    {
        _netAngularImpulse += impulse;
    }
}

void Rigidbody::ApplyImpulseAtPoint(vec2 impulse, vec2 point)
{
    vec3 r = vec3(point.x - _gameObject->GetPosition().x, point.y - _gameObject->GetPosition().y, 0);
    vec3 impulse3 = vec3(impulse.x, impulse.y, 0);

    ApplyAngularImpulse(-cross(r, impulse3).z);
    ApplyLinearImpulse(impulse);
}
```

Figure 83: Rigidbody ApplyLinearImpulse, ApplyAngularImpulse and ApplyImpulseAtPoint methods

Every fixed update (physics step), the Rigidbody will update the GameObject linear velocity and acceleration, angular speed and acceleration, position and orientation based on the force and impulses that have been applied. This will be done unless the GameObject

contains colliders, if it does, it will be updated from the physics engine at the collision detection/resolution phase.

First, the Rigidbody integrates the forces and impulses with the method *integrateForces.* Then, it applies the angular and linear velocities obtained in the previous step to obtain the next position and orientation. This is done in the *applyVelocities* method.

```
void Rigidbody::integrateForces(float fixedTimestep)
{
    //LINEAR
    //Half step velocity
    vec2 halfStepLinearVelocity = _linearVelocity + _linearAcceleration * fixedTimestep * 0.5f;

    //Acceleration
    _linearAcceleration = _netForce / _mass;

    //Velocity
    _linearVelocity = halfStepLinearVelocity + _linearAcceleration * fixedTimestep * 0.5f;
    _linearVelocity += _netLinearImpulse / _mass;


    //ANGULAR
    //Half step speed
    float halfStepAngularSpeed = _angularSpeed + _angularAcceleration * fixedTimestep * 0.5f;

    //Acceleration
    _angularAcceleration = _netTorque / _momentOfInertia;

    //Velocity
    _angularSpeed = halfStepAngularSpeed + _angularAcceleration * fixedTimestep * 0.5f;
    _angularSpeed += _netAngularImpulse / _momentOfInertia;



    //Forces reset
    _netForce = _freezePositionY ? vec2() :
                _PhysicsEngine::GetInstance()->GetGravity() * _mass * _gravityScale;
    _netTorque = 0;

    //Impulses reset
    _netLinearImpulse = vec2();
    _netAngularImpulse = 0;
}
```

Figure 84: Rigidbody integrateForces method

The *integrateForces* method uses an integration method based on the velocity verlet integration. It first calculates the velocity at t/2 (half step velocity). Then, it calculates the acceleration based on the net force. It then obtains the final velocity by adding to the half step velocity the result of multiplying the acceleration by t/2 and also adding the velocity obtained by the net impulse. It does this for both linear and angular velocities. After this, the net force, torque, linear impulse and angular impulse are reset since they have already

been applied. Finally, the *applyVelocities* method is called. It simply sets the new position and orientation of the game object based on the linear velocity and angular speed.

2.6.3 Collision querier

The class that contains all the methods in charge of the collision detection and raycasting is the CollisionQuerier class. This class is used by the physics engine, but it can also be used by the user directly.

2.6.3.1 Raycasting

The CollisionQuerier has two methods for raycasting, *RaycastAll* and *RaycastCollider*. *RaycastCollider* executes a raycast on a specific collider, whilst *RaycastAll* executes a raycast on all colliders of the scene. Since *RaycastAll* simply loops through all the colliders and executes *RaycastCollider* on them, only this last method needs to be explained.

This method simply loops through all the edges of each collider and check if they intersect with the ray. To do so, first the edge is converted into a line and its implicit equation is obtained. Then using the implicit equation of both the ray and the edge line the intersection point can be found. If there is intersection, the distance to the intersection is less than the max ray distance, the intersection is in the direction of the raycast and the intersection point is inside the edge limits (if the point is in the segment defined by the edge), a hit point has been found. The user can also decide to ignore “backfaces”, which means that the edges that cannot be seen from the raycast origin won’t be processed. Once obtained all the raycast hit points, they are sorted by distance to the origin in ascending order. A mask can also be set to only check colliders with that mask.

```
std::vector<RaycastHit> CollisionQuerier::RaycastCollider(glm::vec2 origin, glm::vec2 direction,
                        Collider * collider, bool ignoreBackfaces, float distance, Mask mask)
{
    std::vector<RaycastHit> hits;

    _LineImplicitEquation rayEquation = _LineImplicitEquation(origin, direction);

    if (!collider->GetMask().Matches(mask)) return hits;

    std::vector<vec2>* vertices = collider->GetVerticesWorldSpace();
    std::vector<vec2>* normals = collider->GetNormalsWorldSpace();

    for (int i = 0; i < vertices->size(); i++)
    {
        if (ignoreBackfaces && dot(direction, normals->at(i)) >= 0) continue;

        int j = (i + 1 == vertices->size() ? 0 : i + 1);

        vec2 verticeA = vertices->at(i);
        vec2 verticeB = vertices->at(j);

        _LineImplicitEquation segmentEquation =
                _LineImplicitEquation(verticeA, verticeB - verticeA);

        vec2 intersectionPoint;

        if (checkIntersection(rayEquation, segmentEquation, intersectionPoint))
        {
            vec2 originToIntersection = intersectionPoint - origin;
            float distanceToIntersection = length(originToIntersection);

            if (distanceToIntersection < distance)
            {
                if (dot(direction, originToIntersection) > 0)
                {
                    if (pointInSegment(intersectionPoint, verticeA, verticeB))
                    {
                        hits.push_back({ distanceToIntersection,
                                         intersectionPoint,
                                         normals->at(i),
                                         i,
                                         collider });

                        if (ignoreBackfaces || hits.size() == 2)
                        {
                            break;
                        }
                    }
                }
            }
        }
    }

    std::sort(hits.begin(), hits.end(), [](const RaycastHit& lhs, const RaycastHit& rhs) -> bool
    {
        return lhs.distance < rhs.distance;
    });

    return hits;
}
```

Figure 85: CollisionQuerier RaycastCollider method

2.6.3.2 Collision detection

The method used to check collisions is the *CheckCollision* method. This method first does a broad phase detection calling the *checkCollisionBroadPhase* method, and if a collision is detected, it then executes a narrow phase collision detection algorithm.

The *checkCollisionBroadPhase* method checks if the containing circles of the Colliders collide by simply checking the distance between them.

```
bool CollisionQuerier::checkCollisionBroadPhase(Collider * colliderA, Collider * colliderB)
{
    Circle containingCircleA = colliderA->GetContainingCircleWorldSpace();
    Circle containingCircleB = colliderB->GetContainingCircleWorldSpace();

    float collisionDistance = containingCircleA.radius + containingCircleB.radius;
    vec2 AB = containingCircleB.center - containingCircleA.center;

    return dot(AB, AB) < collisionDistance*collisionDistance;
}
```

Figure 86: CollisionQuerier checkCollisionBroadPhase method

The engine implements two different narrow phase collision detection algorithms: SAT and GJK. Even though GJK is more efficient than SAT, the final version of the engine uses SAT since it is more stable and easier to debug.

2.6.3.2.1 Separating Axis Theorem (SAT)

The Separating Axis Theorem is an algorithm used to determine if two convex shapes are colliding. SAT states that: *"If two convex objects are not penetrating, there exists an axis for which the projection of the objects will not overlap."* This means that we only need to find an axis that fullfill that premise to confirm that two convex shapes are NOT colliding.

The method that executes the SAT algorithm is the *checkCollisionSAT* method. This method uses every edge normal of both colliders as an axis, and checks if the projection of the colliders shapes overlap. All axes are looped and the axis with smaller penetration (with smaller projection overlap) is chosen as the separating axis.

```
bool CollisionQuerier::checkCollisionSAT(Collider * colliderA, Collider * colliderB,
                                         std::vector<Contact> & contacts)
{
    std::vector<vec2> axes;

    for (vec2 axis : *colliderA->GetNormalsWorldSpace())
    {
        axes.push_back(axis);
    }

    for (vec2 axis : *colliderB->GetNormalsWorldSpace())
    {
        axes.push_back(axis);
    }

    float minOverlap = std::numeric_limits<float>().max();
    vec2 minOverlapAxis;

    _Projection projectionA, projectionB;
    for (vec2 axis : axes)
    {
        projectionA = colliderA->getProjection(axis);
        projectionB = colliderB->getProjection(axis);

        if (!projectionA.Overlaps(projectionB))//No overlap
        {
            return false;
        }
        else //Overlap
        {
            if (projectionB.GetCenter() > projectionA.GetCenter())
            {
                axis = -axis;
            }

            float overlap = projectionA.GetOverlap(projectionB);
            if (overlap < minOverlap)
            {
                minOverlap = overlap;
                minOverlapAxis = axis;
            }
        }
    }

    //Clipping
    contacts = getContactPoints(colliderA, colliderB, minOverlapAxis);

    return true;
}
```

Figure 87: CollisionQuerier checkCollisionSAT method

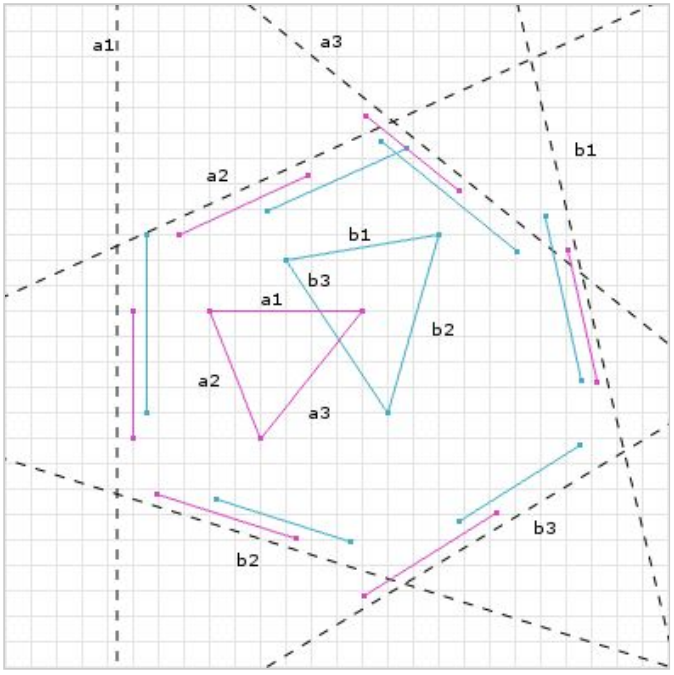

Figure 88: Two convex shapes intersecting. Image from here.

Once the separation axis is obtained it is used to get the contact points using the *getContactPoints* method. This method first obtains the feature edge of each collider with the *getFeatureEdge* method. This method in turn, first obtains the farthest vertex along the inverse normal and then returns either the edge formed by that vertex and the vertex of its right or the one formed by that vertex and the vertex of its left, depending of which edge of the two is more perpendicular to the normal.

Once obtained the feature edges, the *getContactPoints* method checks which of the two edges is the reference edge and which is the incident edge. The reference edge is the one which is more perpendicular to the normal.

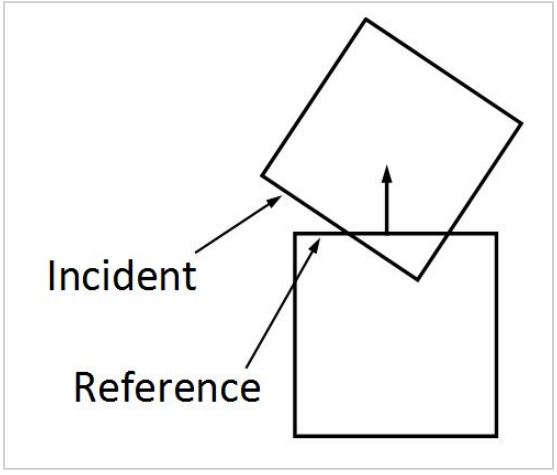

Figure 89: Incident vs Reference edge: Image from here.

Once obtained the reference and incident edges it calls the *clip* method to obtain the collision points.

```
std::vector<Contact> CollisionQuerier::getContactPoints(Collider* colliderA, Collider* colliderB, vec2 normal)
{
    _Edge e1 = getFeatureEdge(colliderA, -normal);
    _Edge e2 = getFeatureEdge(colliderB, normal);
    _Edge incidentEdge, referenceEdge;

    Collider* referenceCollider;

    bool flip = false;

    if (abs(dot(normalize(e1.GetAsVector()), normal)) < abs(dot(normalize(e2.GetAsVector()), normal)))
    {
        referenceEdge = e1;
        incidentEdge = e2;

        referenceCollider = colliderA;

        flip = true;
    }
    else
    {
        referenceEdge = e2;
        incidentEdge = e1;

        referenceCollider = colliderB;
    }

    std::vector<Contact> contacts =  clip(incidentEdge, referenceEdge, referenceCollider, normal, flip);

    return contacts;
}
```

Figure 90: CollisionQuerier getContactPoints method

The *clip* method is in charge of clipping the incident edge (clipped edge) to obtain the final contact points. For each vertex of the incident edge it checks if it is inside the clipping collider. This is done using the *ContainsPoint* method of the Collider class. If it contains the vertex, it is added to the list contact points directly. If it isn't, a raycast is done in the direction of the vertex to the other vertex of the edge. If a raycast hit is obtained, it is the new clipped point and it is added to the list.

Note that the list stores instances of the Contact class, not only positions. So a Contact has to be created using the position of the point. This class also stores the normal and depth of the contact point.

2.6.3.2.2 Gilbert-Johnson-Keerthi algorithm (GJK)

The Gilbert-Johnson-Keerthi algorithm is another algorithm used to determine if two convex shapes are colliding. This algorithm is based on the premise that if two shapes are intersecting, their Minkowski difference will contain the origin. The Minkowski difference of

two shapes is the subtraction of all the points of one shape to all the points of the other shape, that's why if two shapes contains the same point (and thus, are intersecting) their Minkowski difference will contain the origin (a point subtracted by itself is (0,0), the origin).

Since computing the Minkowski difference is a slow process, instead the algorithm tries to iteratively build a polygon inside the Minkowski difference that encloses the origin. If it can be built, it means that the Minkowski difference contains the origin and the shapes are intersecting. This polygon is called simplex.

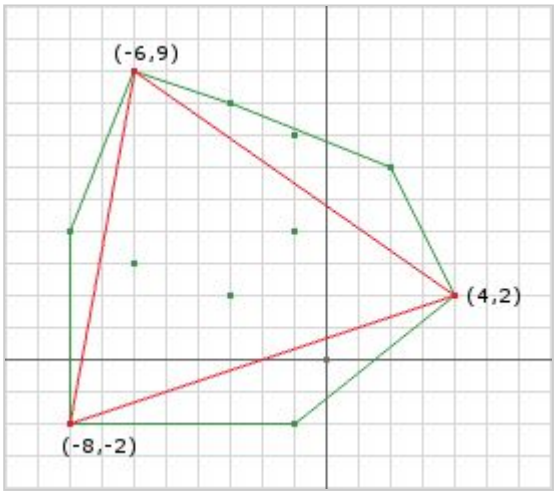

Figure 91: Simplex example. Image from here.

The method that implements the GJK algorithm is the *checkCollisionGJK*. Before understanding this method we first need to understand the *getMinkowskiSupportPoint* and *directionFromTo* method. The *getMinkowskiSupportPoint* simply returns the farthest point of the Minkowski difference of two colliders in a certain direction. On the other hand, if we have two vectors, A and B, the *directionFromTo* method using these two vectors as parameters will return a vector perpendicular to A that points in the direction of B.

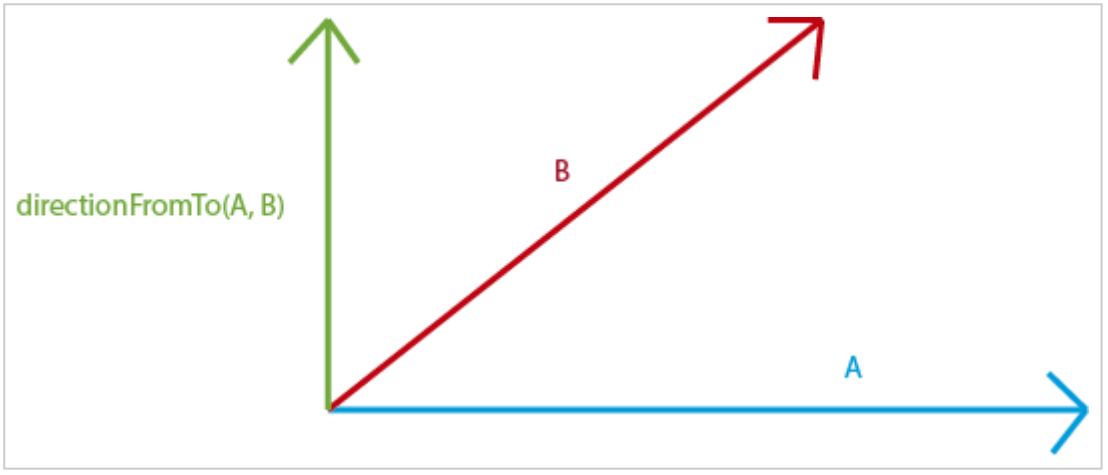

Figure 92: directionFromTo method visual example

The *checkCollisionGJK* uses this methods to try to iteratively build a simplex that encloses the origin. This is done in a loop. Before the loop is executed a point is added to the simplex. This point can be retrieved using *getMinkowskiSupportPoint* in any direction. It also declares three vectors, A, B, and C. A is used to store the last vertex added to the simplex, B to store the previous and C to store the previous of the previous.

```
std::vector<vec2> simplex;
vec2 firstPoint = getMinkowskiSupportPoint(colliderA, colliderB, { 1, 0 });

simplex.push_back(firstPoint);

vec2 direction = -firstPoint;
vec2 nextPoint;

vec2 A, B, C; //A -> last vertex added to simplex, B-> previous vertex added to simplex, -> ...
```

Figure 93: checkCollisionGJK method (first simplex point)

Inside the loop (which loops until the simplex is found), the method first gets a new point using the *getMinkowskiSupportPoint* function, and then it checks if there is no intersection. This is determined by the dot product of the new point with the direction. If it is less than 0 it means that it couldn't get a point that is further along the direction than the origin, so there can not be an intersection. If it is greater than 0, it adds the new point to the simplex and continues with the loop execution.

Then, if the simplex size is two, it simply sets the direction to use to find the next point to the result of *directionFromTo* from AB to AO.

```
if (simplex.size() == 2)
{
    A = simplex[1];
    B = simplex[0];

    vec2 AB = B - A;
    vec2 AO = -A;

    direction = directionFromTo(AB, AO);
}
```

Figure 94: checkCollisionGJK method (simplex of size 2)

If the size of the simplex is three, the process of finding the next direction is more complicated, since it has to check in which of the three possible areas is the origin. This areas are shown in the following figure.

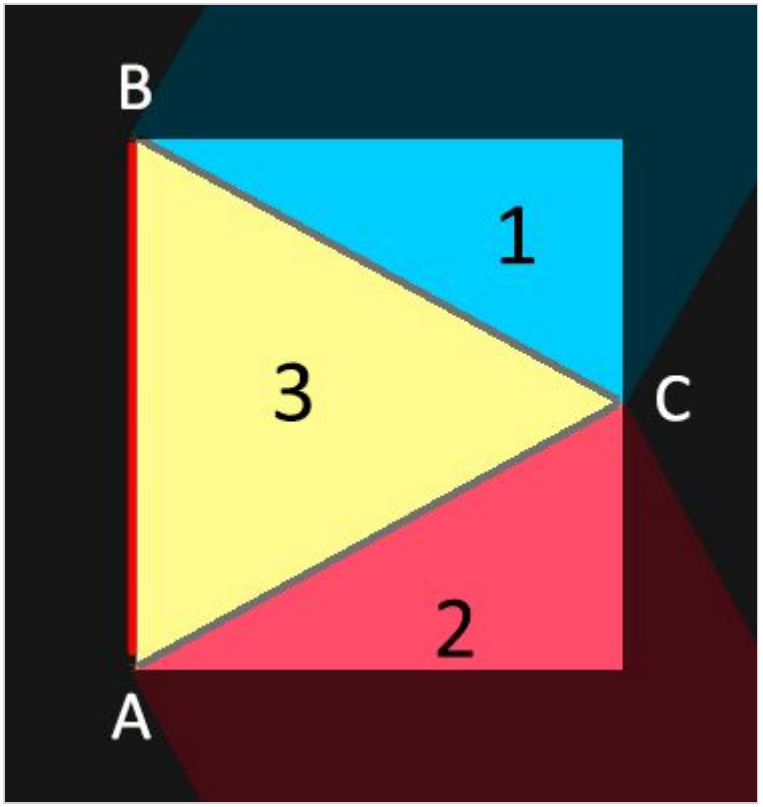

Figure 95: checkCollisionGJK simplex areas (1)

In the previous figure, the red edge represents the simplex in its previous status (when it only had 2 points). The darkened area represents the area where the origin can't be. It can't be at the left of the edge AB because the origin is in the direction of the point C, that's why the next point obtained has been C. It can't be above the point B, under the point A or to the right of the point C because those points are the furthest away in the direction of the origin, so the origin can't be any further.

To find in which area is the origin, a serie of checks are done using the *directionFromTo* method. It the origin is in the area number 3, we have found a simplex that contains the origin, so the shapes are colliding. If it isn't in the area 3, the simplex is updated until it contains the origin.

```
else//Size == 3
{
    A = simplex[2];
    B = simplex[1];
    C = simplex[0];

    vec2 AB = B - A;
    vec2 AC = C - A;
    vec2 AO = -A;

    if (dot(-directionFromTo(AC, AB), AO) > 0)//1
    {
        direction = -directionFromTo(AC, AB);

        //Remove B
        simplex.erase(simplex.begin() + 1);
    }
    else//2,3
    {
        if (dot(-directionFromTo(AB, AC), AO) > 0)//2
        {
            direction = -directionFromTo(AB, AC);

            //Remove C
            simplex.erase(simplex.begin());
        }
        else//3
        {
            //INTERSECTION
            contacts = getContactsEPA(colliderA, colliderB, simplex);

            if (contacts.size() == 0)
            {
                return false;
            }

            return true;
        }
    }
}
```

Figure 96: checkCollisionGJK method (simplex of size 3)

If a collision is found, the contact are retrieves using the *getContactsEPA* method. This method implements the Expanding Polytope Algorithm (EPA). EPA is also an iterative algorithm, it expands the simplex obtained in the GJK algorithm until it hits an edge of the Minkowski difference. The normal of that edge is the normal of the collision.

The *getContactEPA* method simply finds the closest edge of the simplex to the origin, it check if it is an edge of the Minkowski difference by checking if the distance to the edge is the same (with some tolerance) than the distance to the support point of the Minkowski

difference on the direction of the edge normal. If it is, it stores the normal and exits the loops. If it isn't, the support point is added to the simplex and the loop continues.

```
while (true)
{
    _Edge closestEdge = findClosestEdge(simplex);

    vec2 supportPoint = getMinkowskiSupportPoint(colliderA, colliderB, closestEdge.GetNormal());

    double distanceToSupportPoint = dot(supportPoint, closestEdge.GetNormal());

    if (distanceToSupportPoint - distanceToClosestEdge < tolerance)
    {
        normal = -closestEdge.GetNormal();
        break;
    }
    else
    {
        simplex.insert(simplex.begin() + closestEdgeVertexBIndex, supportPoint);
    }
}
```

Figure 97: getContactsEPA loop

Once the normal is obtained, the *getContactPoints* method (which is also used in SAT) is used to retrieve the contact points.

2.6.4 Physics Engine

The _PhysicsEngine class is the one in charge of updating the colliders and invoking the collision events. The collision resolution is done with the help of the _Arbiter class. This class will be explained later, but for now what we need to know is that an arbiter exists for each collision between two colliders, it is in charge of executing the collision resolution and it does it in three phases: a pre step, a step that is executed several times, and a post step.

2.6.4.1 Collision events

As already stated, the collision events are handled by the _PhysicsEngine class. There are four types of events: collision enter, collision exit, trigger enter and trigger exit. The collision enter event is called the first time two colliders collide and none of them is flagged as trigger. The collision exit is called when two colliders that were colliding stop colliding and none of the two is flagged as trigger. The trigger version of the previous events are the same but one or both colliders is flagged as trigger. The events are handled using *createContactEnterEvent*, *createContactExitEvent* and *invokeContactEvents* methods.

The *createContactEnterEvent* and *createContactExitEvent* simply take two colliders and add them to a map (a different map depending on if it is a trigger a collider event and if it is an enter or an exit event), where the key is the collider receiving the event and the value is a ContactInfo struct (which stores the Contact points and the other collider).

```
void _PhysicsEngine::createContactEnterEvent(Collider* colliderA,
                                             Collider* colliderB,
                                             std::vector<Contact> contacts)
{
    std::vector<Contact> inverseNormalContacts = contacts;

    for (int i = 0; i < inverseNormalContacts.size(); ++i)
    {
        inverseNormalContacts[i].normal *= -1;
    }

    if (!colliderA->IsTrigger() && !colliderB->IsTrigger())
    {
        _collisionEnterEvents.push_back(std::pair<Collider*, ContactInfo>
            (colliderA, { contacts, colliderB }));
        _collisionEnterEvents.push_back(std::pair<Collider*, ContactInfo>
            (colliderB, { inverseNormalContacts, colliderA }));

    }
    else
    {
        _triggerEnterEvents.push_back(std::pair<Collider*, ContactInfo>
            (colliderA, { contacts, colliderB }));
        _triggerEnterEvents.push_back(std::pair<Collider*, ContactInfo>
            (colliderB, { inverseNormalContacts, colliderA }));
    }
}

void _PhysicsEngine::createContactExitEvent(Collider * colliderA,
                                            Collider * colliderB)
{
    if (!colliderA->IsTrigger() && !colliderB->IsTrigger())
    {
        _collisionExitEvents.push_back(std::pair<Collider*, ContactInfo>
            (colliderA, { std::vector<Contact>(), colliderB }));
        _collisionExitEvents.push_back(std::pair<Collider*, ContactInfo>
            (colliderB, { std::vector<Contact>(), colliderA }));

    }
    else
    {
        _triggerExitEvents.push_back(std::pair<Collider*, ContactInfo>
            (colliderA, { std::vector<Contact>(), colliderB }));
        _triggerExitEvents.push_back(std::pair<Collider*, ContactInfo>
            (colliderB, { std::vector<Contact>(), colliderA }));
    }
}
```

Figure 98: _PhysicsEngine createContactEnterEvent and createContactExitEvent methods

Then *invokeContactEvents* is used to invoke the events. It simply loops through each map and calls the corresponding method in each one of them.

```
void _PhysicsEngine::invokeContactEvents()
{
    for (std::pair<Collider*, ContactInfo> collisionEvent : _collisionEnterEvents)
    {
        collisionEvent.first->GetGameObject()->OnCollisionEnter(collisionEvent.second);
    }

    for (std::pair<Collider*, ContactInfo> collisionEvent : _collisionExitEvents)
    {
        collisionEvent.first->GetGameObject()->OnCollisionExit(collisionEvent.second);
    }

    for (std::pair<Collider*, ContactInfo> triggerEvent : _triggerEnterEvents)
    {
        triggerEvent.first->GetGameObject()->OnTriggerEnter(triggerEvent.second);
    }

    for (std::pair<Collider*, ContactInfo> triggerEvent : _triggerExitEvents)
    {
        triggerEvent.first->GetGameObject()->OnTriggerExit(triggerEvent.second);
    }

    _collisionEnterEvents.clear();
    _collisionExitEvents.clear();
    _triggerEnterEvents.clear();
    _triggerExitEvents.clear();
}
```

Figure 99: _PhysicsEngine invokeContactEvents method

2.6.4.2 Arbiters update

The physics engine stores a list of all the Colliders of the scene. This list is used to create new arbiters and update the already created ones in the *updateArbitersMethod.* For each collider, it checks for collisions with the rest of colliders using the *CheckCollision* method from the CollisionQuerier class. If a collision is found, if there is already an arbiter that handles the two colliders participating in the collision, it is updated with the new contact points, if there isn't an arbiter, it is created. Arbiters are stored in a map. When a new collision is found (when the arbiter wasn't created) it calls the *createContactEnterEvent* method. When there is no collision between two colliders that had a collision in the previous update, it calls the *createContactExitEvent* and removes their arbiter from the map.

```
void _PhysicsEngine::updateArbiters()
{
    for (int i = 0; i < _colliders.size(); ++i)
    {
        for (int j = i + 1; j < _colliders.size(); ++j)
        {
            Collider* colliderA = _colliders[i];
            Collider* colliderB = _colliders[j];

            if (colliderA->GetRigidbody() == colliderB->GetRigidbody() &&
                colliderA->GetRigidbody() != nullptr)
                continue;

            std::vector<Contact> contacts;

            _ArbiterKey arbiterKey = _ArbiterKey(colliderA, colliderB);

            auto it = _arbiters.find(arbiterKey);

            if (CollisionQuerier::GetInstance()->CheckCollision(colliderA, colliderB, contacts))
            {
                if (it == _arbiters.end())
                {
                    createContactEnterEvent(colliderA, colliderB, contacts);
                    _arbiters.emplace(arbiterKey, _Arbiter(colliderA, colliderB, contacts));
                }
                else
                {
                    it->second.UpdateContacts(contacts);
                }
            }
            else
            {
                if (it != _arbiters.end())
                {
                    createContactExitEvent(colliderA, colliderB);
                    _arbiters.erase(arbiterKey);
                }
            }
        }
    }
}
```

Figure 100: _PhysicsEngine updateArbiters method

2.6.4.3 Physics update

Finally, the physics are updated in the *FixedUpdate* method, which is called every physics update from the Game class. This method first updates the arbiters by calling the *updateArbiters* method. Then it calls the *integrateForces* method of each rigidbody in the scene that has a collider. It then executes the pre-step of each arbiter, the several steps iterations and the post step. It finally calls the *applyVelocities* method of each rigidbody and invokes the contact events.

```
void _PhysicsEngine::FixedUpdate(float fixedTimestep)
{
    //Update arbiters
    updateArbiters();

    //Integrate forces
    for (int i = 0; i < _colliders.size(); ++i)
    {
        Rigidbody* rigidbody = _colliders[i]->GetRigidbody();
        if (rigidbody != nullptr) rigidbody->integrateForces(fixedTimestep);
    }

    //Perform pre-steps
    for (auto it = _arbiters.begin(); it != _arbiters.end(); ++it)
    {
        it->second.PreStep(fixedTimestep);
    }

    //Perform iterations
    for (int i = 0; i < _collisionResolutionSteps; ++i)
    {
        for (auto it = _arbiters.begin(); it != _arbiters.end(); ++it)
        {
            it->second.Step();
        }
    }

    //Perform post-step
    for (auto it = _arbiters.begin(); it != _arbiters.end(); ++it)
    {
        it->second.PostStep();
    }

    //Integrate velocities
    for (int i = 0; i < _colliders.size(); ++i)
    {
        Rigidbody* rigidbody = _colliders[i]->GetRigidbody();
        if (rigidbody != nullptr) rigidbody->applyVelocities(fixedTimestep);
    }

    //Events
    invokeContactEvents();
}
```

Figure 101: _PhysicsEngine FixedUpdate method

2.6.5 Collision resolution

The _Arbiter class is the one in charge of doing the collision resolution. But before understanding how it works, some basic concepts of collision resolution theory must be explained.

A collision occurs when two colliders contact or overlap. When this happens, their linear velocity and angular speed must be updated to simulate a real collision. This can be done using different approaches, but the engine resolves the collision using impulses. When two colliders collide, their mass, moment of inertia, linear velocity, angular speed and the collision normal is used to obtain an impulse *j* that when applied to the colliders will prevent them from penetrating any further.

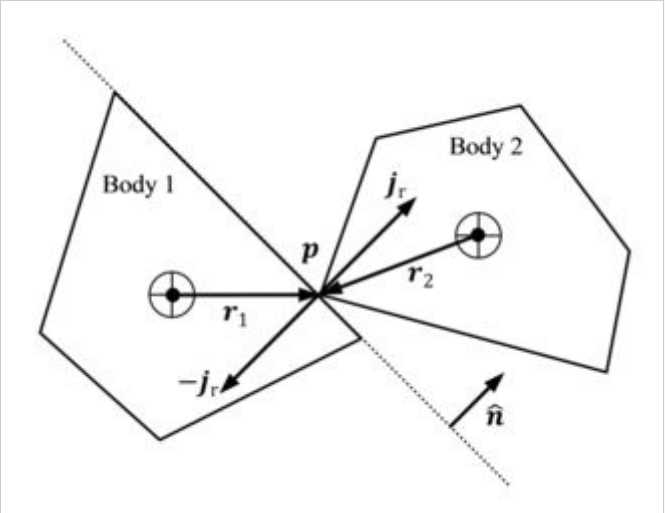

Figure 102: Collision between two shapes. Image from here.

The formula used to obtain the impulse *j* (even though is not exactly the same that the engine uses) is the following one (where *vr* is the relative velocity of the colliders at the point of contact, *m* is the mass, *I* the moment of inertia and *e* is the restitution):

$$j_r = \frac{-(1+e)\mathbf{v}_r \cdot \hat{\mathbf{n}}}{m_1{}^{-1} + m_2{}^{-1} + (\mathbf{I}_1{}^{-1}(\mathbf{r}_1 \times \hat{\mathbf{n}}) \times \mathbf{r}_1 + \mathbf{I}_2{}^{-1}(\mathbf{r}_2 \times \hat{\mathbf{n}}) \times \mathbf{r}_2) \cdot \hat{\mathbf{n}}}$$

Figure 103: J impulse formula. Image from here.

2.6.5.1 Arbiter

The _Arbiter class divides the collision resolution in three methods: a pre step, a step that is executed several times, and a post step. It is heavily based on the sequential impulses collision resolution method, developed by Erin Catto (creator of Box2D). In the original sequential impulses method there is no post step, but the rest of the method is quite similar to what the engine uses.

In the pre step, the denominator of the impulse function is stored in the contact. This is done to avoid recalculating it in each step, since it isn't going to change. A bias is also stored, which depends on the depth of the contact. This bias will be used to apply an extra impulse that will separate colliders that are interpenetrating. The contact restitution impulse is also stored. It will be added in the post step to simulate restitution, and it is calculated using the impulse function and the restitution coefficients of the colliders.

```
void _Arbiter::PreStep(float fixedTimestep)
{
    if (_colliderA->GetRigidbody() == nullptr && _colliderB->GetRigidbody() == nullptr) return;
    if (_colliderA->IsTrigger() || _colliderB->IsTrigger()) return;

    float allowedPenetration = _PhysicsEngine::GetInstance()->_allowedPenetration;
    float biasFactor = _PhysicsEngine::GetInstance()->_penetrationBiasFactor;

    for (int i = 0; i < _contacts.size(); ++i)
    {
        Contact* contact = &_contacts[i];

        vec3 normal = vec3(contact->normal, 0);
        vec3 tangent = vec3(normal.y, -normal.x, 0);

        vec3 rA = vec3(contact->position, 0) - _colliderA->GetGameObject()->GetPosition();
        vec3 rB = vec3(contact->position, 0) - _colliderB->GetGameObject()->GetPosition();

        contact->massNormal = 1 / (((_iMassA + _iMassB) + dot(cross(cross(rA, normal), rA) * _iInertiaA +
                              cross(cross(rB, normal), rB) * _iInertiaB, normal)) * _contacts.size());
        contact->massTangent = 1 / (((_iMassA + _iMassB) + dot(cross(cross(rA, tangent), rA) * _iInertiaA +
                              cross(cross(rB, tangent), rB) * _iInertiaB, tangent)) * _contacts.size());

        contact->bias = biasFactor * max(0.0f, contact->depth - allowedPenetration) / fixedTimestep;

        //Restitution
        float e = min(_colliderA->GetPhysicsMaterial().restitution, _colliderB->GetPhysicsMaterial().restitution);

        if (e == 0)
        {
            contact->restitutionImpulse = vec2();
        }
        else
        {
            vec3 vAP = _rigidbodyA == nullptr ? vec3() : vec3(_rigidbodyA->GetLinearVelocity(), 0);
            vec3 vBP = _rigidbodyB == nullptr ? vec3() : vec3(_rigidbodyB->GetLinearVelocity(), 0);
            vec3 vR = vBP - vAP;

            if (dot(vR, normal) < 0)
            {
                continue;
            }

            contact->restitutionImpulse = e * (dot(vR, normal) * contact->massNormal) * (vec2)normal;
        }
    }
}
```

Figure 104: _Arbiter PreStep method

The step is executed several times in order to get to a more accurate solution. It obtains the normal and tangent *j* impulse. To calculate the normal *j*, it first multiplies the denominator of the impulse function (which has been stored in the pre step) by the relative velocity in the direction of the normal plus the bias of the contact. To calculate the tangent *j*, it multiplies the denominator of the impulse function by the relative velocity in the direction of the tangent, and then the friction is used to clamp the impulse. Once both impulses are calculated, the linear velocity and angular speed of the rigidbodies of the colliders are modified using these values.

```
//NORMAL
float vn = dot(vR, normal);
float dJn = (vn + contact->bias) * contact->massNormal;

//Clamp
float jn0 = contact->jn;
contact->jn = max(jn0 + dJn, 0.0f);
dJn = contact->jn - jn0;

vec2 jn = dJn * vec2(normal);

//Apply
if (_rigidbodyA != nullptr)
{
    _rigidbodyA->SetLinearVelocity(_rigidbodyA->GetLinearVelocity() + jn * _iMassA);
    _rigidbodyA->SetAngularSpeed(_rigidbodyA->GetAngularSpeed() - cross(rA, vec3(jn, 0)).z * _iInertiaA);
}
if (_rigidbodyB != nullptr)
{
    _rigidbodyB->SetLinearVelocity(_rigidbodyB->GetLinearVelocity() - jn * _iMassB);
    _rigidbodyB->SetAngularSpeed(_rigidbodyB->GetAngularSpeed() + cross(rB, vec3(jn, 0)).z * _iInertiaB);
}


//TANGENT
float vt = dot(vR, tangent);
float dJt = vt * contact->massTangent;

//Clamp
float maxJt = _friction * contact->jn;
float jt0 = contact->jt;
contact->jt = clamp(jt0 + dJt, -maxJt, maxJt);
dJt = contact->jt - jt0;

vec2 jt = dJt * vec2(tangent);

//Apply
if (_rigidbodyA != nullptr)
{
    _rigidbodyA->SetLinearVelocity(_rigidbodyA->GetLinearVelocity() + jt * _iMassA);
    _rigidbodyA->SetAngularSpeed(_rigidbodyA->GetAngularSpeed() - cross(rA, vec3(jt, 0)).z * _iInertiaA);
}
if (_rigidbodyB != nullptr)
{
    _rigidbodyB->SetLinearVelocity(_rigidbodyB->GetLinearVelocity() - jt * _iMassB);
    _rigidbodyB->SetAngularSpeed(_rigidbodyB->GetAngularSpeed() + cross(rB, vec3(jt, 0)).z * _iInertiaB);
}
```

Figure 105: _Arbiter Step method, normal and tangent impulse calculation

Finally, the post step simply adds to the rigidbody the linear velocity and the angular speed obtained using the restitution impulse.

```
void _Arbiter::PostStep()//Restitution
{
    if (_colliderA->GetRigidbody() == nullptr && _colliderB->GetRigidbody() == nullptr) return;
    if (_colliderA->IsTrigger() || _colliderB->IsTrigger()) return;

    for (int i = 0; i < _contacts.size(); ++i)
    {
        Contact* contact = &_contacts[i];

        vec3 rA = vec3(contact->position, 0) - _colliderA->GetGameObject()->GetPosition();
        vec3 rB = vec3(contact->position, 0) - _colliderB->GetGameObject()->GetPosition();

        //Apply restitution
        if (_rigidbodyA != nullptr)
        {
            _rigidbodyA->SetLinearVelocity(_rigidbodyA->GetLinearVelocity() +
                contact->restitutionImpulse * _iMassA);
            _rigidbodyA->SetAngularSpeed(_rigidbodyA->GetAngularSpeed() -
                cross(rA, vec3(contact->restitutionImpulse, 0)).z * _iInertiaA);
        }
        if (_rigidbodyB != nullptr)
        {
            _rigidbodyB->SetLinearVelocity(_rigidbodyB->GetLinearVelocity() -
                contact->restitutionImpulse * _iMassB);
            _rigidbodyB->SetAngularSpeed(_rigidbodyB->GetAngularSpeed() +
                cross(rB, vec3(contact->restitutionImpulse, 0)).z * _iInertiaB);
        }
    }
}
```

Figure 106: _Arbiter PostStep method

2.7 The Game class

The Game class acts like a hub between the different modules of the engine and the user side and it also initializes, updates, and terminates them.

2.7.1 Initializing the engine

The Game class initializes the engine modules using the values obtained from the configuration file. This file looks like this:

```
#SCENE
firstScene = CollisionTestScene

#DISPLAY
referenceWindowWidth = 1024
referenceWindowHeight = 624
windowWidth = 1024
windowHeight = 624
screenFitMode = widthFit
windowMode = windowed
windowTitle = test
textureFilteringMode = nearest
maxCameraDepth = 1000

#TEXT
fontSize = 100

#PHYSICS
gravityX = 0
gravityY = -1000
fixedTimestep = 0.005
collisionResolutionSteps = 10
allowedPenetration = 0.1
penetrationBiasFactor = 0.1
minRigidbodySpeed = 5

#SOUND
minAttenuationDistance = 256
maxAttenuationDistance = 512
minPanningDistance = 256
maxPanningDistance = 512
masterGain = 0.2
```

Figure 107: Configuration file

To read the values of this file the game uses the *parseConfigurationFile* method. This method simply reads line by line the file and sets the values readed into the respective variables.

```
void Game::parseConfigurationFile(std::string  fileName)
{
    std::string line;
    std::ifstream configurationFile(fileName);

    if (!configurationFile.good())
    {
        printf("\nWARNING: Game::parseConfigurationFile -> Can't load configuration file");
        return;
    }

    printf("\n\nConfiguration file loaded correctly\n-----------------------------------");

    getline(configurationFile, line);
    while (!configurationFile.eof())
    {
        if (line[0] != '#' && line.size() > 0)
        {
            line.erase(std::remove_if(line.begin(), line.end(), ::isspace), line.end());

            int pos = (int)line.find("=", 0);
            std::string variable = line.substr(0, pos);
            std::string value = line.substr(pos + 1, line.size() - pos - 1);

            if (variable == "windowWidth")
            {
                _windowWidth = atoi(value.c_str());

                printf("Window width: %i\n", _windowWidth);
            }
            else if (variable == "windowHeight")
            {
                _windowHeight = atoi(value.c_str());

                printf("Window heigth: %i\n", _windowHeight);
            }


...
```

Figure 108: Fragment of the Game parseConfigurationFile method

These variables are then used in the *Initialize* method to initialize the respective modules. This method is in charge of initializing everything that is needed for the execution of the game. It seeds the random number generator, calls the *parseConfigurationFile* method, initializes the GLFW library, creates the window and sets its parameters (size, title and window mode), initializes the rendering engine, sets the camera parameters, sets the base font size of the _FontLoader class, sets the parameters of the audio and physics engine, sets

the callbacks that will be called by the InputManager, sets the first scene (using the *setScene* method) and then starts the game loop.

2.7.2 Input callbacks

The methods that are called back by the _InputManager class are the *onButtonTrigger*, *onControllerButtonTrigger*, *onMouseButtonTrigger* and *onMouseMoveTrigger.* These methods simply call the respective scene event methods.

```
void Game::onButtonTrigger(int button, int action)
{
    if (action == GLFW_PRESS)
    {
        _currentScene->OnButtonDown(button);
    }
    else if (action == GLFW_RELEASE)
    {
        _currentScene->OnButtonUp(button);
    }
}

void Game::onControllerButtonTrigger(int controller, int button, int action)
{
    if (action == GLFW_PRESS)
    {
        _currentScene->OnControllerButtonDown(controller, button);
    }
    else if (action == GLFW_RELEASE)
    {
        _currentScene->OnControllerButtonUp(controller, button);
    }
}

void Game::onMouseButtonTrigger(int button, int action, double x, double y)
{
    glm::vec2 worldPosition = Camera::GetActiveCamera()->ScreenToWorldPosition({ x, y });

    if (action == GLFW_PRESS)
    {
        _currentScene->OnMouseButtonDown(button, { x, y }, worldPosition);
    }
    else if (action == GLFW_RELEASE)
    {
        _currentScene->OnMouseButtonUp(button, { x, y }, worldPosition);
    }
}

void Game::onMouseMoveTrigger(double x, double y)
{
    glm::vec2 worldPosition = Camera::GetActiveCamera()->ScreenToWorldPosition({ x, y });
    _currentScene->OnMouseMove({ x, y }, worldPosition);
}
```

Figure 109: Input callbacks

2.7.3 Scenes

When the user wants to change the scene, it calls the *ChaangeToScene* method from the Scene class, which in turn calls the *ChangeToScene* method from the Game clas. This last method simply stores the next scene name in a variable and sets a flag that indicates that the scene must be changed. At the start of the next frame, the *setScene* method is called. This method gets the next scene using the SceneManager, terminates the current scene calling the *Terminate* method of the scene, clears the engine modules using the *clear* method and initializes the next scene using the *Initialize* method of the scene.

```
void Game::setScene(std::string  scene)
{
    printf("\nLoad scene: %s", scene.c_str());

    Scene* nextScene = SceneManager::GetInstance()->GetScene(scene);

    if (nextScene == nullptr)
    {
        printf("\nWARNING: Game::setScene -> Scene not valid");
        return;
    }

    if (_currentScene != nullptr)
    {
        _currentScene->Terminate();
    }

    clear();

    _currentScene = nextScene;
    _currentScene->Initialize();
    _changeScene = false;
}

void Game::ChangeToScene(std::string  scene)
{
    _changeScene = true;
    _nextScene = scene;
}
```

Figure 110: Game setScene and ChangeToScene methods

The *clear* method simply calls the *Clear* method of the rendering, audio and physics engine, and the _FontLoader, SpriteSheetManager and Drawer. These methods in turn, delete all the resources allocated by those classes and clear their lists and maps.

2.7.4 The game loop

The game loop is executed until the *Terminate* method of the Game class is called (which sets the _terminate variable to true) or the game window is closed. In the loop, first, the scene is changed if the _changeScene flag is set to true. Then, the delta time (the time elapsed since the previous loop) is calculated. Once this is done, the _InputManager is updated, then the physics are updated, the *Update* method of the scene is called and finally the rendering engine is updated.

Since the physics are updated in a fixed timestep, they can be updated more than once in a game loop, or even not be updated in several loops. Every loop, the delta time will be added to the timeAccumulator variable. Then, in the physics update, while the accumulated time is greater than the delta time, the physics are updated (first the *FixedUpdate* method of the current scene is called and then the physics engine is updated) and the delta time is subtracted from the accumulated time.

```
while (!_terminate && !_window->ShouldClose())
{
    if (_currentScene == nullptr) continue;
    if (_changeScene) setScene(_nextScene);

    double currentTime = Timer::GetInstance()->GetTime();
    double deltaTime = currentTime - previousTime;
    previousTime = currentTime;

    timeAccumulator += deltaTime;

    _inputManager->Update();

    while (timeAccumulator >= _fixedTimestep)
    {
        _currentScene->FixedUpdate(_fixedTimestep);

        _physicsEngine->FixedUpdate(_fixedTimestep);

        timeAccumulator -= _fixedTimestep;
    }

    _currentScene->Update(deltaTime);

    _renderingEngine->Update();

    glfwPollEvents();
}
```

Figure 111: The game loop

2.8 Utility classes

Aside from the different modules, the engine also includes a serie of utility classes that help the user to do certain tasks.

2.8.1 Random

The Random class is a static class which allows the user to obtain random numbers. The *Seed* method has to be called to seed the random number generator. This is done from the *Initialize* method of the Game class.

```
class Random
{
    public:
        static void Seed()
        {
            srand(std::chrono::high_resolution_clock::now().time_since_epoch().count());
        }

        static float Range(float minValue, float maxValue)
        {
             return Range01() * (maxValue - minValue) + minValue;
        }

        static int Range(int minValue, int maxValue)
        {
            return (int)Range((float)minValue, (float)maxValue + 1.0f - std::numeric_limits<float>::min());
        }

        static float Range01()
        {
            return static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
        }
};
```

Figure 112: The Random class

The class has three different methods to obtain a random number. One to obtain a random integer between two values, another to obtain a random float between two numbers and the last one to obtain a float between 0 and 1 (both included).

2.8.2 Timer

The Timer class is a singleton that simply acts as a wrapper around the *glfwGetTime* method from the GLFW library. It also allows the user to set a time scale, which will scale the time obtained by the value defined. For instance, if the time scale is 0.5 and the time elapsed

since the start of the program (the value returned by the *glfwGetTime* method) is 10 seconds, the method *GetTime* will return 5 seconds. This method is the one used by the Game class to obtain the delta time, so changing the time scale will affect to everything that depends on it. If the user wants to get the real time, it can be retrieved using the *GetTimeUnscaled* method.

```
double Timer::GetTime()
{
    return glfwGetTime() * _timeScale;
}

double Timer::GetTimeUnscaled()
{
    return glfwGetTime();
}

double Timer::GetTimeScale()
{
    return _timeScale;
}

void Timer::SetTimeScale(double timeScale)
{
    _timeScale = timeScale;
}
```

Figure 113: The Timer public methods

2.8.3 Masks

Masks are used to group elements in a logical way. This can be used when raycasting, to avoid raycasting with colliders of certain masks, or to only raycast colliders of a certain mask. But since masks can have a general purpose, the Mask class and the MaskManager class have been defined outside the physics module.

Mask store an integer (a long variable in this case) (called bitmask) whose bits are used to make the comparisions. When two masks are “equal” we say they match, but the truth is they don’t need to be equal to match. For two masks, if one of their non-zero bits are equal, they match. To do so, the AND logical operator is used. This is done in the *Matches* method of the Mask class. Two masks can be added using the OR logical operator. If a mask C is a combination of mask A and mask B, C matches both A and B, even if A doesn’t match B. This is done by overloading the + operator.

```
Mask Mask::operator+(const Mask & otherMask)
{
    Mask mask = Mask(_bits | otherMask._bits);

    return mask;
}

Mask Mask::operator+=(const Mask & otherMask)
{
    _bits = _bits | otherMask._bits;
    return *this;
}

bool Mask::Matches(Mask otherMask)
{
    return (_bits & otherMask._bits);
}
```

Figure 114: Mask class Matches method and + operator overload

The MaskManager is in charge of creating and managing the masks. The *CreateMask* method takes the mask string, creates a new mask and stores it in a map with the name as key. The *GetMask* method returns a mask that has already been created using the mask map.

```
void MaskManager::CreateMask(std::string name)
{
    if (_currentMaskIndex <= MAX_MASKS)
    {
        Mask mask = Mask(glm::pow(2, _currentMaskIndex));
        _currentMaskIndex++;

        _maskMap[name] = mask;
    }
    else
    {
        printf("\nWARNING: MaskManager::CreateMask() -> Cannot create another mask, mask limit reached");
    }
}

Mask MaskManager::GetMask(std::string name)
{
    std::map<std::string, Mask>::iterator it = _maskMap.find(name);
    if (it != _maskMap.end())
    {
        return  it->second;
    }

    printf("\nWARNING: MaskManager::GetMask -> _maskMap doesn't contain the mask named: %s", name.c_str());

    return Mask();
}
```

Figure 115: MaskManager class CreateMask and GetMask method

2.8.4 State Machine

The engine also defines a state machine abstract class that can be used by the user to create its own state machines. A state machine is formed by a serie of state nodes, wich in turn have an state and a serie of conditions that are used to check if the current state node has to be updated.

The State class is an interface class that declares the basic methods of a state: *OnEnterState,* which is called when the state starts, *Update*, which is called each game loop and *OnExitState*, which is called when the current state is changed.

```
class __declspec(dllexport) State
{
    public:
        virtual void OnEnterState() = 0;
        virtual void Update(double deltaTime) = 0;
        virtual void OnExitState() = 0;
};
```

Figure 116: State class

The struct StateNode simply stores a state and a map of StateNode and Condition (the first one as key and the second as value). A Condition is simply a function that returns a boolean and takes no parameters.

```
using Condition = std::function<bool()>;

struct StateNode
{
    State* state;
    std::map<StateNode*, Condition> childNodes;
};
```

Figure 117: StateNode struct and Condition definition

The state machine simply stores the current StateNode and checks if any of its Condition returns true. If it happens, the current StateNode is changed to the one that the Condition is related to.

```
void StateMachine::changeCurrentStateNode(StateNode * stateNode)
{
    if (_currentStateNode != nullptr)
    {
        _currentStateNode->state->OnExitState();
    }

    _currentStateNode = stateNode;

    _currentStateNode->state->OnEnterState();
}


void StateMachine::Update(double deltaTime)
{
    _currentStateNode->state->Update(deltaTime);

    for (auto it = _currentStateNode->childNodes.begin(); it != _currentStateNode->childNodes.end(); ++it)
    {
        if (it->second())//If condition == true
        {
            changeCurrentStateNode(it->first);
            break;
        }
    }
}
```

Figure 118: StateMachine methods

The StateMachine class inherits from Component, so it has the *Update* event method. Every update, it calls the *Update* method of the State of the current StateNode and checks is a Condition is met. If it happens, the *OnExitState* method of the State of the current StateNode is called, the current StateNode is set to the new one and the *OnEnterState* method of its State is called.

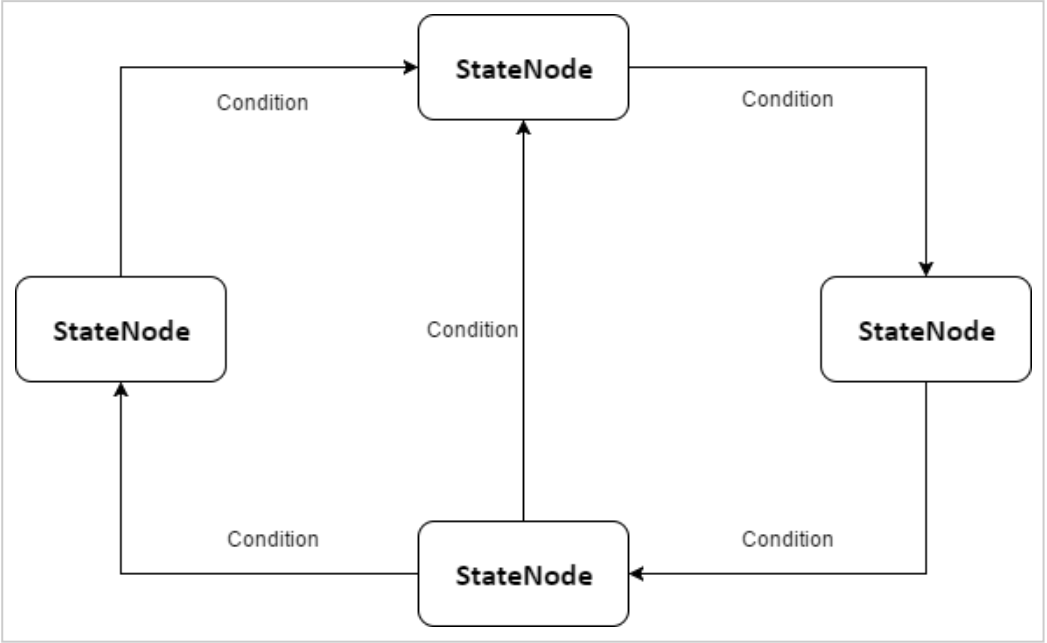

Figure 119: State machine example scheme

3. Final results

To test the engine and prove that it can be used to create games, a serie of demos and a game prototype have been developed using the engine. Video

3.1 Demos

The demos have been divided in three types: graphics, audio and physics. There are three demos of graphics and two demos of audio and physics.

The main objective of the graphics demos is to show the most important capabilities of the rendering engine: efficiency, animated sprites, use of sprite sheets, transparency and parallax.

The first demo renders 12.000 animated sprites at a resolution of 1024x720 and shows the number of frames per second. The fps will vary depending of the computer, but in my computer (3.2Ghz i7 processor and Nvidia GeForce GTX 770m of 2GB) it renders at approximately 60fps. Its objective is to show the efficiency of the engine and the use of sprite sheets and animated sprites.

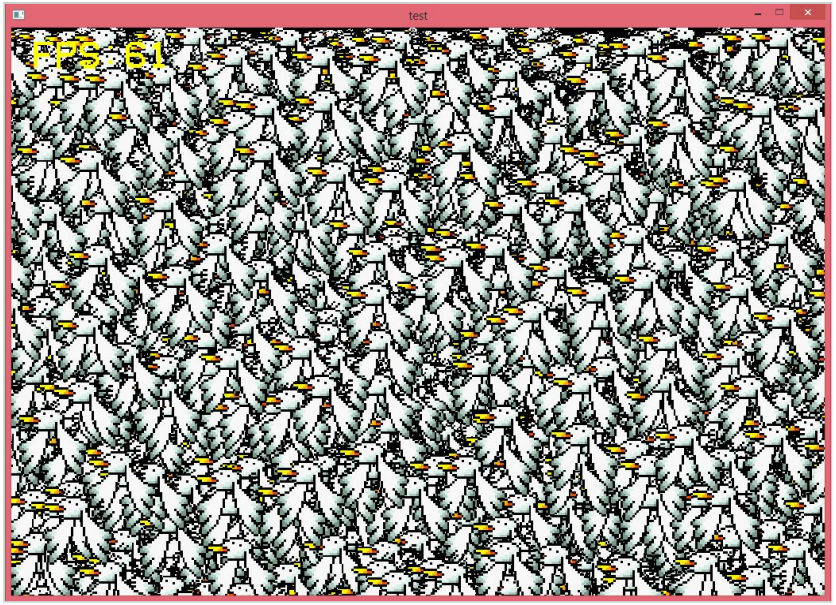

Figure 120: Graphics demo 1

The second demo renders a serie of transparent circles with parallax effect. Its objective is to show the parallax effect and the sprite transparency.

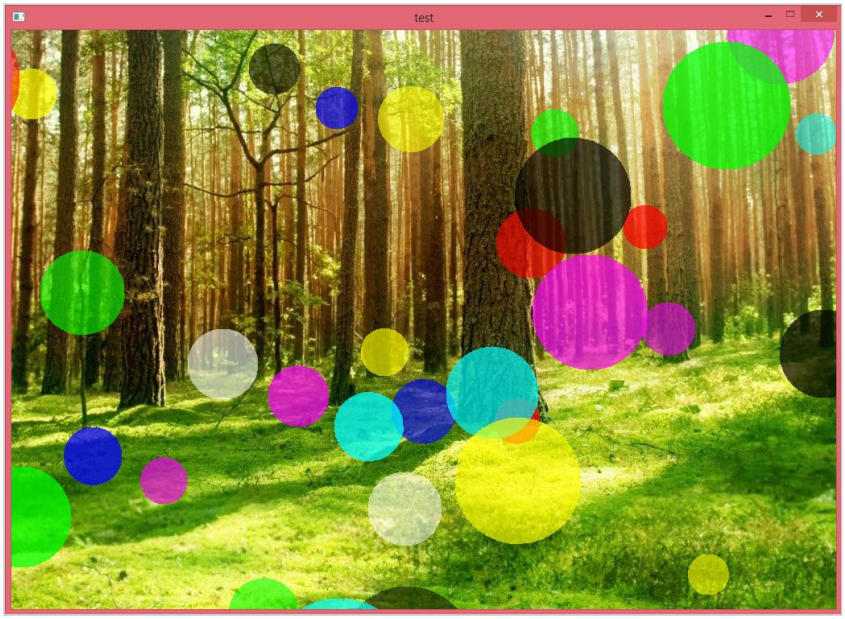

Figure 121: Graphics demo 2

The third demo simply renders a text in the screen and allows the user to change its horizontal and vertical alignment.

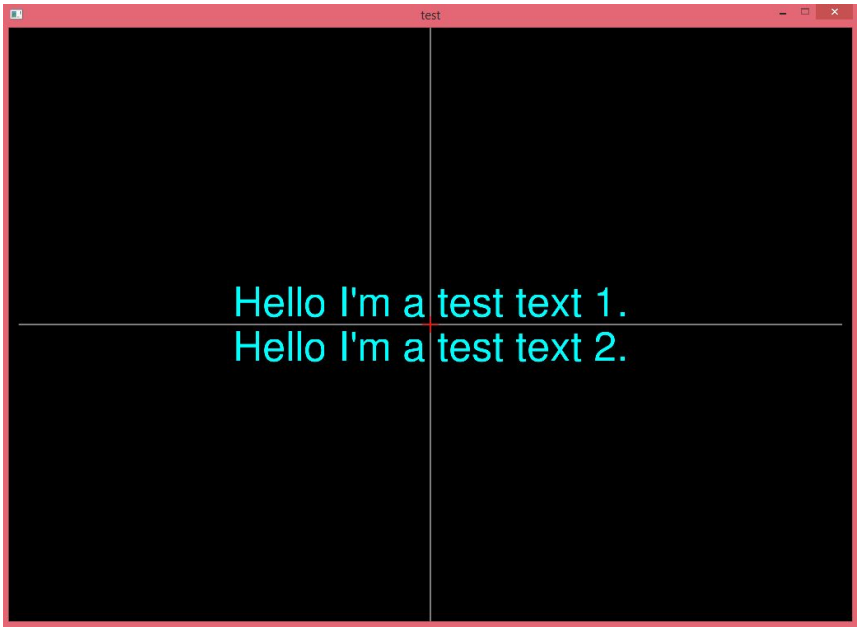

Figure 122: Graphics demo 3

The objective of the audio demos, in turn, is to show how the spatial audio works and also to help understand the concept of audio source and audio listener and the attenuation and panning effects.

The first demo simply has an audio source (represented by a blue circle) that can be moved using the mouse and a serie of circles and lines that represent the min and max attenuation and panning distances. When the user moves the audio source, it gets attenuated and panned depending on the position

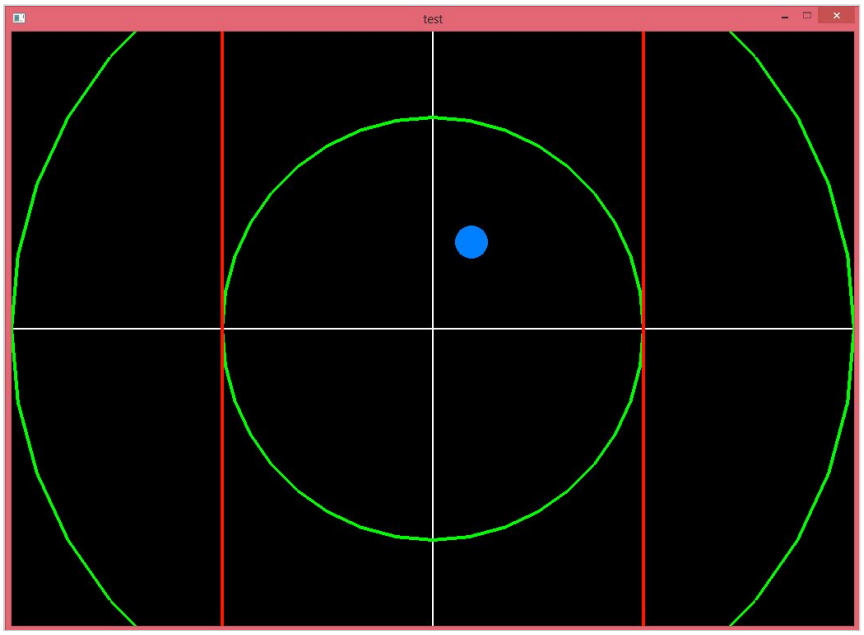

Figure 123: Audio demo 1

The second demo has two different audio sources (represented by a blue and an orange circle) and an audio listener attached to the camera, which can be moved using the keyboard. It also have the circles and lines representing the distances. When the user moves the listener the audios of the audio sources are attenuated and panned depending on the listener position.

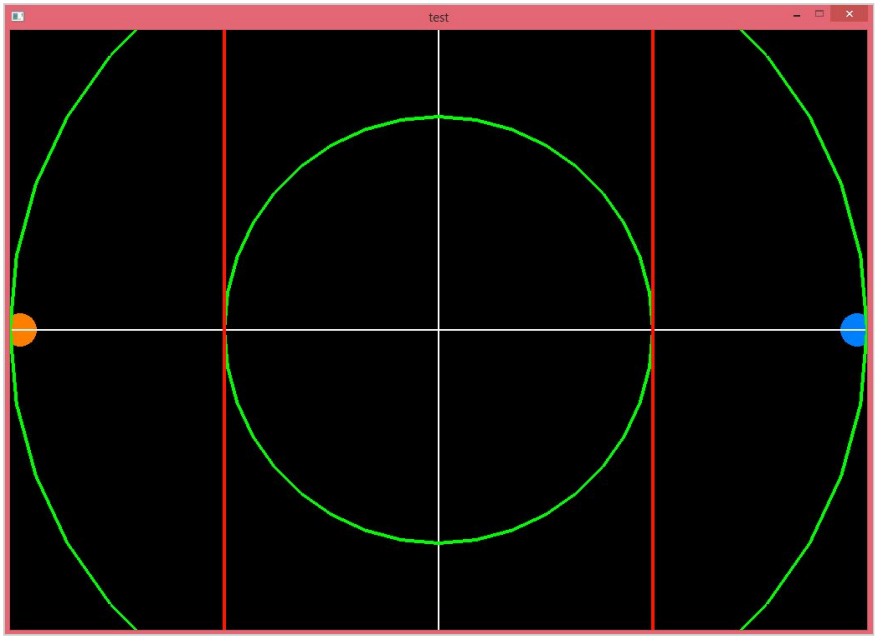

Figure 124: Audio demo 2

Finally, the main objective of the physics demos is to show the collision detection, collision resolution, rigid body dynamics, the use of physics materials, raycasts, etc.

The first demo allows the user to create colliders of X sides (from 3 to whatever the user chooses), apply impulses to them using raycasts and instance projectiles to make them collide with the colliders. The colliders can also have different mass to allow the user to see how the projectiles and impulses interact with colliders of different mass.

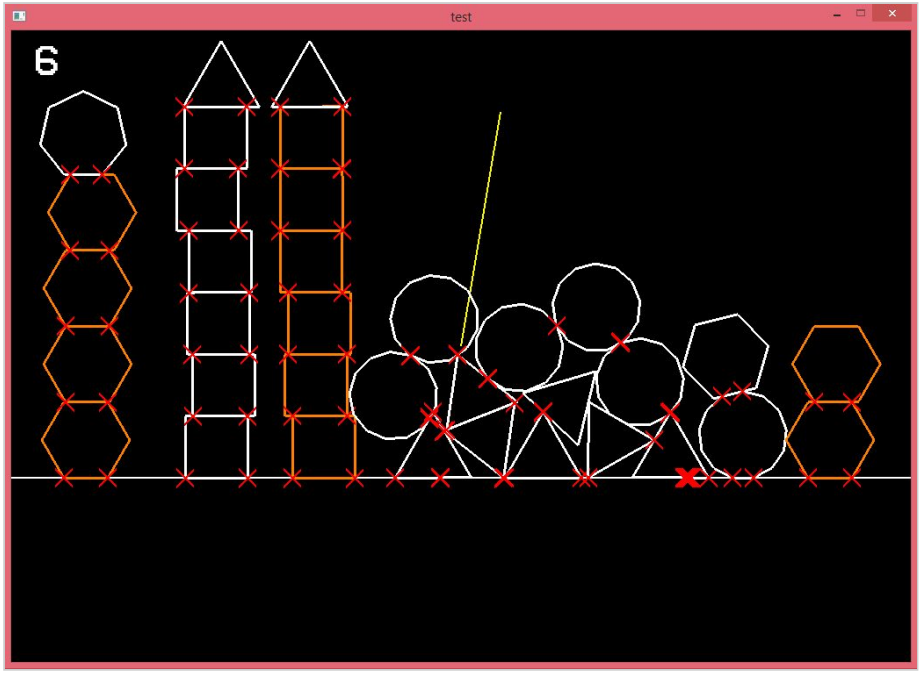

Figure 125: Physics demo 1.

The second demo allows the user to instance colliders with different physic materials. Its objective is to show how colliders with different physic materials react to collisions.

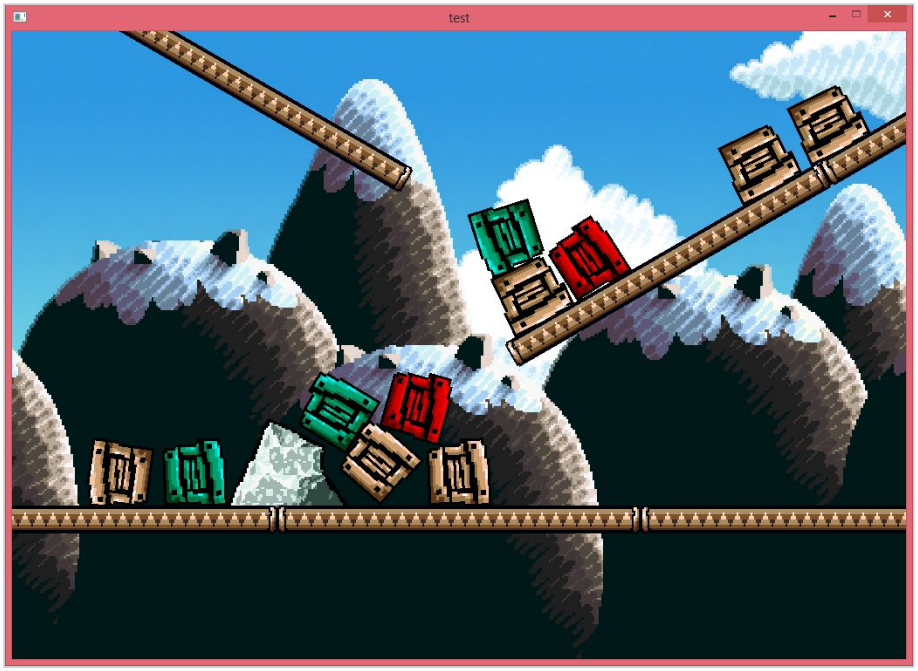

Figure 126: Physics demo 2

3.2 Game prototype

Along with the demos, a simple game prototype has been developed from scratch using the engine. It is a simple 2D lateral game where the player takes control of a character that can move, jump and interact with some elements of the game. It combines every feature of the engine and shows that it can be used to make relatively complex games. Video

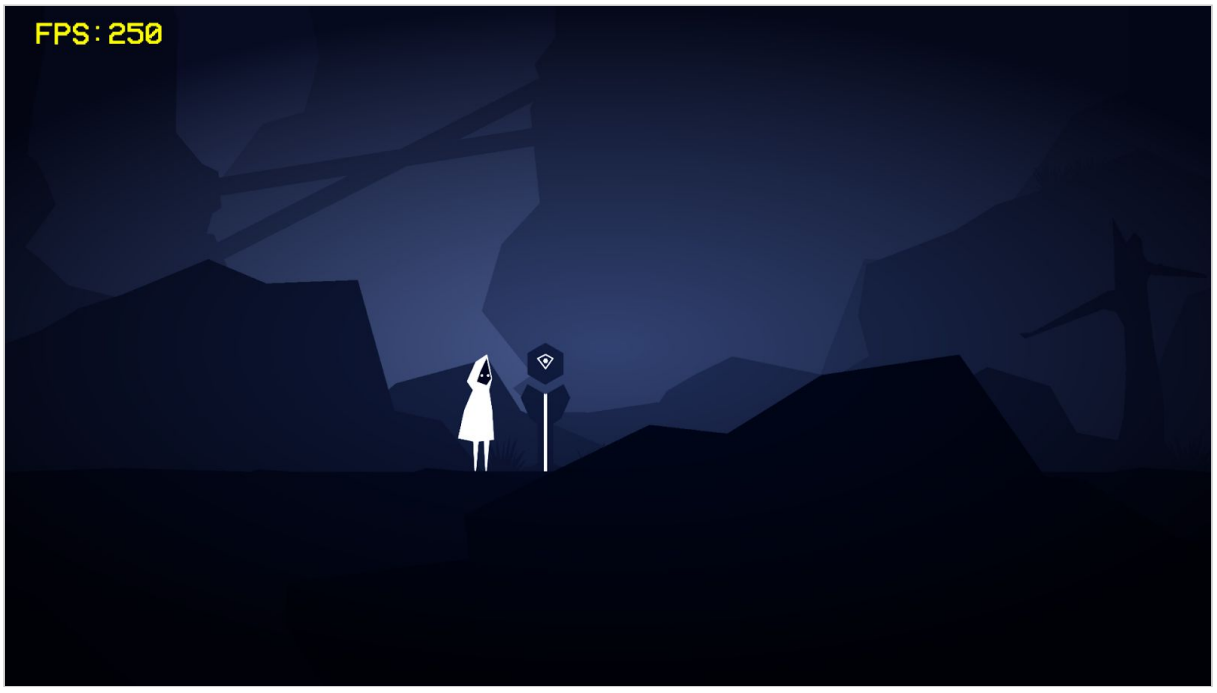

Figure 127: Game prototype

4. Conclusions

This project was started with one objective in mind, to learn about how game engines work internally. The main personal reason for this was that, to me, it feels bad to use an engine without knowing what it is really doing internally. And finally, after several month of development, this objective has been successfully accomplished. I've not only learned how an engine works at its low level but I've also increased my overall knowledge about graphics, audio and physics programming and also about code structuring and design patterns. I've also increased my knowledge about the C++ programming language, which is, in my opinion, the most powerful language nowadays.

To be honest, when I started this project I wasn't expecting to learn that much, and also I wasn't sure if I was going to be able to accomplish it, since creating a game engine seemed very complex and I wasn't sure if I could do it by myself. Furthermore, the results obtained have also been much better than expected. Even though the engine isn't perfect and could be improved greatly, it is more powerful that I thought it would be.

On the other hand, I would have liked to include more features, like a scene editor (similar to the Unity editor, but obviously much simpler), but it would have required probably more than 100h to get it working, since it is very complex, and I didn't have time to do that. I also would have liked to include post processing, 2D lighting, audio effects, particle systems, etc, but I also didn't have enough time to do that.

In conclusion, I'm quite happy with the results and I personally think that more students should do similar projects and go beyond simply doing videogames using a pre existing engine. It is a very interesting learning experience and you will obtain knowledge that could be used in future projects.

I've made public the source code of the engine here.

5. Bibliography

Rendering:

http://ogldev.atspace.co.uk/www/tutorial33/tutorial33.html
https://www.khronos.org/opengl/wiki/Blending#Source.2C_Destination.2C_and_the_buffer
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/
https://open.gl/textures
https://r3dux.org/2010/11/2d-c-openglglfw-basecode/
https://learnopengl.com/#!Getting-started/Camera
http://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/orthographic-projection-matrix
https://learnopengl.com/#!Advanced-OpenGL/Advanced-GLSL
https://learnopengl.com/#!In-Practice/Debugging
https://www.khronos.org/opengl/wiki/Transparency_Sorting
https://www.freetype.org/
https://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Text_Rendering_01
https://www.freetype.org/freetype2/docs/tutorial/step1.html

Audio:

http://www.topherlee.com/software/pcm-tut-wavformat.html
http://www.dunsanyinteractive.com/blogs/oliver/?p=72
https://openal.org/documentation/openal-1.1-specification.pdf
https://openal.org/documentation/OpenAL_Programmers_Guide.pdf
https://ffainelli.github.io/openal-example/

Physics:

https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331
http://www.gotoandplay.it/_articles/2005/08/advCharPhysics.php
http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_GJK_notes.pdf
https://mollyrocket.com/849
http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_GJK_notes.pdf
http://www.dyn4j.org/2011/11/contact-points-using-clipping/
https://code.google.com/archive/p/box2d/downloads
http://chrishecker.com/images/e/e7/Gdmphys3.pdf
http://gafferongames.com/virtual-go/collision-response-and-coulomb-friction/

https://en.wikipedia.org/wiki/Collision_response#Impulse-Based_Reaction_Model
http://www.wyrmtale.com/blog/2013/115/2d-line-intersection-in-c
http://allenchou.net/game-physics-series/
http://www.dyn4j.org/2010/04/gjk-distance-closest-points/
http://twvideo01.ubm-us.net/o1/vault/gdc09/slides/04-GDC09_Catto_Erin_Solver.pdf
http://www.dyn4j.org/2010/05/epa-expanding-polytope-algorithm/
http://www.dyn4j.org/2010/01/sat/