



UNIVERSITY OF PIRAEUS – DEPARTMENT OF INFORMATICS
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc “Informatics”
ΠΜΣ «Πληροφορική»

MSc Thesis
Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	“Wand: Design and Development of a Game Engine for Visual Novels” «Wand: Σχεδίαση και Ανάπτυξη μιας Μηχανής Παιχνιδιών για Οπτικά Μυθιστορήματα»
Student’s name-surname: Ονοματεπώνυμο φοιτητή:	Maria Violaki Μαρία Βιολάκη
Father’s name: Πατρώνυμο:	Christos Χρήστος
Student’s ID No: Αριθμός Μητρώου:	ΜΠΠΛ19007
Supervisor: Επιβλέπων:	Themistoklis Panagiotopoulos, Professor Θεμιστοκλής Παναγιωτόπουλος, Καθηγητής

July 2022 / Ιούλιος 2022

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

**Themistoklis Panagiotopoulos,
Professor**

Θεμιστοκλής
Παναγιωτόπουλος, Καθηγητής

**Dionisios Sotiropoulos,
Assistant Professor**

Διονύσιος Σωτηρόπουλος,
Επίκουρος Καθηγητής

**Ioannis Tasoulas,
Assistant Professor**

Ιωάννης Τασούλας,
Επίκουρος Καθηγητής

ABSTRACT

The games industry is undergoing a rapid growth in the recent years, resulting in an increasing demand for high-quality video games. Nowadays, games need to be built on top of complex architectures whose goal is make development simpler, faster, and more accessible to creators. This is the job of a game engine.

The present thesis analyzes the most important aspects of Wand, a simple visual novel engine. Due to the central role of storytelling and visuals in this particular game genre, heavy focus is placed on graphics and how they are managed by the engine. Rendering graphics is a key component of any development environment for video games, and this is especially true for visual novels.

Aside from the main features of a game engine, this study also summarizes Wand's functionality and provides examples as to how a game programmer might use it. The primary goal of the API is to provide a set of flexible and easy-to-use tools to the visual novel creator while ensuring the efficiency of the end product.

TABLE OF CONTENTS

1. Introduction	5
1.1. The Need for Game Engines.....	5
1.2. Project Description	5
2. What Is a Game Engine?	7
2.1. Definition.....	7
2.2. Popular Game Engines	8
2.3. Visual Novel Engines	8
2.3.1. Definition of a Visual Novel	8
2.3.2. Popular Engines.....	9
3. Main Aspects of a Game Engine	10
3.1. Window	10
3.2. Graphics	11
3.2.1. Definition and Popular APIs.....	11
3.2.2. The Graphics Pipeline.....	12
3.2.3. Rendering Shapes.....	13
3.2.4. Rendering Sprites and Text.....	16
3.3. Audio	17
3.4. State and Serialization	18
3.5. Secondary Engine Features	18
4. Engine Development	20
4.1. Project Details	20
4.1.1. Details and Limitations	20
4.1.2. Project Structure	21
4.1.3. External Libraries	22
4.2. Main Classes and Functions	23
4.2.1. Core Classes	23

4.2.2. Events Classes	28
4.2.3. Input Classes	31
4.2.4. Graphics Classes	32
4.2.5. State Classes	39
4.2.6. UI Classes	39
4.2.7. VN Classes	41
4.2.8. Audio Classes	43
4.2.9. Precompiled Headers	44
5. An Example Game	46
5.1. Game Details	46
5.1.1. Summary	46
5.1.2. Graphics	46
5.1.3. Project Structure	48
5.2. Main Classes and Functions	50
5.3. Game Code Analysis	54
5.3.1. Full Scene Example	54
5.3.2. Example Functions	57
6. Conclusion	59
6.1. Afterword	59
6.2. Future Work	59
References	61

1. INTRODUCTION

1.1. The Need for Game Engines

The fast-paced evolution of technology over the past few decades has caused the production of video games to skyrocket. Millions of games have been created until this day, and this number only keeps growing. The multitude of different platforms combined with the release of game engines, easily accessible to independent developers, has certainly played an important role in this advancement.

However, the more games are released, the higher the standards of the gaming community. Many successful games are nowadays produced by large teams of programmers, designers, artists, and other professionals in the field. Modern hardware can support heavier operations, causing software developers to exploit this opportunity in order to create high-quality products.

Eventually, the need for game engines arrived. Most modern games are very complex to be built from the ground up, and their developers often require an architecture that hides away this complexity and makes room for game logic. By managing feature-specific operations and providing helpful tools to the creator, game engines can greatly reduce the development time required for the production of a game.

1.2. Project Description

This paper aims to explore some of the most important aspects of a game engine. Its scope is limited to a few basic components of a visual novel engine, but its features can easily be generalized to more complex architectures. Although this study will analyze some key concepts that can be applied to many similar projects, the focus will be mostly on the specific features of Wand.

Regarding its technical details, the engine was created with C++ and Visual Studio, two of the most commonly used tools in the games industry. The graphics-API chosen for this project is OpenGL, mainly due to its simplicity and cross-platform availability. Wand also integrates several third-party libraries which will be described in one of the following sections. As for the Visual Studio solution, it consists of two projects: *Wand* and *Game*.

Wand Engine provides tools for the creation of 2D games, and more specifically, visual novels. Due to their story-driven gameplay and limited action and world exploration, most games that belong in this genre do not require the entirety of features offered by many popular game engines, such as Unity. Very often, the player only has to click on the screen in order to progress in the story and, for this reason, the need for a level editor and other complex GUI elements is less imperative.

Ren'Py, the most popular visual novel engine, serves as this project's inspiration. It has a relatively simple Graphical User Interface, and most of its development involves writing code in scripts. Although Wand does not support scripting, it shares *Ren'Py*'s simplicity in the sense that the entire visual novel can be created inside Visual Studio. As an engine, Wand is statically linked to the Game project and has no GUI of its own.

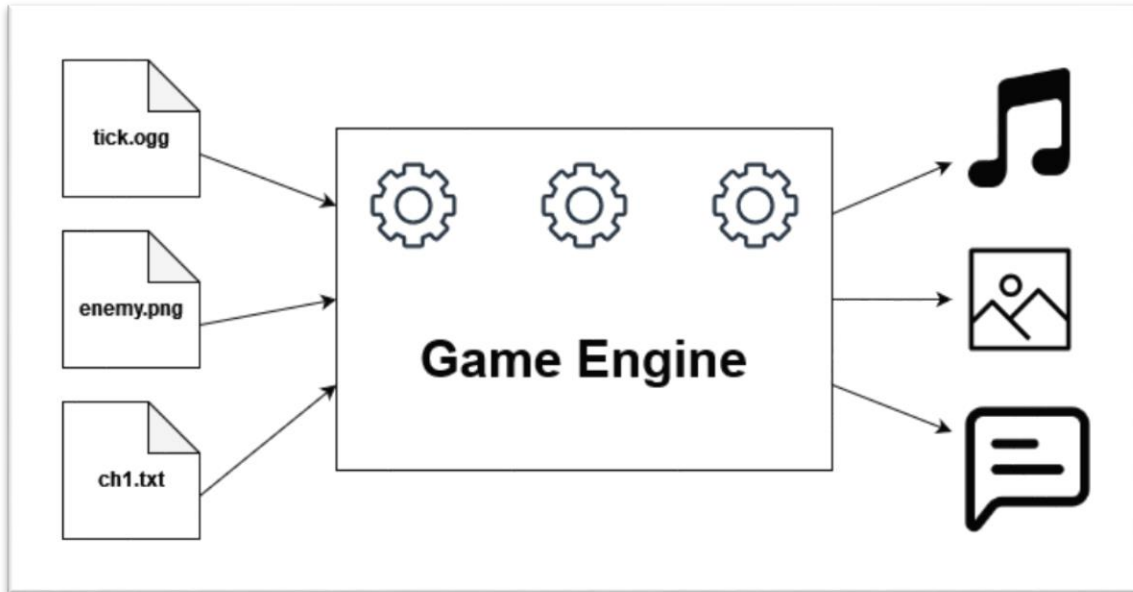
Along with the engine, one separate piece of software was developed: a simple visual novel named *Wand Tutorial* that makes use of Wand's tools. Although this paper will analyze the main features of both projects, the full code is freely available on GitHub. Their online repositories can be found at the following URLs:

Wand: <https://github.com/mariaviolaki/wand>

Wand Tutorial: <https://github.com/mariaviolaki/wand-tutorial>

2. WHAT IS A GAME ENGINE?

2.1. Definition



A game engine can be defined as a set of tools that the programmer can use to build an interactive application. Some engines, such as Unity or Unreal, are commercialized and publicly released. Many others, though, often developed by big game studios like Valve and Ubisoft, are internal to the company, built as platforms that can be used for the development of video games. It is important to note, however, that the same engines can be used for products other than games: Virtual Reality applications, architectural visualizations, simulations, and many other interactive programs also display graphics in real time and usually require the same set of tools that are built into a game engine.

A more specific definition would be centered around data transformation. The core function of a game engine is the conversion of data from one form to another: It is responsible of reading in a variety of files from the disk and transforming them in a way that can be easily perceived by humans. Text, images, audio files, and many other forms of data are parsed by the engine at runtime and are presented to the user along with ways to interact with them (Chernikov, What is a GAME ENGINE?, 2018).

2.2. Popular Game Engines

Nowadays, there is a variety of engines that the developer can use to build their game. One of the most well-known and widely used ones is Unity. Launched in 2005, Unity Engine has gained a sizable community and enjoys constant updates. It is often perceived as the top choice for beginners and indie game developers, and the support for the creators is significant. It can be used for free by anyone who earns less than 100K dollars per year and has a store with a great variety of free and paid assets. In addition, not only can the engine be used for the development of 2D and 3D games, it is also popular for its support for VR and AR applications. At the same time, it is well-suited for the creation of games that run on multiple platforms, including Android and iOS.

A slightly older yet equally popular choice for game developers is Unreal Engine. Like Unity, it is cross-platform and can be used for AR and VR applications. However, while it provides tools for the development of both 2D and 3D games, it is best suited for the latter. Unreal Engine is considered more performant than other commercial engines, but this comes at the cost of greater system requirements. Although it can also be used by indie creators as well, it works best when it comes to the development of team projects. Many triple-A games have been built with this engine, in fact, since it handles complicated tasks more efficiently. Lastly, it is important to note that it also includes a Marketplace with free assets and also offers a variety of useful tools, such as blueprints which can be valuable for non-programmers.

Besides Unity and Unreal Engine, there are several other options that are less popular and more limiting in one way or another. Godot, Phaser, GameMaker Studio, and RPG Maker are some of the most well-known among them. (Schardon, 2022)

2.3. Visual Novel Engines

2.3.1. Definition of a Visual Novel

Visual Novels are a video game genre where the story plays a central role in development. While there are several published 3D games of this type, the majority of Visual Novels are 2D applications. The characters and locations featured in them can be static or animated, but the gameplay is rather limited compared to other types of games. Very often, Visual Novels also include music, sound effects, as well as cutscenes, although as a rule, the emphasis is put on narrative and visuals.

2.3.2. Popular Engines

When it comes to the development of Visual Novels, Ren'Py is the most widely used engine. Released in 2004, it is a free and open-source tool that only requires some basic knowledge of Python. It is a straightforward engine that can be easily learned through tutorials since it lacks much of the complexity of a typical game engine. It is very flexible and uses Python for scripting, making it fairly easy to use by beginners.

There are also a few other known options for this genre, such as TyranoBuilder which allows for less customization but is easier to use, or Visual Novel Maker which is pricier and has a steeper learning curve but includes a full suite of features (Vincent, 2020). Additionally, since Visual Novels and common video games share many similar features, the creator can also use more complex engines such as Unity or Unreal for development. Such options, however, are often considered unnecessarily hard and heavy on the system since Visual Novels are typically a lot simpler.

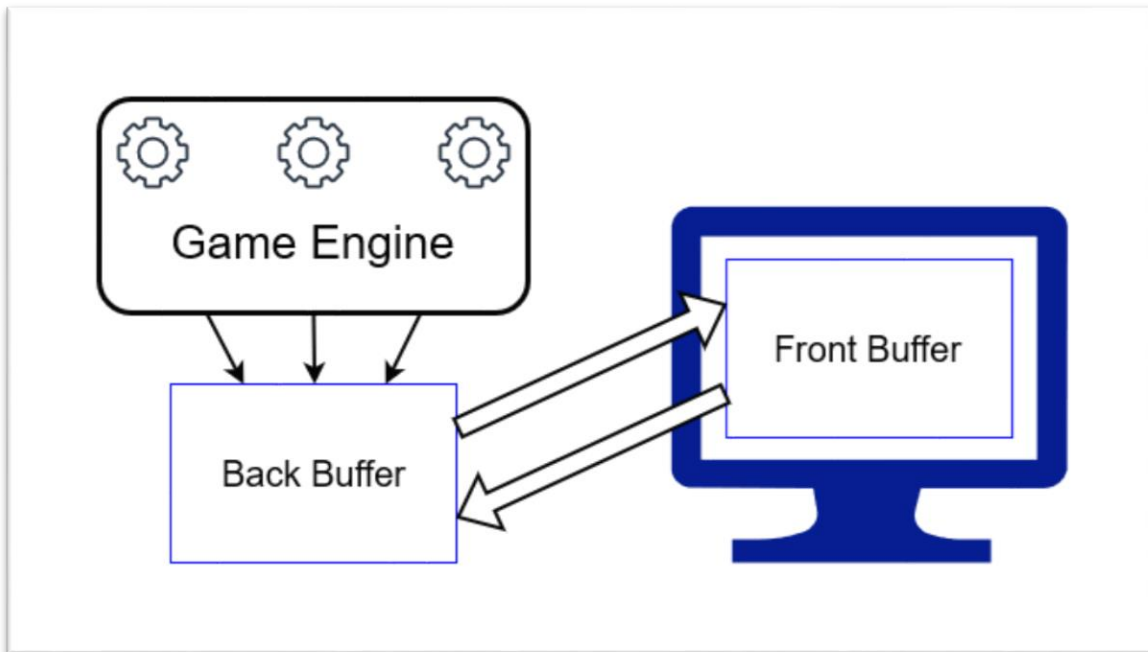
3. MAIN ASPECTS OF A GAME ENGINE

3.1. Window

One of the most crucial components in a game engine is the creation of a window, which can be perceived as the basis of many other major subsystems, including graphics and user input. For this reason, the developer often needs to use an external library that is responsible of clearing and updating the graphics drawn every frame as well as keeping track of the events triggered when the user interacts with it.

These events include keyboard presses, mouse button clicks, scrolling, and movement, as well as others which are related to the window itself, such as setting the fullscreen mode, resizing, and closing. The engine either needs to notify its various subsystems with its own events when these window API events occur, or save their data so that they can be available when the user—or any other part of the engine—requests them.

As for the graphics-related functions offered by the library API, they range from clearing the existing graphics with a certain color to setting the blending mode for the shapes that are drawn on top of it. In order to achieve the former, the window keeps track of two buffers; the one on the back that is meant for writing and the front one that will be displayed. This mechanism allows the engine to fill the back buffer with graphics while the front buffer shows to the user the graphics of the latest frame. When the next frame is about to be rendered, these buffers are swapped and the pixels of the new back buffer are overwritten with new information (Gordan, 2021). The window follows these same steps continuously until it is closed.



Blending can produce different results depending on the blending function and the blending equation set by the engine developer. It determines how the fragment shader's output color is combined with the color that is already in the buffer. In this way, the new colored pixel can be set to cover the already existing one according to its transparency, or it can even replace the old color entirely, regardless of the value of its alpha parameter.

3.2. Graphics

3.2.1. Definition and Popular APIs

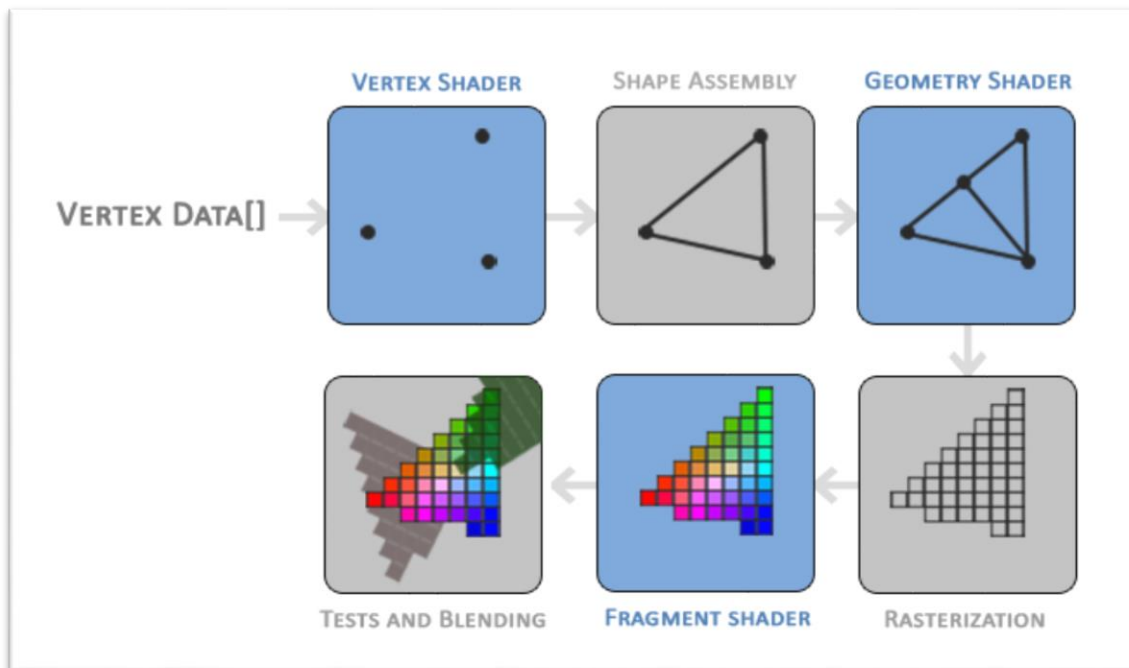
Graphics is one of the most important systems in a game engine. It is essentially everything that is displayed on the screen, including text, images, videos, or even basic shapes. The hardware that is responsible for rendering graphics on the screen is the GPU, and it is accessed and programmed by game engines in order to render 2D and 3D shapes as efficiently as possible.

There are several APIs that allow the engine developer to access the graphics card: OpenGL which is cross-platform and fairly easy to use by beginners, Vulkan which is more powerful as well as cross-platform, DirectX which is available on Microsoft platforms, and Metal which is used for iOS and macOS platforms (Chernikov, 2017).

3.2.2. The Graphics Pipeline

The programs that run on the GPU and are responsible of rendering graphics are called shaders. Shaders are part of the rendering pipeline, a series of stages that converts data into the final image that is displayed on the screen.

The Graphics Pipeline (de Vries)



The data that is given as input to the graphics pipeline is in the form of vertices. As opposed to those defined in mathematics, graphics vertices contain a lot more information than just position coordinates. Color and texture coordinates are two such examples.

The vertex shader, that is, the first part of the graphics pipeline as seen above, is responsible of receiving the vertices supplied by a program that runs on the CPU and transforming them according to the needs of the application. In the next stage, the shape assembler connects the final positions supplied by the vertex shader according to a primitive, i.e. a point, a line, or a triangle. As for the geometry shader, it creates additional vertices, thus forming more primitives from the already existing ones.

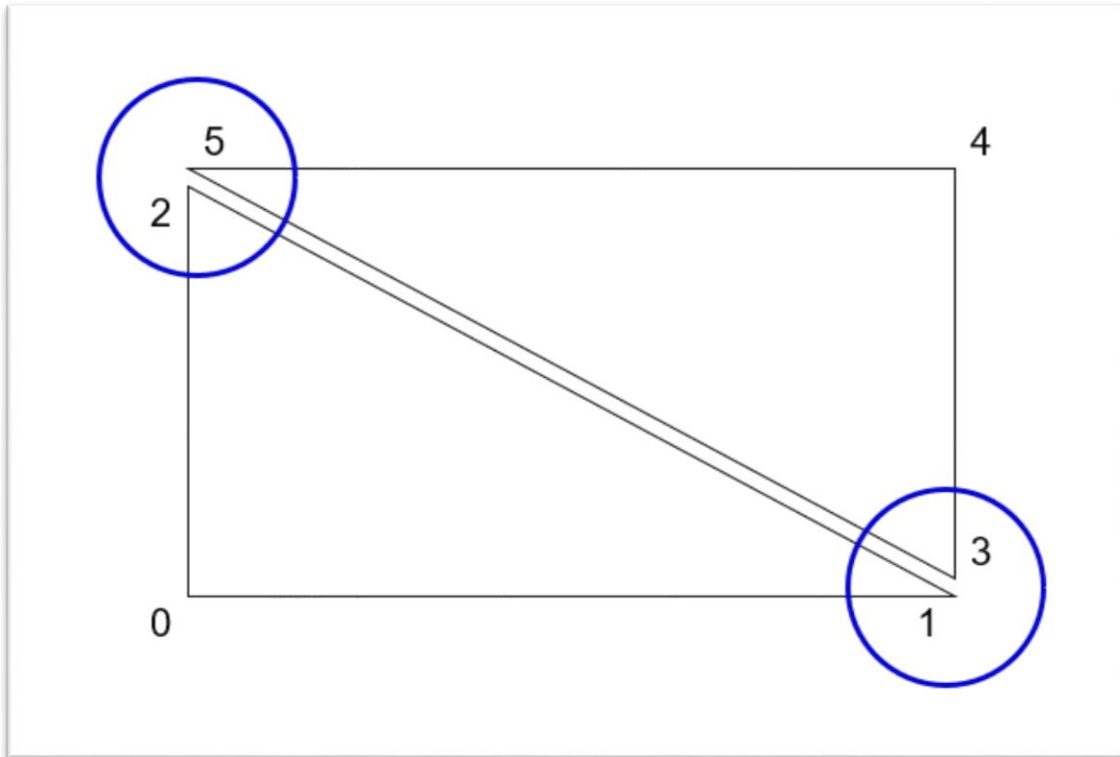
This data is then rasterized. In other words, the accurate, mathematical shapes, created in the following stages are transformed into pixels that will be displayed on the screen. These pixels are colored in the fragment shader. The color of each pixel depends on multiple factors such as lighting, shadows, and texture coordinates. However, there may be more than one color corresponding to the same pixel since there may be multiple objects overlapping in the same place. The final color is determined in the last stage of the graphics pipeline where the blending of transparent, semi-transparent, and non-transparent objects takes place (Gordan, 2021).

Taking everything into consideration, the vertex and fragment shaders are usually considered as the most important ones and are often the only ones that need to be adjusted in a game engine.

3.2.3. Rendering Shapes

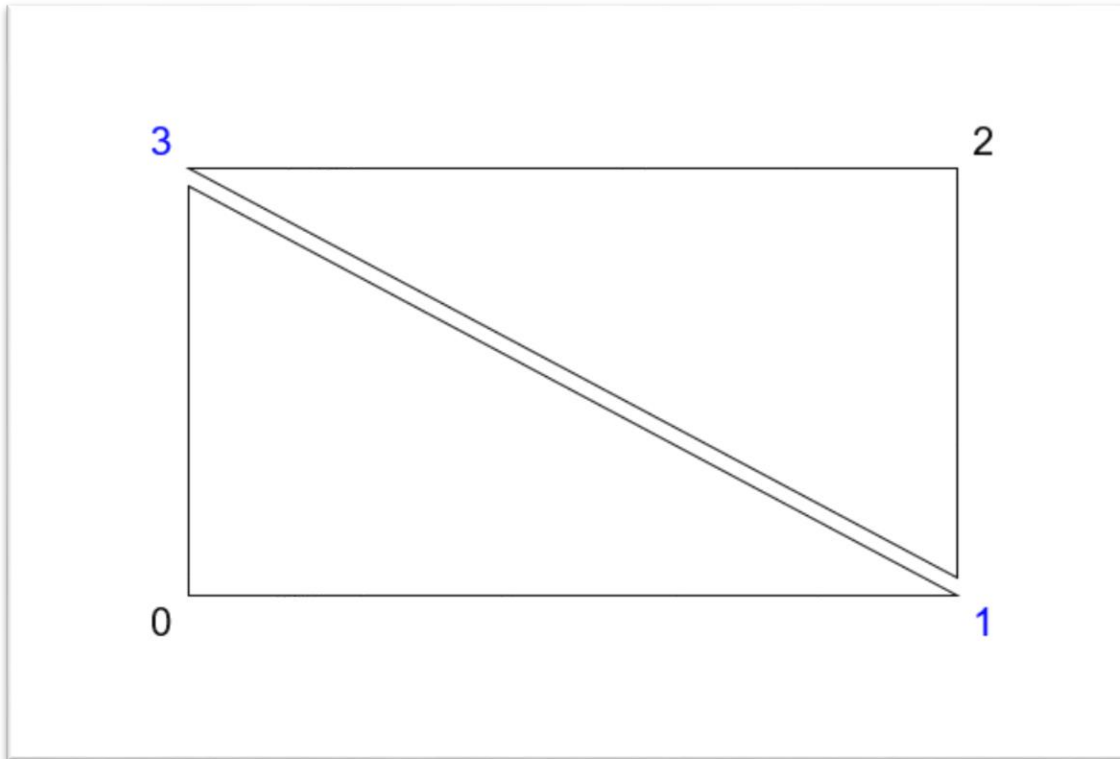
In order to render any kind of shape in a graphics engine, the shape first needs to be divided into triangles. This is the simplest arrangement of vertices in space so that they can form a flat surface. The game developer should be able to create any kind of geometry depending on their needs.

The most common shape that can be rendered by a graphics engine is a rectangle, which can then be broken down into two triangles. Since each triangle consists of three vertices, six vertices are needed in total for the shape to be drawn. However, two of these vertices are duplicate and can result in a significantly lower performance when multiple and more complex shapes need to be rendered in a single frame.



In this case, in order to draw the rectangle depicted above, one would have to supply to the GPU the vertices of these two triangles. That is, the vertices [0, 1, 2] and [3, 4, 5].

To avoid this waste of memory, the graphics engineer can make use of indices. As a first step, instead of creating an array with vertices that hold the same data, they can supply to the GPU four vertices for each rectangle. But since every shape needs to be broken down into triangles, the engineer should also create a set of indices that point to the appropriate vertex.



If the resulting array of vertices then was [0, 1, 2, 3] the array of indices to draw two triangles would be [0, 1, 3, 1, 2, 3]. The repetition of numbers in the index array is much more preferable to a repetition in the vertex array because, as it has already been mentioned, each vertex holds a lot more information than a single integer. For instance, the data for a white point in the bottom left corner of the window would be: [0, 0, 0, 255, 255, 255, 255]. The first three numbers correspond to the coordinates on the x, y, and z axis, and the last four correspond to the RGBA color of non-transparent white. If a single vertex contains all these seven numbers, the data needed to draw a rectangle would require 28 numbers in total.

The array of vertices is stored in an array buffer within the GPU, and the index array is also stored in a respective buffer. The entity that keeps track of every number within a single vertex is called Vertex Array, and it manages all the vertex buffers that are created in the graphics card. It is also possible for the developer to store multiple shapes within the vertex buffer, in which case the Vertex Array only manages the data of a single buffer.

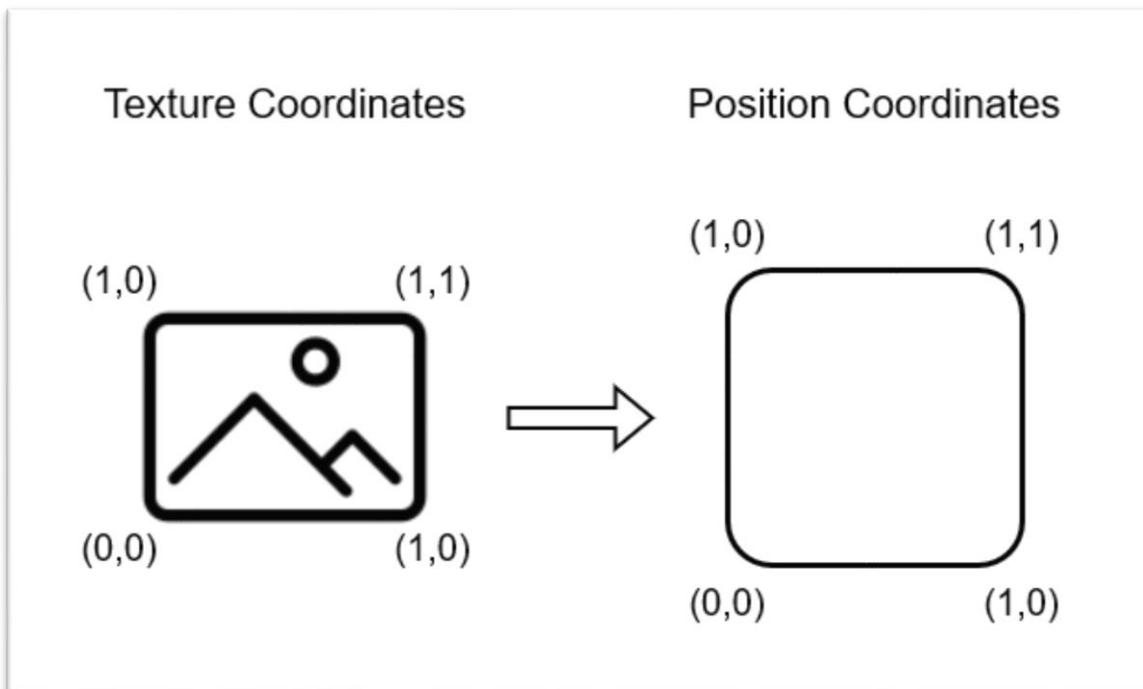
It is important to note, however, that the larger the vertex buffer grows, the larger the index buffer becomes as well. For example, a vertex buffer containing data for a single rectangle

stores four vertices, and the index buffer that corresponds to it holds six indices. A buffer with two rectangles, and thus, eight vertices would require twelve indices, and so on.

3.2.4. Rendering Sprites and Text

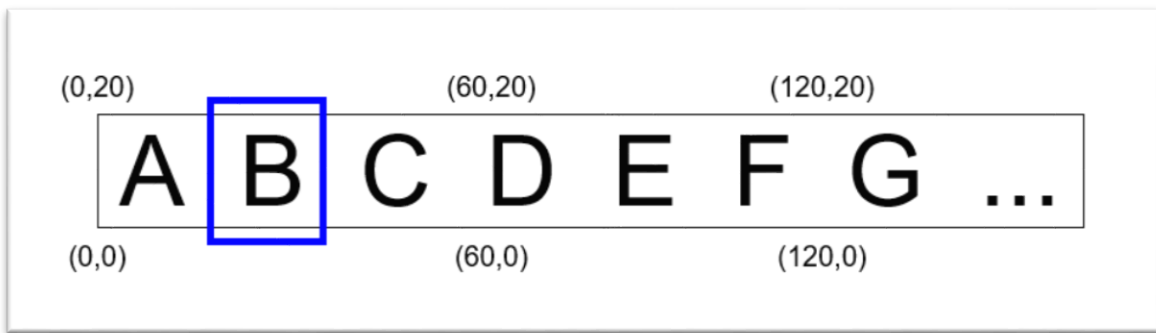
Aside from displaying basic shapes, a game engine often needs to load images and fonts from disk and draw them as well. For this purpose, it is often useful for the developer to use external libraries to read in these files and convert them into a set of bytes. Similarly to basic shapes, these bytes are also saved in buffers and sent to the GPU for rendering.

Two-dimensional sprites as well as three-dimensional models are also referred to as textures. Sprites can be rendered on top of shapes, often rectangles. As it has already been mentioned, part of the data that may be included in a vertex are the texture coordinates. In this way, each edge of the rectangle can be associated to the respective edge of the sprite. As a result, the final color of a pixel is determined not by an RGBA color, but by the appropriate color in the texture.



As for text rendering, a similar procedure is followed. In order to draw characters on the screen, the engine should at least take as input a font file, a text size, a color, and the string that needs to be displayed. While the last parameters are usually straightforward, the font needs to be loaded into memory and be used for the creation of a font atlas, which is essentially a large texture. A font atlas, contains all the different characters, or glyphs, in a font. These can be letters, numbers, and special characters.

Font Atlas



To render a single character, the developer would still need to supply the data of four vertices to the GPU. Each vertex would at least contain position, color, and texture data. For example, and according to the image above, in order to draw a red 'B' in a 60x80 rectangle placed at the bottom left corner of the window, the input to the graphics pipeline would be the following for the top-right vertex: [60, 80, 0, 255, 0, 0, 255, 40, 20].

The first three numbers refer to the x, y, and z coordinates of the top-right corner of the rectangle. The next four describe the color red. As for the last two numbers, they correspond to the top-right corner of the glyph for the letter 'B' inside the font atlas.

3.3. Audio

Depending on the needs of the engine, audio can either be a simple or a very complex subsystem. The use of dedicated libraries is often necessary for the engine programmer. The application would need to read in an audio file at runtime, save it in memory, and adjust the parameters of its playback. Both music and sound effects might need to be played at different volumes, panning, and speed. Whether the track should loop or not is also a

valuable piece of information since some audio sources are only meant to be played once. This is often the case for sound effects and voice lines as opposed to background music.

3.4. State and Serialization

There can be various implementations when it comes to this particular aspect. Since the majority of applications make use of data that should persist even after the user exits the program, it is crucial that these data be stored on disk or even backed up online. This conversion of data into a format which is then saved or transmitted into some form of storage is known as serialization. The reverse process, deserialization, makes use of the backed-up information to recreate the original objects and data types needed by the application.

The engine needs to store data in a clear and concise way. For this reason, although it is possible to save the state of the application in files that contain plain, unstructured text, it is rarely the preferred method. The engine programmer can choose to serialize their data in other formats, such as binary or JSON. While the former can be faster and more efficient in terms of space, JSON serialization creates more portable files that can be parsed and read by humans more easily.

3.5. Secondary Engine Features

A game engine can include many additional subsystems depending on its scope and goals set by its programmer. For many engines, physics and collision detection is a core component since a great number of games contain objects that move around in space, overlap, and interact with one another. This feature is not particularly useful for a visual novel because its developer rarely needs to apply physics to the relationship between game objects or even to their relationship with the world. The visual novel developer often needs to only tell a story and not include battles or similar mechanics in the gameplay. However, it is also possible that some visual novels may require such a feature, and therefore the lack of a physics and collision detection subsystem would be restrictive.

On the contrary, animation is something that almost every application may need to include. Visual novels could make use of animated sprites, and even a simple UI program might require this functionality for operations such as the click of a button. In regard to visual novels, the applications of animation vary. Some UI-related functions would be the resizing of a button when the user interacts with it, the release of particles or ripples starting from

the position of a mouse click, and a smooth transition between menus. Moreover, there have been released multiple visual novels that implement character animation for functions such as blinking and breathing. Background animations can also add to the user experience for such games and can be used in various occasions within a story. These include natural phenomena like rain and snow, moving objects such as curtains and wind chimes, and even background particles to create a certain atmosphere.

There is a lot more functionality that can be added to a game engine, and depending on the needs of the project, some subsystems may play a more central role compared to others. Particles, for instance, are also an important component for most games, and networking can be also essential although it is mainly implemented by engines which aim to provide tools for online multiplayer games. Another basic feature that the engine developer should consider is scripting: Although the end product would be slower in performance and less flexible, its development would require less time and technical knowledge. Lastly, AI is also considered as an essential subsystem even though its usefulness depends on the types of games that can be made with the engine. Other examples of secondary or smaller features would be a mechanism generating random numbers and a logger that prints information and error messages to the screen or a file.

In any case, one thing that all engines have in common is the system that binds them all together and initializes the various components and main resources of the application. After being initialized, the main components of the engine will in turn set up smaller subsystems dependent on them, and this process will continue in the same manner until every crucial feature of the engine is operational.

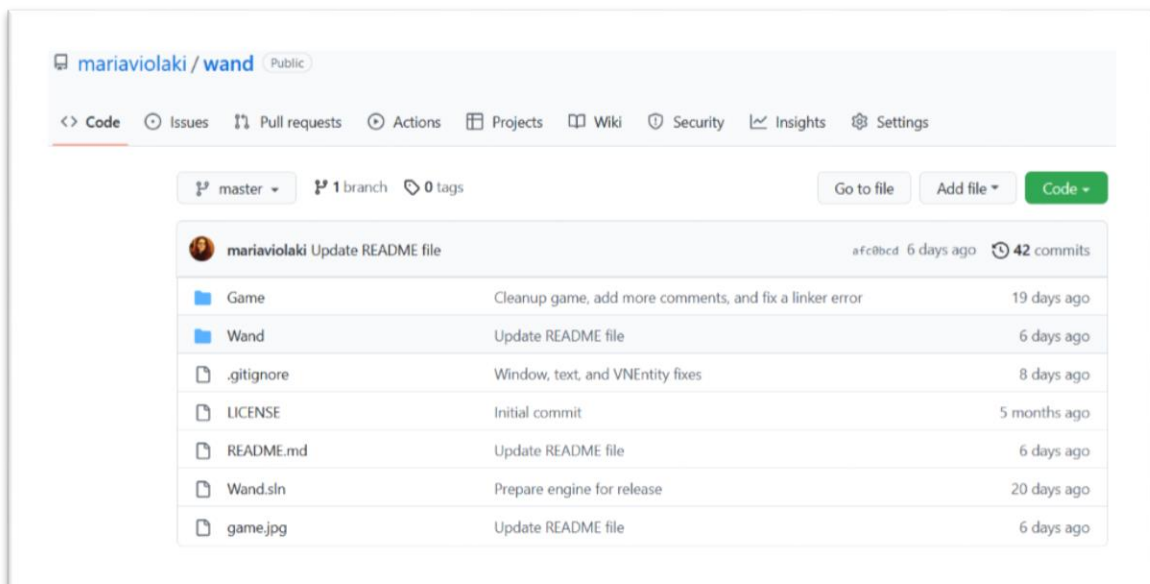
4. ENGINE DEVELOPMENT

4.1. Project Details

4.1.1. Details and Limitations

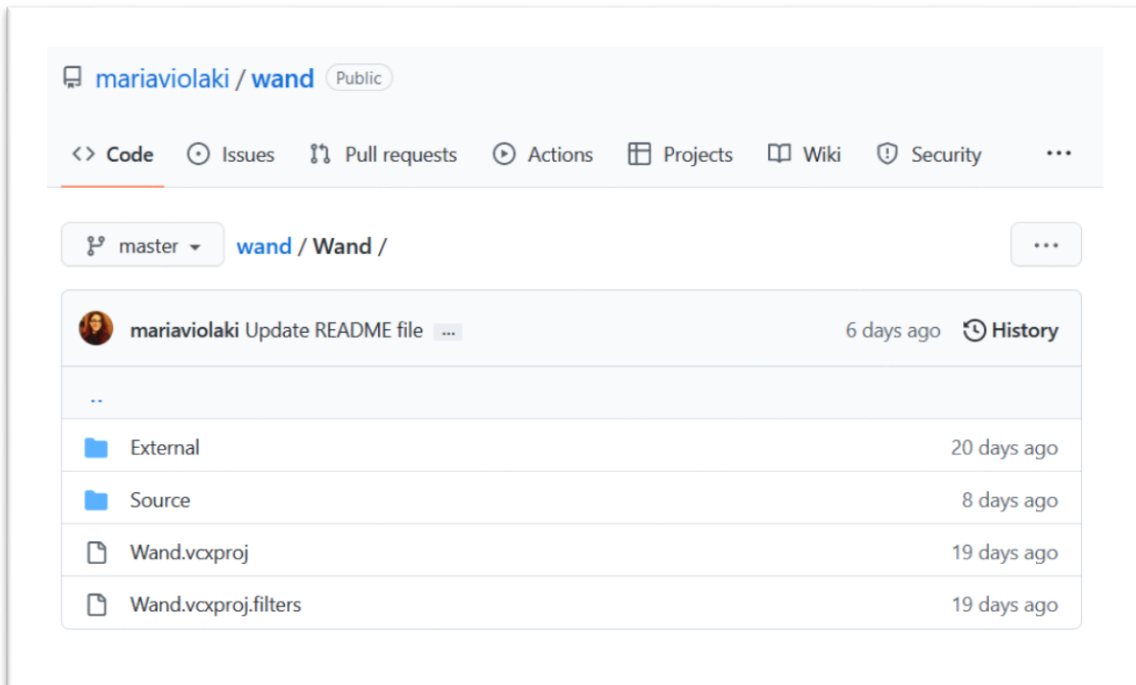
Wand was developed with Visual Studio and is written in C++. Although the external libraries used for this project are cross-platform, the games made with this engine are aimed for Windows x64 machines and the game developer is encouraged to use Visual Studio and C++ for programming.

The engine is also available on GitHub, and the root directory of the repository corresponds to the root directory of the Visual Studio solution. It is composed of two main projects, each of them in its own separate folder: *Wand*, which contains all the engine code, and *Game*, which is intended for the game developer and includes the game's code, assets, and states which are generated at runtime. The repository of Wand Engine can be found at this URL: <https://github.com/mariaviolaki/wand>

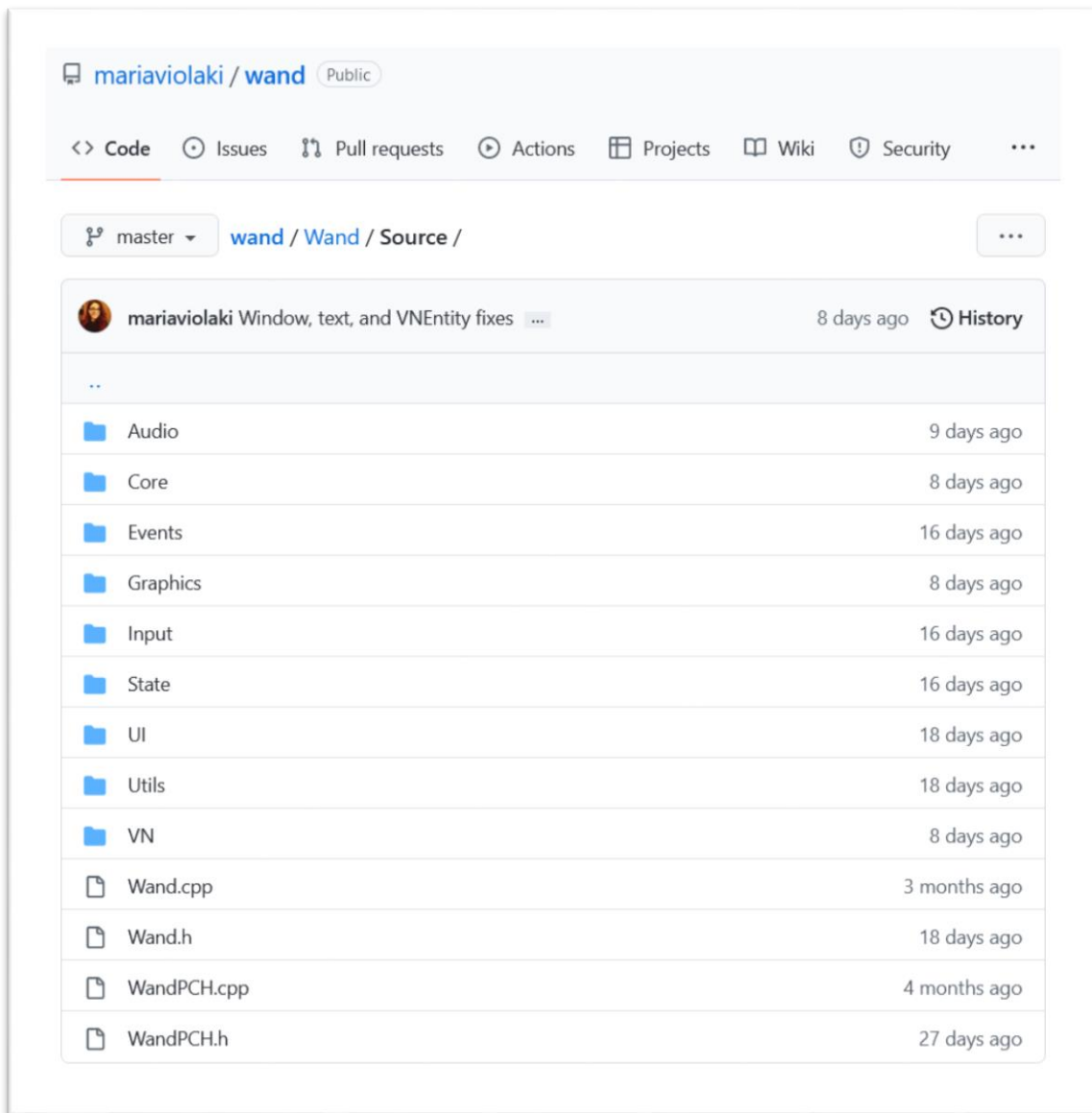


4.1.2. Project Structure

As it was already mentioned, the *Wand* project folder contains all the engine code. Each file is placed into a different folder, according to the category it belongs to. The main folders in this directory are *External* and *Source*. The former includes all the external libraries that have been linked to the engine, and the latter contains all the original engine code.

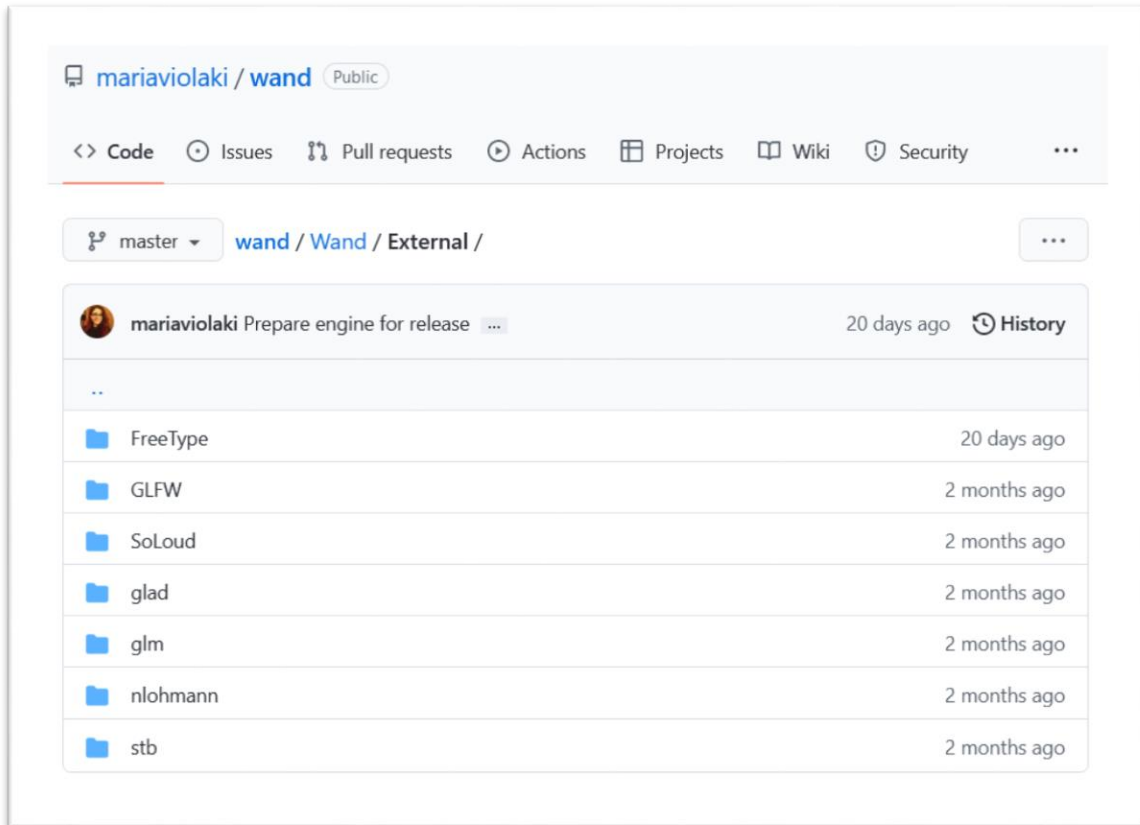


The contents in *Source* summarize the engine's main features, grouped in different folders: *Core* contains all the major components. The majority of files in this directory contain classes and functions that are called in the beginning of the program and are required for the initialization of other subsystems. There are also several other major features which do not belong in any of the other categories. The majority of the other directories within *Source* are focused on some particular aspect of the engine: *Events*, *Input*, *Graphics*, *Audio*, *State*, *UI*, and *VN* have limited or no interaction with each other. *Utils* contains various tools that are useful for the engine and the game developer. Their scope is small enough and they do not belong in any other category.



4.1.3. External Libraries

The folder *External* contains the following libraries and APIs: OpenGL for graphics rendering, Glad for OpenGL initialization, GLFW for window creation and management, GLM for mathematics for OpenGL, nlohmann/json for JSON serialization, stb_image for image loading, FreeType for font rendering, and SoLoud for audio playback.

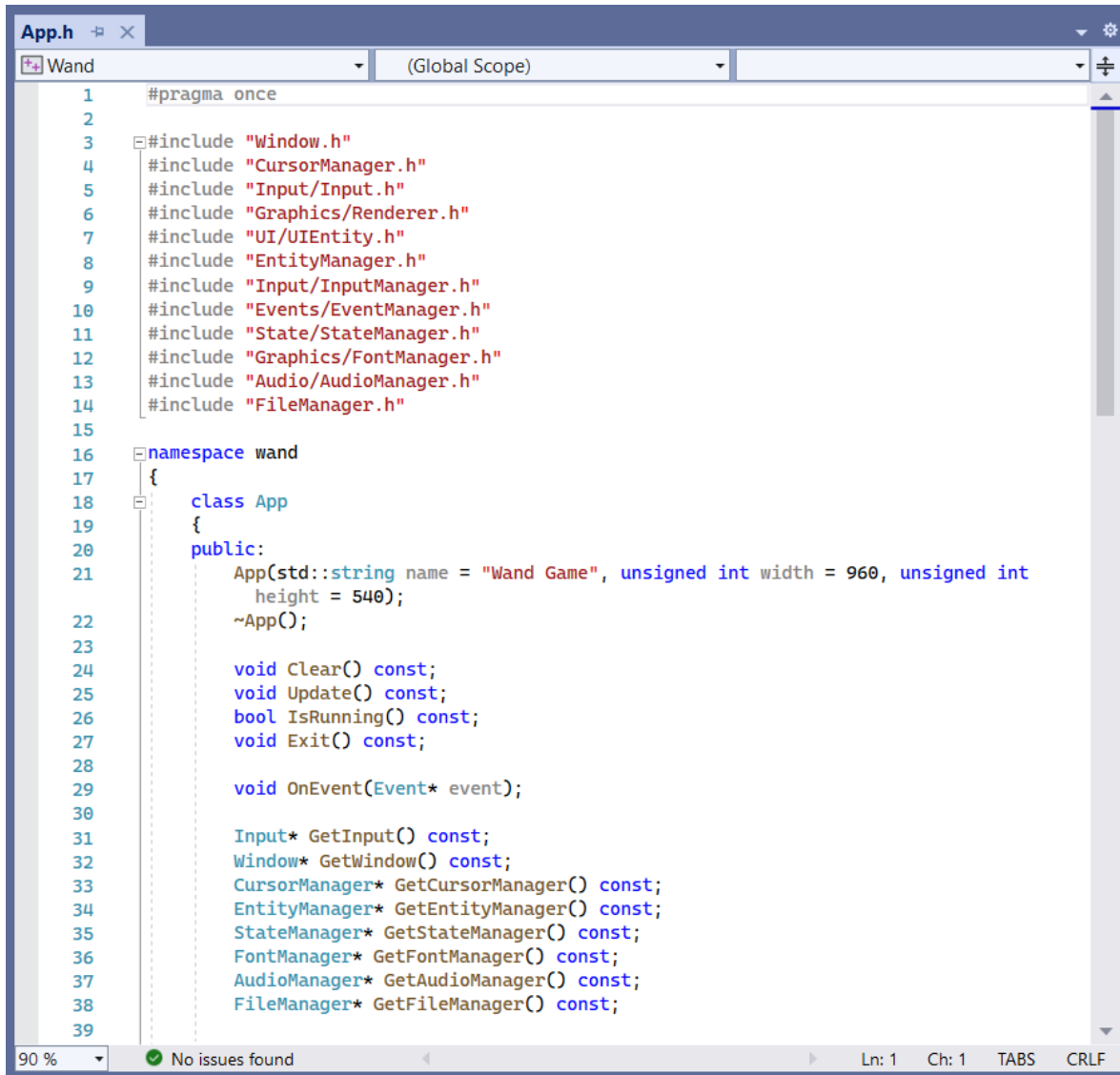


Some libraries only consist of header files, and every library that was linked to the game engine was linked statically. In the cases when a *.lib* file was required for the project to build, two separate files were provided; one for Debug and one for Release mode.

4.2. Main Classes and Functions

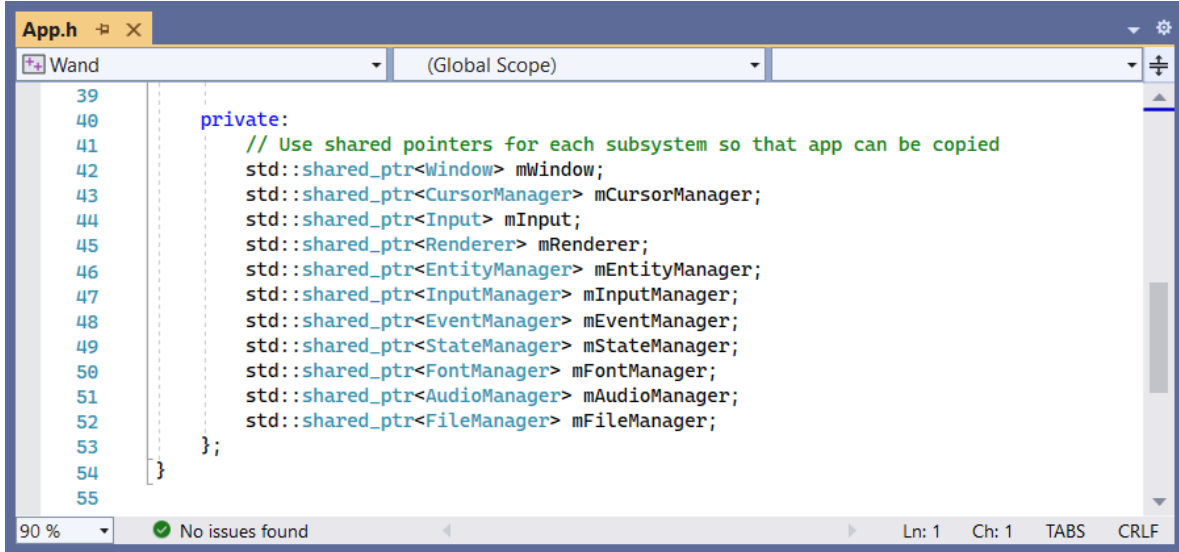
4.2.1. Core Classes

All the code for the game engine is included within the *wand* namespace. For every class in the project there is a header file and a source file created. Although the engine is built as a static library and doesn't have a starting point of execution, there is a main component from which all the other major subsystems derive. As it will be explained later in the *Example Game* section, the first object that should be instantiated in a Wand game is the *App*. *App* contains the majority of features that the engine or the game developer might need while the program runs.

App.h

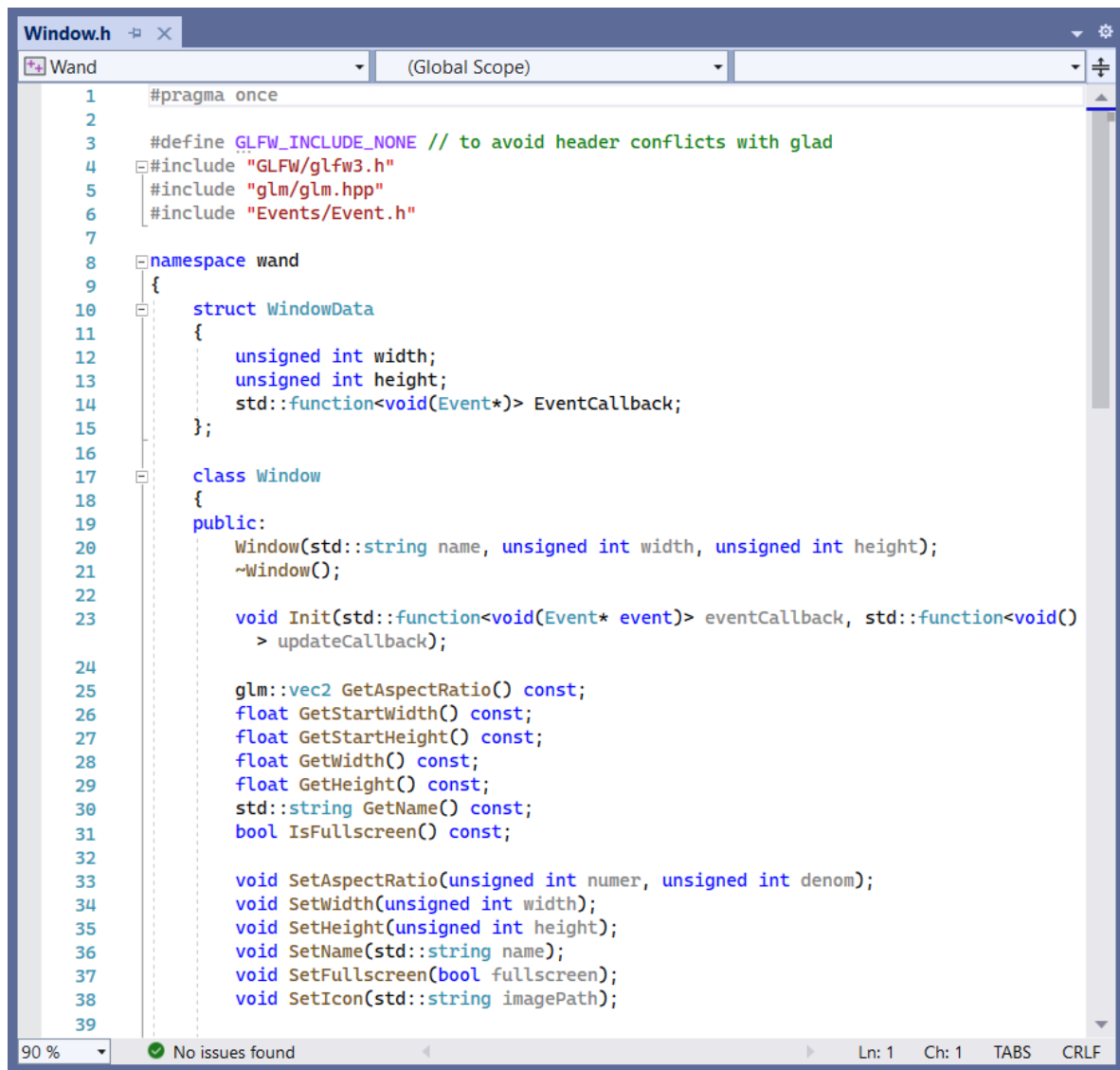
```
1 #pragma once
2
3 #include "Window.h"
4 #include "CursorManager.h"
5 #include "Input/Input.h"
6 #include "Graphics/Renderer.h"
7 #include "UI/UIEntity.h"
8 #include "EntityManager.h"
9 #include "Input/InputManager.h"
10 #include "Events/EventManager.h"
11 #include "State/StateManager.h"
12 #include "Graphics/FontManager.h"
13 #include "Audio/AudioManager.h"
14 #include "FileManager.h"
15
16 namespace wand
17 {
18     class App
19     {
20     public:
21         App(std::string name = "Wand Game", unsigned int width = 960, unsigned int
            height = 540);
22         ~App();
23
24         void Clear() const;
25         void Update() const;
26         bool IsRunning() const;
27         void Exit() const;
28
29         void OnEvent(Event* event);
30
31         Input* GetInput() const;
32         Window* GetWindow() const;
33         CursorManager* GetCursorManager() const;
34         EntityManager* GetEntityManager() const;
35         StateManager* GetStateManager() const;
36         FontManager* GetFontManager() const;
37         AudioManager* GetAudioManager() const;
38         FileManager* GetFileManager() const;
39     };
40 }
```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF



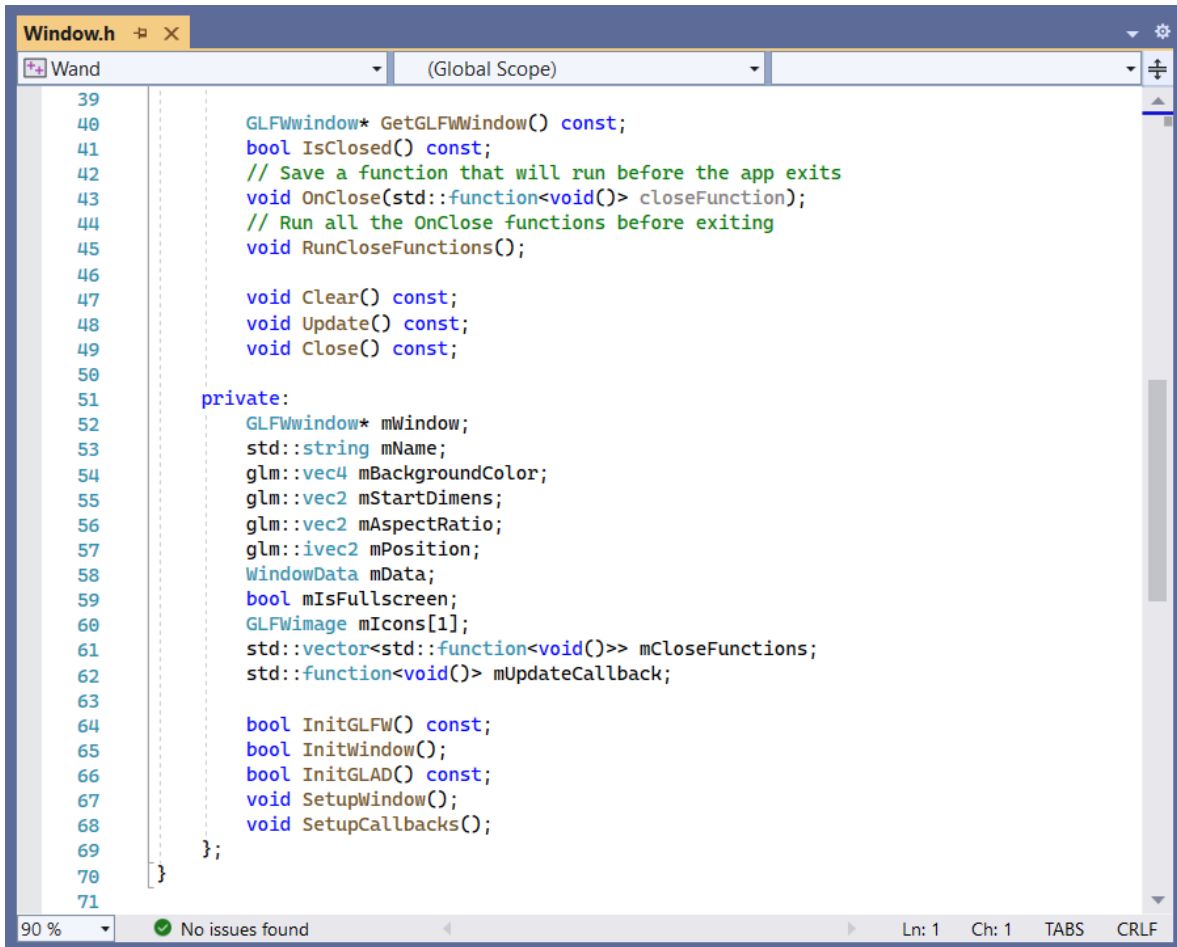
```
39
40     private:
41         // Use shared pointers for each subsystem so that app can be copied
42         std::shared_ptr<Window> mWindow;
43         std::shared_ptr<CursorManager> mCursorManager;
44         std::shared_ptr<Input> mInput;
45         std::shared_ptr<Renderer> mRenderer;
46         std::shared_ptr<EntityManager> mEntityManager;
47         std::shared_ptr<InputManager> mInputManager;
48         std::shared_ptr<EventManager> mEventManager;
49         std::shared_ptr<StateManager> mStateManager;
50         std::shared_ptr<FontManager> mFontManager;
51         std::shared_ptr<AudioManager> mAudioManager;
52         std::shared_ptr<FileManager> mFileManager;
53     };
54 }
55
```

As it can be seen above, the game developer can request some major subsystems directly from the *App*. One of these is the *Window* which initializes and manages the most important graphics-related resources of the engine. By accessing the *Window*, the programmer can change the window's title, size, icon, fullscreen mode, and aspect ratio. This class also keeps track of the events that are directly related to it.

Window.h

```
1 #pragma once
2
3 #define GLFW_INCLUDE_NONE // to avoid header conflicts with glad
4 #include "GLFW/glfw3.h"
5 #include "glm/glm.hpp"
6 #include "Events/Event.h"
7
8 namespace wand
9 {
10     struct WindowData
11     {
12         unsigned int width;
13         unsigned int height;
14         std::function<void(Event*)> EventCallback;
15     };
16
17     class Window
18     {
19     public:
20         Window(std::string name, unsigned int width, unsigned int height);
21         ~Window();
22
23         void Init(std::function<void(Event* event)> eventCallback, std::function<void()
24             > updateCallback);
25
26         glm::vec2 GetAspectRatio() const;
27         float GetStartWidth() const;
28         float GetStartHeight() const;
29         float GetWidth() const;
30         float GetHeight() const;
31         std::string GetName() const;
32         bool IsFullscreen() const;
33
34         void SetAspectRatio(unsigned int numer, unsigned int denom);
35         void SetWidth(unsigned int width);
36         void SetHeight(unsigned int height);
37         void SetName(std::string name);
38         void SetFullscreen(bool fullscreen);
39         void SetIcon(std::string imagePath);
40     };
41 }
```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF



```
Window.h
Wand (Global Scope)
39
40     GLFWwindow* GetGLFWWindow() const;
41     bool IsClosed() const;
42     // Save a function that will run before the app exits
43     void OnClose(std::function<void()> closeFunction);
44     // Run all the OnClose functions before exiting
45     void RunCloseFunctions();
46
47     void Clear() const;
48     void Update() const;
49     void Close() const;
50
51 private:
52     GLFWwindow* mWindow;
53     std::string mName;
54     glm::vec4 mBackgroundColor;
55     glm::vec2 mStartDimens;
56     glm::vec2 mAspectRatio;
57     glm::ivec2 mPosition;
58     WindowData mData;
59     bool mIsFullscreen;
60     GLFWimage mIcons[1];
61     std::vector<std::function<void()>> mCloseFunctions;
62     std::function<void()> mUpdateCallback;
63
64     bool InitGLFW() const;
65     bool InitWindow();
66     bool InitGLAD() const;
67     void SetupWindow();
68     void SetupCallbacks();
69 };
70
71
90 % No issues found Ln: 1 Ch: 1 TABS CRLF
```

The last major class within *Core* is the *EntityManager*. The purpose of this class is to save and keep track of all the UI entities created in a game. These are Rectangles, Sprites, TextBoxes, and Buttons which can either be created with a Rectangle and a TextBox, or a Sprite and a TextBox. All these entities can be altered during runtime, and *EntityManager* always keeps the updated version of them. It can send them to other subsystems within the engine or even to the game programmer if they request them.

EntityManager.h

```

1  #pragma once
2
3  #include "UI/UIEntity.h"
4  #include "UI/Rectangle.h"
5  #include "UI/Sprite.h"
6  #include "UI/TextBox.h"
7  #include "UI/Button.h"
8
9  namespace wand
10 {
11     class EntityManager
12     {
13     public:
14         EntityManager();
15
16         void Init(FontManager* fontManager);
17         // Create and render a new rectangle
18         Rectangle* AddRectangle(Color color = Color(255, 255, 255, 0));
19         // Create and render a new sprite
20         Sprite* AddSprite(const std::string& imagePath);
21         // Create and render a new textbox
22         TextBox* AddTextBox(const std::string& fontName, unsigned int fontSize, Color
            color);
23         // Create and render a new image button
24         Button* AddButton(std::string imagePath, std::string fontName, unsigned int
            fontSize, Color textColor);
25         // Create and render a new rectangle button
26         Button* AddButton(Color bgColor, std::string fontName, unsigned int fontSize,
            Color textColor);
27         // Get all the UI entities that have been created
28         std::vector<std::unique_ptr<UIEntity>>& GetEntities();
29
30     private:
31         std::vector<std::unique_ptr<UIEntity>> mEntities;
32         FontManager* mFontManager;
33
34         // Set the font manager needed for the text's initialization
35         void SetFontManager(TextGFX* textGFX) const;
36     };
37 }
38

```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF

4.2.2. Events Classes

Even though Events are dependent on other engine components, they remain a central subsystem that affects many other classes, including the Window, the Input, the UIEntities, and the Renderer. There are many different types of Events as shown below:

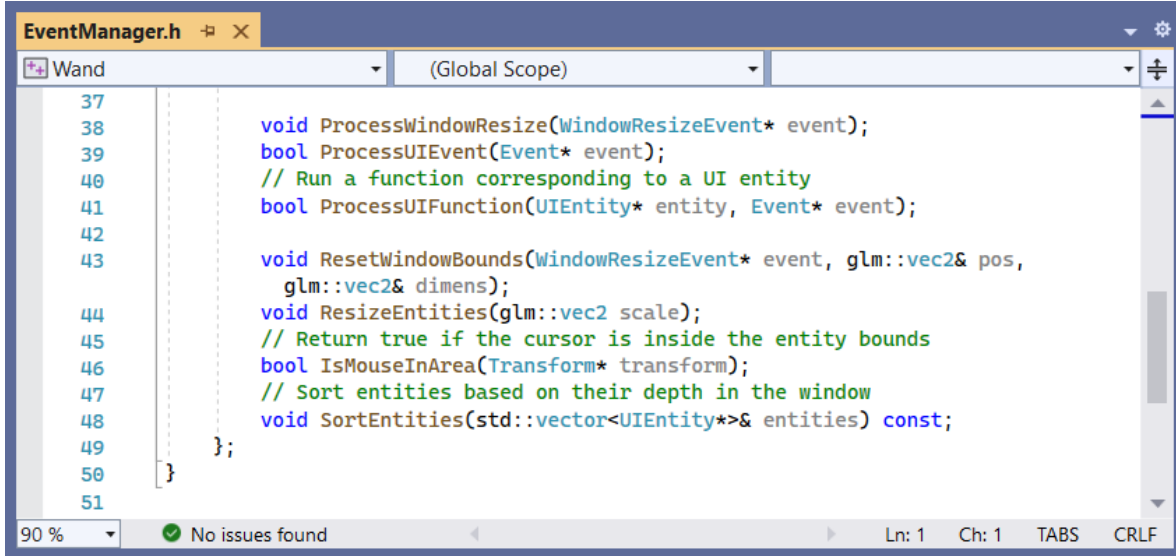
```
enum class EventCategory
{
    Window, Input
};

enum class EventType
{
    WindowResize, WindowClose,
    KeyRelease, KeyPress, MouseButtonRelease, MouseButtonPress,
    MouseScrollX, MouseScrollY, MouseMove
};
```

Events that belong to the category *Input* have their mouse and keyboard data saved in the *Input* class from which the game programmer can later request the current state. *Window* Events can be used to adjust the window's aspect ratio after resizing, change the scale of the entities displayed, or notify the engine that the application is about to close. The class that keeps track of Events is called *EventManager*, and it is responsible of notifying and altering all the interested components of the engine every time some change is detected on the window.

EventManager.h

```
1  #pragma once
2
3  #include "Event.h"
4  #include "UI/UIEntity.h"
5  #include "Core/Window.h"
6  #include "Input/Input.h"
7  #include "Graphics/Renderer.h"
8  #include "Graphics/Base/Transform.h"
9  #include "Core/CursorManager.h"
10
11 namespace wand
12 {
13     class EventManager
14     {
15     public:
16         EventManager();
17         void Init(Window* window, Input* input, Renderer* renderer,
18             CursorManager* cursorManager);
19         // Clear the events of the last frame
20         void Clear();
21         // Save the entities for this frame
22         void SetEntities(std::vector<std::unique_ptr<UIEntity>>& entities);
23         // Handle window and input events
24         void HandleEvent(Event* event);
25     private:
26         std::vector<UIEntity*> mEntities;
27         std::vector<UIEntity*> mActiveEntities;
28         Window* mWindow;
29         Input* mInput;
30         Renderer* mRenderer;
31         CursorManager* mCursorManager;
32         double mXPos;
33         double mYPos;
34
35         void HandleWindowEvent(Event* windowEvent);
36         void HandleInputEvent(Event* inputEvent);
37     }
```



```
EventManager.h
Wand (Global Scope)
37
38 void ProcessWindowResize(WindowResizeEvent* event);
39 bool ProcessUIEvent(Event* event);
40 // Run a function corresponding to a UI entity
41 bool ProcessUIFunction(UIEntity* entity, Event* event);
42
43 void ResetWindowBounds(WindowResizeEvent* event, glm::vec2& pos,
44                       glm::vec2& dims);
45 void ResizeEntities(glm::vec2 scale);
46 // Return true if the cursor is inside the entity bounds
47 bool IsMouseInArea(Transform* transform);
48 // Sort entities based on their depth in the window
49 void SortEntities(std::vector<UIEntity*>& entities) const;
50 };
51
```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF

4.2.3. Input Classes

As it was explained in the previous sections, the window detects any interaction of the player with the user interface and sends the appropriate event to the engine. Those of the category *Input* are related to mouse and keyboard events. The most recent ones are temporarily stored in the *Input* class from which the game developer may request the current state at any given time. This class is being kept up to date by the *EventManager* which sends it any new input-related changes every frame.

Input.h

```

1  #pragma once
2
3  #include "Events/Event.h"
4  #include "InputMacros.h"
5
6  namespace wand
7  {
8      class Input
9      {
10     public:
11         Input();
12
13         /* Search this frame's events for a specific type of input */
14         bool MouseMoved() const;
15         int GetX() const;
16         int GetY() const;
17         bool KeyPressed(int key) const;
18         bool KeyReleased(int key) const;
19         bool MouseButtonPressed(int button) const;
20         bool MouseButtonReleased(int button) const;
21         bool ScrollX() const;
22         bool ScrollY() const;
23         float GetScrollX() const;
24         float GetScrollY() const;
25
26         // Clear the last frame's events
27         void ClearEvents();
28         // Add a new event for this frame
29         void AddEvent(Event* event);
30         // Set the latest mouse position
31         void SetMousePos(double xPos, double yPos);
32
33     private:
34         std::vector<std::unique_ptr<Event>> mEvents;
35         double mXPos;
36         double mYPos;
37     };
38 }

```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF

4.2.4. Graphics Classes

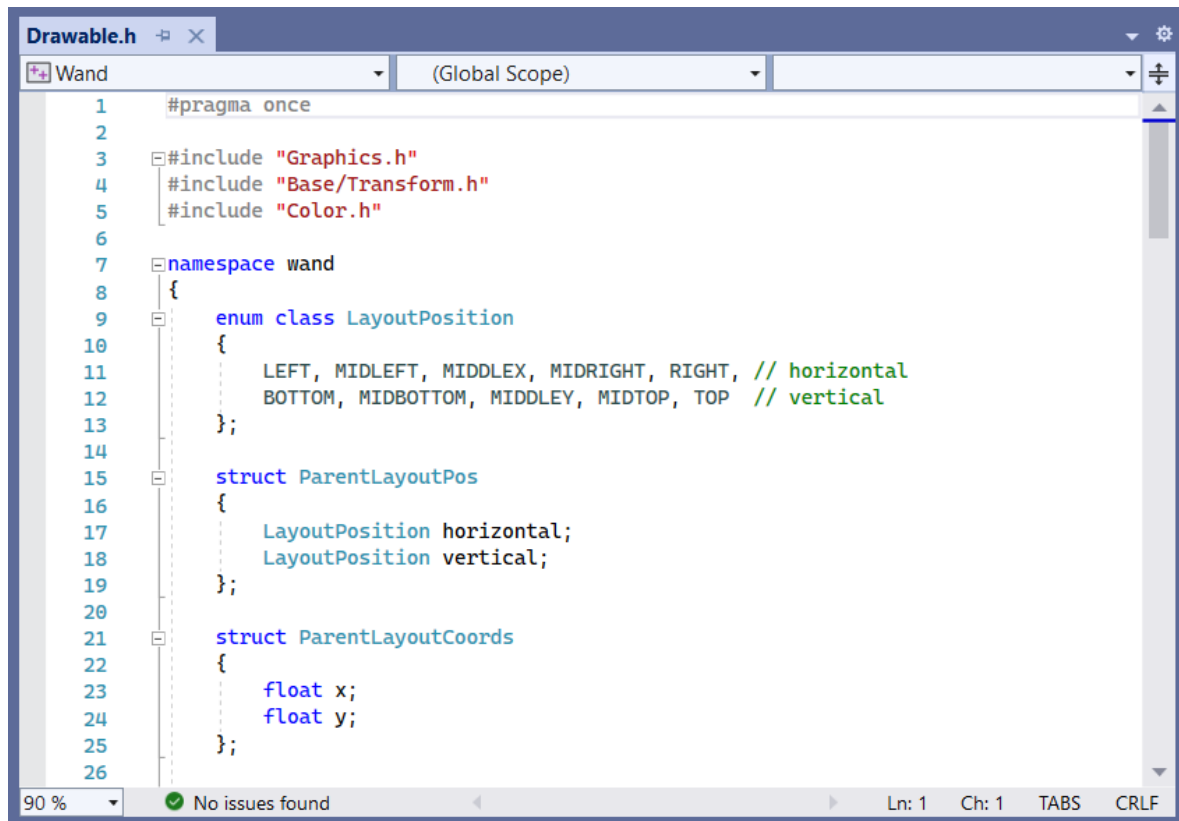
Graphics is a substantial component of a game engine. The classes in this category contain all the necessary data for rendering rectangles, sprites, and text, as these are the main building blocks of all the UI entities in Wand Engine. These three types of 2D graphics are

also considered *drawables* and their data is sent to the GPU for rendering once per frame— unless the window is resized, in which case they are rendered again.

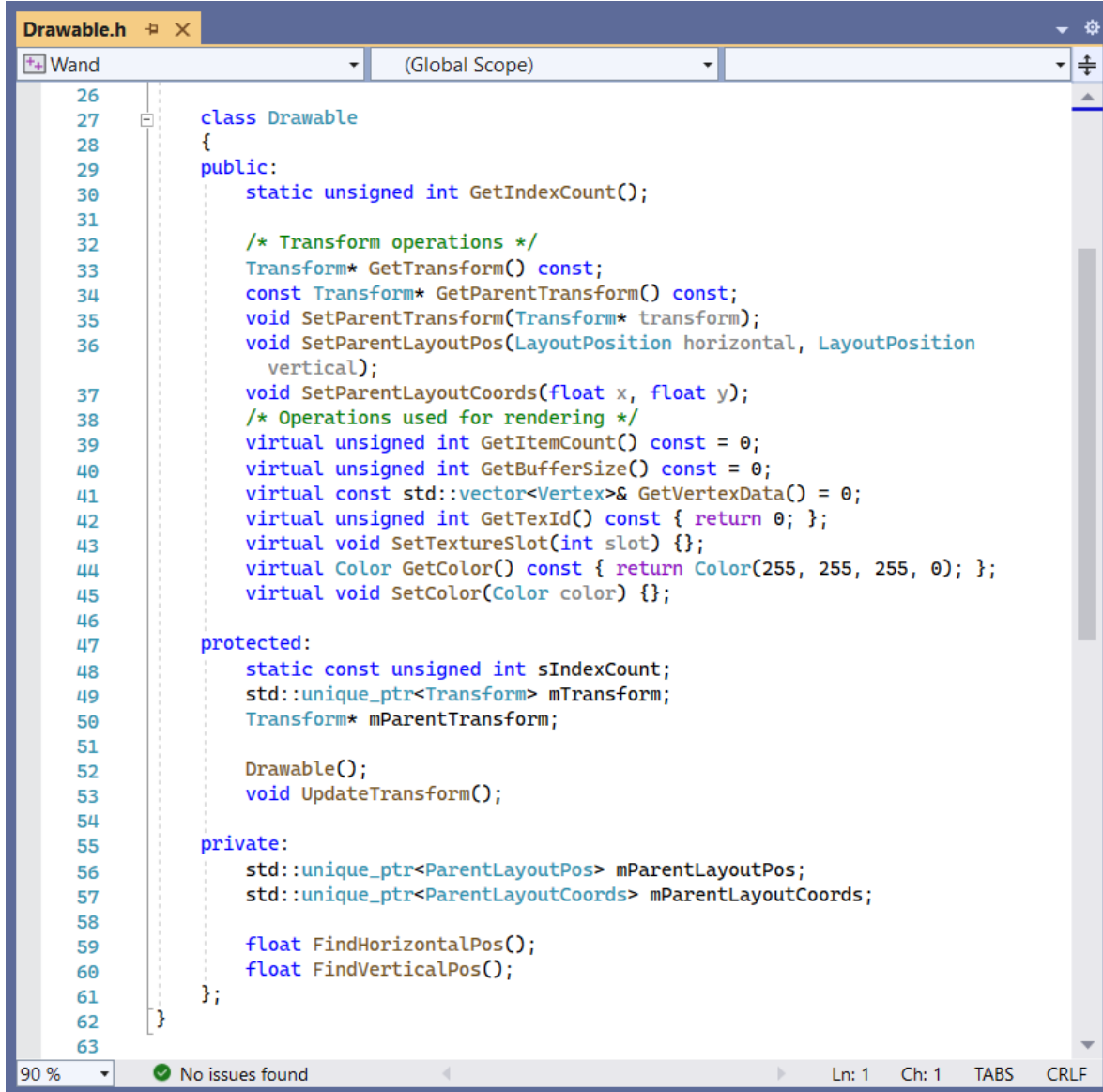
Some of the most important classes in this category is the *Drawable* from which rectangles, sprites, and text inherit their common features, the *FontManager* that saves and searches for different *Fonts* with different names and sizes, the *Transform* which contains position, size, and rotation data for a single drawable, and the *Renderer* that collects all the relevant information in order to display a set of drawables to the screen.

It is also important to note that the Graphics directory contains many graphics API-specific classes, such as *VertexArray*, *VertexBuffer*, *IndexBuffer*, *Texture*, and *ShaderProgram*. Lastly, there are two additional files for the vertex and fragment shader code that is required by *ShaderProgram*. These two files contain instructions to the GPU about how to determine the final position of each vertex on the window and the final pixel color respectively.

Drawable.h



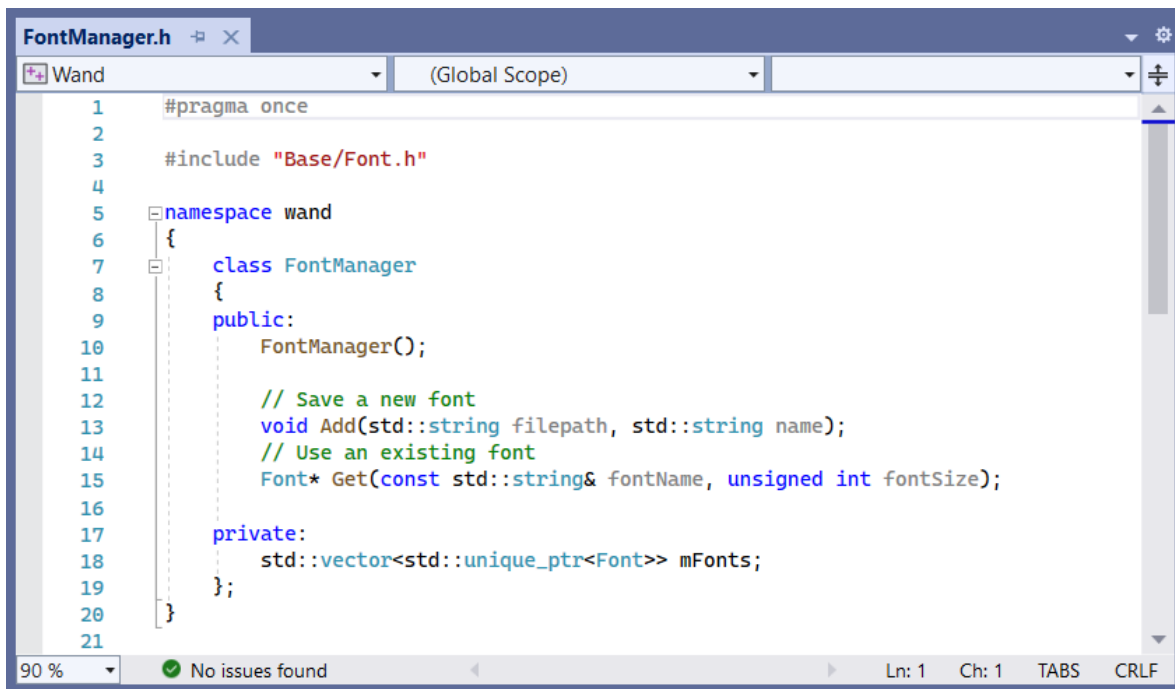
```
1  #pragma once
2
3  #include "Graphics.h"
4  #include "Base/Transform.h"
5  #include "Color.h"
6
7  namespace wand
8  {
9      enum class LayoutPosition
10     {
11         LEFT, MIDDLEFT, MIDDLEX, MIDRIGHT, RIGHT, // horizontal
12         BOTTOM, MIDBOTTOM, MIDDLEY, MIDTOP, TOP // vertical
13     };
14
15     struct ParentLayoutPos
16     {
17         LayoutPosition horizontal;
18         LayoutPosition vertical;
19     };
20
21     struct ParentLayoutCoords
22     {
23         float x;
24         float y;
25     };
26 }
```



```
26
27 class Drawable
28 {
29 public:
30     static unsigned int GetIndexCount();
31
32     /* Transform operations */
33     Transform* GetTransform() const;
34     const Transform* GetParentTransform() const;
35     void SetParentTransform(Transform* transform);
36     void SetParentLayoutPos(LayoutPosition horizontal, LayoutPosition
        vertical);
37     void SetParentLayoutCoords(float x, float y);
38     /* Operations used for rendering */
39     virtual unsigned int GetItemCount() const = 0;
40     virtual unsigned int GetBufferSize() const = 0;
41     virtual const std::vector<Vertex>& GetVertexData() = 0;
42     virtual unsigned int GetTexId() const { return 0; };
43     virtual void SetTextureSlot(int slot) {};
44     virtual Color GetColor() const { return Color(255, 255, 255, 0); };
45     virtual void SetColor(Color color) {};
46
47 protected:
48     static const unsigned int sIndexCount;
49     std::unique_ptr<Transform> mTransform;
50     Transform* mParentTransform;
51
52     Drawable();
53     void UpdateTransform();
54
55 private:
56     std::unique_ptr<ParentLayoutPos> mParentLayoutPos;
57     std::unique_ptr<ParentLayoutCoords> mParentLayoutCoords;
58
59     float FindHorizontalPos();
60     float FindVerticalPos();
61 };
62
63
```

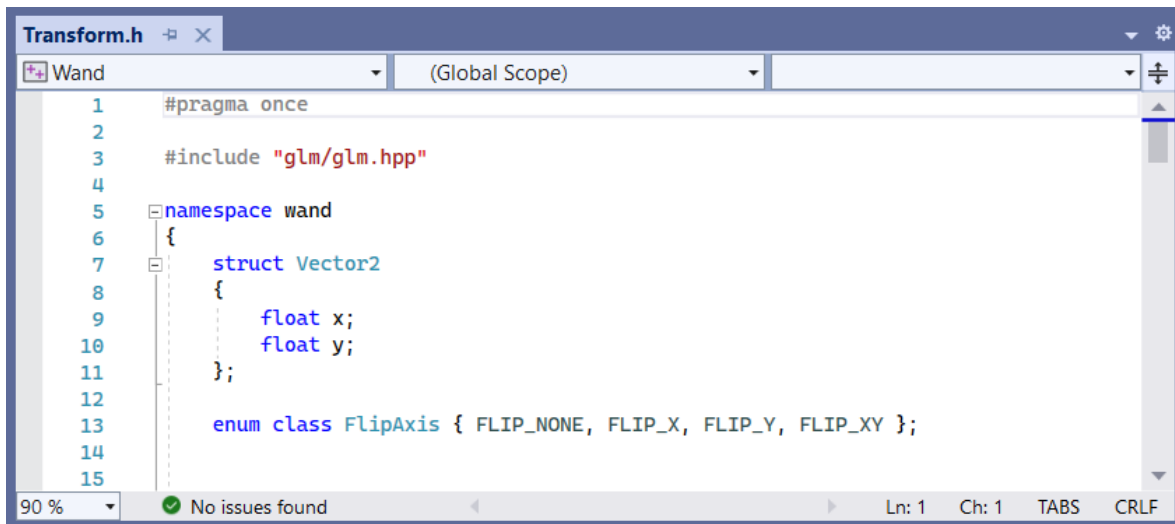
90 % No issues found Ln: 1 Ch: 1 TABS CRLF

FontManager.h

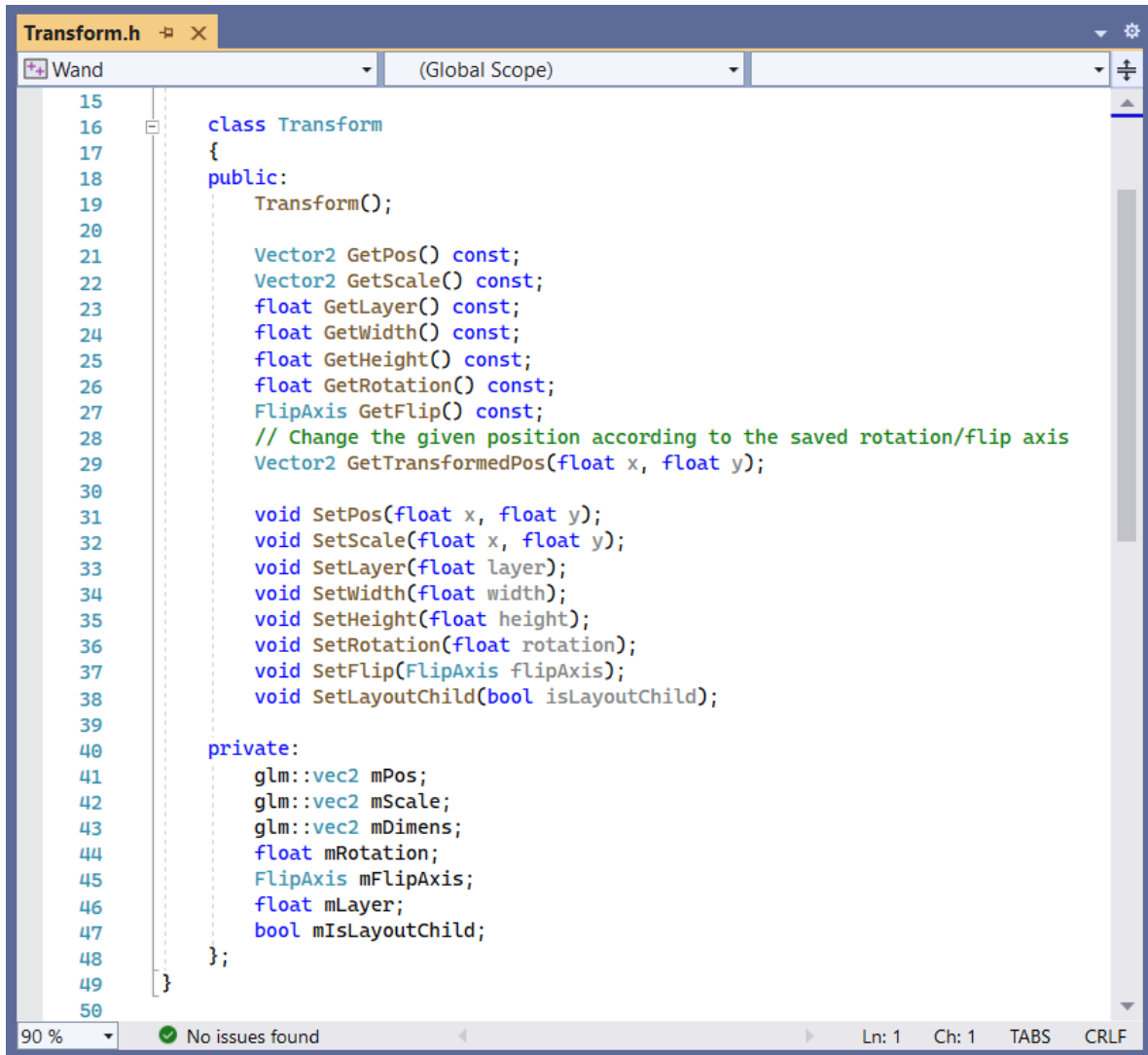


```
1  #pragma once
2
3  #include "Base/Font.h"
4
5  namespace wand
6  {
7      class FontManager
8      {
9      public:
10         FontManager();
11
12         // Save a new font
13         void Add(std::string filepath, std::string name);
14         // Use an existing font
15         Font* Get(const std::string& fontName, unsigned int fontSize);
16
17     private:
18         std::vector<std::unique_ptr<Font>> mFonts;
19     };
20 }
21
```

Transform.h



```
1  #pragma once
2
3  #include "glm/glm.hpp"
4
5  namespace wand
6  {
7      struct Vector2
8      {
9          float x;
10         float y;
11     };
12
13     enum class FlipAxis { FLIP_NONE, FLIP_X, FLIP_Y, FLIP_XY };
14
15 }
```



```
15
16 class Transform
17 {
18 public:
19     Transform();
20
21     Vector2 GetPos() const;
22     Vector2 GetScale() const;
23     float GetLayer() const;
24     float GetWidth() const;
25     float GetHeight() const;
26     float GetRotation() const;
27     FlipAxis GetFlip() const;
28     // Change the given position according to the saved rotation/flip axis
29     Vector2 GetTransformedPos(float x, float y);
30
31     void SetPos(float x, float y);
32     void SetScale(float x, float y);
33     void SetLayer(float layer);
34     void SetWidth(float width);
35     void SetHeight(float height);
36     void SetRotation(float rotation);
37     void SetFlip(FlipAxis flipAxis);
38     void SetLayoutChild(bool isLayoutChild);
39
40 private:
41     glm::vec2 mPos;
42     glm::vec2 mScale;
43     glm::vec2 mDimens;
44     float mRotation;
45     FlipAxis mFlipAxis;
46     float mLayer;
47     bool mIsLayoutChild;
48 };
49
50
```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF

Renderer.h

```

1  #pragma once
2
3  #include "Base/VertexArray.h"
4  #include "Base/VertexBuffer.h"
5  #include "Base/IndexBuffer.h"
6  #include "Base/ShaderProgram.h"
7  #include "Drawable.h"
8  #include "UI/UIEntity.h"
9
10 namespace wand
11 {
12     class Renderer
13     {
14     public:
15         Renderer();
16
17         void Init(float windowHeight, float windowWidth, std::string shaderPath);
18         // Adjust the projection matrix in the shader when the window is resized
19         void ResetProjectionMatrix(float xMin, float yMin, float xMax, float yMax);
20         // Submit the entities that should be considered for rendering in this frame
21         void Submit(std::vector<std::unique_ptr<UIEntity>>& entities);
22
23     private:
24         // Variables
25         std::unique_ptr<VertexArray> mVAO;
26         std::unique_ptr<VertexBuffer> mVBO;
27         std::unique_ptr<IndexBuffer> mIBO;
28         std::unique_ptr<ShaderProgram> mShaderProgram;
29         std::array<int, MAX_TEXTURES> mSavedTexSlots;
30         std::vector<Drawable*> mRenderQueue;
31
32         // Private Methods
33         void SetupBuffers();
34         void SetupShaderProgram(float windowHeight, float windowWidth, std::string
35             shaderPath);
36         void SaveTextureSlot(Drawable* sprite, unsigned int& slotIndex);
37         void FillVertexBuffer(unsigned int& drawablesInBuffer, unsigned int&
38             itemsInBuffer);
39         void Render();
40     };
41 }

```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF

Standard.vert – Vertex Shader

```
Standard.vert  ⇄ ×
1  #version 330 core
2
3  layout(location = 0) in vec3 position;
4  layout(location = 1) in vec4 color;
5  layout(location = 2) in vec2 texCoords;
6  layout(location = 3) in float texSlot;
7  layout(location = 4) in float isText;
8  out vec4 vColor;
9  out vec2 vTexCoords;
10 out float vTexSlot;
11 out float vIsText;
12
13 uniform mat4 uProjection;
14
15 void main()
16 {
17     gl_Position = uProjection * vec4(position, 1.0);
18     vColor = color;
19     vTexCoords = texCoords;
20     vTexSlot = texSlot;
21     vIsText = isText;
22 }
23
90 %  No issues found  Ln: 1  Ch: 1  TABS  CRLF
```

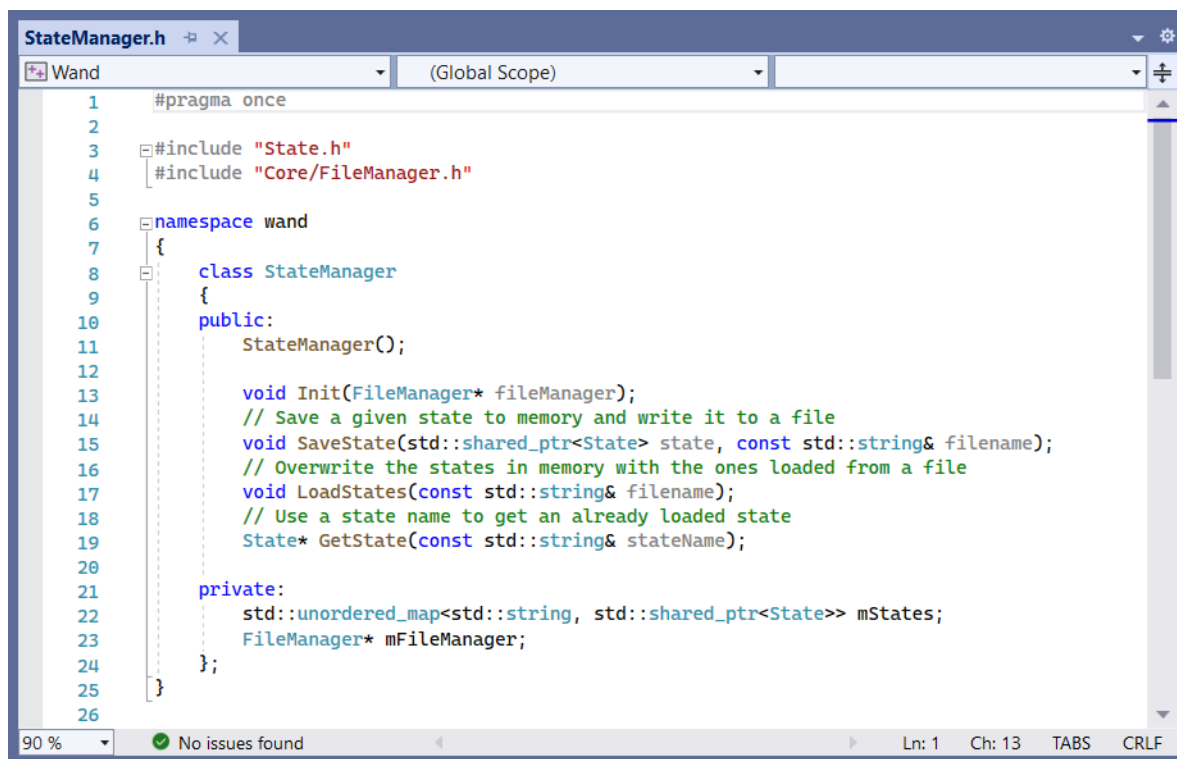
Standard.frag – Fragment Shader

```
Standard.frag  ⇄ ×
1  #version 330 core
2
3  in vec4 vColor;
4  in vec2 vTexCoords;
5  in float vTexSlot;
6  in float vIsText;
7  out vec4 color;
8
9  uniform sampler2D uTexSlots[16];
10
11 void main()
12 {
13     if (vIsText == 0.0)
14     {
15         color = texture(uTexSlots[int(vTexSlot)], vTexCoords) * vColor;
16     }
17     else
18     {
19         color = vec4(vColor.rgb, texture(uTexSlots[int(vTexSlot)], vTexCoords).r *
20             vColor.a);
21     }
22 };
23
90 %  No issues found  Ln: 1  Ch: 1  SPC  CRLF
```

4.2.5. State Classes

The game state in Wand Engine is saved in JSON format in text files on the disk. A single state can include a series of key-value pairs, corresponding to the name of the variable that is being saved and its respective value. The data types allowed to be saved in a pair are integers, doubles, bools, and strings. Although there can be multiple states in a game, a new state will overwrite an old one if they both have the same name.

StateManager.h



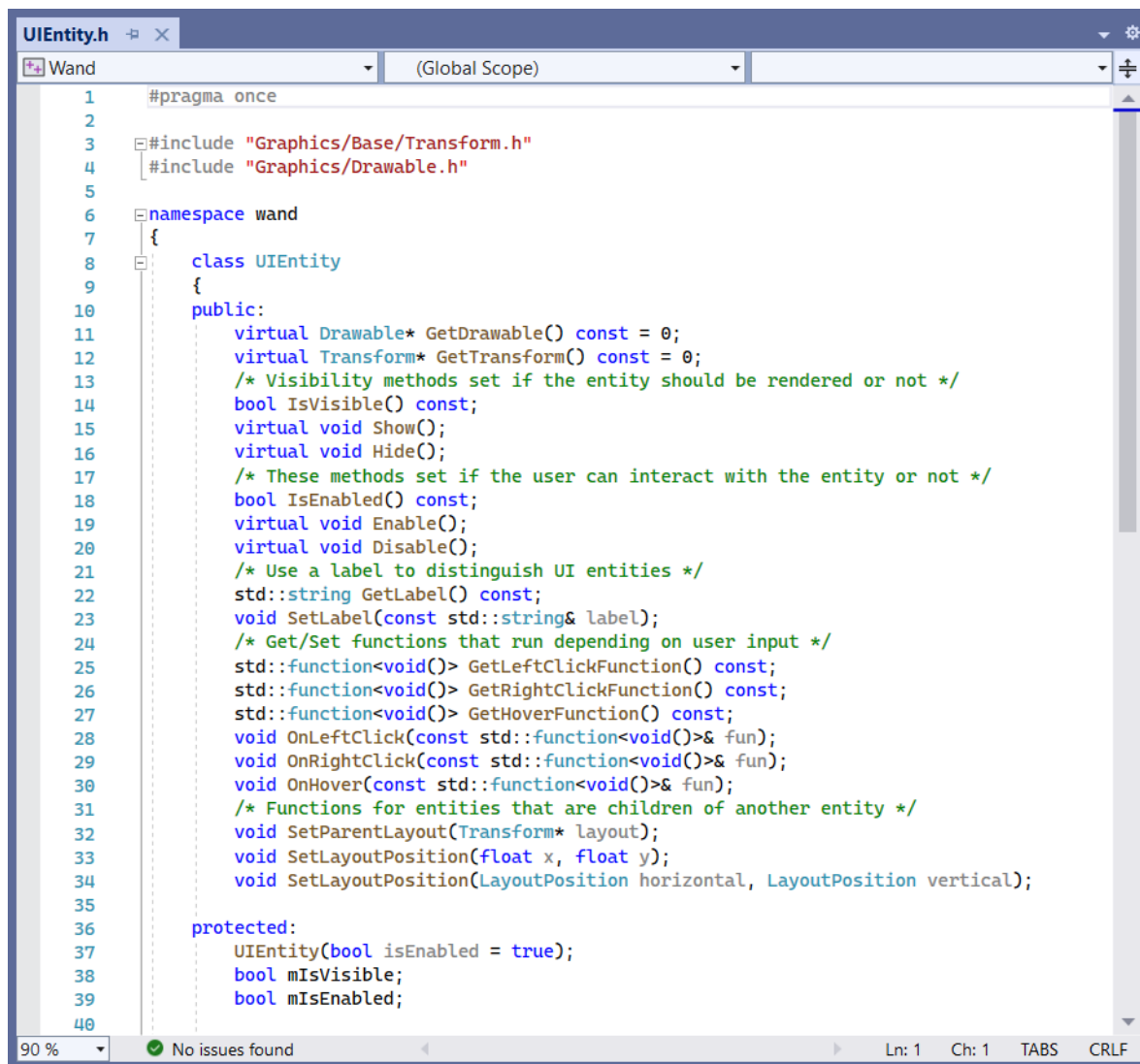
```
1  #pragma once
2
3  #include "State.h"
4  #include "Core/FileManager.h"
5
6  namespace wand
7  {
8      class StateManager
9      {
10     public:
11         StateManager();
12
13         void Init(FileManager* fileManager);
14         // Save a given state to memory and write it to a file
15         void SaveState(std::shared_ptr<State> state, const std::string& filename);
16         // Overwrite the states in memory with the ones loaded from a file
17         void LoadStates(const std::string& filename);
18         // Use a state name to get an already loaded state
19         State* GetState(const std::string& stateName);
20
21     private:
22         std::unordered_map<std::string, std::shared_ptr<State>> mStates;
23         FileManager* mFileManager;
24     };
25
26 }
```

4.2.6. UI Classes

As it was already mentioned, drawables are the building blocks of UI entities. The former include all the necessary information for graphics to be drawn to the window, whereas the latter are the objects that the player interacts with. One or more drawables can be used for a single UI entity, depending on its complexity. In addition, every UI entity contains functions that run when the user clicks or hovers over them and optional labels that can be

used by the programmer in order to distinguish between them. These objects can also use other entities as their layouts and be positioned relative to them. They can be hidden, in which case they are not chosen for rendering, or disabled so that the player cannot interact with them.

UIEntity.h



```

1  #pragma once
2
3  #include "Graphics/Base/Transform.h"
4  #include "Graphics/Drawable.h"
5
6  namespace wand
7  {
8      class UIEntity
9      {
10     public:
11         virtual Drawable* GetDrawable() const = 0;
12         virtual Transform* GetTransform() const = 0;
13         /* Visibility methods set if the entity should be rendered or not */
14         bool IsVisible() const;
15         virtual void Show();
16         virtual void Hide();
17         /* These methods set if the user can interact with the entity or not */
18         bool IsEnabled() const;
19         virtual void Enable();
20         virtual void Disable();
21         /* Use a label to distinguish UI entities */
22         std::string GetLabel() const;
23         void SetLabel(const std::string& label);
24         /* Get/Set functions that run depending on user input */
25         std::function<void()> GetLeftClickFunction() const;
26         std::function<void()> GetRightClickFunction() const;
27         std::function<void()> GetHoverFunction() const;
28         void OnLeftClick(const std::function<void()& fun);
29         void OnRightClick(const std::function<void()& fun);
30         void OnHover(const std::function<void()& fun);
31         /* Functions for entities that are children of another entity */
32         void SetParentLayout(Transform* layout);
33         void SetLayoutPosition(float x, float y);
34         void SetLayoutPosition(LayoutPosition horizontal, LayoutPosition vertical);
35
36     protected:
37         UIEntity(bool isEnabled = true);
38         bool mIsVisible;
39         bool mIsEnabled;
40

```

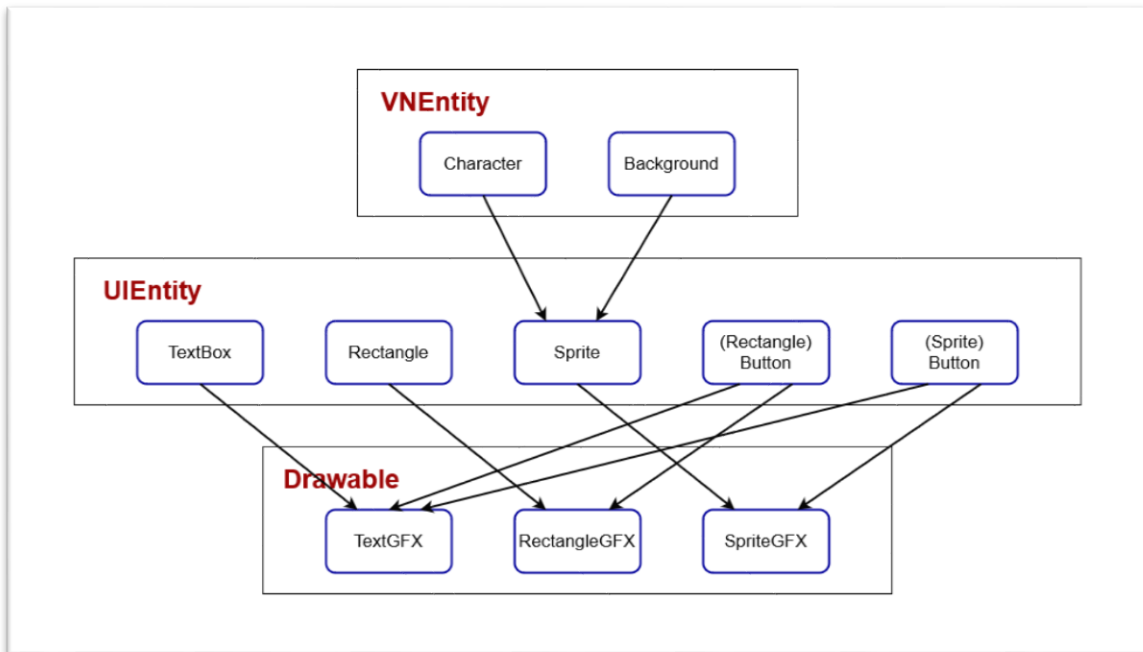
```

40
41     private:
42         std::function<void()> mLeftClickFunction;
43         std::function<void()> mRightClickFunction;
44         std::function<void()> mHoverFunction;
45         std::string mLabel;
46     };
47 }
48

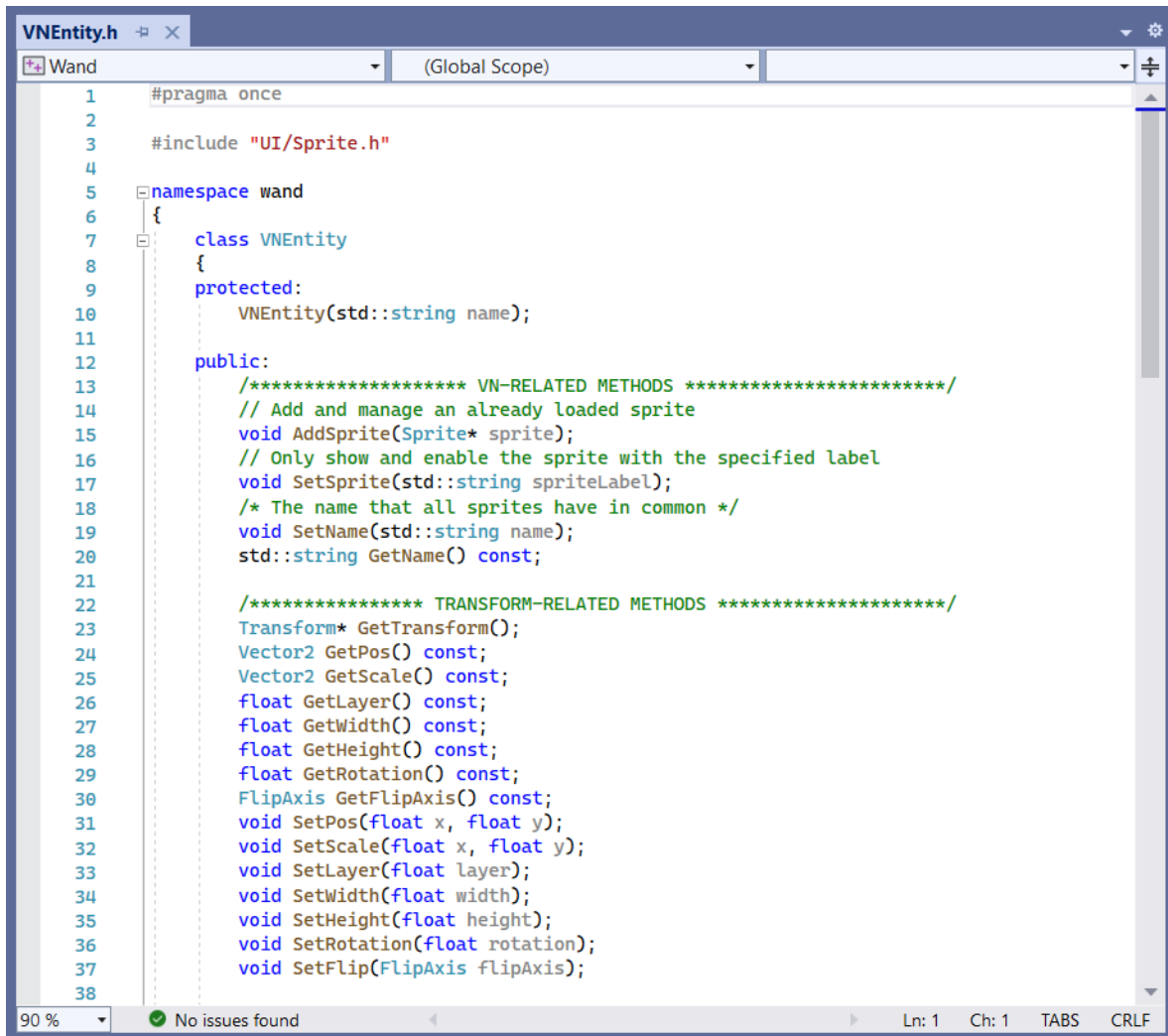
```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF

4.2.7. VN Classes

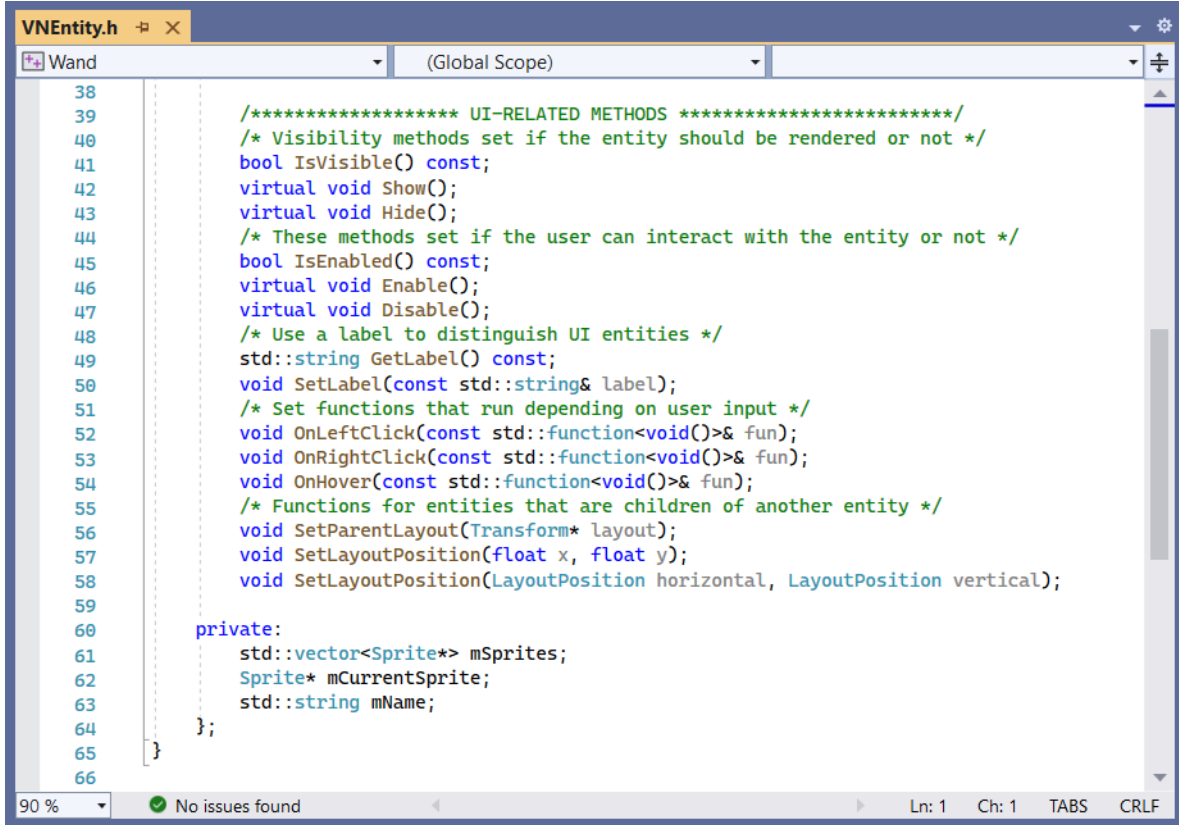


Visual Novel entities are constructs that manage and group together a set of sprites. There are two types of *VNEntities* that can be created in Wand Engine and these are either Characters or Backgrounds. Their purpose is to offer the developer an easy way to alternate between expressions for the same character or different backgrounds that belong in the same category. They function similarly to *UIEntities*, but require different labels so that every sprite has an original name within the group.

VNEntity.h

```
1 #pragma once
2
3 #include "UI/Sprite.h"
4
5 namespace wand
6 {
7     class VNEntity
8     {
9     protected:
10         VNEntity(std::string name);
11
12     public:
13         /***** VN-RELATED METHODS *****/
14         // Add and manage an already loaded sprite
15         void AddSprite(Sprite* sprite);
16         // Only show and enable the sprite with the specified label
17         void SetSprite(std::string spriteLabel);
18         /* The name that all sprites have in common */
19         void SetName(std::string name);
20         std::string GetName() const;
21
22         /***** TRANSFORM-RELATED METHODS *****/
23         Transform* GetTransform();
24         Vector2 GetPos() const;
25         Vector2 GetScale() const;
26         float GetLayer() const;
27         float GetWidth() const;
28         float GetHeight() const;
29         float GetRotation() const;
30         FlipAxis GetFlipAxis() const;
31         void SetPos(float x, float y);
32         void SetScale(float x, float y);
33         void SetLayer(float layer);
34         void SetWidth(float width);
35         void SetHeight(float height);
36         void SetRotation(float rotation);
37         void SetFlip(FlipAxis flipAxis);
38     }
```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF



```

38
39      /****** UI-RELATED METHODS *****/
40      /* Visibility methods set if the entity should be rendered or not */
41      bool IsVisible() const;
42      virtual void Show();
43      virtual void Hide();
44      /* These methods set if the user can interact with the entity or not */
45      bool IsEnabled() const;
46      virtual void Enable();
47      virtual void Disable();
48      /* Use a label to distinguish UI entities */
49      std::string GetLabel() const;
50      void SetLabel(const std::string& label);
51      /* Set functions that run depending on user input */
52      void OnLeftClick(const std::function<void()>& fun);
53      void OnRightClick(const std::function<void()>& fun);
54      void OnHover(const std::function<void()>& fun);
55      /* Functions for entities that are children of another entity */
56      void SetParentLayout(Transform* layout);
57      void SetLayoutPosition(float x, float y);
58      void SetLayoutPosition(LayoutPosition horizontal, LayoutPosition vertical);
59
60 private:
61     std::vector<Sprite*> mSprites;
62     Sprite* mCurrentSprite;
63     std::string mName;
64 };
65
66

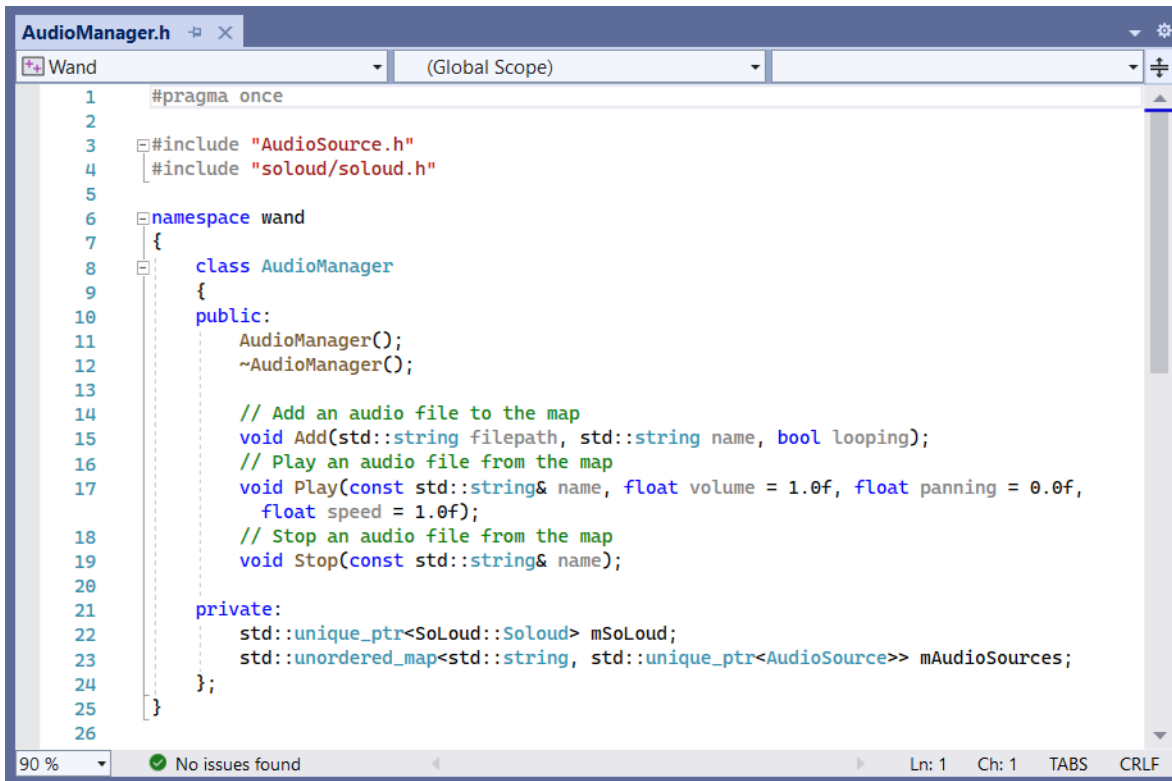
```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF

4.2.8. Audio Classes

When the game developer first loads an audio file to the engine, they need to set a name and whether the playback should be looping or not. This name can then be used to start or stop the playback. Optionally, the developer can also choose to set a custom volume, panning, and speed when the audio source begins playing.

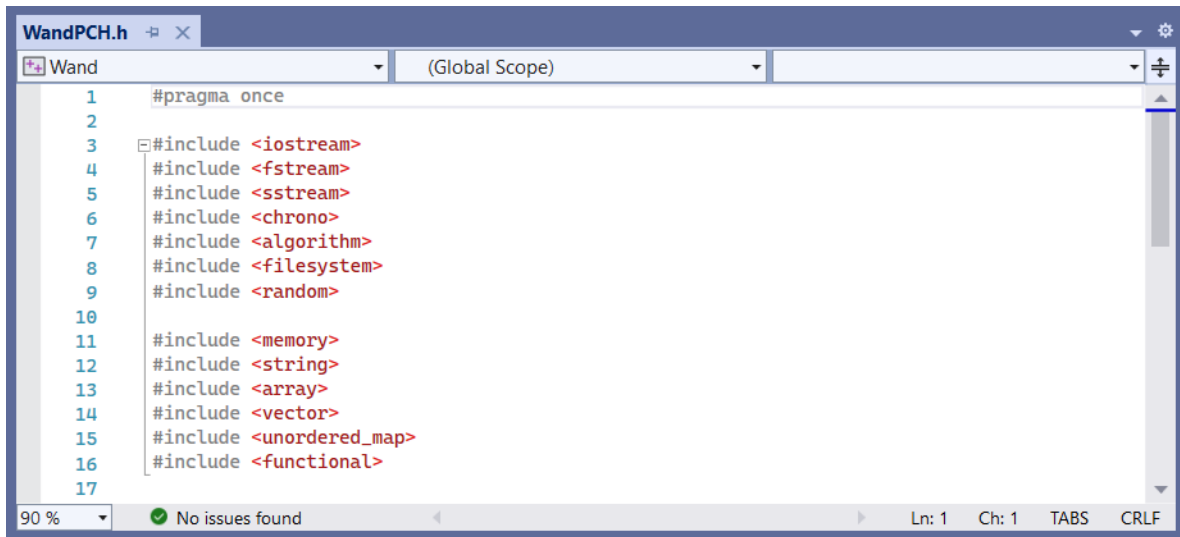
AudioManager.h



```
1 #pragma once
2
3 #include "AudioSource.h"
4 #include "soLoud/soLoud.h"
5
6 namespace wand
7 {
8     class AudioManager
9     {
10     public:
11         AudioManager();
12         ~AudioManager();
13
14         // Add an audio file to the map
15         void Add(std::string filepath, std::string name, bool looping);
16         // Play an audio file from the map
17         void Play(const std::string& name, float volume = 1.0f, float panning = 0.0f,
18                 float speed = 1.0f);
19         // Stop an audio file from the map
20         void Stop(const std::string& name);
21     private:
22         std::unique_ptr<SoLoud::SoLoud> mSoLoud;
23         std::unordered_map<std::string, std::unique_ptr<AudioSource>> mAudioSources;
24     };
25 }
26
```

4.2.9. Precompiled Headers

Precompiled headers is a technique that reduces compilation time and is often used in large projects. Long header files that are unlikely to change during the development of an application can be added to a single header file called precompiled header. This type of file is processed faster and is only compiled when any of the headers it contains is modified. It should be included in every source file of the project and it gives access to all its headers. However, while it is advised to add mostly external libraries to a precompiled header, not everything should be included in it. Headers that are only used in a few files in the project's codebase should often be kept in the original file as this improves its readability. In such a case, it would be easy to deduce which library is used in this particular file and how it relates to the developer's code.

WandPCH.h

```
WandPCH.h
Wand (Global Scope)
1 #pragma once
2
3 #include <iostream>
4 #include <fstream>
5 #include <sstream>
6 #include <chrono>
7 #include <algorithm>
8 #include <filesystem>
9 #include <random>
10
11 #include <memory>
12 #include <string>
13 #include <array>
14 #include <vector>
15 #include <unordered_map>
16 #include <functional>
17
90 % No issues found Ln: 1 Ch: 1 TABS CRLF
```

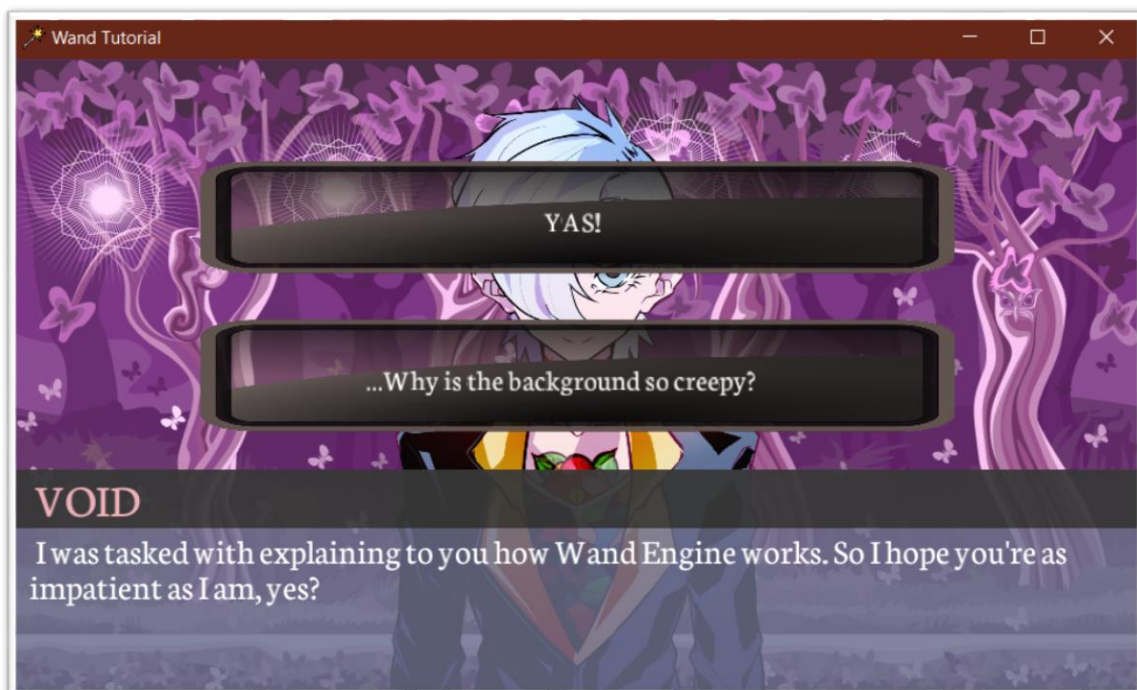
5. AN EXAMPLE GAME

5.1. Game Details

5.1.1. Summary

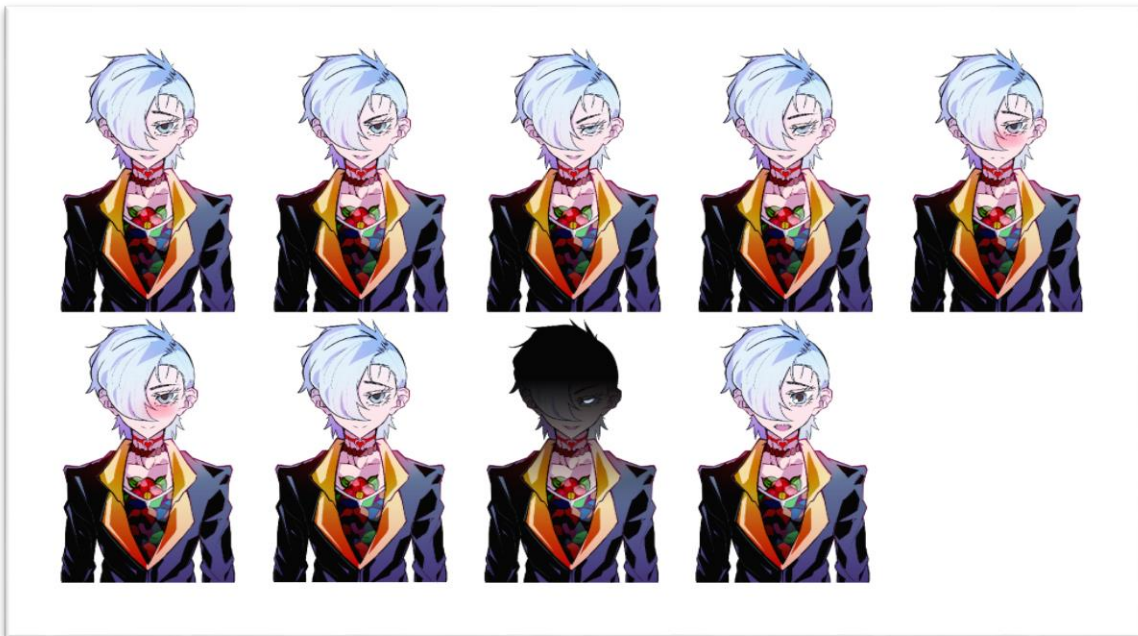
Wand Tutorial is a simple visual novel made with Wand. The game's main character, Void, explains the main features of the engine, including choice buttons, game states, audio, and positioning sprites on the screen. Depending on how the player reacts to Void, his likability increases or decreases in the duration of the game, and he will give different answers according to its level. The game runs on Windows x64 platforms and is developed in C++ with Visual Studio.

5.1.2. Graphics



As can be seen in the image above, Wand Tutorial contains many different elements. The semi-transparent grey rectangles, the textboxes on top of them, and the two buttons are all UIEntities. The main character as well as the background are VNEntities. The window title is set to 'Wand Tutorial' and its icon is set to a custom image.

The need to create a VNEntity becomes clear upon watching all the different expressions a character might have. Even though every image that is loaded to the engine is stored as a single sprite, it is often more convenient for the developer to refer to the character just by using their name. In order to alternate between the different expressions, it would then make sense to use the label of the appropriate sprite.



In the same logic, for a simple game, only a single background is needed. Backgrounds are usually rendered behind other UI entities and have the same width and height as the window. Therefore, every time the developer needs to move the plot to a different place, they would only need to choose the sprite with the appropriate label.

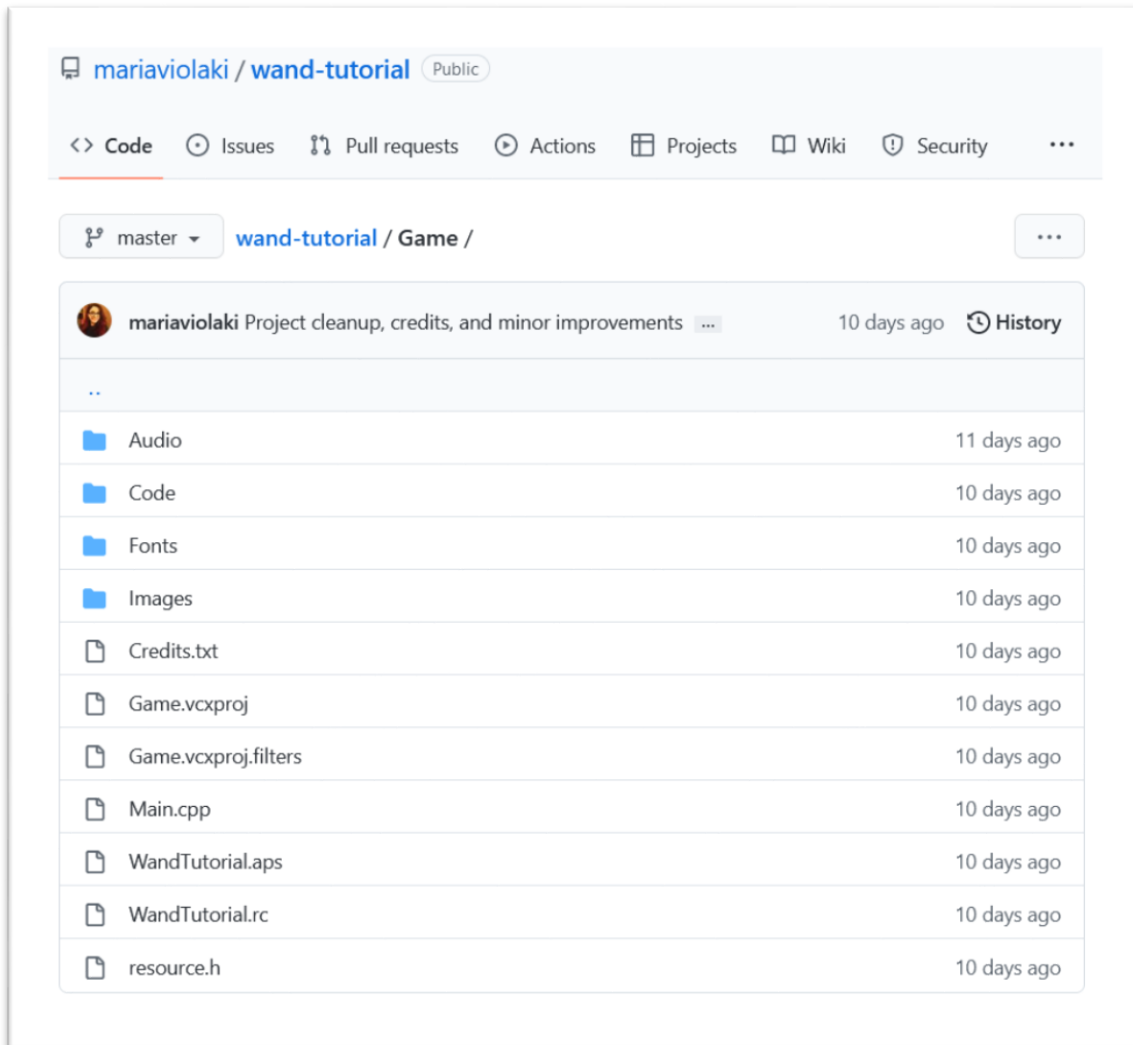


Images that have very few variations can be loaded as simple sprites. Except for the sprite-changing functionality, they have the same functions available and it is not necessary to set custom labels. Button images and random objects often belong in this category.



5.1.3. Project Structure

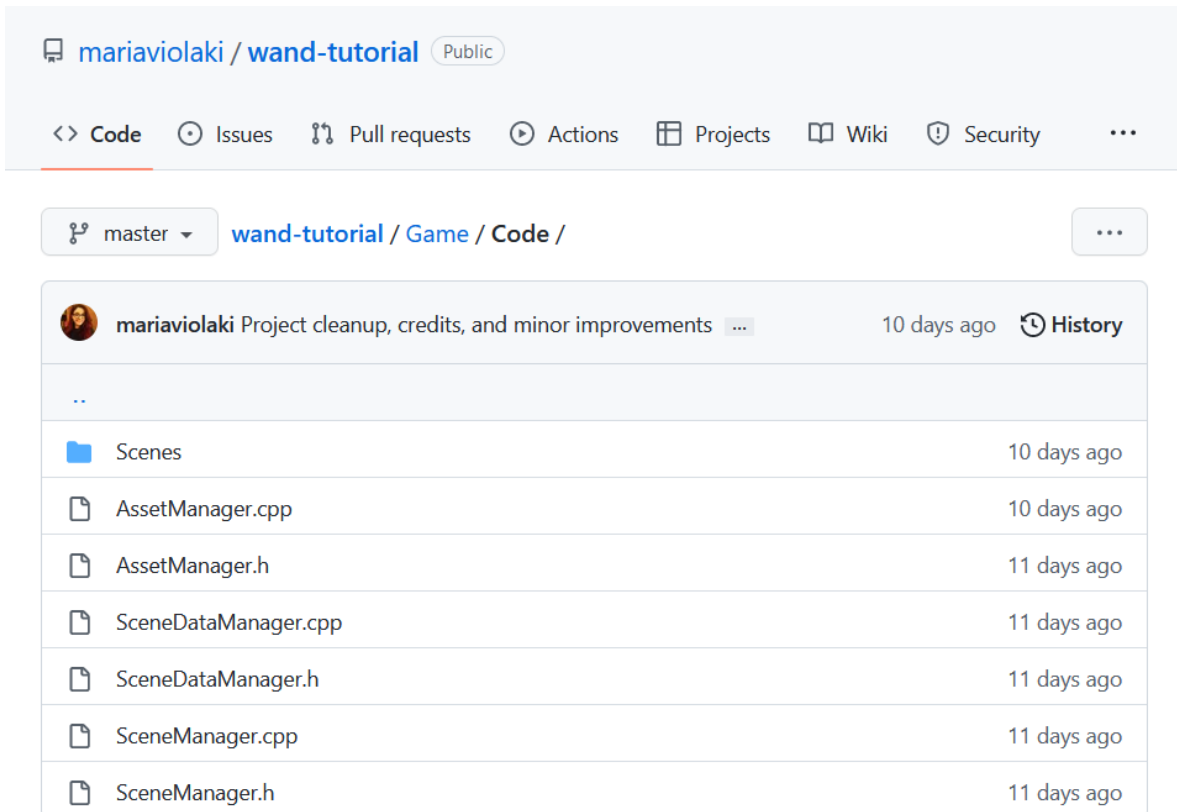
In order to create a game with Wand Engine, the developer would first have to download the *wand* repository from GitHub. Any changes to the already existing Visual Studio solution would then need to be made within the *Game* project. All the code, assets, and saved states should be able to be found in the respective folder. The online repository of the example game made with Wand can be found here: <https://github.com/mariaviolaki/wand-tutorial>



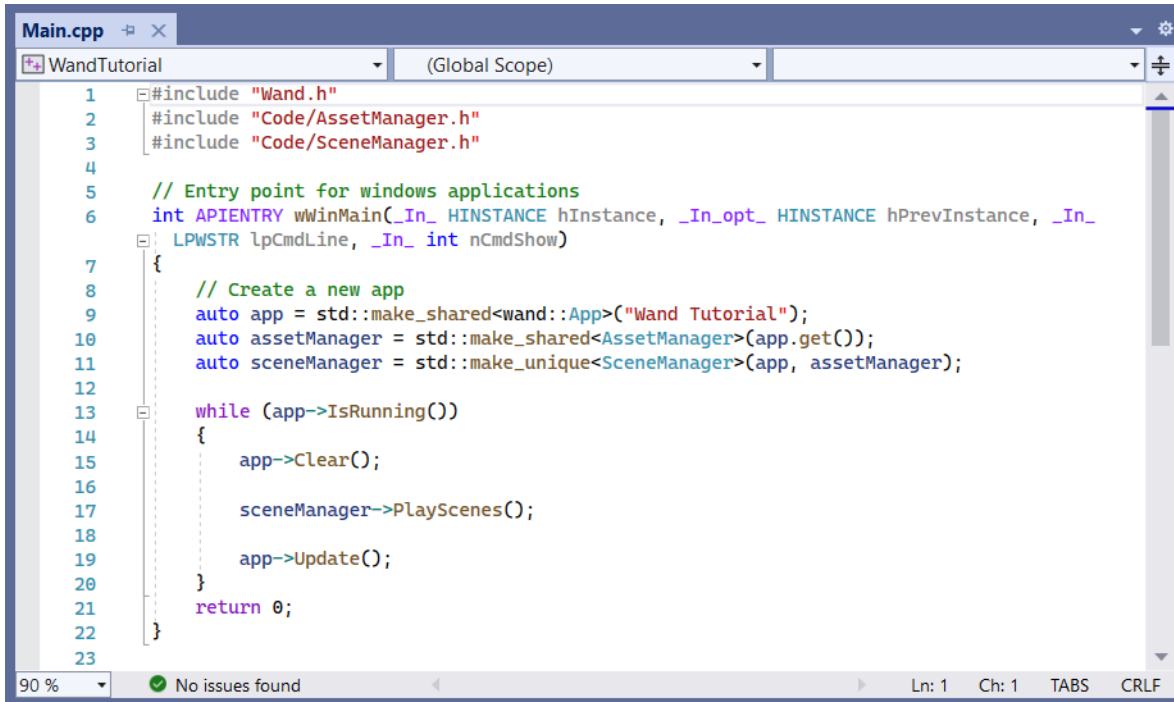
Most of the source and header files for *Wand Tutorial* can be found inside *Code*. *Audio*, *Fonts*, and *Images* are all directories generated by the engine automatically as it will search for any assets in these specific paths. The game developer can request these paths from the *App's* file manager. Similarly, even though it is not shown above, as soon as the game runs, one additional folder will be created inside *Game*: the folder *Saves*, where the engine will store any states saved at runtime. *Main.cpp* is the file from which the program's execution begins, and the last three files shown in the image are created by Visual Studio automatically when the developer sets a custom icon for the executable.

5.2. Main Classes and Functions

The main classes for *Wand Tutorial* is the *AssetManager* that is responsible of loading and initializing game assets, the *SceneDataManager* which saves and loads scene data in a single state, the *SceneManager* which plays all the scenes in the correct order and starts from the last saved one, as well as the *Scene*. *Scene* contains the information that all the different scenes have in common, including variables and functions.



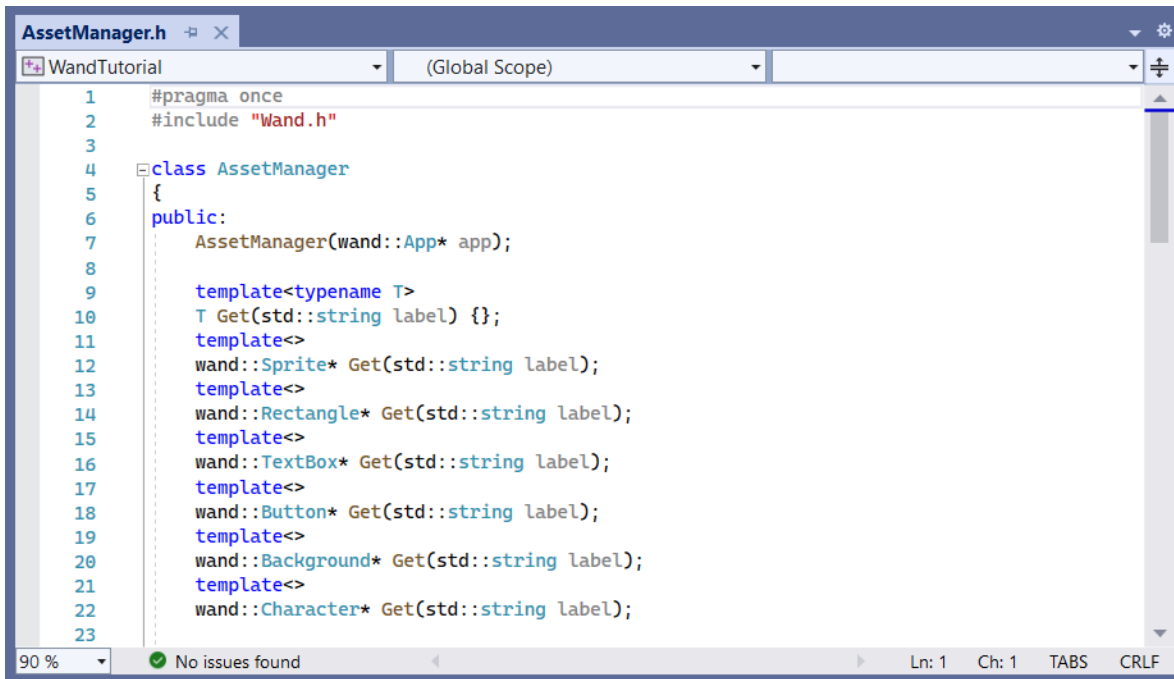
It is important to note at this point that the codebase for this game is only an example; a possible implementation using the tools that Wand Engine has to offer.

Main.cpp


```

1  #include "Wand.h"
2  #include "Code/AssetManager.h"
3  #include "Code/SceneManager.h"
4
5  // Entry point for windows applications
6  int APIENTRY wWinMain(_In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance, _In_
  LPWSTR lpCmdLine, _In_ int nCmdShow)
7  {
8      // Create a new app
9      auto app = std::make_shared<wand::App>("Wand Tutorial");
10     auto assetManager = std::make_shared<AssetManager>(app.get());
11     auto sceneManager = std::make_unique<SceneManager>(app, assetManager);
12
13     while (app->IsRunning())
14     {
15         app->Clear();
16
17         sceneManager->PlayScenes();
18
19         app->Update();
20     }
21     return 0;
22 }
23

```

AssetManager.h


```

1  #pragma once
2  #include "Wand.h"
3
4  class AssetManager
5  {
6  public:
7      AssetManager(wand::App* app);
8
9      template<typename T>
10     T Get(std::string label) {};
11     template<>
12     wand::Sprite* Get(std::string label);
13     template<>
14     wand::Rectangle* Get(std::string label);
15     template<>
16     wand::TextBox* Get(std::string label);
17     template<>
18     wand::Button* Get(std::string label);
19     template<>
20     wand::Background* Get(std::string label);
21     template<>
22     wand::Character* Get(std::string label);
23

```

```

AssetManager.h
WandTutorial (Global Scope)
23
24 private:
25     wand::App* mApp;
26     std::vector<wand::Sprite*> mSprites;
27     std::vector<wand::Rectangle*> mRectangles;
28     std::vector<wand::TextBox*> mTextBoxes;
29     std::vector<wand::Button*> mButtons;
30     std::vector<std::shared_ptr<wand::Background*>> mBackgrounds;
31     std::vector<std::shared_ptr<wand::Character*>> mCharacters;
32
33     // Load fonts
34     void LoadFonts();
35     // Load UI entities
36     void LoadSprites();
37     void LoadRectangles();
38     void LoadTextBoxes();
39     void LoadButtons();
40     // Load visual novel entities
41     void LoadBackgrounds();
42     void LoadCharacters();
43     // Load audio files
44     void LoadAudio();
45 };
46
90 % No issues found Ln: 1 Ch: 1 TABS CRLF

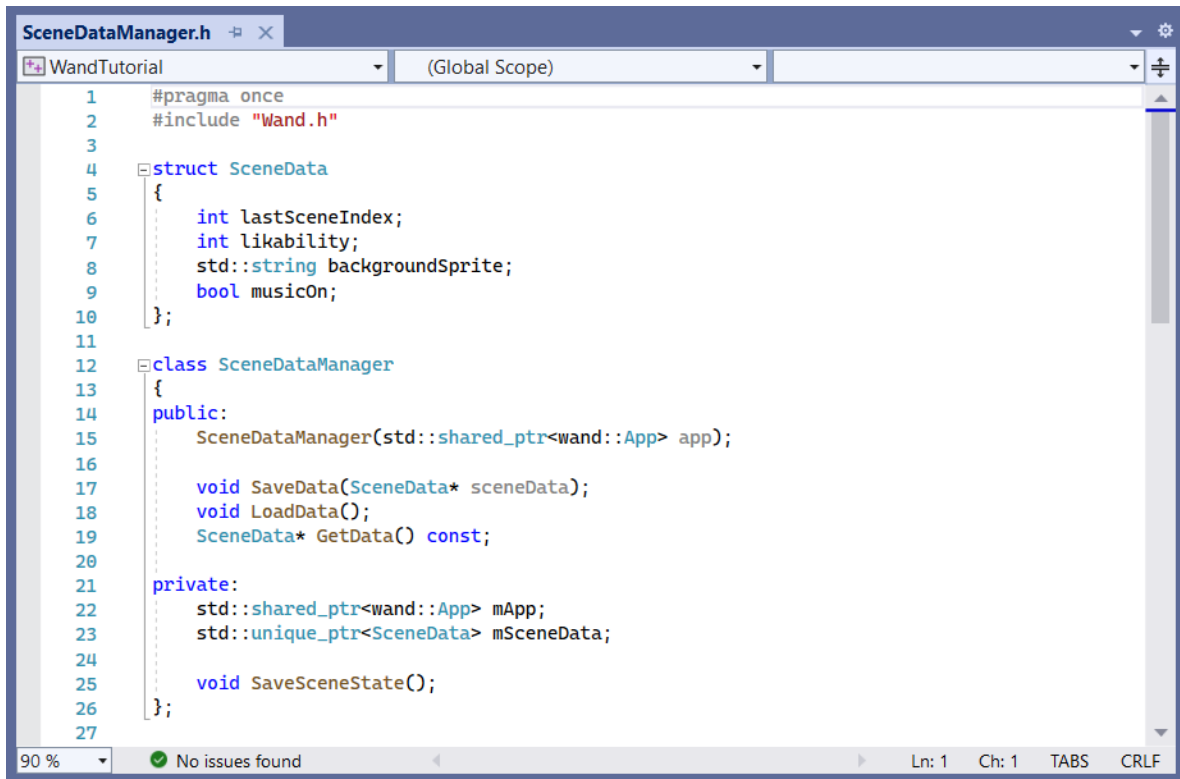
```

SceneManager.h

```

SceneManager.h
WandTutorial (Global Scope)
1 #pragma once
2 #include "Wand.h"
3 #include "AssetManager.h"
4 #include "SceneDataManager.h"
5 #include "Scenes/Scene.h"
6
7 class SceneManager
8 {
9 public:
10     SceneManager(std::shared_ptr<wand::App> app, std::shared_ptr<AssetManager>
        assetManager);
11
12     void PlayScenes();
13
14 private:
15     std::shared_ptr<wand::App> mApp;
16     std::shared_ptr<AssetManager> mAssetManager;
17     std::shared_ptr<SceneDataManager> mSceneDataManager;
18     std::unordered_map<unsigned int, std::unique_ptr<Scene*>> mScenes;
19     unsigned int mSceneIndex;
20 };
21
90 % No issues found Ln: 1 Ch: 1 TABS CRLF

```

SceneDataManager.h

```
1 #pragma once
2 #include "Wand.h"
3
4 struct SceneData
5 {
6     int lastSceneIndex;
7     int likability;
8     std::string backgroundImage;
9     bool musicOn;
10 };
11
12 class SceneDataManager
13 {
14 public:
15     SceneDataManager(std::shared_ptr<wand::App> app);
16
17     void SaveData(SceneData* sceneData);
18     void LoadData();
19     SceneData* GetData() const;
20
21 private:
22     std::shared_ptr<wand::App> mApp;
23     std::unique_ptr<SceneData> mSceneData;
24
25     void SaveSceneState();
26 };
27
```

90 % No issues found Ln: 1 Ch: 1 TABS CRLF

Scene.h

```

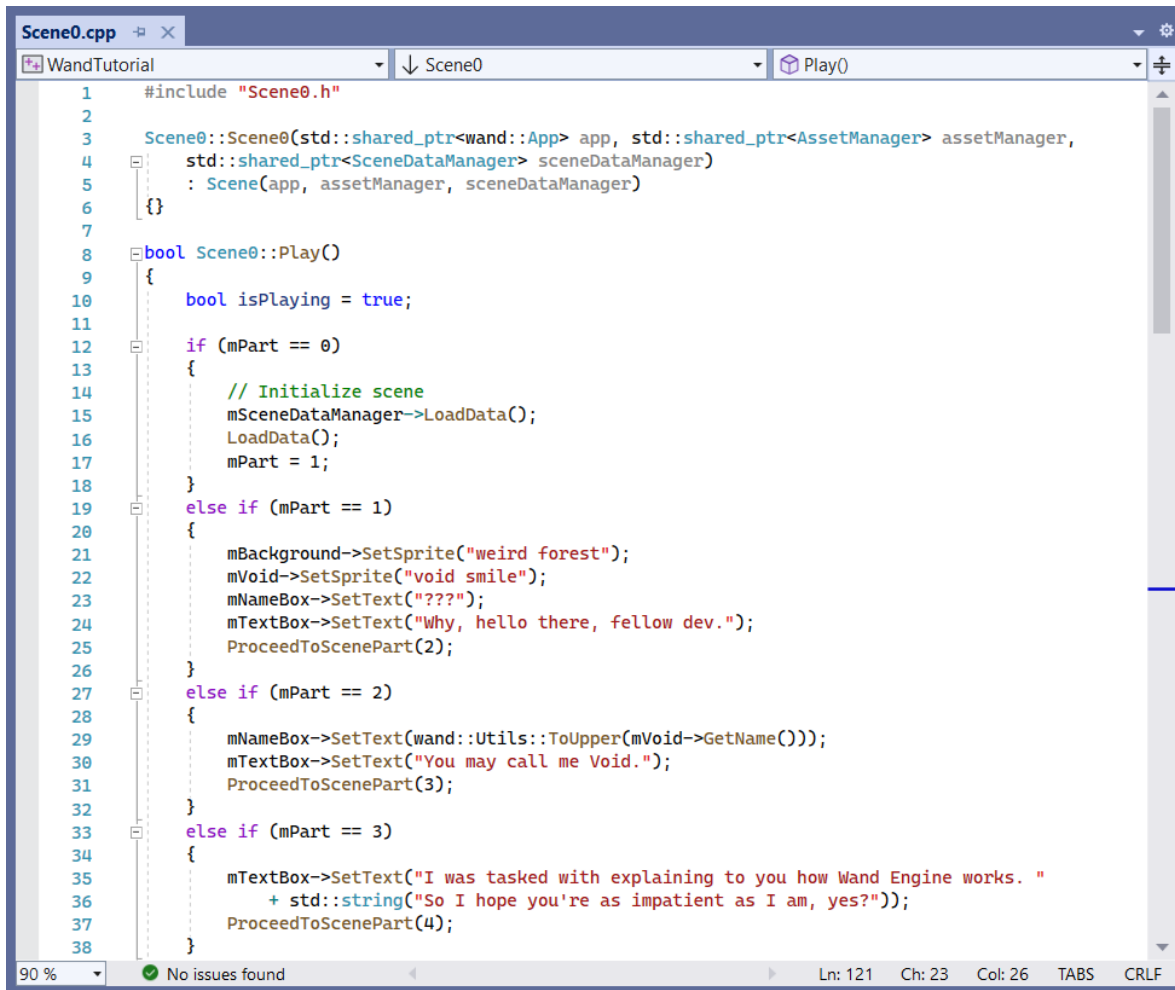
1 #pragma once
2 #include "Wand.h"
3 #include "../AssetManager.h"
4 #include "../SceneDataManager.h"
5
6 class Scene
7 {
8 public:
9     // Returns true when the scene is finished
10    virtual bool Play() = 0;
11    void ProceedToScenePart(unsigned int scenePart, bool playSound = false);
12    void ProceedAndPlaySound(unsigned int scenePart);
13
14 protected:
15    std::shared_ptr<wand::App> mApp;
16    std::shared_ptr<AssetManager> mAssetManager;
17    std::shared_ptr<SceneDataManager> mSceneDataManager;
18    unsigned int mPart; // the specific part of the scene
19    bool mIsDataSaved;
20    std::unique_ptr<SceneData> mSceneData;
21    // Assets that all scenes should have access to
22    wand::Rectangle* mChoiceButtonRect;
23    wand::TextBox* mNameBox;
24    wand::TextBox* mTextBox;
25    wand::Background* mBackground;
26    wand::Character* mVoid;
27    wand::Button* mChoiceButton1;
28    wand::Button* mChoiceButton2;
29    wand::Sprite* mBlob;
30
31    Scene(std::shared_ptr<wand::App> app, std::shared_ptr<AssetManager> assetManager,
32          std::shared_ptr<SceneDataManager> sceneDataManager);
33    void LoadData();
34    void SaveData();
35 };
36

```

5.3. Game Code Analysis

5.3.1. Full Scene Example

Due to the nature of the game genre which is centered around storytelling, the developer of a visual novel usually spends most of their time working on code that contains a lot of repetition. Therefore, it is crucial that the steps they need to follow are as clear and concise as possible.

Scene0.cpp

```
1 #include "Scene0.h"
2
3 Scene0::Scene0(std::shared_ptr<wand::App> app, std::shared_ptr<AssetManager> assetManager,
4 std::shared_ptr<SceneDataManager> sceneDataManager)
5 : Scene(app, assetManager, sceneDataManager)
6 {}
7
8 bool Scene0::Play()
9 {
10     bool isPlaying = true;
11
12     if (mPart == 0)
13     {
14         // Initialize scene
15         mSceneDataManager->LoadData();
16         LoadData();
17         mPart = 1;
18     }
19     else if (mPart == 1)
20     {
21         mBackground->SetSprite("weird forest");
22         mVoid->SetSprite("void smile");
23         mNameBox->SetText("???");
24         mTextBox->SetText("Why, hello there, fellow dev.");
25         ProceedToScenePart(2);
26     }
27     else if (mPart == 2)
28     {
29         mNameBox->SetText(wand::Utils::ToUpper(mVoid->GetName()));
30         mTextBox->SetText("You may call me Void.");
31         ProceedToScenePart(3);
32     }
33     else if (mPart == 3)
34     {
35         mTextBox->SetText("I was tasked with explaining to you how Wand Engine works. "
36             + std::string("So I hope you're as impatient as I am, yes?"));
37         ProceedToScenePart(4);
38     }
39 }
```



```

Scene0.cpp
WandTutorial | Scene0 | Play()
39  else if (mPart == 4)
40  {
41      mChoiceButton1->SetText("YAS!");
42      mChoiceButton1->OnLeftClick([this]() { (mSceneData->likability)++; this->mPart = 5; });
43      mChoiceButton1->Show();
44      mChoiceButton2->SetText("...Why is the background so creepy?");
45      mChoiceButton2->OnLeftClick([this]() { (mSceneData->likability)--; this->mPart = 6; });
46      mChoiceButton2->Show();
47  }
48  else if (mPart == 5)
49  {
50      mChoiceButton1->Hide();
51      mChoiceButton2->Hide();
52      mVoid->SetSprite("void shy smile");
53      mTextBox->SetText("I'm glad.");
54      ProceedToScenePart(14);
55  }
56  else if (mPart == 6)
57  {
58      mChoiceButton1->Hide();
59      mChoiceButton2->Hide();
60      mVoid->SetSprite("void surprised");
61      mTextBox->SetText("Oh? Let's change it, then.");
62      ProceedToScenePart(7);
63  }
64  else if (mPart == 7)
65  {
66      mBackground->SetSprite("creepy forest");
67      mVoid->SetSprite("void evil");
68      mTextBox->SetText("How about now? Is this more to your liking?");
69      ProceedToScenePart(8);
70  }
90 % | No issues found | Ln: 121 | Ch: 23 | Col: 26 | TABS | CRLF

```

```

Scene0.cpp
WandTutorial | Scene0 | Play()
71  else if (mPart == 8) { ... }
81  else if (mPart == 9) { ... }
89  else if (mPart == 10) { ... }
95  else if (mPart == 11) { ... }
101 else if (mPart == 12) { ... }
109 else if (mPart == 13) { ... }
115 else if (mPart == 14)
116 {
117     mVoid->SetSprite("void smile");
118     mTextBox->SetText("So let's move on with the main part then, shall we?");
119     ProceedToScenePart(15);
120 }
121 else if (mPart == 15)
122 {
123     mSceneData->lastSceneIndex = 0;
124     SaveData();
125     isPlaying = false;
126 }
127
128     return isPlaying;
129 }
130
90 % | No issues found | Ln: 62 | Ch: 25 | Col: 31 | TABS | CRLF

```

As can be seen in the images above, some of the most common methods being called for displaying graphics are the *SetSprite* for VNEntities and *SetText* for TextBoxes. *Show/Hide*, *OnLeftClick*, as well as many others can be used for any UI or VN entity.

Scene0 is an example of how all the scenes in Wand Tutorial are set up. Each of them consists of several parts, and every time the player clicks on the left mouse button *mPart* is set to a different number. Waiting for user input is often the only way to progress through a scene, except for a few cases when *mPart* is explicitly changed immediately in order to avoid executing a block of code more than once.

5.3.2. Example Functions

Although the game does not explore every feature that Wand has to offer, it can be useful to list all the different functions that were used for its development:

Scene Functions

```
int i = wand::Random::GetInt(1, 10);
bool isButtonClicked = mApp->GetInput()->MouseButtonReleased(MOUSE_BUTTON_LEFT);
bool isKeyPressed = mApp->GetInput()->KeyReleased(KEY_SPACE);
float scale = mBackground->GetTransform()->GetScale().x;
mVoid->SetSprite("void smile");
mNameBox->SetText(wand::Utils::ToUpper(mVoid->GetName()));
mChoiceButton1->Show();
mChoiceButton2->Hide();
mChoiceButton1->OnLeftClick([this]() { mSceneData->likability++; this->mPart = 5; });
mChoiceButton1->GetTextTransform()->SetWidth(mChoiceButton1->GetTransform()->GetWidth() - 20);
mChoiceButton1->GetTextTransform()->SetHeight(mChoiceButton1->GetTransform()->GetHeight() - 20);
mVoid->SetParentLayout(mBackground->GetTransform());
mVoid->SetLayoutPosition(wand::LayoutPosition::MIDDLEX, wand::LayoutPosition::BOTTOM);
mBlob->GetTransform()->SetFlip(wand::FlipAxis::FLIP_Y);
mBlob->GetTransform()->SetRotation(45.0f);
mApp->GetAudioManager()->Play("whip", 0.5f, 1.0f, 1.5f);
mApp->GetAudioManager()->Stop("whip");
```

AssetManager Functions

```

std::string root = mApp->GetFileManager()->GetRootFolder();
mApp->GetWindow()->SetIcon(root + "Game/Images/wand.png");
mApp->GetAudioManager()->Add(
    root + "Game/Audio/whip.ogg", "whip", true);
mApp->GetFontManager()->Add(
    root + "Game/Fonts/neuton-regular.ttf", "neuton-regular");
wand::TextBox* nameBox = mApp->GetEntityManager()->AddTextBox(
    "neuton-regular", 40, wand::Color(237, 175, 184, 255));
wand::Rectangle* textRect = mApp->GetEntityManager()->AddRectangle(
    wand::Color(102, 106, 134, 210));
wand::Button* choiceButton = mApp->GetEntityManager()->AddButton(
    root + "Game/Images/choice_button.png", "neuton-regular", 25, wand::Color(255, 255, 255, 255));
wand::Sprite* blob = mApp->GetEntityManager()->AddSprite(
    root + "Game/Images/blob.png");
std::shared_ptr<wand::Background> background = std::make_shared<wand::Background>("main background");
wand::Sprite* weirdForest = mApp->GetEntityManager()->AddSprite(
    root + "Game/Images/weird_forest.png");
weirdForest->SetLabel("weird forest");
weirdForest->GetTransform()->SetPos(0, 0);
weirdForest->GetTransform()->SetLayer(0);
weirdForest->GetTransform()->SetWidth(mApp->GetWindow()->GetStartWidth());
weirdForest->GetTransform()->SetHeight(mApp->GetWindow()->GetStartHeight());
background->AddSprite(weirdForest);

```

SceneDataManager Functions

```

std::shared_ptr<wand::State> state = std::make_shared<wand::State>("State0");
state->Add(new wand::Pair("likability", mSceneData->likability));
state->Add(new wand::Pair("lastSceneIndex", mSceneData->lastSceneIndex));
state->Add(new wand::Pair("backgroundSprite", mSceneData->backgroundSprite));
state->Add(new wand::Pair("musicOn", mSceneData->musicOn));
mApp->GetStateManager()->SaveState(state, "states.txt");
mApp->GetStateManager()->LoadStates("states.txt");
wand::State* state = mApp->GetStateManager()->GetState("State0");
for (const std::shared_ptr<wand::Pair>& pair : state->GetStateData())
{
    if (pair->GetName() == "likability")
        mSceneData->likability = pair->GetIntValue();
    else if (pair->GetName() == "lastSceneIndex")
        mSceneData->lastSceneIndex = pair->GetIntValue();
    else if (pair->GetName() == "backgroundSprite")
        mSceneData->backgroundSprite = pair->GetStringValue();
    else if (pair->GetName() == "musicOn")
        mSceneData->musicOn = pair->GetBoolValue();
}

```

6. CONCLUSION

6.1. Afterword

This paper analyzed the role of game engines as well as their most basic components. More specifically, special emphasis was put on visual novels and the features that set them apart from common games. Taking into account all these factors was a crucial step before starting to build a visual novel engine. Similarly to any problem that is in need of a solution, the first steps that ought to be taken is understanding all its different parameters and clearly defining its boundaries as well as the end goal.

Game engines need to implement various features, including window and state management, graphics rendering, and audio playback. The more each of these features is optimized, the more efficient and flexible the end product can be. This is the reason why many popular engines are maintained by extensive development teams working on improving performance and adding new features.

In any case, however, the creation of a game engine—however small—can be a valuable experience even to developers who are working outside teams. Apart from the engine-specific skills that can be acquired, there are many others that can be learned throughout this process, such as linking different libraries, optimizing the program's performance, searching for help online, and even debugging the project.

6.2. Future Work

The optimization of Wand's already existing features will always have a high priority in the list of future improvements. While it is important for an engine to offer a wide range of tools to game developers, it is often better to establish a robust and high performant architecture first. Wand's various subsystems could be adjusted so that they become more flexible and efficient, a change that would also affect the quality of the end product.

Regarding changes on a greater scope, it was already mentioned that Wand applications need to be developed in Visual Studio and, upon being built, they create Windows x64 executables. Since all the external libraries that are currently linked into the engine can run on multiple platforms, one of the primary goals of this project would be to make Wand

cross-platform. As it is the case for many other popular architectures in the field, both the engine and its games would need to be able to function on different platforms.

Moreover, one new feature that would need to be added eventually is animation. The addition of movement to already existing graphics would render the gameplay much more engaging and would make the stories come to life. Similarly, the use of a particle system could cause the end product to feel more vibrant and less static.

There are several other features that could be added to Wand, although these would be less useful given the needs of this particular engine. 3D rendering, for example, is not preferred by the majority of applications of this genre, and complex physics and AI subsystems are even more rarely used.

The less a game focuses on simple storytelling and static images, the less it reminds the player of a visual novel. Very often, giving the developer more options can be helpful in the sense that it does not limit their creativity. Nevertheless, very much like the developer of a game engine, every creator ought to consider all the aspects of the problem they are trying to solve: The features and limitations of the game need to be known beforehand so that the appropriate engine can be selected. If the application is going to require more tools than a visual novel engine can offer, it might be a better choice for the developer to choose an engine with a greater variety of features, such as Unity.

REFERENCES

- Chernikov, Y. (2017, September 17). *OpenGL*. Retrieved from The Chernobyl: https://www.youtube.com/playlist?list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2
- Chernikov, Y. (2017, September 17). *Welcome to OpenGL*. Retrieved from The Chernobyl: <https://www.youtube.com/watch?v=W3gAzLwfIP0>
- Chernikov, Y. (2018, September 30). *Game Engine*. Retrieved from The Chernobyl: <https://www.youtube.com/playlist?list=PLlrATfBNZ98dC-V-N3m0Go4deliWHPFwT>
- Chernikov, Y. (2018, October 14). *What is a GAME ENGINE?* Retrieved from The Chernobyl: <https://www.youtube.com/watch?v=vtWdgtMo1T4>
- de Vries, J. (n.d.). *Hello Triangle*. Retrieved from LearnOpenGL: <https://learnopengl.com/Getting-started/Hello-Triangle>
- Glaiel, T. (2021, November 18). *How to make your own game engine (and why)*. Retrieved from Game Developer: <https://www.gamedeveloper.com/blogs/how-to-make-your-own-game-engine-and-why->
- Gordan, V. (2021, April 27). *OpenGL Course - Create 3D and 2D Graphics With C++*. Retrieved from freeCodeCamp.org: <https://www.youtube.com/watch?v=45MIykJWJ-C4>
- Schardon, L. (2022, February 22). *Best Game Engines for 2022 – Which Should You Use?* Retrieved from Zenva Pty Ltd: <https://gamedevacademy.org/best-game-engines/>
- Vincent, B. (2020, April 3). *The best engines for making your own visual novel*. Retrieved from PC Gamer: <https://www.pcgamer.com/the-best-visual-novel-engines/>