FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Zeman

Cross-platform 2D game framework

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer Science

Specialization: General Computer Science

Prague 2022

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature

I would like to thank my supervisor Mgr. Jakub Gemrot, Ph.D., my family and friends for all their support and advices.

Title: Cross-platform 2D game framework

Author: Tomáš Zeman

Department / Institute: Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: One of the most useful tools for game development is a game framework. It is usually a complex software which offers abstraction of game components such as rendering, physics, sound, user input or AI. The goal of this thesis is to create a simple game framework for 2D games, focused on performance, extensibility and multiplatformity. A second goal of this thesis is implementation of an example game for demonstration of functions and functionality of the framework.

Programming language C++ was chosen for development of the framework along with a portion of SDL library. Target platforms were chosen to be Windows and Linux. The example game was successfully implemented and tested on both platforms using most of the framework's capabilities.

Keywords: game framework, 2D, C++, cross-platform

Contents

Introduction....5
Thesis Goals....6
1. Existing Frameworks....7
1.1. Overview....8
Unity 3D....8
Unreal Engine....10
Godot....12
GameMaker Studio 2....13
Cocos2d-x....14
libGDX....15
MonoGame....16
2.1. Comparison....17
3. Analysis....20
3.1. Programming Language....21
C++....21
C#....21
Java....22
Rust....22
Selection....22
3.2. Target platforms....23
Linux....23
Windows....23
3.3. Framework components....24
Scene handling....25
Rendering....25
Physics....25
Audio....25
Scripting....26
User interface....26
Networking....26
Entity Component System support....26
3.4. Libraries available....28
SDL....28
SFML....28
GLFW....28
Box2D....28

Chipmunk2D....29
Dear ImGui....29
NanoGUI....29
Gainput....29
GameNetworkingSockets....29
3.5. Selection....30
4. Development documentation....31
4.1. Architecture overview....31
SceneGraph....35
Utility Classes....35
Framework Subsystems (Engines)....35
Entity Component System....35
Resources....36
Scene....36
4.2. Scene....37
4.3. Scene Graph....38
4.4. Utility classes....40
Resources....40
Settings....40
Camera....40
Texture....40
Logger....40
Timer....41
4.5. Framework subsystems (Engines)....42
RenderEngine....42
SoundEngine....42
InputEngine....42
FileEngine....43
LuaEngine....43
4.6. Components....44
Console Component....45
GUI Component....45
Input Component....45
Physics Component....45
Sound Component....46
Sprite Component....46
Type Component....46
4.7. Systems....47
Physics System....47

Render System....47
PlayerInput System....47
Audio System....48
Console System....48
GUI System....48
4.8. Compilation....49
5. Example game....51
5.1. Example 1 - Empty Game....51
5.2. Example 2 - Player....55
5.3. Example 3 - Console....57
5.4. Example 4 - Physics....59
5.5. Example 5 - Audio....61
5.6. Example 6 - GUI....62
5.7. Example 7 - Animations & Scripting....64
5.8. Final Example....65
Conclusion....66

Introduction

Videogames have been made for decades. As hardware performance has increased over the years, the complexity of modern cutting-edge videogames shot up drastically. The most-played videogames are no longer developed by individuals, but by whole teams of developers. As developing costs skyrocket, it has become imperative to simplify the development process to save resources. Software tools are the elegant solution to save on development time and resources. These tools, big or small, simplify certain parts of the videogame development process so that developers can focus on more important elements of a game. This thesis seeks to implement such a tool.

One of the most common and most useful tools is the so-called game framework or engine. Both of these are often used interchangeably since they lack proper definition. However, there are differences. While framework is used for lower level software and software without its own graphical interface, engine is used for larger, more robust, software bundles, usually using their own graphical application. This thesis focuses on the former concept of a game framework.

Game framework is a software tool or a set of tools, typically consisting of subsystems for graphics, sound, physics, user input, AI, scene representation etc. Game frameworks can be categorized based on many criteria, such as target platforms, programming language, focus on 2D or 3D games or licensing.

Problems may arise when choosing the right framework or engine for developing a particular game. For beginner developers existing engines, such as Unity or Unreal Engine might be too complex or hard to navigate. For large videogame development companies creating the latest AAA games using cutting-edge technology no framework can be flexible or robust enough. The solution to both of these issues is to create a custom framework.

Thesis Goals

- Create a 2D multi-platform (Linux and Windows) game framework
- Use the framework to create a simple game to demonstrate framework‘s functions

Structure of This Thesis

Chapter One analyzes the existing software, its features, performance, platforms available and other specifics. Because the difference between a game engine and a game framework is slight, both software presenting as a framework and software presenting as an engine are considered. At the end of the chapter, there is a comparison of the engines and frameworks discussed along with their summary.

Chapter Two discusses reasoning behind programming language choice, platform selection and libraries used. This chapter also describes components of the framework built in this thesis.

Chapter Three describes implementation of the framework built here. This chapter starts by providing an overview of the framework‘s architecture and continues by detailed description of each component and their interfaces and interactions.

Chapter Four presents a simple example game. The purpose of the example game is to demonstrate functionality and features of the framework. The body of this chapter mainly provides code snippets and screenshots of the game.

Chapter Five, the last chapter of this thesis, discusses the results and options for future work.

1. Existing Frameworks

Currently, there exist dozens of publicly usable game engines and frameworks[1]. These pieces of software differ in many aspects. Supported platforms, licensing options, offered performance, programming languages supported, technical support availability and more can vary significantly. To provide a better understanding of modern engine or framework capabilities, several popular frameworks and engines have been selected for comparison. Here, each of the selected is summarized with focus on licensing options, features, supported languages and supported platforms. Last section compares them.

1 https://en.wikipedia.org/wiki/List_of_game_engines

1.1. Overview

Unity 3D

Unity 3D is the most used game engine in the world with 48% market share [1] and $234.8 million revenue in Q1 2021[2]. It offers wide platform support including iOS, Android, WebGL, Nintendo Switch, Google Stadia, PS4, Xbox, Oculus and more are added to support next-generation platforms.

Unity is currently (2022) available for free for companies or developers whose revenue or funding is less than $100,000 in the last 12 months. The "Plus plan" provides developers with splash screen customization, advanced diagnostics and analytics. This plan is only available to subjects with revenue or funding less than $200,000 in the last 12 months. Above the $200,000 revenue level, subjects are required to use Pro or Enterprise plans. These plans offer advanced technical support, high-end art assets, build server capacity, and more. Unity does not take royalties.

2 https://unity.com/our-company/newsroom/unity-announces-first-quarter-2021-financial-results

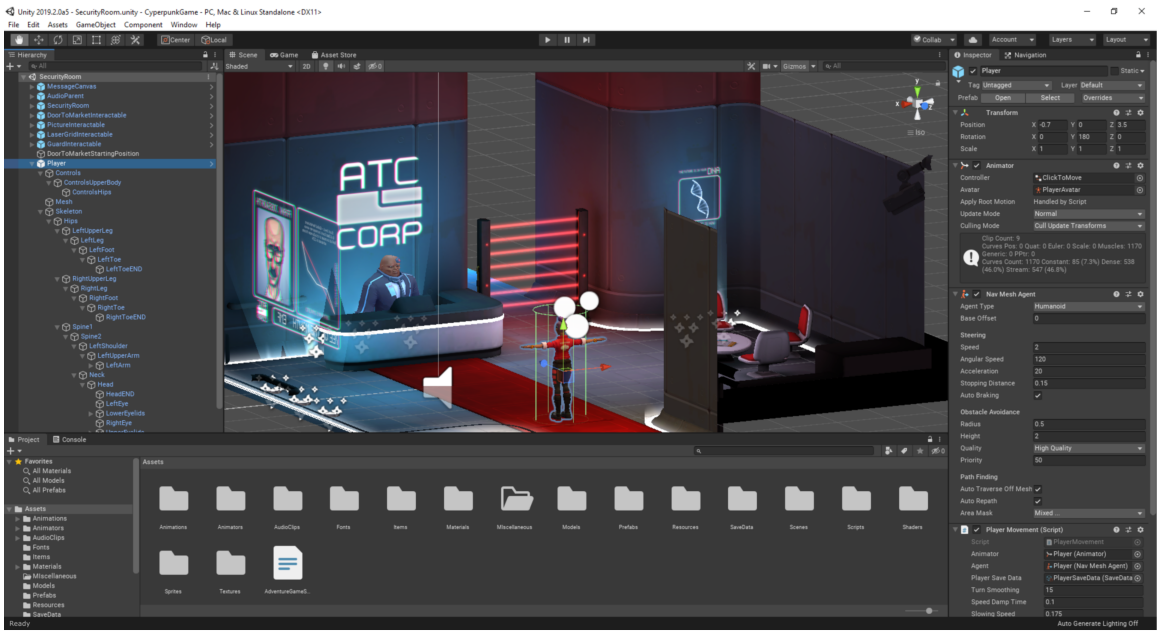

Illustration 1: Unity Editor 2019 - Windows

This game engine comes with an editor for creation of both 2D and 3D games. It provides developers with many tools such as code editor, scene visualization, asset management, particle system visualization and much more.

Unity supports scripting in C# natively and in any other language with support for compiling into a .NET compatible DLLs (Dynamic Link Library). Unity provides two scripting backends: Mono and IL2CPP (Intermediate Language To C++). Mono, the default, compiles scripts written in C# using JIT (Just-in-time) compiler. IL2CPP compiles intermediate language (in this case C#) to C++, which is then compiled to native binary file using AOT (Ahead-of-time) compiler.

Unity is well optimized for development of games of various genres and scales from small mobile games to large AAA games. Many successful games, for example Kerbal Space Program (over 2 million copies sold [2]), HearthStone (over 23 million active players in 2020 [3]) or Subnautica (over 5 million copies sold [4]), were made with Unity.

Unreal Engine

This second most commercially used engine supports slightly less platforms, but still supports development for iOS, Android, Windows, Linux, Google Stadia, Xbox, Nintendo Switch, PS4 and similar platforms. Unreal Engine revenue was $97 million in 2019 [5] and its market share in 2021 was 13%.

Unreal Engine is currently (2022) free to use with 5% royalties if your revenue exceeds $1,000,000. Unreal Engine also offers a royalty-free licence for internal projects, free projects and non-interactive content (films, TV shows, videos or still images). In all cases, full access to C++ source code of the engine is provided.

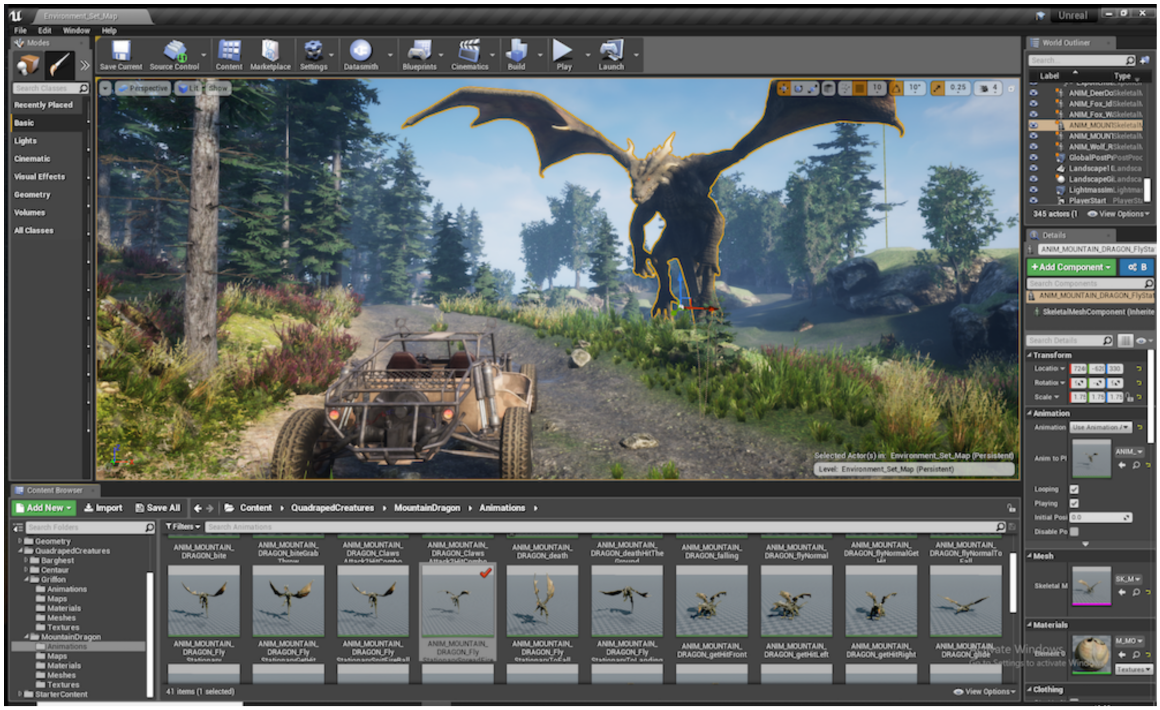

Illustration 2: Unreal Engine 4 Editor

Just like Unity, Unreal Engine also comes with an editor supporting 2D and 3D games. The editor offers similar features as Unity Editor.

Unreal Engine supports scripting in Blueprint Visual Scripting system. This node-based system is used by developers to interactively create scripts, which are then transformed to C++ classes. These classes will then be compiled during the build phase for a specific platform. Scripting natively in C++ is also supported, and results in faster code at the cost of more development time.[3]

Unreal Engine is well optimized for large and complex games of different genres, especially first person games. Among successful games developed with this engine are Life Is Strange [6], Fortnite [7] and Ark: Survival Evolved [8].

3 http://awforsythe.com/unreal/blueprints_vs_cpp

Godot

Godot engine is an open-source engine with focus on both 2D and 3D games. Unlike the previous two engines discussed, Godot is completely free without any royalties. While not many games are created using Godot, in comparison with Unity and Unreal Engine, it offers an editor with similar functionalities as Unity and Unreal Engine. Platforms supported are similar to Unreal Engine‘s: Windows, Linux, iOS, Android, Nintendo Switch, PS4, Xbox One, various VR platforms, and others.

In terms of scripting, Godot offers the widest support of languages. Developers are able to use GDScript, its own scripting language, C#, C/C++, block-based visual scripting and more languages by community provided bindings.

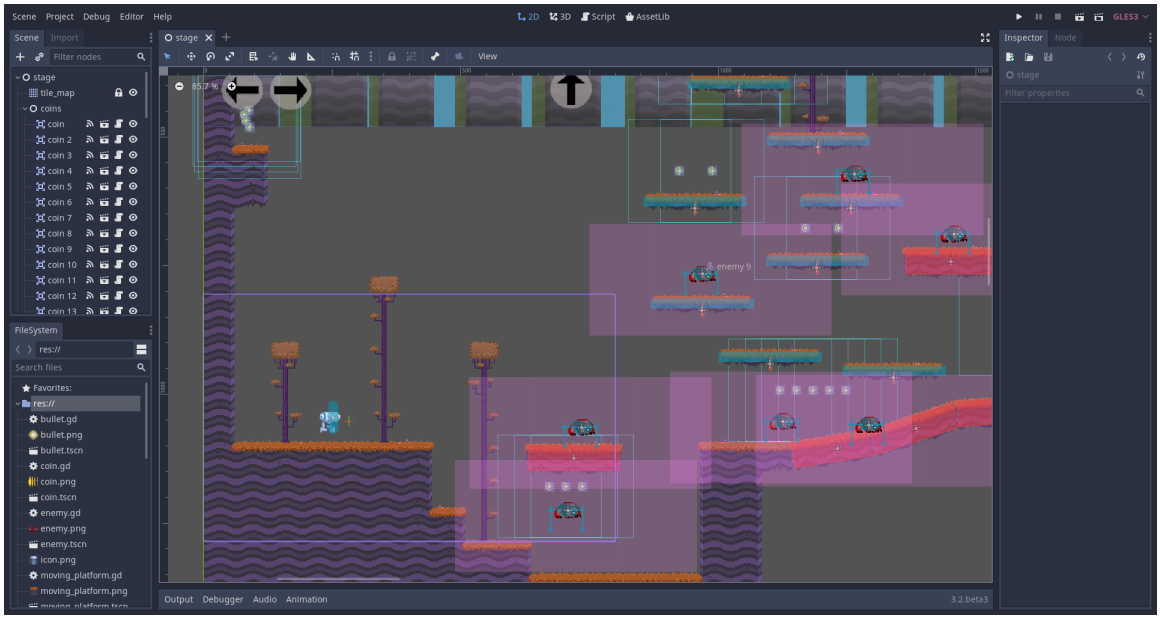

Illustration 3: Godot Editor

Godot Engine is designed to be easy to use, but still flexible and complex enough for medium scale games. There aren‘t many well-known examples of games made with this engine due to its low market share, caused by its low funding, in comparison to Unity or Unreal Engine. On top of that, Godot was released in 2014, making it a relatively new engine. An example of a succesful game made with Godot is Deponia [9].

GameMaker Studio 2

GameMaker Studio 2 (GMS) is a game engine focused on 2D games (with limited 3D capability). Its original focus was on novice programmers, but with recent releases, more advanced features, were implemented.

This engine supports development for multiple platforms such as Windows, Linux, consoles or mobile phones. Its editor is easy to use with focus on visual development. For scripting, GameMaker Studio created GameMaker language, based on C programming language. Other scripting option is DnD™ (Drag and Drop™) visual scripting.

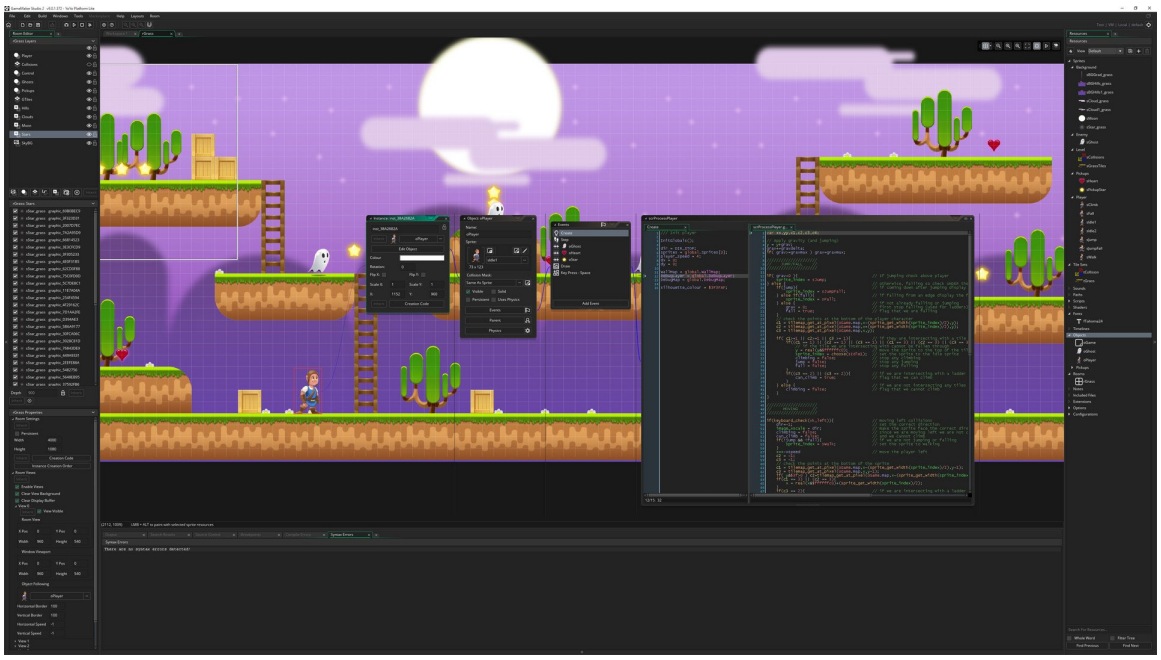

Illustration 4: GameMaker Studio 2 - Editor

Pricing is less friendly for novice programmers. It's only free licensing options allows exporting games only to GXC, browser-based platform owned by Opera. For exporting to desktop platforms, Windows, macOS and Linux, the cost of a licence is 50$/year. For $100/year, developers can buy a licence that allows for exporting games to desktop platforms, mobile platforms, web (HTML5) and UWP (Universal Windows Platform). Development for three console platforms, Playstation (4 and 5), Xbox (One and Series X|S) or Nintendo Switch, costs $800/year. Best example of a succesful GMS game is Undertale [10].

Cocos2d-x

Cocos2d-x is a free, open source, game framework for creating 2D games. It does not provide an editor (editor is available as a standalone tool - Cocos Creator). For developers it offers easy to use and fast API.

The only supported platforms are mobile (iOS, Android) and desktop (Windows, Linux, macOS). Among its features are: scene management, sprite management, physics (uses Box2d or Chipmunk physics engines), animations, sound support, basic GUI (Graphical User Interface), user input support (touch/ mouse/keyboard/accelerometer depending on platform) and others.

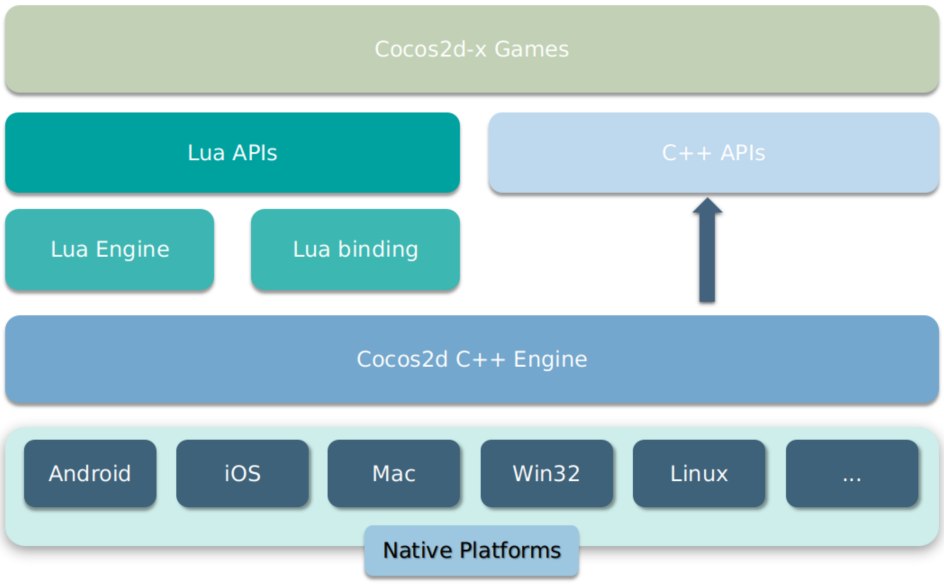

Illustration 5: Cocos2d-x - Architecture

This framework is written in C++, but provides bindings for Lua and JavaScript programming languages. Good example of a game made with this framework is Hill Climb Racing mobile racing game with more than a billion downloads in total [11].

libGDX

This game development framework is written in Java, offers cross-platform development and open-source code at no cost with comprehensive documentation. Just like previous framework, it does not offer an editor.

Supported platforms are Windows, Linux, macOS, Android, iOS and Web (HTML5). Its development features consist of audio, input handling, 2D and 3D graphics, file system abstraction, networking, integration of services such as Google Play Games, in-app purchases or various analytics, physics and more.

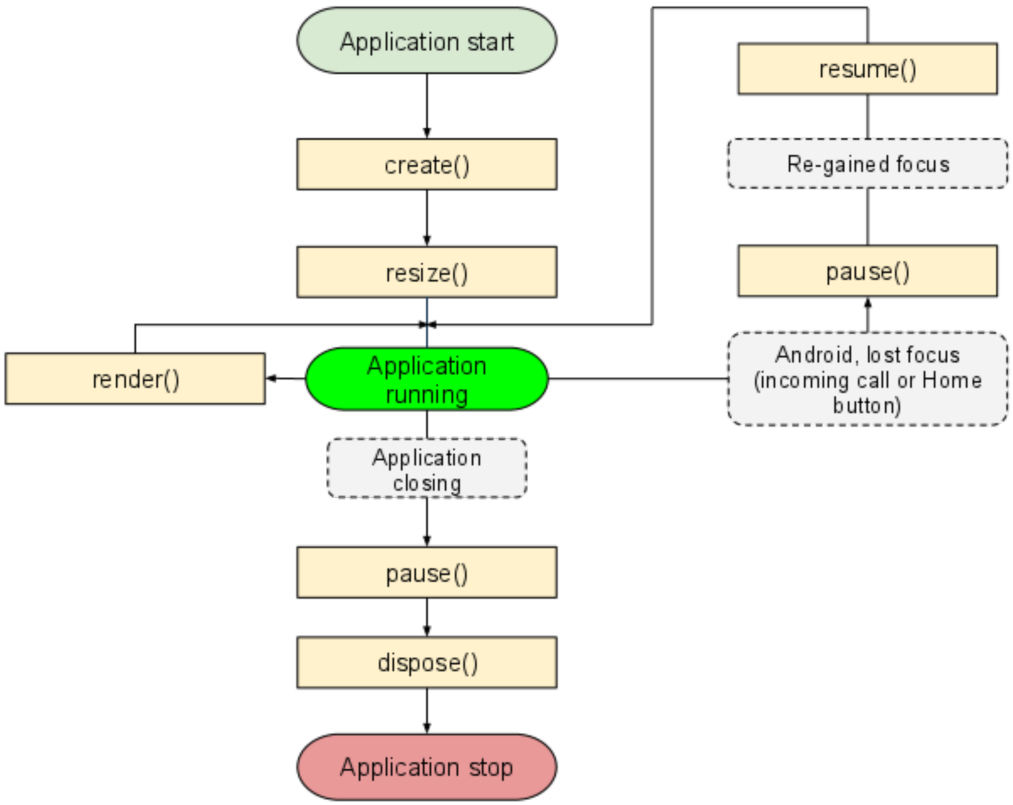

Illustration 6: libGDX - life cycle of a game

Examples of games written using this framework are Slay the Spire [12] or mobile game Sandship (with more than 5 million downloads[4]).

4 https://play.google.com/store/apps/details?id=com.rockbite.deeptown&hl=en&gl=US

MonoGame

MonoGame is another example of a game framework. Written in C#, based on Microsoft's XNA (XNA's Not Acronymed) Framework, it is open-source and supports multiple platforms: Android, iOS, Windows, macOS, Linux, PS4, Xbox and Nintendo Switch with plans to support more.

Its offered features are similar to other frameworks, 2D and 3D graphics, audio, input handling and its own math library. Same as previous frameworks, it does not offer a game editor, only GUI editor for project management.

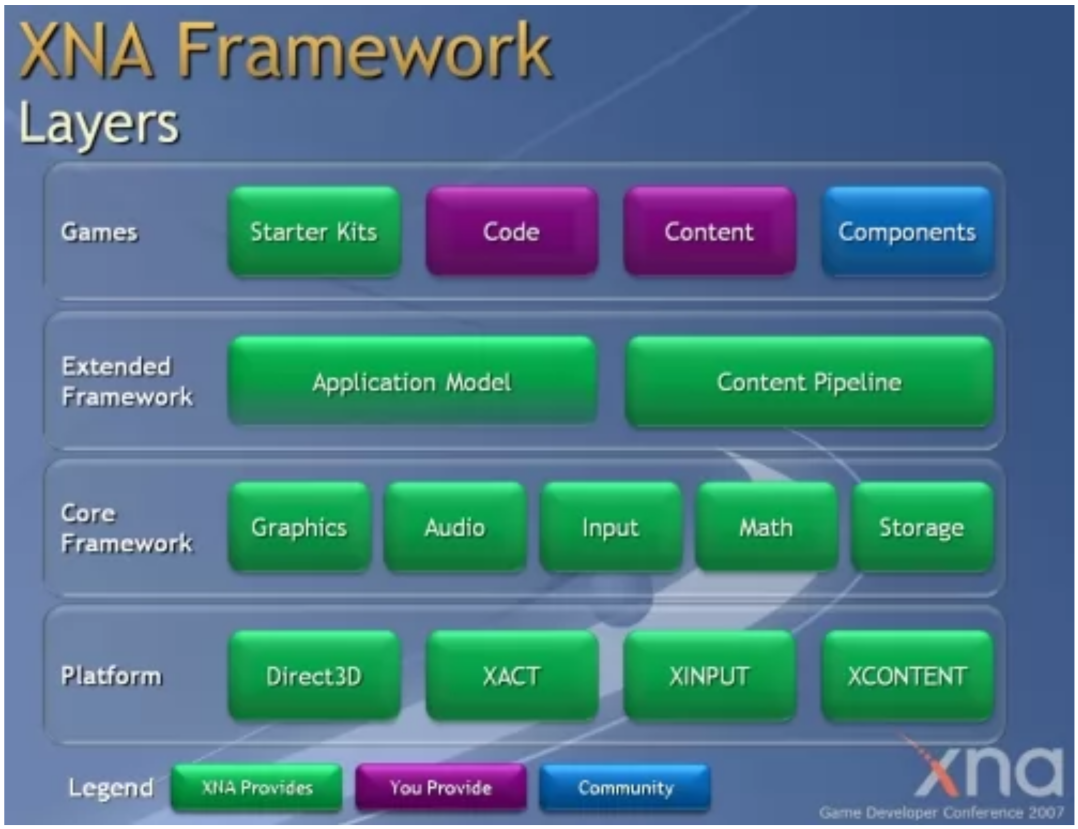

Illustration 7: XNA Framework - Architecture

2. Noteworthy games made using this framework are Celeste [13], Barotrauma [14] or Stardew Valley, which sold over 10 million copies in 4 years[5].

5 https://www.stardewvalley.net/press/

2.1. Comparison

From these examples we can see that game engines provide developers with powerful editors to separate game development from its coding aspect. On the other side, frameworks tend to be more transparent about their architecture and external libraries used. Engines are often sold to developers with many licensing options. Frameworks are often free and open-source, enabling its further development by community. Engines usually provide integration with third party tools.

Pricing comparison	Non-commercial use	Commercial use	Price range
Unity 3D	Free	Paid	$399/year/seat - $4000/month/20 seats
Unreal Engine	Free	Free + 5% royalties	5% royalties + $0 - $1500/seat/year
Godot	Free	Free	N/A
GameMaker Studio 2	Free trial (30 days)	Paid	$39 - $1500/year
Cocos2d-x	Free	Free	N/A
libGDX	Free	Free	N/A
MonoGame	Free	Free	N/A

Table 1.1: Framework/Engine comparison - Pricing

Platform support	Windows	Linux	macOS	Android	iOS	Fire	PS4	PS5	PSVita	Xbox	Nintendo Switch	VR	Web	Google Stadia	Raspberry Pi
Unity 3D	✔	✔	✔	✔	✔	✘	✔	✘	✘	✔	✔	✔	✔	✔	✘
Unreal Engine	✔	✔	✔	✔	✔	✘	✔	✔	✘	✔	✔	✔	✔	✔	✘
Godot	✔	✔	✔	✔	✔	✘	✘[6]					✔	✔	✘	✘
GameMaker Studio 2	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✘	✔	✘	✘
Cocos2d-x	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
libGDX	✔	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✔	✘	✔
MonoGame	✔	✔	✔	✔	✔	✘	✔	✘	✔	✔	✔	✘	✘	✔	✘

Table 1.2: Framework/Engine comparison - Platform support

6 Unofficial support

Language support	Native language	Scripting options	Open-source
Unity 3D	C#	C#	No
Unreal Engine	C++	C++, Blueprint	Yes
Godot	C++	C++, C#, GDScript, Visual Script	Yes
GameMaker Studio 2	C++	GML, DnD™	No
Cocos2d-x	C++	Bindings for Lua & JavaScript	Yes
libGDX	Java	N/A	Yes
MonoGame	C#	N/A	Yes

Table 1.3: Framework/Engine comparison - Language support

Development options	2D/3D games	Editor	Notable features
Unity 3D	2D/3D	Yes	Asset store
Unreal Engine	2D/3D	Yes	Complex graphics
Godot	2D/3D	Yes	Wide scripting support
GameMaker Studio 2	2D (limited 3D)	Yes	Easy to use, Wide platform support
Cocos2d-x	2D	No	Lightweight
libGDX	2D/3D	No	Java development
MonoGame	2D/3D	No	Based on XNA

Table 1.4: Framework/Engine comparison - Development options

3. Analysis

This chapter first summarizes the cons and pros and discusses the selection of commonly used programming languages for framework development.

Second section discusses target platforms (Linux and Windows). For each target platform I will summarize their advantages and disadvantages in terms of game framework development, their specific subsystems and other platform specific features.

In the third section, distinct components of a general game framework are described. Not every framework contains all of the described components and most frameworks contain more components.

In the fourth section, some third party libraries, providing abstraction of one or more components mentioned in the third section, are summarized.

In the last section libraries selected for the framework are discussed.

3.1. Programming Language

The choice of a programming language affects most aspects of the development process. Performance, library selection, memory overhead, platform support, scripting language integration, extensibility, or the speed of the development process all largely depend on the framework‘s native language.

C++

C++ is one of the most commonly chosen languages. It offers great performance by compiling code directly into platform specific machine code. Other advantages for of C++ include the numerous third party libraries available and the control over optimizations and memory allocation resulting in very small memory overhead.

Disadvantages of C++ are for example: possibility of memory leaks, need to compile for each platform independently and harder debugging compared to C# or Java.

C#

Very popular language for game and game framework development with many libraries providing high level abstraction. C# compiles code to intermediate language which is then interpreted by a virtual machine. This process slows its execution, but makes debugging easier and allows it to run the same code on multiple platforms.

Allocated memory is managed by a garbage collector. This approach relieves the developers from manually managing memory, but introduces a performance penalty and memory overhead.

Java

Similar to C#, Java compiles code to intermediate language, called bytecode, which is then run using Java VM. Java emphasizes being a cross-platform, high level language. As with C#, Java uses a garbage collector for memory management.

Game framework development in Java makes games easily portable. The exception to this is consoles, since Java VM is not available for most major consoles. In terms of performance, Java programs are not as fast as C++ programs and a framework developed using Java will most likely need parts of it have rewritten in C/C++.

Rust

Rust language offers similar performance as C++ with easier and safer memory management. However, it is a relatively new language from 2010 and thus there is not as many libraries or as much community support as there is for languages like C# or C++.

Selection

C++ was selected as the framework language. It was chosen mainly for offering a wide range of third party libraries, having the best performance while supporting OOP and having a rich standard library.

3.2. Target platforms

Here positives and negatives of game development for two main desktop platforms, Windows and Linux, will be summarized.

Linux

Supporting the Linux platform can be rather easy due to it‘s more technical community, mostly open-source libraries, and easy compiling. Games for this platform can use either OpenGL API (Application Programming Interface) for 2D or 3D graphics or Vulkan for 3D graphics.

Windows

Windows is the most used desktop gaming platform[7], supporting this platform should be a priority for every framework aiming for success. Thanks to this, the gaming market on Windows is also full of competition as vast majority of desktop games support this platform.

Windows also offers the use of Microsofts proprietary software DirectX. This is a set of APIs for multimedia and game programming. Part of this is Direct3D API for 2D and 3D graphics. Games for Windows can also use Vulkan or OpenGL.

7 https://www.statista.com/statistics/265033/proportion-of-operating-systems-used-on-the-online-gaming-platform-steam/

3.3. Framework components

Game frameworks can be divided into components that interact with each other, providing unified interface and abstraction. This approach makes development of a framework easier since different components do not usually depend on each other and a component can be developed without being burdened by unnecessary dependencies. Abstraction provided by the framework then separates developers from platform dependent or low-level code.

A good overview of various framework components can be found in the Game Engine Architecture book by Jason Gregory (Gregory, J. 2018, fig. 1.16.) [15]. From the figure, a set of components most often used in a framework has been selected:

- Scene handling
- RenderingEngine
- Physics
- Audio
- Scripting
- User interfaces
- Networking
- Entity Component System support

Many components mentioned in the book are not covered by this thesis for multiple reasons such as being too advanced, unnecessary or specific for 3D games.

There is a short comparison of popular frameworks from the previous chapter at the end of this section (Table 3.1). It summarizes their support of the following components. Only native support is considered.

Scene handling

A common approach for dividing games into smaller parts is using scenes. Each scene is basically a screen displaying different information. Scenes can vary in granularity from small and specialised, displaying each level or menu screen, to big and complex, displaying the entire game or all menu screens.

A game is then a collection of screens with a starting screen. Player then navigates through the scenes, for example by pressing the "Start" button in the menu screen.

Rendering

Rendering subsystem takes care of rendering textures, user interface and other game elements onto the screen. This uses graphic APIs such as OpenGL or Direct3D either directly or through a graphics library. Graphic APIs then interact with OS and hardware to send rendered data to GPU, which then renders it on the screen.

Physics

Physics simulation is a useful but complex and demanding feature of a framework. For this reason, there are dedicated physics libraries that try to optimize this as much as possible. Not every game uses physics, but among frameworks it is a common feature.

This component provides functionalities such as rigid-body simulations, collision detection, sensors for proximity detection or joints.

Audio

The audio component of a framework functions as abstraction for playing music or short sounds. It takes care of decoding various audio formats and streaming them to audio devices.

Scripting

Scripting is a way for game developers to write code that does not have to be compiled with the game and can be used to either modify the behaviour of the game or act as a game itself. Scripts can also be used by third party developers to modify or extend the game.

Scripting languages are typically interpreted at runtime. Because of this, their performance is worse than that of compiled languages.

User interface

This components provides abstraction for handling user input from various sources such as keyboard, mouse, controller or touchscreen.

Networking

All online multiplayer games need to connect to either a server or directly to another client through the internet. Even some single-player games use networking to check for updates or display real-time news in-game. Networking component abstracts networking to provide a simpler interface than what OS or framework‘s language library offer.

Entity Component System support

Recently, game developers have started to use a software architecture pattern called Entity „Component System“ more often, rather than using a pure OOP approach. This patter prioritizes composition over inheritance and offers several advantages:

- Better scalability
- Separates game logic from game data, making changes to either one easier
- Code flexibility

Components	**Physics**	**Audio**	**Scripting**	**Scene Support**	**Networking**	**User interface**	**ECS**
Unity 3D	✔	✔	✔	✔	✔	✔	✔
Unreal Engine	✔	✔	✔	✔	✔	✔	✘
Godot	✔	✔	✔	✔	✔	✔	✘
GameMaker Studio 2	✔	✔	✔	✘	✔	✔	✘
Cocos2d-x	✔	✔	✔	✔	✔	✔	✘
libGDX	✔	✔	N/A	✔	✔	✔	✘
MonoGame	✔	✔	N/A	✘	✔	✘	✘

Table 3.1: Framework Components - Comparison

3.4. Libraries available

In this section several known examples of libraries are introduced.

SDL

Simple DirectMedia Layer, SDL, is a cross-platform library providing low level interface to hardware or system components such as audio, input devices, graphics (using OpenGL and Direct3D) or networking [8]

It is written in C with bindings to other languages, for example C#, Go, Python or Rust.[9]

SFML

This multimedia library is very similar to SDL. Provides modules for graphics, audio, networking and window management. Compared to SDL, this library is written in C++ and implements object oriented design [10]

GLFW

GLFW[11] is another library providing a simple API, written in C. It uses OpenGL and Vulkan backends for rendering and includes support for user input devices. Many frameworks, Cocos2d-x or libGDX for example, use GLFW.[12]

Box2D

Box2D[13] is a 2D physics engine, developed by Erin Catto. It is written in C++ with community support for other languages. Several popular frameworks and engines use Box2D to simulate 2D physics. Examples of these can be libGDX[14] or Unity[15].

8 https://www.libsdl.org/
9 https://www.libsdl.org/languages.php
10 https://www.sfml-dev.org/faq.php#grl-whatis
11 https://www.glfw.org/
12 https://www.glfw.org/community.html
13 https://box2d.org/
14 https://github.com/libgdx/libgdx/wiki/Physics
15 https://docs.unity3d.com/Manual/PhysicsSection.html

Chipmunk2D

This physics library is written in C and compared to Box2D focuses on multi-threading support and performance.[16] Chipmunk2D is included in Cocos2D framework.[17]

Dear ImGui

Dear ImGui is a Graphical User Interface (GUI) library for C++.[18] It is sponsored by large companies such as Blizzard, Google, Nvidia or Ubisoft.[19] Focus of this library is on easy integration. This library is not very flexible in customizating the look of the interface. For this reason it is best suited for making tools rather than games.

NanoGUI

NanoGUI is a GUI library written in C++.[20] Unlike Dear ImGui, NanoGUI depends on GLFW and a few other libraries.

Gainput

Gainput is a C++ input library focused on games.[21] It provides a unified interface and besides desktop or console devices (keyboard, mouse, gamepad), it supports multi-touch screens or built-in sensors commonly found in mobile devices.

GameNetworkingSockets

GameNetworkingSockets is a networking library written in C++.[22] Among its features are reliable and unreliable connection over UDP, encryption or IPv6 support.

16 https://chipmunk-physics.net/aboutChipmunk.php
17 https://chipmunk-physics.net/games.php
18 https://github.com/ocornut/imgui
19 https://github.com/ocornut/imgui/wiki/Sponsors
20 https://github.com/mitsuba-renderer/nanogui
21 https://github.com/jkuhlmann/gainput
22 https://github.com/ValveSoftware/GameNetworkingSockets

3.5. Selection

The SDL library was chosen for offering many subsystems, for user input, filesystem access, sound, networking and more.

For physics implementation, Box2D library was selected since it is written in C++ and for that reason, its integration in the framework will be easier.

Dear ImGui was selected to provide a GUI implementation. While not as suitable for games as different GUI libraries with more customization options, it is standalone and is easy to integrate and replace with a different library in future versions of the framework.

Additionally, an ECS library[23] was used to provide a simple Entity Component System implementation. Library plf::nanotimer[24] was used to provide a cross-platform high-precision timer. As the scripting language, Lua was chosen, LuaState[25] binding library for C++ in particular.

23 https://github.com/SRombauts/ecs
24 https://www.plflib.org/nanotimer.htm
25 https://github.com/AdUki/LuaState

4. Development documentation

This chapter serves as the development documentation of the framework. The first section summarizes the architecture and general framework structure. The following sections further document inner structure and interface of individual framework components. The last section documents filesystem hierarchy and compilation process.

4.1. Architecture overview

The architecture of the framework is built around Entity Component System [16] (ECS), further explained in this subchapter. The reason for using this architectural pattern is to allow for easy extensibility and scalability. Further subchapters starting at 3.2. refer to parts of Illustrations 8, 9 and 10.

Illustration 8 shows a visualization of the architecture.

The Class diagram in Illustration 9 represents a simplified overview of the most important classes including their attributes and functions.

Illustration 10 shows order of initialization of the framework‘s components.

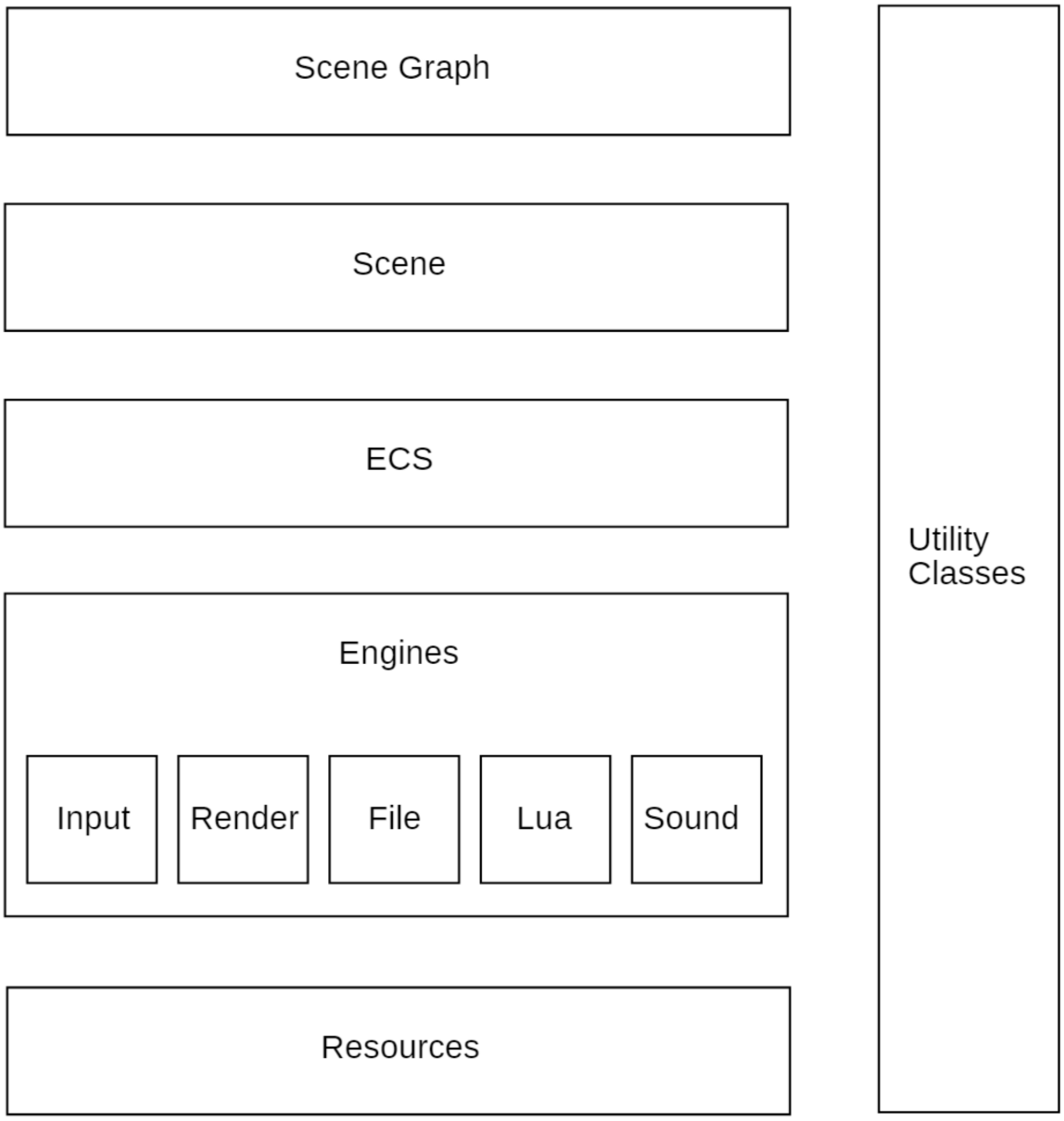

Illustration 8: Framework Architecture

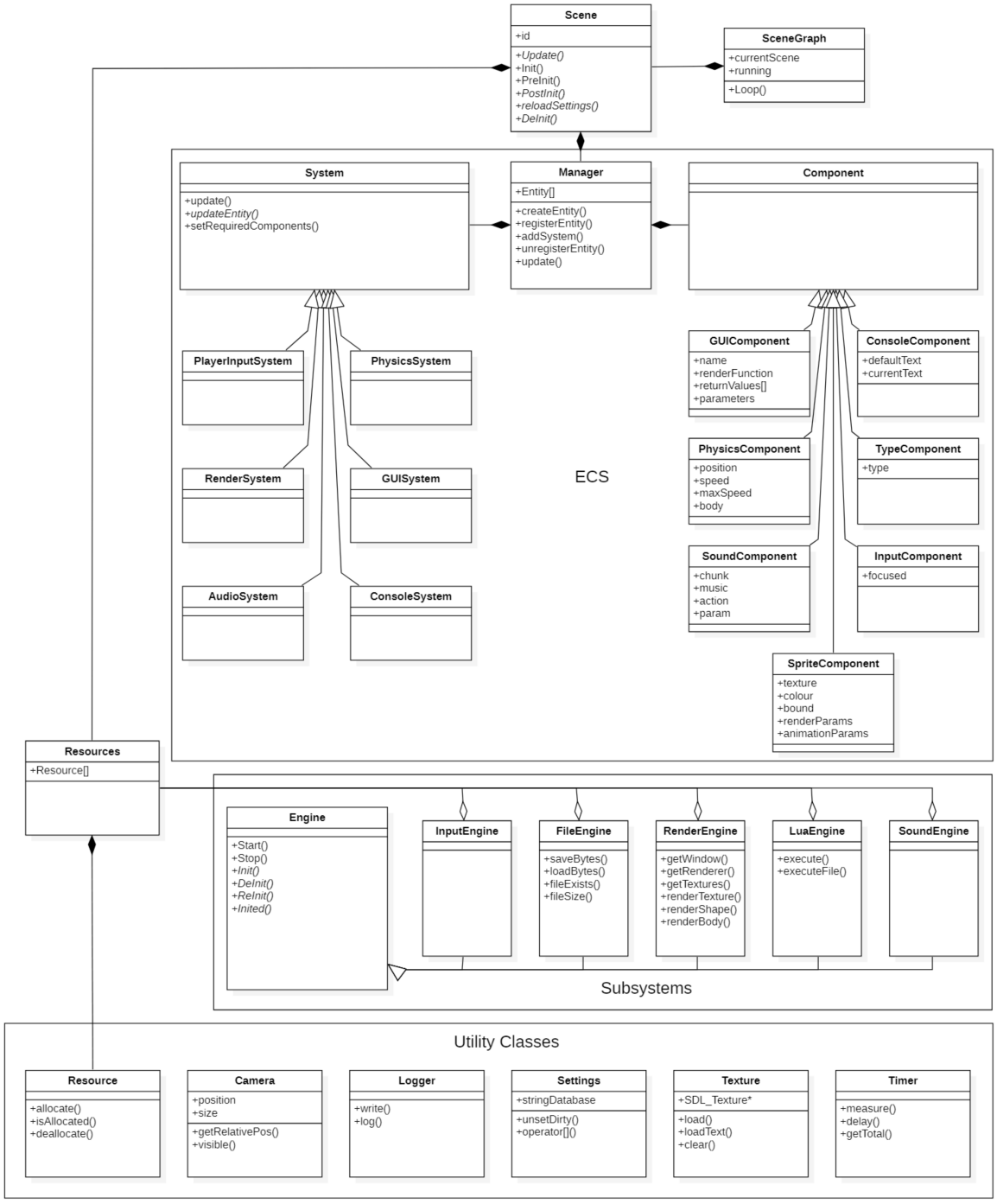

Illustration 9: Class Diagram

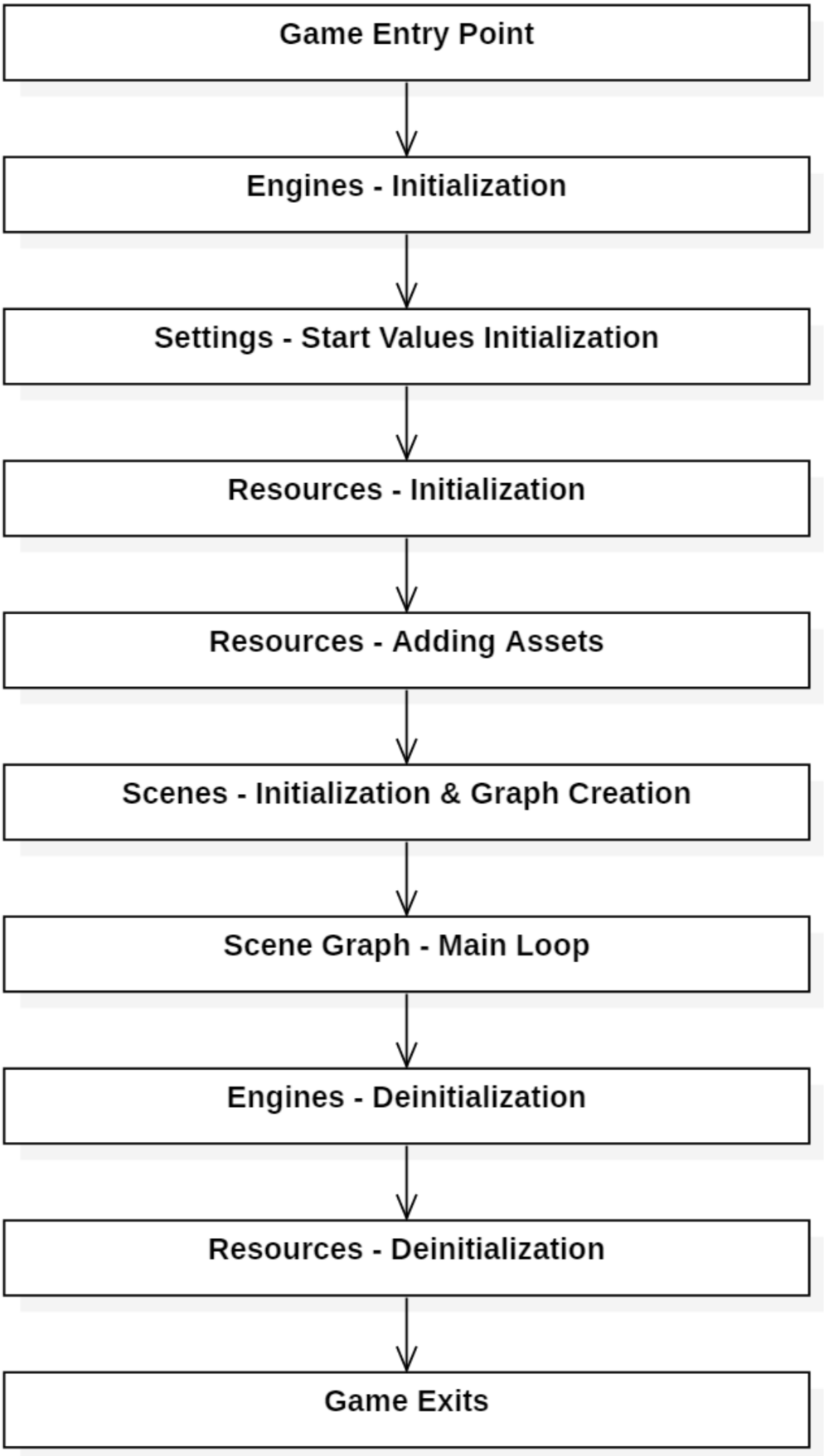

Illustration 10: Lifecycle of a Game

SceneGraph

Each game or program using this framework is created via instancing the SceneGraph class. This object contains a graph of game‘s scenes and main loop function - the entry point. Scenes are logical wholes of each game, for example, main menu, tutorial, first level, second level, credits, or cutscene, can all be different scenes.

Utility Classes

There are several, often general-purpose, classes that can be used anywhere in the code. Their purpose is mainly to abstract complex functionality behind simple interface.

Framework Subsystems (Engines)

These classes provide high-level interface to low-level functions.

Entity Component System

This architectural pattern can be characterized by the following:

- Prioritizes composition over inheritance.
- Consists of Entity, Component, System, and in most implementation also Manager.
- Game objects are sets of Components, represented by Entity.
- Components hold object‘s data.
- Each System acts on every Entity with particular set or subset of Components.
- Manager then provides an interface to managing entities and their data.
- Most implementations are compatible with data-oriented approach by storing Components closer together in memory.

In this implementation, Entities are simple identifiers: 32-bit numbers. Game objects data is stored in a Manager class. Expression „Entity contains Component“ here means that a Component storage inside the Manager has registered the Entity and associated it with a Component object. Systems process

entities with required components, set by each System, in a loop. Creation, destruction and Component association of Entities is handled by the Manager.

Resources

For simplification of loading assets (and similar objects) into the game, this framework provides a Resources class. Essentially, it is a storage of pointers to object‘s data, along with pointers to functions for allocating and deallocating them

Scene

Finally, the Scene class contains game logic, an ECS Manager, and a set of Resources that need to be loaded in the scene when it starts.

4.2. Scene

In general, games can be divided into various number of scenes, for example: one scene (the entire game), three scenes (menu, loading screen, game) or hundreds of scenes (for each level/stage). Its purpose is mainly to divide game data and logic into smaller parts.

This framework‘s implementation of a scene is rather simple. Each Scene has a unique **`SceneID (size_t data type)`** and a set of Resources that are required to be allocated before the Scene can be initialized (SceneGraph handles the initialization).

Each Scene defines several functions (virtual functions are intended to be implemented by game developers in their projects by inheriting Scene class):

- virtual **`Update()`** function called every game tick by the main loop
- **`Init()`** function that first calls **`PreInit()`** and then **`PostInit()`**
- **`PreInit()`** which allocates required resources if they are not allocated
- virtual **`PostInit()`** function
- virtual **`reloadSettings()`** function that gets called upon modifications in values stored in the Settings class
- virtual **`DeInit()`** function called by Scene Graph when Scene gets deinitialized

Additionally, Scene class stores a pointer to **`SceneGraph`**, an instance of the ECS Manager and a vector of **`SDL_Event`** objects for user input. Using the pointer to **`SceneGraph`**, scene can call **`Change()`** function to change the current Scene.

4.3. Scene Graph

Scene Graph is an entry point for the game, containing the main loop. It stores pointers to scene, and for each (ordered) pair of Scenes allows for specifying a transition function between them.

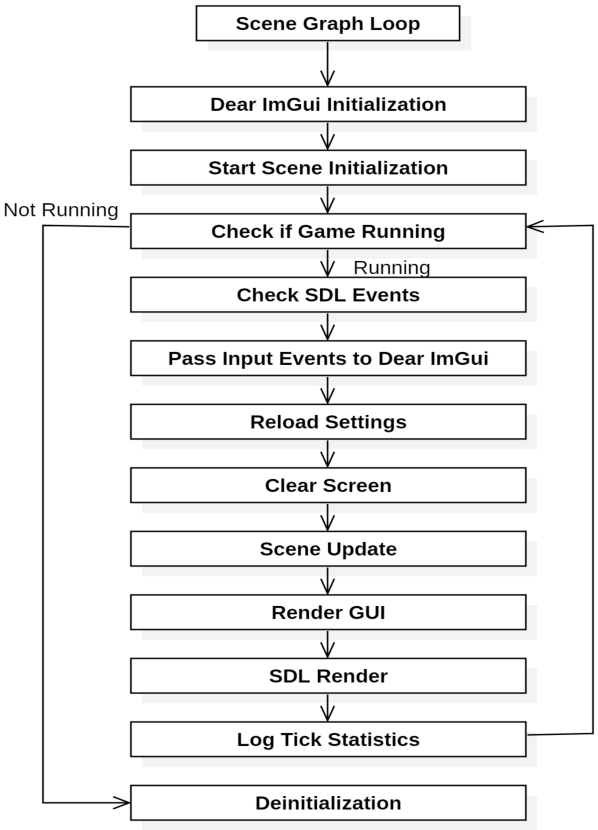

Illustration 11: Scene Graph - Main Loop

Illustration 11 shows how the main loop works. First, Dear ImGui is initialized, then start Scene is initialized. Next the loop checks if the game is running (boolean value stored in SceneGraph). If not, current Scene and Dear ImGui are deinitialized. If the game is running, the next step is checking SDL events for window changes and executing **`reloadSettings()`** if Settings have changed. Events indicating the game has exited are also checked. Then input events are passed to Scene and Dear ImGui. Next, Settings are checked, screen is cleared and **`Update()`** function of current Scene is called. Then Dear ImGui is rendered and after that, **`SDL_RenderPresent()`** is called which renders sprites

that were scheduled for rendering in the Scene. Final step of the loop is writing how long each phase took and other statistics to system console.

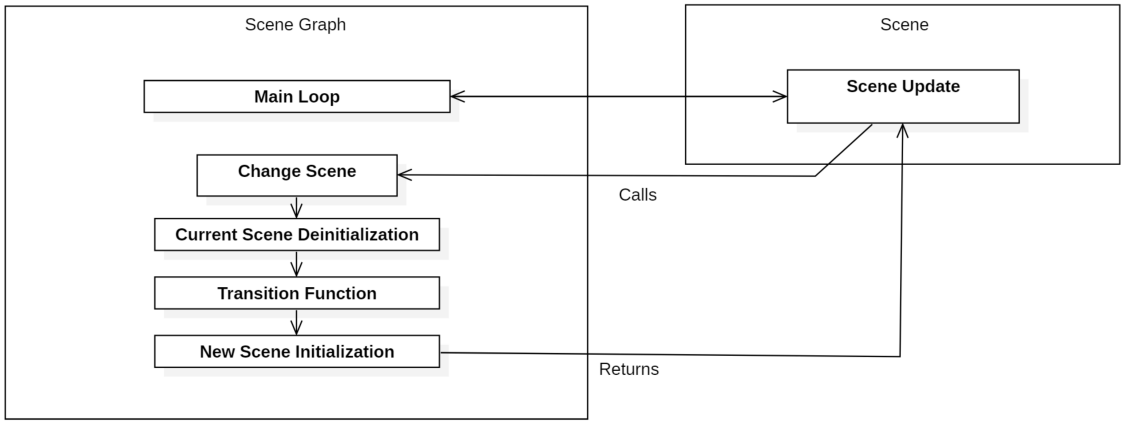

Illustration 12: Scene Change

The process of changing scenes is depicted on Illustration 12. To change the current scene, game logic calls **`Change()`** member function of the SceneGraph. First, current Scene gets deinitialized by calling the **`DeInit()`** function of the current scene. Then transition function, if present, is called and new Scene is initalized, via **`Init()`** call, and **`Change()`** returns. When using the **`DeInit()`** function to deallocate resources, **`Change()`** must be called after all operations with these resources have been finished.

4.4. Utility classes

To provide more abstraction, several global utility classes with various usages were implemented.

Resources

The Resources class functions as database of Resource objects. Resource is an object containing providing a function for allocating the respective resource (e.g., renderer object, Texture, or sound file), storing pointer to it, deallocating the resource and a function for checking whether the resource is allocated. In particular, assets in each scene are using this storage along with some subsystems, for example the RenderEngine subsystem which stores a **`SDL_Renderer`**.

Settings

This class also stores objects. It is a storage of **`std::string`** objects. In addition to that, it provides functions for conversion into different data types. For simple synchronization it provides a dirty bit and **`unsetDirty()`** functions which unsets it. The bit gets set when a stored value is modified.

Camera

This class stores the position and size of the camera and provides functions for converting global coordinates to relative and checking if a rectangle is visible by the camera.

Texture

Texture class provides high-level interface to loading a texture either from an image on disk or from text using a given font.

Logger

Logger is a class for writing messages to system console. Offers three fatality levels: Error, Warning, Info. It has an internal buffer that is written into using **`log()`** function and then is flushed into console using **`write()`** function.

Timer

Timer is a wrapper class providing a cross-platform precise timer using the plf::nanotimer library. In addition to it‘s functions for measuring time, it has a **`delay()`** function that waits a given amount of time and then returns.

4.5. Framework subsystems (Engines)

Apart from utility classes, subsystems in this section (further called Engines) are intended to be more complex and have a **`Start()`** function to initialize the Engine with startup options and needed Resources for it's functioning. Each Engine provides abstraction of lower-level libraries.

Each Engine is a singleton[26] accessed using either Engine template class or directly for static functions. This class provides static functions **`Start<T>`** and **`Stop<T>`** (where **`T`** is one of the Engine classes) for starting/stopping each Engine specified by the template parameter. These two functions operate over Engine instances and Initialize (if the Engine was not running), Deinitialize (if the Engine was running) or Reinitialize (using new options).

RenderEngine

RenderEngine initializes game window using SDL API calls: **`SDL_Init()`**, **`SDL_CreateWindow()`** and **`SDL_CreateRenderer()`**. After initialization provides access to **`SDL_Window`** and **`SDL_Renderer`** objects. RenderEngine also provides static funtions for rendering Textures and Box2D shape and body objects.

SoundEngine

SoundEngine initializes game audio using SDL API calls: **`Mix_Init()`** and **`Mix_OpenAudio()`**.

InputEngine

InputEngine currently serves as an example of an empty Engine implementation. In future work, this Engine will provide support for unified user input interface.

26 https://refactoring.guru/design-patterns/singleton/cpp/example

FileEngine

FileEngine serves as interface for reading and writing bytes from/to files (**`saveBytes()`** and **`loadBytes()`**). It also defines **`fileExists()`** and **`fileSize()`** functions for querying whether a file exists in the filesystem and how big it is in bytes.

LuaEngine

LuaEngine provides simple interface for scripting in Lua. It contains a lua::State object from the LuaState library and functions **`execute()`** and **`executeFile()`** which execute a script either directly from a string or from a file.

4.6. Components

Each Component class has the purpose to store data for game objects. Most only provide a constructor. Each Component class is also inherited from a base Component class that contains a component identifier. Illustration 12 shows the associations between Components and Systems.

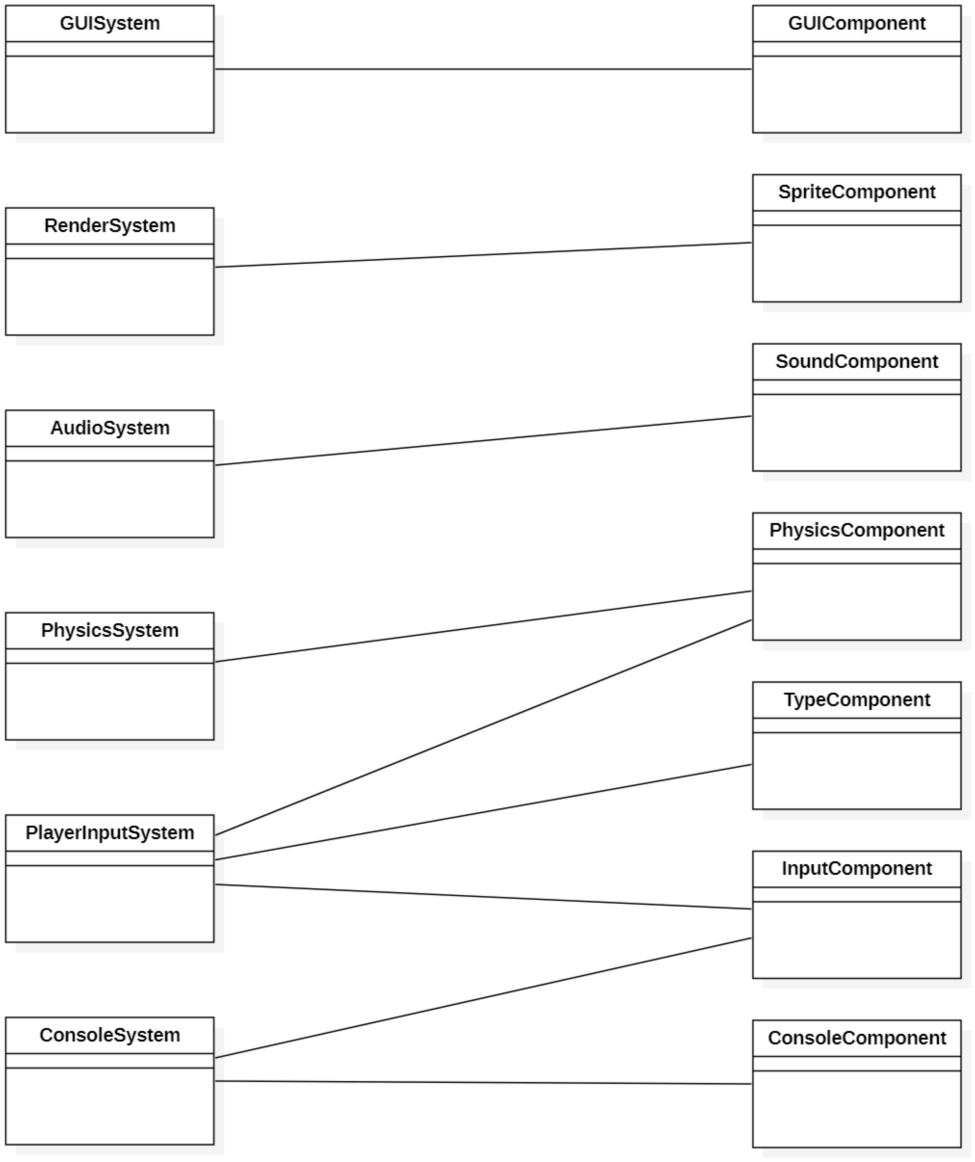

Illustration 13: Component - System associations

Console Component

Console Component stores two strings: default console text and current console text. It is used by the Console System.

GUI Component

GUI Component stores a name of a window, function to be called for its rendering, vector of return values and an arbitrary pointer for passing parameters to the function. It is used by the GUI System and helps with abstracting the complexity of the Dear ImGUI library which displays it‘s GUI using a sequence of calls to global functions (see Example 6) usually called from a single function per window each tick.

Input Component

Input Component stores a boolean value indicating whether the entity containing this component should process user input. It is used by the PlayerInput System and Console System.

Physics Component

Physics Component, used by the Physics System and, when needed, by the Render System. This component represents a physical object, either using Box2D physics engine or internal simple physics and either solid (not moving) or dynamic. Stores it‘s size, position, speed, maxSpeed and optionally a pointer to a Box2D‘s **`b2Body`** object. Provides three constructors. Two for simple physics and one for Box2D physics. In addition to constructors, it also implements functions for getting or setting the object‘s position, size and velocity.

Sound Component

Sound Component stores a pointer to **`Mix_Chunk`** and **`Mix_Music`** objects (from SDL). These two objects are independent. The difference between these two types is in the API. Sound component also stores a 32bit integer **`param`** and an 8bit unsigned integer representing a bit array of actions that needs to be taken. Actions that can be taken by the Sound Component are: play chunk, play chunk **`param`** times, play chunk infinitely, play music, pause music or stop music. Actions regarding chunks are independent from actions regarding music. This component is used by the Audio System.

Sprite[27] Component

Sprite Component stores either a pointer to a Texture object or a colour. Plus it stores rendering information: whether the sprite is bound to a physics body. source rectangle (which part of a texture should be rendered), destination rectangle (where on screen to render, if not bound to a physics body), angle; and information for animating. For animating it stores size of a tile, starting tile, how many consecutive tiles are part of the animation, how long each frame of animation lasts in game ticks, frame offset (which frame should be the starting frame) and tick offset (for synchronization purposes). It is used by the Render System.

Type Component

Type Component stores an 8bit integer for general purposes and to provide a minimal example of a component. It is only used by the PlayerInputSystem to distinguish player object from other objects that may have the same components.

27 Sprite - two-dimensional image representing an individual part of a larger whole

4.7. Systems

Each System processes (by calling **`updateEntity()`** function) every entity that contains its required Components (set in Systems constructor). Update function is called every game tick (every iteration of the main loop) for every entity separately. This function has a second parameter besides the entity: time since last tick (in seconds), which can be used by some systems.

Physics System

Physics System requires Physics Component. Box2D physics are handled separately by creating a Box2D's **`b2World`** object and updating it using it's **`Step()`** function, either directly in a scene's implementation or by creating a separate Component/System for it (see Final Example). This System only updates position of dynamic Physics Components using simple physics equations without handling collisions.

Render System

Render System requires Sprite Component and when needed, Physics Component. It renders each Sprite Component according to its settings. If the Component is set to be bound to Box2D body, it uses the entity's Physics Component for rendering over the physics body with correct angle.

PlayerInput System

PlayerInput System requires Input Component, Type Component and PhysicsComponent. For every entity it checks Input and Type Components, if Type contains 0 (indicating the entity is a player) and Input is set to be focused, user input is read and forces are applied to Physics Component based on keys defined in this class (w/a/s/d to move up/left/down/right). Maximum speed is then checked and corrected. Lastly, player body is angled so that it faces the mouse cursor.

Audio System

Audio System requires only the Sound Component. Depending on the action from the Components data, SDL API calls **`Mix_PlayMusic()`**, **`Mix_ResumeMusic()`** or **`Mix_HaltMusic()`** are called if **`Mix_Music`** is stored in the Component and the appropriate action is set. Independently, for **`Mix_Chunk`**, alternative way of storing audio, playing is handled by **`Mix_PlayChannel()`** function and the respective action in the Sound Component is reset.

Console System

Console System requires Input Component and Console Component. It handles reading keyboard input and displaying the console on screen. After entering a command and pressing enter, Console System parses the entered text and if it finds a matching command, calls the command with entered parameters. Each command is stored as a function with a vector of string as it‘s only parameter with a corresponding name. User functions can be defined in each scene.

GUI System

GUI System requires GUI Component. Each tick, function from GUI Component is called and values returned are stored in the Component. This way, Dear ImGUI windows are properly displayed.

4.8. Compilation

Framework is header-only and therefore does not require to be compiled, only included. When compiling a game using this framework, respective libraries must be linked or included (see example/README.md).

For compilation, GCC[28] supporting C++20 is required (at least GCC version 9.2.). Cross-compilation for Windows is supported using Mingw-w64[29]. Ubuntu 20.04 is recommended.

Steps to compile the frameworks examples (located in the example/ subdirectory):

1. Check if libraries in libs/ subdirectory exist. If not, download missing libraries and place them in corresponding sub:
   - https://gitlab.com/ZemanTomas/getoptcpp in libs/getoptcpp
   - https://github.com/spurious/SDL-mirror.git in libs/SDL2
   - https://github.com/bminor/SDL_ttf in libs/SDL2_ttf
   - https://github.com/bminor/SDL_image in libs/SDL2_image
   - https://github.com/mattreecebentley/plf_nanotimer.git in libs/plf_nanotimer
   - https://github.com/erincatto/box2d.git in libs/box2d
   - https://github.com/Tyyppi77/imgui_sdl.git in libs/imgui_sdl
   - https://github.com/ocornut/imgui.git in libs/imgui
   - https://github.com/SDL-mirror/SDL_mixer in libs/SDL2_mixer
   - https://github.com/Rapptz/sol.git in libs/sol
   - https://github.com/AdUki/LuaState.git in libs/LuaState
2. Install libraries: *sudo apt-get install liblua5.2-dev libsdl2-dev libsdl2-mixer-dev libsdl2-ttf-dev libsdl2-image-dev libbox2d2.3.0*

28 https://gcc.gnu.org/
29 http://mingw-w64.yaxm.org/doku.php/start

3. Install getopctpp library (from libs/getoptcpp subdirectory): *sudo make install* (if building for Windows: *sudo make win* and then move *getoptcpp.dll* in libs/win64/lib)
4. If not present, install GNU Make: *sudo apt-get install make*
5. For cross-compiling for Windows, install *g++-mingw-w64-x86-64*
6. For cross compiling for Windows, place Windows development libraries for SDL2, SDL2_image, SDL2_mixer, and SDL2_ttf in libs/win64
7. For cross compiling for Windows, place lua52.dll, liblua52.a, libBox2D.dll and libBox2D.a archives built for Windows in libs/win64/libs
8. From the example/ subdirectory:
    1. For Linux: *make EXAMPLEDIRECTORY=<example_directory>*
    2. For Windows: *make EXAMPLEDIRECTORY=<example_directory> win*
9. Game executable *game* will be created. (For Windows build, it is needed to put runtime DLLs next to the executable.)

5. Example game

In this chapter, an overview of example games is provided along with code examples and commentary. For game developers, these examples can serve as a basic guide on usage of the framework.

There are total of 8 example games, from the simplest example to the final example game. Each example builds on top of the previous one and adds new features. Assets for each example are stored in example/data subdirectory.

- First example shows how to create a game window
- Second example shows how to create a player character using ECS
- Third example shows how to work with the built-in console
- Fourth example shows how to create a physics object
- Fifth example shows how to use the audio system
- Sixth example shows how to create a simple GUI
- Seventh example shows how to implement animations and execute basic LUA scripts
- Final example shows how to extend the ECS by additional System and Component on a basic game prototype.

5.1. Example 1 - Empty Game

The first example is focused on doing the bare minimum to provide a working game window. Only two files containing code will be need for now (until the last example): *main.cpp* and *Game.hpp*. The first file should contain the **`main()`** function and contents of the second file will be discussed in this section. First thing in *main.cpp*, after including SceneGraph.hpp, that needs to be done is set default values using the Settings utility class.

```
Settings::getInstance()[Settings::FPS] = "60"; //Maximum FPS
Settings::getInstance()[Settings::SCREEN_WIDTH] ="800"; //Width of default window
Settings::getInstance()[Settings::SCREEN_HEIGHT] = "600"; //Height of default window
Settings::getInstance()[Settings::MEDIA_PATH] = "data"; //Location of assets
```

After that, RenderingEngine needs to start.

```
Engine::Start<RenderEngine>(REOptions()); //Start rendering engine
```

The next step is to initialize all resources. (And exit the program if this step fails)

```
if(!Resources::Initialize()){
    Logger::write();
    return 1;
}
```

Now that resources are ready, next step is to create SceneGraph instance and add a pointer to a scene to it.

```
SceneGraph graph;
graph.Add(new ExampleGame(&graph));
```

Before further describing implementation of the ExampleGame class, last line of code that will enter the scene graph loop.

```
graph.Loop(0);
```

This function takes one parameter - SceneID, and will start the game loop using the respective class as current scene.

In Scene implementation, two files need to be included: ecs/ECS.hpp - containing all needed Components and Systems; and utils/Scene.hpp - containing Scene interface definition.

Any user implemented Scene must inherit Scene class and provide its constructor with a pointer to the SceneGraph instance declared earlier. Scene, base class, also requires unique integer as its SceneID and a pointer to vector of pairs **<ResourceID,Resource*>** used as a requirement for this scene to allocate during the initialization.

Function **reloadSettings()** should be called every time there is a change in any Settings variable that needs to apply changes in a special way. For example if current SceneID would be stored in a Settings variable, then this function would check if current SceneID is equal to the one stored and if not, change the current scene to match.

Function **Update(const float dt)** is called every game tick. Parameter dt here is the amount of time elapsed since the last game tick.

Function **`PostInit()`** and get called after the scene is loaded (when the current scene in SceneGraph changes to it, including at the start as entry point).

Function **`DeInit()`** is called after the scene ends (game quits or current scene changes to a different scene).

```
class ExampleGame : public Scene{
    public:
     ExampleGame(SceneGraph* _sg):
        Scene(_sg,0,new std::vector<std::pair<size_t,Resource*>>({}))
    {}
    void PostInit() override{}
    void reloadSettings() override{
        //Required to be called here to unset dirty bit indicating change
        Settings::unsetDirty();
    }
    void Update(const float dt) override{}
    void DeInit() override{}

};
```

Illustration 14: Example 1 - Empty Game

5.2. Example 2 - Player

The goal of the second example is to create a player character. This example uses *player.png* as the player texture. After resource initialization, function **`Resources::AddResource(Resource*,size_t,bool)`** is called with a pointer to a Resource instance containing functions needed to allocate a player texture, Resource ID and a true value, indicating that the resource should be immediatelly allocated.

In the scene constructor there are 4 main parts:

- Creating Component storage for all Components that are going to be used in the scene.
- Creating Systems, that are needed. Physics and Render System in this case.
- Creating a Box2D world needed for PhysicsComponent.
- Creating the Player entity. In this case it requires PhysicsComponent to be able to be interacted with later and SpriteComponent so it can be rendered.

```
//Create Component storages
manager.createComponentStore<PhysicsComponent>();
manager.createComponentStore<TypeComponent>();
manager.createComponentStore<SpriteComponent>();
//Create Systems
manager.addSystem(System::SysPtr(new PhysicsSystem(manager)));
manager.addSystem(System::SysPtr(new
RenderSystem(manager,RenderEngine::getRenderer(),textures,&cam)));
//Create Box2D world
world = new b2World({0.0f,0.0f});
//Create player
player = manager.createEntity();//Creating entity.
float_2D size = {24.0f,24.0f};
float_2D pos = {1000.0f,1000.0f};
manager.addComponent(player,PhysicsComponent(world,pos,size,true,true,
{0.0f,1.0f,0.1f,0.0f,15.0f}));
manager.addComponent(player,SpriteComponent(RenderEngine::getPlayerTexture(),
{10,3,13,26}));
manager.registerEntity(player);//Registering entity in the ECS
```

Next, update function needs to contain code to update the ECS and Box2D world. And since the player can be anywhere in the map, RenderSystem needs a Camera object reference to know which SpriteComponents to render. For that reason, implementation of player tracking is added to this example.

```
void Update(const float dt) override{
    manager.update(dt);
    world->Step(1/60.0f,6,3);//Box2D documentation suggests constant update delta
    updateCamPos(dt,player,cam);
}
```

Additionally, **`reloadSettings()`** function should now contain code for updating the size of the camera, since window size is contained in Settings variables and after window size changes, the variables are updated.

Illustration 15: Example 2 - Player

5.3. Example 3 - Console

In this example, the console is implemented along with functions for printing the player location.

First, the game needs to know where the font is located relative to the media directory:

```
Settings::getInstance()[Settings::FONT] = "font.ttf";
```

Then, similarly to adding player textures, font Resource is added. In the constructor, ConsoleComponent storage is created, custom console functions are to a map container that is then passed to ConsoleSystem.

Console is an entity and as such needs to be created, have components added to it (in this case Input and Console Components) and then registered using the ECS manager object.

```
console = manager.createEntity();
manager.addComponent(console,InputComponent(false));
manager.addComponent(console,ConsoleComponent(""));
manager.registerEntity(console);
```

In the example, there are 2 functions intended for use by the console.

- **`teleportPlayer(float,float)`** - changes location of the player
- **`writePlayerPos()`** - writes current location of the player to system console

Reference to these functions is passed along with the console function name mapping in the ConsoleSystem constructor. Every function passed in the console takes a vector of strings as a parameter. Additional informations passed with the functions make sure there is a correct number of arguments when using the commands.

```
map<string,pair<function<void(vector<string>&)>,size_t>> consoleFunctions = {
    {"teleportPlayer",
        {[this](vector<string>& par){teleportPlayer(stof(par[0]),stof(par[1]));},2}
    },
    {"displayPosition",
        {[this](vector<string>& par){writePlayerPos();},0}
    }
};
manager.addSystem(System::SysPtr(new ConsoleSystem(manager,&events,
                                                   RenderEngine::getFont(),
                                                   RenderEngine::getRenderer(),
                                                   _sg,&consoleFunctions))
                                                );
```

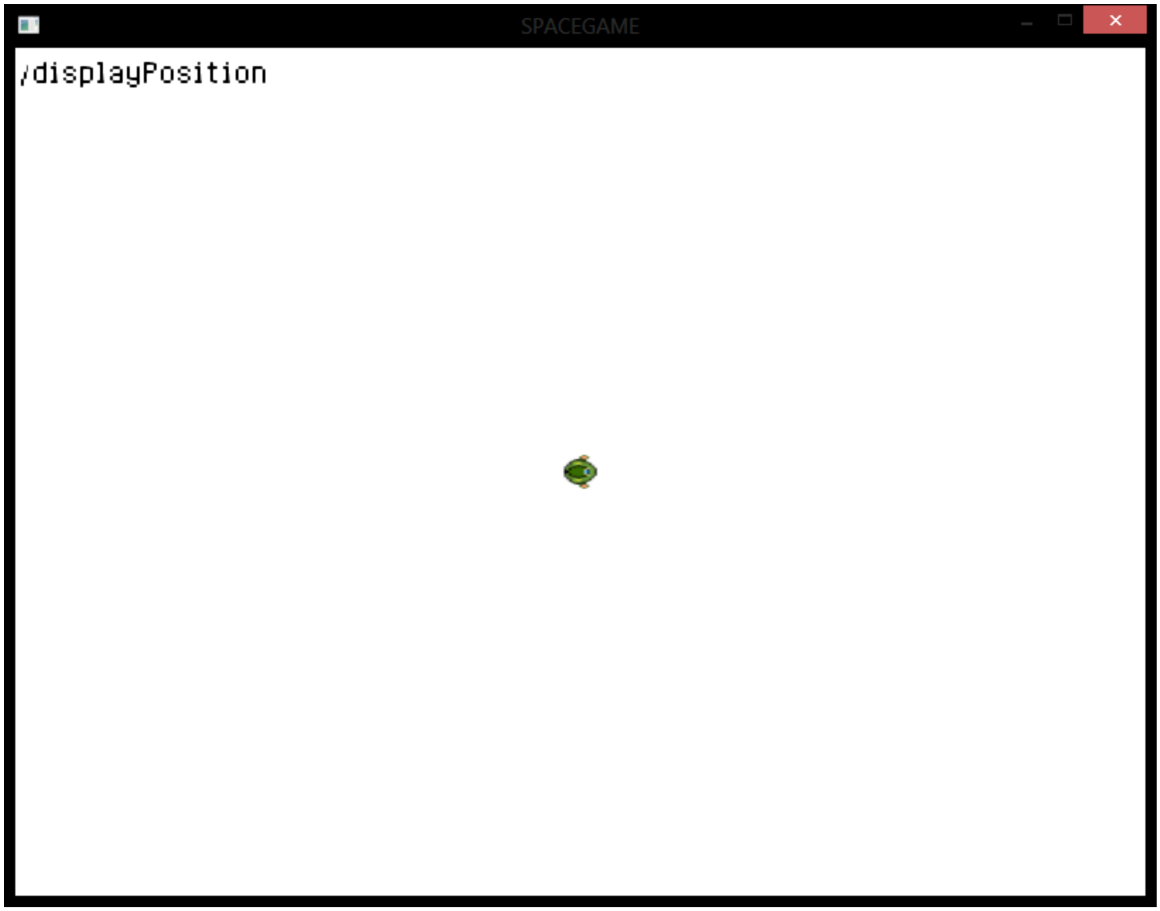

Illustration 16: Example 3 - Console

5.4. Example 4 - Physics

Fourth example implements another physics object, a box, and controls of the player entity. Player controls are handled by PlayerInputSystem. Its contructor requires, besides a reference to the ECS manager required by all Systems, pointer to a vector of events from the Scene base class and a pointer to a camera instance.

Next, PlayerInputSystem only processes entities with TypeComponent with 0 as type and InputComponent. These two Components should be added to player entity in order to control it.

Creating a box is similar to creating the player entity. In this case the difference is that a box only requires Physics and Spire Component. Another difference is that box is a rectangle and not moving, and for its rendering only a colour is needed now.

```
box = manager.createEntity();
float_2D size_b = {34.0f,34.0f};
float_2D pos_b = {1050.0f,1050.0f};
manager.addComponent(box,PhysicsComponent(world,pos_b,size_b,false,false,
{0.0f,1.0f,0.1f,0.0f,15.0f}));
manager.addComponent(box,SpriteComponent(0xA00000FF));
manager.registerEntity(box);
```

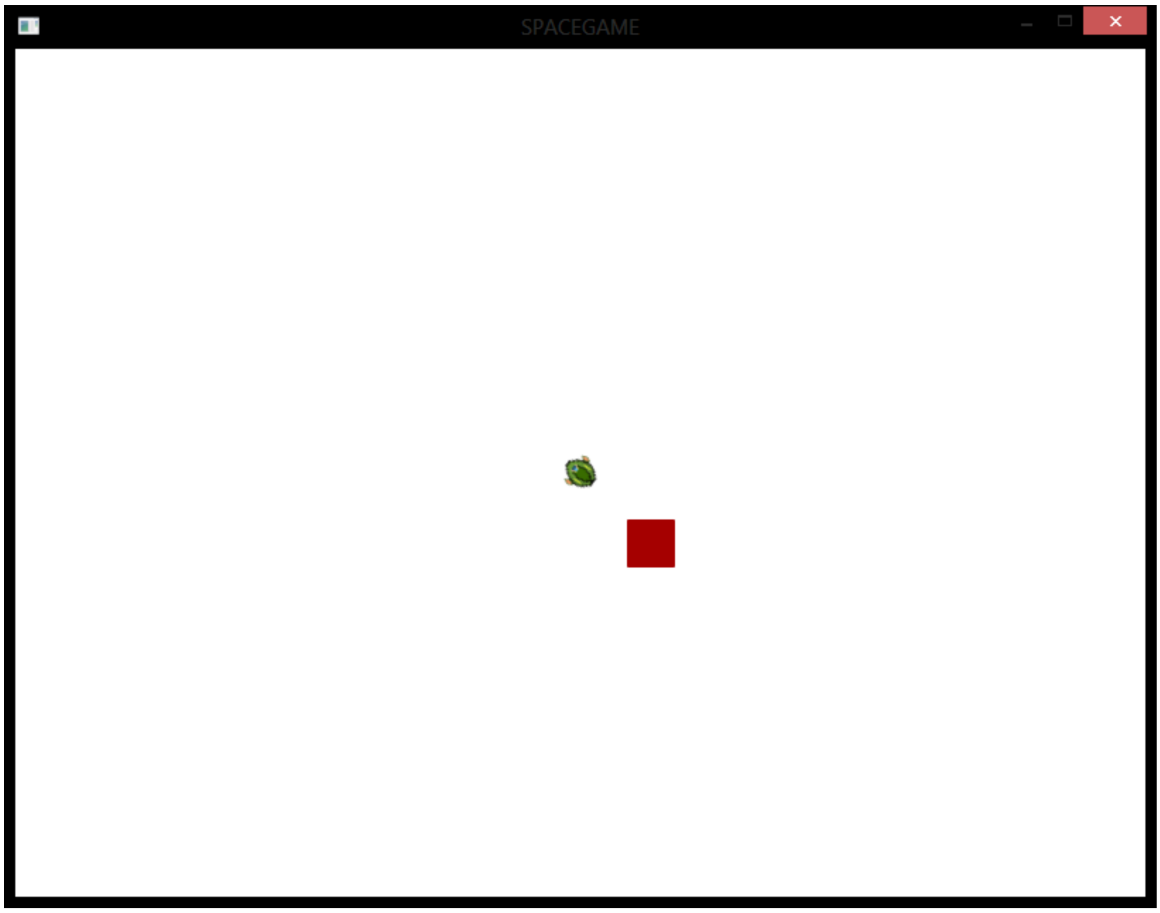

Illustration 17: Example 4 - Physics

5.5. Example 5 - Audio

Playing audio requires a working sound engine and a sound file (*beat.wav*).

```
Settings::getInstance()[Settings::MUSIC] = "beat.wav";
Settings::getInstance()[Settings::MUSIC_ON] = "true";//Required framework variable
Engine::Start<SoundEngine>(SEOptions());
```

The sound file is loaded using **`Mix_LoadMUS(char*)`** function which returns pointer to a Mix_Music object.

```
//Mix_LoadMUS for loading music files, Mix_LoadWAV for chunks, both are from SDL_mixer
Mix_LoadMUS((Settings::getMEDIA_PATH()/Settings::getMUSIC()).string().c_str())
```

Music is then created as an entity with only SoundComponent added to it. SoundComponents constructor takes a pointer to Mix_Music or Mix_Chunk object.

To be able to turn music on and off, the variable *action* in SoundComponent needs to be modified. In function **`reloadSettings()`**:

```
if(music != INVALID_ENTITY){
    SoundComponent& snd = manager.getComponentStore<SoundComponent>().get(music);
    if(Settings::getMUSIC_ON()){snd.action |= MUSIC_PLAY;}else{snd.action &=
~MUSIC_PLAY;}
}
```

5.6. Example 6 - GUI

GUI implementation consists of GUISystem and GUIComponent, which takes two parameters, window name and a window function. The window function returns a vector of arbitrary pointers - output variables, and takes window name and an arbitrary pointer as parameters. The goal of this example is to create a window with a slider for controlling players max speed.

```
static vector<void*> InitGUI1(string name, [[maybe_unused]] void* user_data){
    static float slider1 = 15.0f;
    static float color[4] = {1.0f,1.0f,1.0f,1.0f};
    vector<void*> params = {&slider1,color};
    ImGui::Begin(name.c_str());//Create window
    ImGui::SliderFloat("maxSpeed", (float*)params[0],1.0f,40.0f);
    ImGui::ColorEdit3("blockColor",(float*)params[1]);//Unused
    ImGui::End();//End creating window
    return params;//Output pointers to slider values and colour picker values
}
```

Now, to access window variables and modify players maxSpeed, in Update function:

```
PhysicsComponent& p = manager.getComponentStore<PhysicsComponent>().get(player);
GUIComponent& w1 = manager.getComponentStore<GUIComponent>().get(window1);
p.maxSpeed = *(float*)w1.values[0];//Accessing window value
```

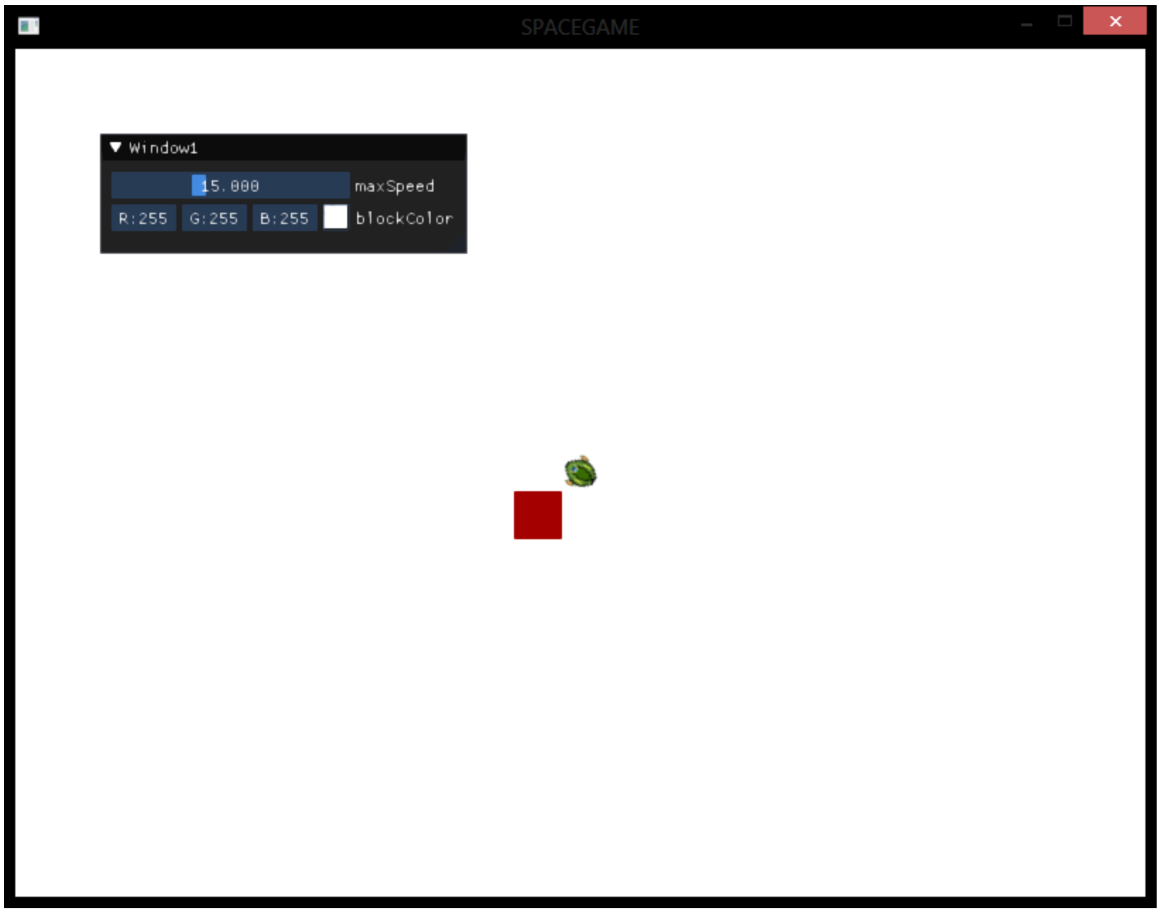

Illustration 18: Example 6 - GUI

5.7. Example 7 - Animations & Scripting

This example has two goals, animating a box texture and executing a script. Animation is set in the SpriteComponent. only requirement is that the component should contain a tilemap with the animation frames.

```
//Sets SpriteComponent animation
//parameters: tile size 32x32, starting tile at (6,2), two frames long animation (left to right)
manager.getComponentStore<SpriteComponent>().get(box).SetAnimation({32,32},{6,2},2);
```

Scripting requires LuaEngine. After that, scripts can be executed globally, either directly or by specifying a script file to execute.

```
//For binding functions or variables, use getState() and work with lua::State
Engine::getInstance<LuaEngine>().execute("print 'Hello lua!'");
```

5.8. Final Example

The last example expands on the previous example by three main features. First feature is a procedurally generated world. This is done by implementing WorldSystem and WorldComponent (see example/ExampleFinal/WorldComponent.hpp and WorldSystem.hpp). Assets needed for this example are *player.png, tiles_small.png, tiles.png* and *beat.wav*.

Second feature is a mouse user input. Mouse clicks modify world tiles.

Third feature is a console command for saving the modified world on disk.

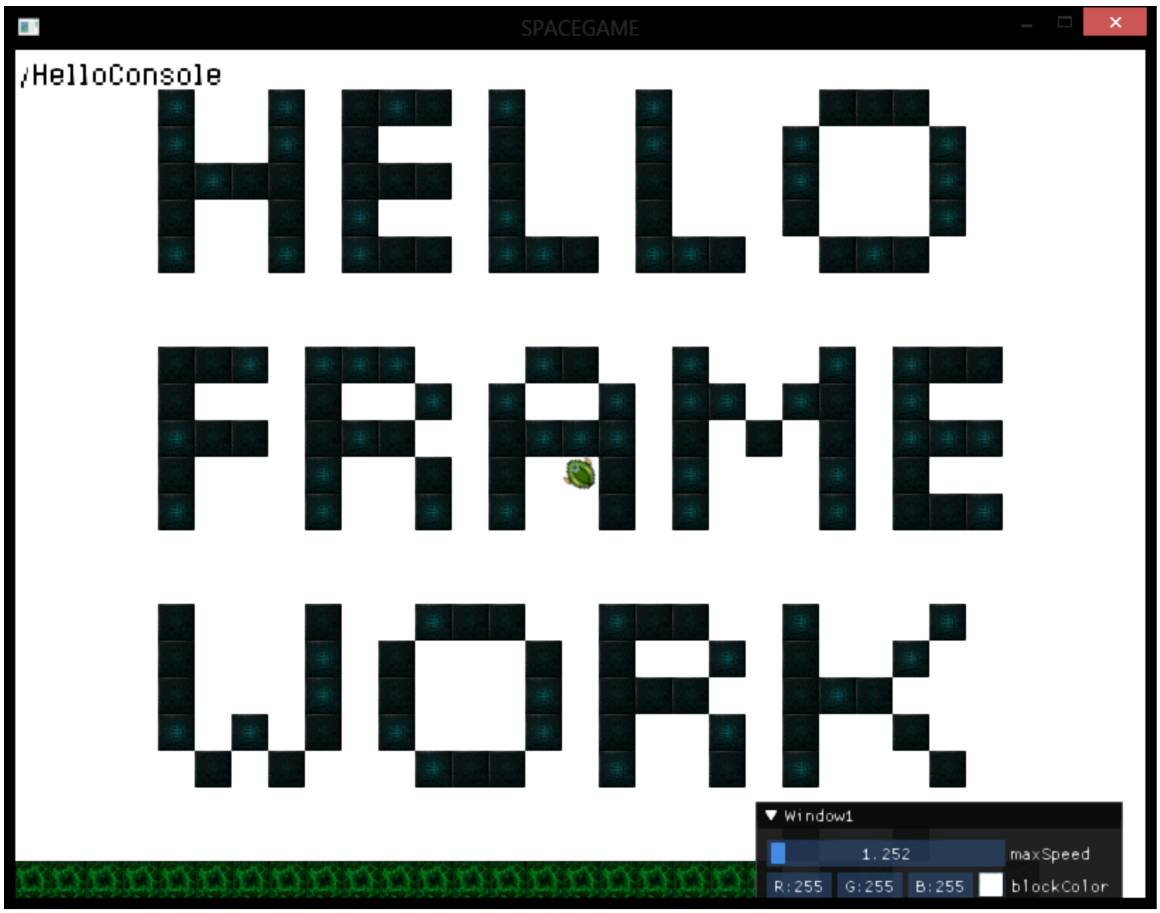

Illustration 19: Final Example

Conclusion

Creating a game framework is a very difficult task even for dedicated groups of people. After analysing currently used game engines and frameworks and selecting required components, a framework that encompasses minimum needed functionality for game developers has been created. Resulting framework supports, among others, dividing game logic and data into scenes, physics simulation using the Box2D library, audio, scripting in Lua using the LuaState library, basic GUI provided by the Dear ImGui library, rendering textures using the SDL library. By natively developing in C++, developers can fully use hardware and optimize games. Yet it can be easily extended to provide the same features as popular commercial frameworks.

Literature

[1]: Malhotra, M. (2021). Unreal Engine vs Unity 3D Games Development: What to Choose?. https://www.valuecoders.com/blog/technology-and-apps/unreal-engine-vs-unity-3d-games-development/

[2]: Sirani, J. (2017). GTA Publisher Acquires Space Sim Kerbal Space Program. https://www.ign.com/articles/2017/05/31/gta-publisher-acquires-space-sim-kerbal-space-program

[3]: Blizzard Entertainment. (2021), Year of the Phoenix in Review. https://playhearthstone.com/en-us/news/23625669/year-of-the-phoenix-in-review

[4]: Valentine, R. (2020). Subnautica has sold over 5m copies. https://www.gamesindustry.biz/articles/2020-01-14-subnautica-has-sold-over-5m-copies

[5]: McAloon, A. (2021). Epic v. Apple trial offers rare look into Epic financials, billions of Fortnite revenue. https://www.gamedeveloper.com/business/epic-v-apple-trial-offers-rare-look-into-epic-financials-billions-of-i-fortnite-i-revenue

[6]: Handrahan, M. (2017). Life is Strange was bought by 3 million people. https://www.gamesindustry.biz/articles/2017-05-19-life-is-strange-was-bought-by-3-million-people

[7]: Patra, I. (2021). Epic Games' Fortnite earned $9 billion in its first two years . https://www.thehindu.com/sci-tech/technology/epic-games-fortnite-earned-9-billion-in-its-first-two-years/article34489818.ece

[8]: Batchelor, J. (2019). How Ark: Survival Evolved "fell into sustainable revenue" without skins or loot boxes. https://www.gamesindustry.biz/articles/2019-08-16-how-ark-survival-evolved-fell-into-sustainable-revenue-without-skins-or-loot-boxes

[9]: manos426f. (2016). Interviews with the Daedalic Entertainment team. https://ragequit.gr/specials/item/ragequit-gr-at-daedalic-days-2016-part-2-interviews-english-text-edition/

[10]: Orland, K. (2018). Valve leaks Steam game player counts; we have the numbers. https://arstechnica.com/gaming/2018/07/steam-data-leak-reveals-precise-player-count-for-thousands-of-games/

[11]: Chapple, C. (2018). Fingersoft's Hill Climb Racing franchise rolls past one billion downloads. https://www.pocketgamer.biz/interview/68010/fingersofts-hill-climb-racing-franchise-rolls-past-one-billion-downloads/

[12]: Boudreau, I. (2019). Slay the Spire has sold more than 1.5 million copies. https://www.pcgamesn.com/slay-the-spire/slay-the-spire-sales

[13]: Marks, T. (2019). Celeste Sequel (Probably) Won't Happen, Developer Says. https://www.ign.com/articles/2019/09/07/celeste-developer-doesnt-want-to-make-a-sequel-new-game-in-the-works

[14]: Regalis. (2021). Two years of Early Access. https://barotraumagame.com/announcements/two-years-of-early-access/

[15]: Gregory, J. (2018). 1.6 Runtime Engine Architecture. In Game Engine Architecture, Third Edition. Taylor & Francis Ltd.

[16]: West, M. (2007). Evolve Your Hierarchy. https://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

List of Illustrations

Illustration 1: Unity Editor 2019 - Windows....9
Illustration 2: Unreal Engine 4 Editor....10
Illustration 3: Godot Editor....12
Illustration 4: GameMaker Studio 2 - Editor....13
Illustration 5: Cocos2d-x - Architecture....14
Illustration 6: libGDX - life cycle of a game....15
Illustration 7: XNA Framework - Architecture....16
Illustration 8: Framework Architecture....32
Illustration 9: Class Diagram....33
Illustration 10: Lifecycle of a Game....34
Illustration 11: Scene Graph - Main Loop....38
Illustration 12: Scene Change....39
Illustration 13: Component - System associations....44
Illustration 14: Example 1 - Empty Game....54
Illustration 15: Example 2 - Player....56
Illustration 16: Example 3 - Console....58
Illustration 17: Example 4 - Physics....60
Illustration 18: Example 6 - GUI....63
Illustration 19: Final Example....65

List of Tables

Table 1.1: Framework/Engine comparison - Pricing....18
Table 1.2: Framework/Engine comparison - Platform support....18
Table 1.3: Framework/Engine comparison - Language support....19
Table 1.4: Framework/Engine comparison - Development options....19
Table 3.1: Framework Components - Comparison....27

Attachments

1. GitLab link to the resulting framework:
   https://gitlab.com/ZemanTomas/spacegame
2. Source Code in a .zip archive.