

Cloud Engine

Requirements Specification & Technical Design Document

William Dewaele

Charly Jeauc

Saad Raouf

ISART'
DIGITAL

Supervised by : **Stephan Rampin**

Summary

Summary	2
Team members & Organisation:	5
1.1 Team Members	5
1.2 Organisation	5
1.2.1 Methodology:	5
1.2.2 Version Control:	6
Introduction:	7
2.1 Purpose:	7
2.2 Document Conventions:	7
2.3 Intended Audience and Reading Suggestions:	8
2.4 Project Scope:	8
Nomenclature & Naming Conventions:	9
3.1 Coding Conventions :	9
3.1.1 Language standard :	9
3.1.2 Explicit Naming :	9
3.1.3 Naming Convention :	9
3.1.4 Include Convention :	10
3.1.5 Structs & Classes :	10
3.2 Typical file format :	11
Overall Description:	13
4.1 Product Features:	13
4.1.1 Editor:	13
4.1.2 Scripting:	13
4.1.3 Executable Game:	13
4.2 User Classes and Characteristics:	13
4.2.1 Game Programmers:	13
4.2.2 Game Designers:	14
4.2.3 Artists:	14
4.2.4 Sound Designers:	14
4.3 Operating Environment:	14
4.3.1 Operating System:	14
4.3.2 Hardware Requirements:	14
4.4 Design and Implementation Constraints:	15
4.5 Technology Enablers:	15

System Architecture:	16
5.1 System Architecture Diagram:	16
5.2 System Components:	17
5.2.1 Third party Software Development Kits (SDKs)	17
5.2.2 Platform Independence Layer	17
5.2.3 Core Systems	18
5.2.4 Resources	18
5.2.5 Low Level Renderer	19
5.2.5 Profiling & Debugging	19
5.2.7 Physics & Collisions	20
5.2.8 Audio	20
5.2.9 Scene graph & Culling Optimization	20
UML	21
6.1 General Diagram	21
6.2 Entity Component System UML Diagram	22
6.3 Resource Manager UML Diagram	23
6.4 Scene Manager UML Diagram	24
6.5 Rendering UML Diagram	25
6.6 Physics UML Diagram	26
6.7 AltMath	27
6.8 Editor	28
Technical Issues	30
7.1 Rendering	30
7.2 Core	31
7.3 Tools	32
References:	33

Revision History

Name	Date	Reason For Changes	Version
Jeauc	07/01/19	Update APIs versions, remove boost and update layout	0.2
Dewaele	18/02/19	Update following the end of the first part.	0.3

1. Team members & Organisation:

1.1 Team Members

Saad Raouf	s.raouf@student.isartdigital.com
Charly	c.jeauc@student.isartdigital.com
William	w.dewaele@student.isartdigital.com

1.2 Organisation

1.2.1 Methodology:

The team will adopt an Agile methodology using Scrum. This iterative approach has proven its merits especially to its ability to embrace changes; which is rather frequent in game programming. The team uses HacknPlan online tool to organise the product backlog and sprints. The team also meets every morning (after coffee) in a brief stand-up meeting where each member answers three questions:

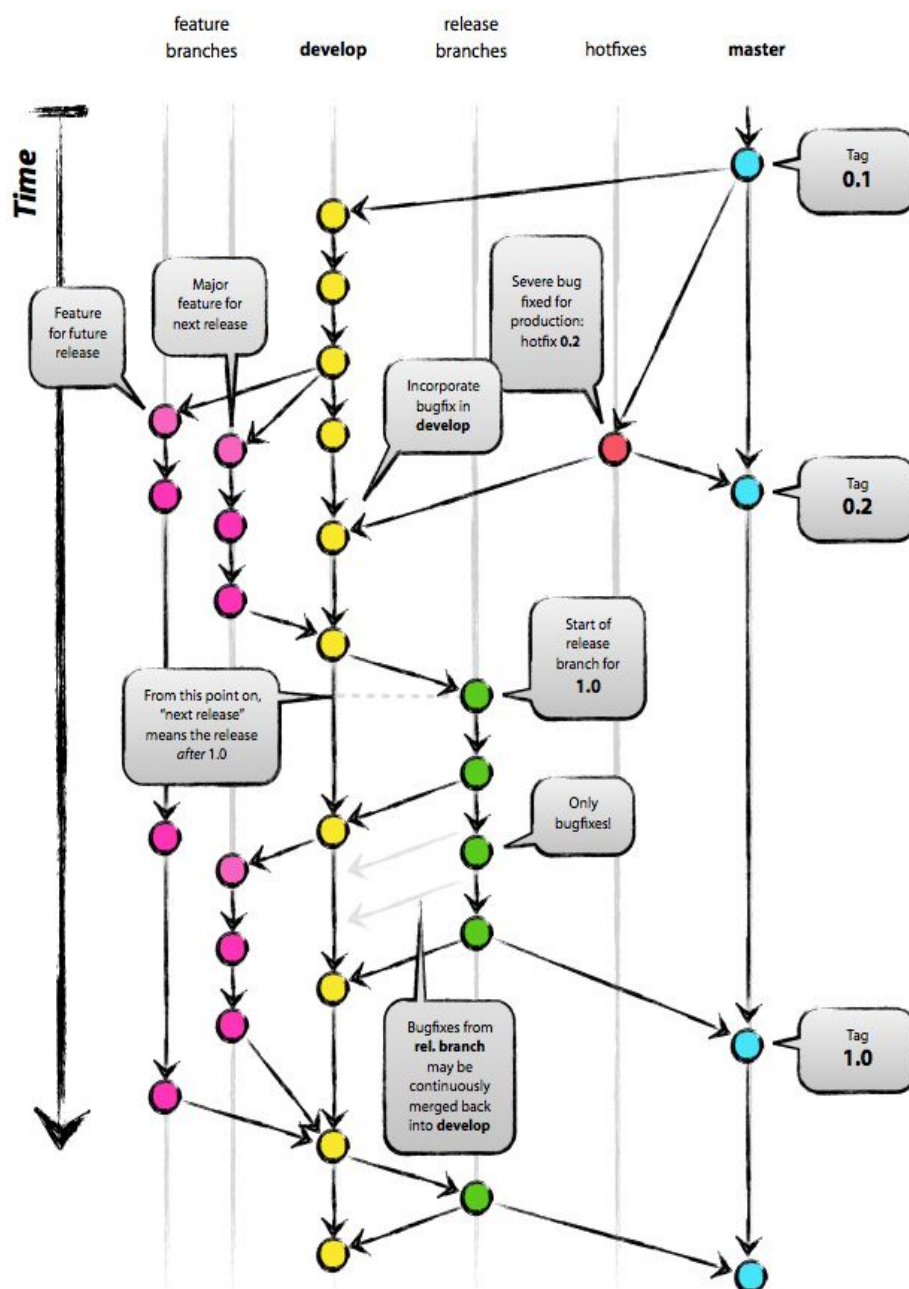
What did you do yesterday ?

What are you planning to do today ?

Do you have any difficulty ?

1.2.2 Version Control:

For the engine development, a version control system is primordial. It will allow the restoration of previous versions of files or of the whole project and also simply the work as a team. Git is a tool responding to all our needs and with which we are familiar. We will use a gitflow so each team member can work on different features at the same time without conflicting with other developers.



gitflow exemple

New development are built in feature branches, which are branched off of develop branch.

When features are ready for release, they are merged into develop branch.

If it's time for a release, a release branch is created off of develop branch. The point of this branch is to test the features onto a suitable test environment and to fix them if necessary.

When the release is finished, the release branch is merged into master and into develop so we can keep the fixes of the release.

The master branch will only have the code merged from release and hotfix branches.

The hotfix branches are used to create emergency fixes from the master and when done are merged back into master and develop.

2.Introduction:

2.1 Purpose:

The Cloud Engine is the capstone project for our AEC Game Engine Programming. The purpose is to develop a game engine with a graphic interface similar to Unity and Unreal Engine. The ultimate goal is to make a first-person puzzle game using the Cloud Engine. The pedagogical purpose is to perfect our knowledge in key engine development areas.

The first phase consists of the documentation: presenting the project, produce UML diagrams, establishing coding conventions (naming convention, namespaces, files structure..), listing external libraries and technology enables.

This project presents multiple challenges; some of these challenges:

- Investigation and Research
- Technical
- Architectural
- Organizational and communication

2.2 Document Conventions:

The table below lays definitions, acronyms and abbreviations:

Cloud Engine	The finished product, the game engine.
Cloud Tech	The team constituted by its members as specified in the Team members section of this document
Engine	
Editor	The graphical user interface for interacting with Cloud Engine
Product	The end result of the project: Engine + Editor

DLL	Dynamically Linked Library
API	Application programming interface: simplifies programming by abstracting the underlying implementation and exposing objects or actions the developer needs

2.3 Intended Audience and Reading Suggestions:

The primary intended audience of this document would be the team itself for organizational reasons; but also for the capstone committee in order to clearly demonstrate the steps, decisions and approaches towards the end goal.

This document is also intended for other game developers, students and enthusiasts that are interested in the process of making game engines. In case that is you, in case of questions or any other related matters feel free to write an email to any team member. We would be happy to help :)

Better resolution diagrams can be found within the documentation folder.

2.4 Project Scope:

The project spans on 6 months period, with various phases to guide the development. In order for the project to be achievable within the time frame, only key features relevant to making a first-person puzzle game will be developed.

The end product should be consisted of an engine DLL that is separate and a graphical editor that exploits the engine.

3.Nomenclature & Naming Conventions:

3.1 Coding Conventions :

3.1.1 Language standard :

In order to have access to the latest C++ standard features, The cloud engine will be coded with C++ 17.

3.1.2 Explicit Naming :

Classes, structs, variables and methods naming must be explicit, no name abbreviation is allowed. Naming should not bring any confusion and should describe as much as possible its purpose.

Header files should contain only one class and both should have the same name.

3.1.3 Naming Convention :

- Classes' name should be in UpperCamelCase
- Structs' name should be in UpperCamelCase
- Classes' members should start with "m_" and be in lowerCamelCase (m_myVariable)
- Const static classes' members should be in UpperCamelCase
- Structs' members should be in lowerCamelCase
- Methods should be in UpperCamelCase
- Methods' parameters should start with "p_" and be in lowerCamelCase (p_myParameter)
- Files' name should be in UpperCamelCase
- Abstract classes should start with "A" and be in UpperCamelCase (AMyClass)
- Interface should start with "I"(capital "i") and be in UpperCamelCase (IMyInterface)

- Namespaces should be in UpperCamelCase
- Macros should be in SCREAMING_SNAKE_CASE

3.1.4 Include Convention :

- Libraries includes should be between chevrons (`#include <stdio>`)
- Project files includes should be between double quotes (`#include "MyClass.h"`)

In header files (.h), includes should be avoided; Forward declarations are encouraged.

3.1.5 Structs & Classes :

Structs are used to encapsulate data (usage is data driver). Classes are used to encapsulate data where functionalities and methods are relevant to data it self. With this in mind classes should not have any member public and structs should not have any member private.

3.2 Typical file format :

```

C MyClass.h x
1  /**
2   * Cloud Engine
3   * @author CloudTech
4   */
5
6  #pragma once
7
8  #define MACRO_WITH_INTERACTION 0
9
10 #include <vector>
11 #include "MyStruct.h"
12
13 #define MY_MACRO 65823
14
15 namespace MyNameSpace
16 {
17     /**
18     * MyClass is an exemple of struct format
19     */
20     class MyClass
21     {
22     private:
23         /**
24         * MyPrivateMethod is a private Methode exemple
25         */
26         void MyPrivateMethod();
27
28     public:
29         /**
30         * MyPublicMethod is a public Methode exemple
31         * @param p_myParameter here describe the parameter
32         */
33         int MyPublicMethod(int p_myParameter);
34
35     private:
36         std::vector<MyStruct> m_myStructs;
37         float m_floatAttribute;
38
39     public:
40         int m_thisCaseMustBeAvoid;
41         static const int StaticConstAttribute;

```

- File header
- Include guards
- Macro changing behavior of includes
- Includes files
- Macro not changing behavior of includes
- Namespace
- Class description
- Class name
- Private methods description
- Private methods parameters
- Private methods declaration
- Public methods description
- Public methods parameters
- Public methods declaration
- Private attributes
- Public attributes are to avoid except for static const ones

Class nomenclature

```
namespace MyNameSpace
{
    /**
     * MyStruct is an exemple of struct format
     */
    struct MyStruct
    {
        public:
            int myAttribute;

        private:
            std::string thisCaseMustBeAvoid;

        public:
            /**
             * MyPublicMethod is a public Methode exemple
             * @param p_myParameter here describe the parameter
             */
            int MyPublicMethod(int p_myParameter);

        private:
            /**
             * MyPrivateMethod is a private Methode exemple
             */
            void MyPrivateMethod();
    };
}
```

- Namespace
- Struct description
- Struct name
- Public attributes
- Private attributes are to avoid
- Public methods description
- Public methods parameters
- Public methods declaration
- Private methods description
- Private methods parameters
- Private methods declaration

Struct nomenclature

4. Overall Description:

4.1 Product Features:

This section is not intended to be an exhaustive list but rather a high level summary of major features or significant functions that the product should perform or lets the user perform.

4.1.1 Editor:

The end product must have an Editor graphical interface composed of: assets browsing window, scene window, selectable scene objects with gizmos and inspector window.

4.1.2 Scripting:

The end product must support scripting in order to let the user implement different gameplay mechanics (equivalent to MonoBehaviour from Unity).

4.1.3 Executable Game:

The end product must build a first-person puzzle game as an external .exe.

4.2 User Classes and Characteristics:

The team anticipates different user classes of the end product. These users are differentiated by the five basic disciplines in Game development: Game Programmers, Game Designers, Artists, and Sound Designers.

4.2.1 Game Programmers:

In addition to scripting, Cloud Engine should have a debug console. This console should be able to display engine classes such as vectors, display entities names, and various useful information.

4.2.2 Game Designers:

Cloud Engine should be game designer friendly: Easy manipulation of objects for level design, and the possibility to tweak values for balancing.

4.2.3 Artists:

Cloud Engine should support most notable file formats for models (fbx, obj). For textures png and jpeg, and also light manipulation.

4.2.4 Sound Designers:

Cloud Engine should support basic sound manipulation: Sound Level, fading...

4.3 Operating Environment:

4.3.1 Operating System:

Windows 10, 64 bits.

4.3.2 Hardware Requirements:

Minimum Hardware:

- 2.5 GHz Processor (i5)
- 3GB RAM
- 3D Accelerator Card with DirectX 9.0 support. 512 MB onboard video ram, hardware transformation and lighting support
- 300 MB Hard-Drive space available

Recommended Hardware:

- 3.5 GHz Processor (i7 - 7700)
- 4 GB RAM
- Nvidia GTX 860m / AMD Radeon R9
- 300 MB Hard-Drive space available

4.4 Design and Implementation Constraints:

Most of the following constraints are extracted from the assignment document:

- Cloud Engine should be a Windows executable
- The engine should be as a separate DLL
- Only one rendering API should be supported
- Integrate a physics engine (PhysX / Bullet)
- Sound API
- In-house Math library
- Inspector with visible properties
- Play button to test the level ingame
- Full-screen mode

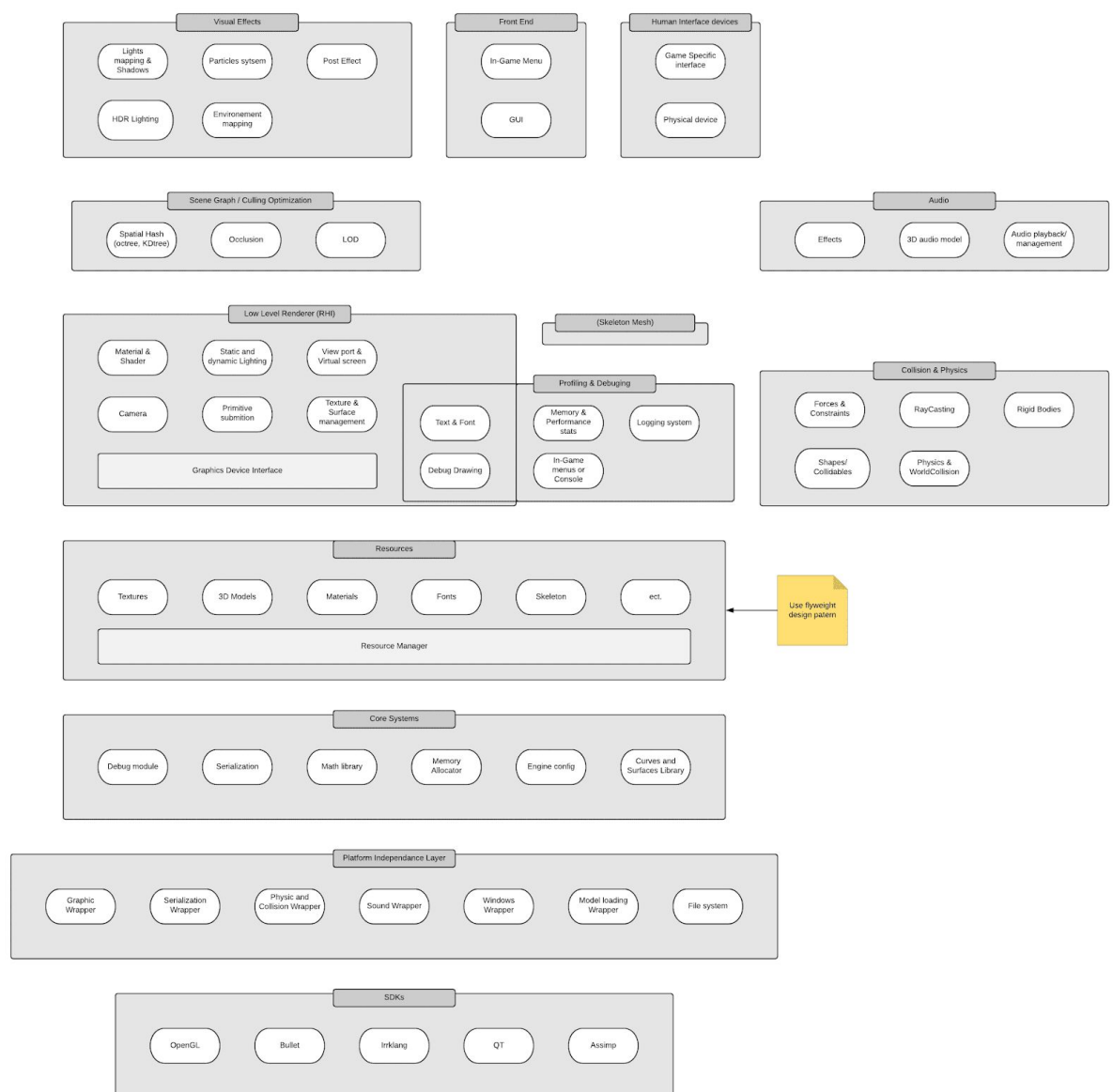
4.5 Technology Enablers:

- Visual Studio 2017 15.7.5
- Qt 5.12.0
- LucidChart
- HacknPlan
- Git
- GitKraken
- Visual Studio Code 1.25.1

5. System Architecture:

5.1 System Architecture Diagram:

System Architecture Diagram is an overview of the runtime game engine architecture split into various modules.



System architecture diagram

5.2 System Components:

5.2.1 Third party Software Development Kits (SDKs)

CloudEngine pursues the same path most game engines do, it means using 3rd party Software Development Kits. The engine uses a combination of well-known SDKs such as:

- OpenGL 4.5: As a graphic SDK. Our experience with OpenGL leads us to use it as a 3D graphic SDK. Its easiness, portability and large documentation make it a first-rate choice in the engine.
- Bullet 2.85: As a Physics API, our choice is based its large number of features. The effectiveness of this SDK is proved by its use in different well-known games.

The bullet physics library is free, cross-platform, flexible and doesn't suffer much from idiosyncrasies behavior.

- Irrklang 1.6: High level 2D & 3D cross-platform sound engine and audio library supporting all file formats.
- QT 5.12: Graphical application development framework.
- Assimp 4.1.0: Open asset import library. It provides a full asset conversion pipeline to use in our engine.

5.2.2 Platform Independence Layer

The usage of SDKs required us to wrap their functions in custom function created by us. Providing us a consistent API and permitting us to not be blocked by a possible change of library. It's composed by:

- Graphic, Serialization, Physic & Collision Wrapper:
Basically, it's wrapping the SDKs to a customized set of functions.
- File System: Its principal areas of functionality are manipulating file names and paths, opening/closing/reading and writing individual files, scanning content and handling asynchronous file I/O requests.

5.2.3 Core Systems

Such a complex C++ software Application as a game engine requires a decent number of software utilities. It's possible to list certain of those software utilities.

- Debug module: Composed essentially by the Assertions who are line of error-checking code. Its purpose is to catch mistakes and violations of our original assumptions.
- Math Library: Since a game is by nature highly mathematics-intensive. A Math library offers facilities for vector, matrix, quaternion, ... manipulation.
- Memory Allocator: Ensure high-speed allocation and deallocation. Also, it's limiting the negative effects of memory fragmentation.
- Serialization: Writing data and objects on a support (File, buffer ...), so that they can be reconstructed later in the memory of the same or another computing host. The reconstruction process is also known as deserialization.

5.2.4 Resources

A game is composed by a multitude of different kind of media. It means an engine must ensure to not waste memory on having multiple copies of the same media at the same time. Each engine is handling the resources on its way. For CloudEngine, we decided to let a ResourcesManager handles our resources managers, it means a texture is managed by a TextureManager, the TextureManager itself is managed by the ResourcesManager. Resources are commonly identified in this way:

- Textures
- 3D Models
- Materials
- Fonts
- Skeleton
- Audio

The Resources Manager relies on the FileSystem. We wrap the Native File System API in the CloudEngine API. The reason is the NFS might lack of tools oriented for our game engine. To help us reducing the weight of processing those

resources we minimize the memory usage by sharing the data as much as possible with other similar objects, using the FlyWeight design pattern.

5.2.5 Low Level Renderer

The low-level rendering gathers all the raw rendering facilities of the engine. For this, the design is focused on rendering geometric primitives. The component is separate into various subcomponents as:

- Graphics Device Interface: OpenGL needs a fairly amount of code to be written to show any result (initializing the devices, setting up the render surfaces [buffers, ...]). All this is managed by the Graphics Device Interface.
- Other Renderer subcomponents: Debug UI, ImGui?

5.2.5 Profiling & Debugging

In such a complex, intricate software the engine is. A set of debugging tools is obviously needed. Firstly, to make the development process easier and secondly to reduce the possibilities that any programmer, using the engine or making it, to be confronted to an error. An example of debugging purpose would be the Debug Drawing: conceptualize ray casts for any object might be a bit difficult since they rely on a continuation of complex mathematics calculations. The utilization of Debug Drawing is a good way to represent those calculations on the screen and how they operate. Obviously, the debugging doesn't stop there, it contains:

- Logging system: printing out the information for debugging is a good and easy way to have any information regarding the code the developer wrote. Calling `printf()` is surely a good idea but we can push a bit farther.
- Memory & Performance Stats: it permits us to annotate blocks of code that should be timed and give them a human-readable names. It means, the Profiler measures the execution time wherever it's called and stores the results. This way, it's giving us, programmers, a human-understandable feedback.

5.2.7 Physics & Collisions

In the real world, objects avoid doing impossible things like intangible things and so on. In game, it's mostly the same but the objects need to be told what to do. For this, a Physics & Collision system is needed to simulate all the possible interactions between objects. This system will consist of a wrap of the bullet physics library into classes such as Rigidbody, box collider ...

5.2.8 Audio

Using an external API (Irrklang) to generate the sounds requires us to wrap it. Wrapping it makes possible to uniformize the functions and in case of changing the API, it won't impact the code.

5.2.9 Scene graph & Culling Optimization

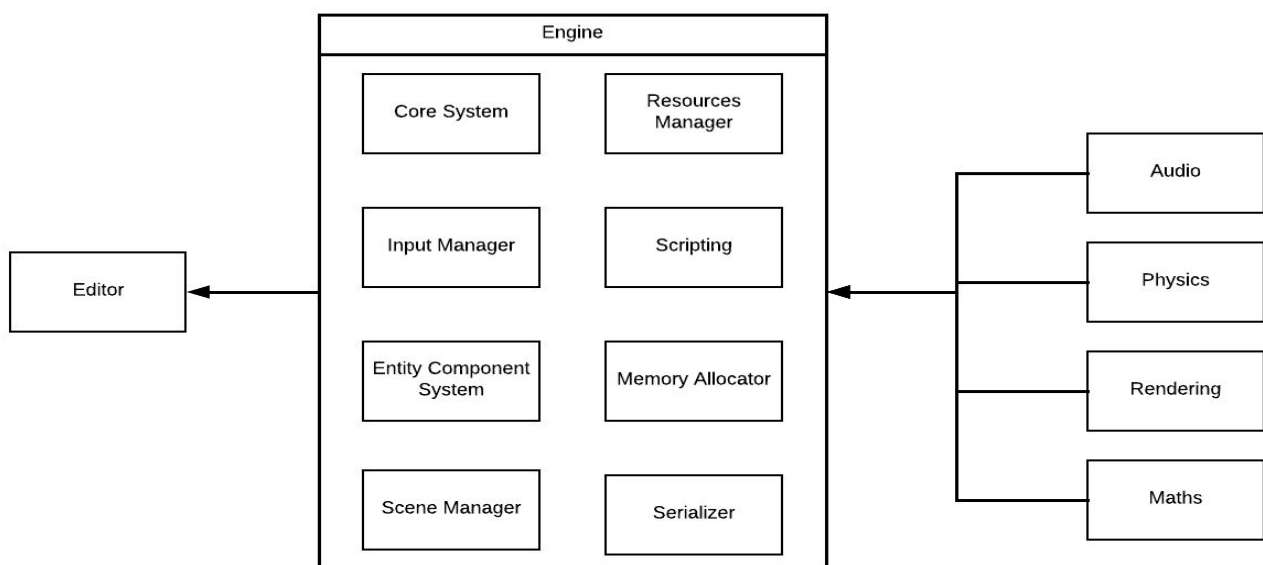
Rendering a scene and all the objects present in the scene has a cost and a heavy one. Some ways exist to reduce the rendering weight applies on the performances. The culling and the scene graph are two possible examples.

- Culling Optimization: Based on some form of visibility to limit the number of primitives submitted for rendering, the Frustum Culling removes object that the camera can't "see".
- Scene Graph: A data structure that manages all the geometry in the scene and allows us to quickly discard large swaths. Such a data structure is often called a scene graph; However, a game's scene graph needn't be a graph, and in fact the data structure of choice is usually some kind of tree.

6. UML

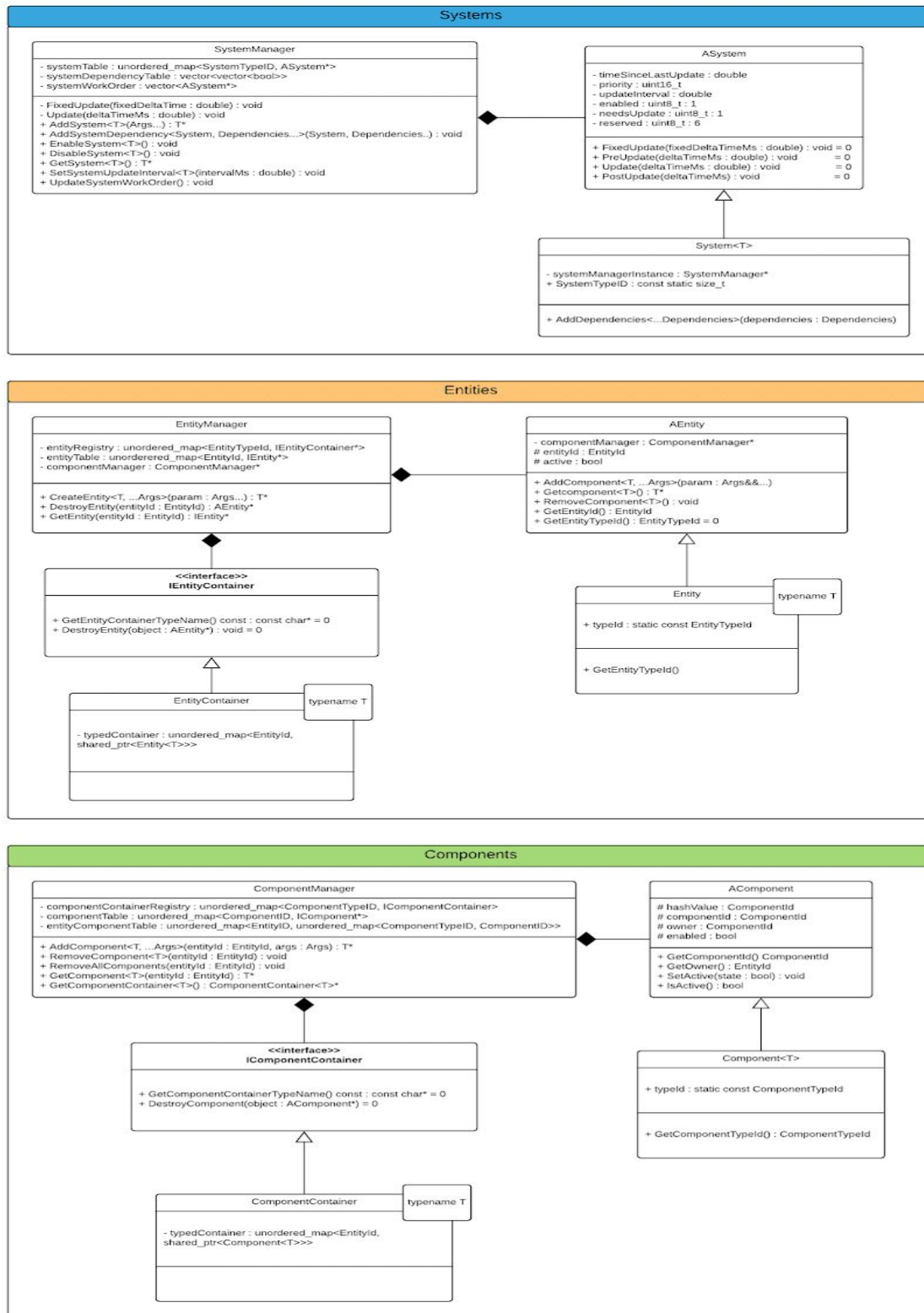
6.1 General Diagram

The diagram below serves as a general overview of the different *systems* that compose the Cloud Engine. Instead of having one huge diagram, for readability reasons, each sub part has been split into its own UML. Certain parts that require further research and investigation are a black box for the team at the moment; thanks to the iterative approach of *Scrum*, those black boxes will be detailed further down the line.



Undetailed Component diagram

6.2 Entity Component System UML Diagram



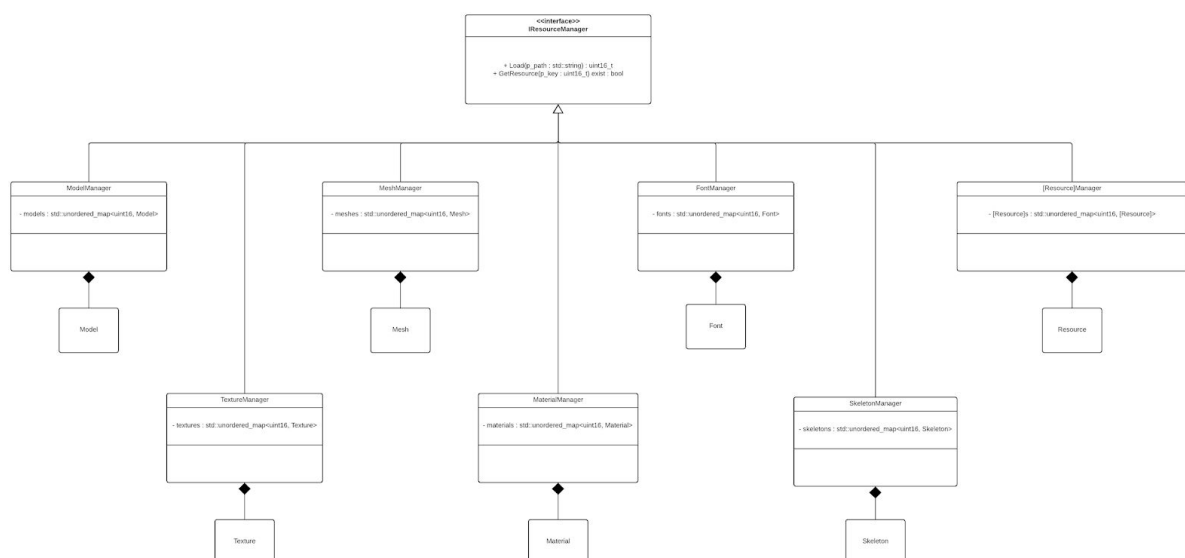
ECS diagram

Entity Component System (ECS) is a design pattern mostly used in game development which follow the composition over inheritance principle. That allow a greater flexibility of the code. In an ECS, every objects are an entity, they consists of an id. By adding components, the functionalities of an entity can be changed. Each systems runs continuously and performs actions on entities that possesses a component of the same aspect as that system.

The use of managers for the instantiation, the life and the destruction of each entity, component and system is easier and a faster implementation.

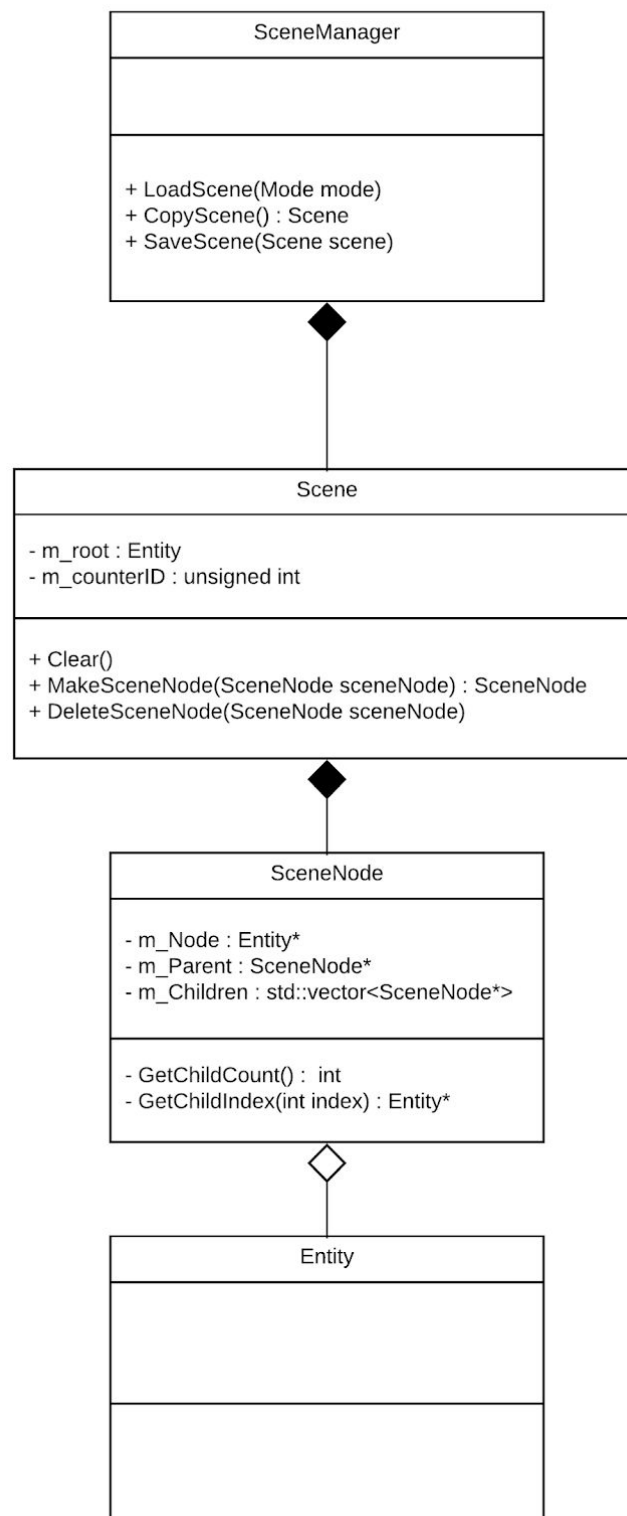
6.3 Resource Manager UML Diagram

The Resources Manager is the place where all the used resources are centralized and managed. This manager purpose is to handle each sub manager (i.e.”Mesh Manager”, “Sound Manager”, ...). One of the particularity of the Resources Manager is the implementation and utilization of the design pattern Flyweight. It means all the resources *entities* use is not duplicate but rather shared with the entities using the same resource.



Resource manager uml

6.4 Scene Manager UML Diagram

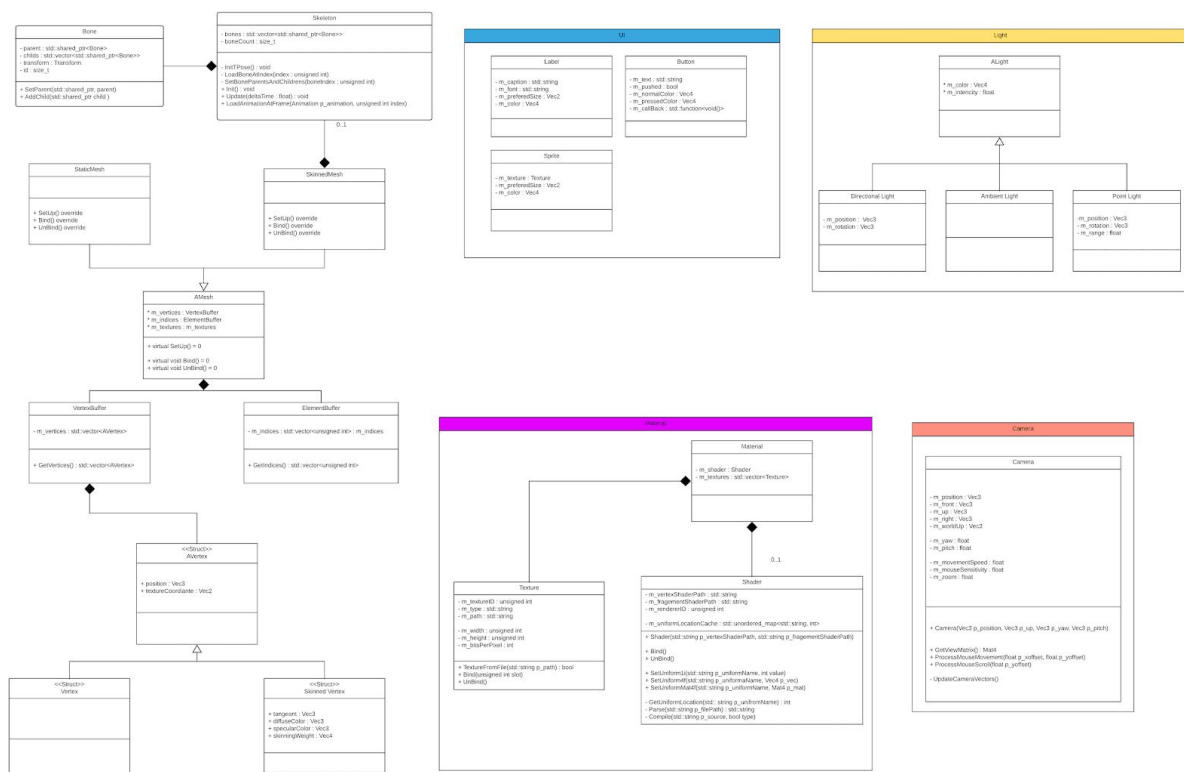


Scene Manager uml

A scene node is a wrapping of our entities from the ECS. This wrapping provides relationship of parent/children between entities, plus various useful methods such as accessing children of entity. The scene itself is just a tree-like structure of all entities with their parenting links. The scene manager is the *entity* that manages scenes, it also has various methods for loading scenes (additive/single). The scene manager also has a copy scene functionality that Cloud engine will use in order to simulate the game: When the user presses the play button to test the game, Cloud engine makes a copy of the scene and simulates the game. Once the simulation is stopped, the copied scene is removed.

6.5 Rendering UML Diagram

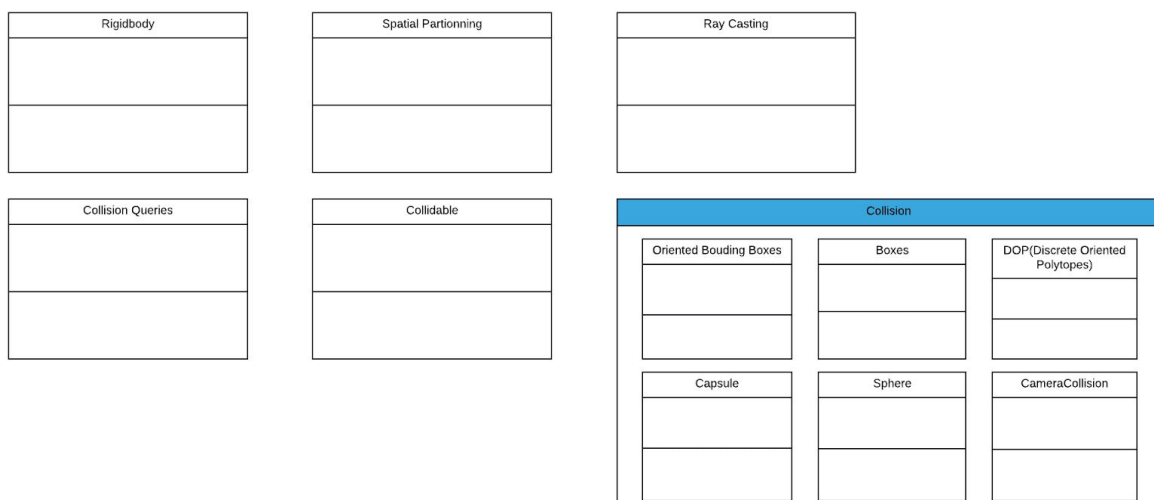
Thanks to the experience we have from our first year of school, the rendering section of Cloud Engine was the most straight forward for the team to design.



Rendering uml

6.6 Physics UML Diagram

Since we're using the SDK "Bullet", the physics engine is mostly a wrapping of bullet's function. Of course, a lot a functionality inherited, derived, from the application of physics function is also implemented. For now, the uml is empty for a simple reason, we didn't planify how we will wrap and use Bullet yet.



Physics uml

6.7 AltMath

AltMath is the math library that all students (9 students) of my class developed in collaboration. Documentation of AltMath can be found in a separate document (AltMath Documentation)



AltMath uml

6.8 Editor

For now the editor remains a black box. The editor part comes later in the development cycle. Nonetheless, the team has engaged in preliminary research; for example: a primary research about undo/redo revealed interesting ideas.

Undo/Redo functionality is a common feature in most contemporain softwares. Implementation and efficiency varies greatly. The Undo/Redo mechanism needs to record every actions and informations to be able to roll back the system to its previous

state (undo) or roll forward to state that have just been undone (redo). To make this mechanism work, a couple of ways exists such as:

- Create a stack of objects & add a “undo” operation to each objects (command pattern).
- Store every state of the systems (memento pattern).

Those two patterns are the most known ways to implement a Undo/Redo mechanism.

The command pattern looks like a good idea but implementing a “undo” function in every object of the engine might be bothersome and redundant. That’s why the memento pattern looks more interesting thanks the “easiness” of creating a stack of states. Knowing the baseline and remember every single change that was made so when something is undone, we just have to throw away the “Top-Item” and regenerate the current view from the baseline plus all the changes. Clicking the redo then just puts that item anywhere in the stack without messing up other undo/redo options. One of the downside of the memento pattern is the consumption of memory it takes. That’s why, Snapshot is only taken every few actions.

7 System Architecture:

7.1 Rendering:

Part	Description	What Problems I encountered	How I did overcome them
Core Window	GLFW window for the final game	X	X
Input System	Wrapping Input Similar to Unity where you can Do Input::GetKey(Key::KEY_K)	GLFW and Qt dont use same codes for keys	For now the editor remains RnD. The editor could launch events to trigger The existing input system in the core
Rendering	Wrapping OpenGL classes	Using more advanced techniques than last year: (UniformBufferObject)	LearnOpengl.com and the help from my class mates
Shader class	A wrapping for Opengl shader	Smart shader that can detect uniforms and samplers	Reading the documentation for glGetActiveAttrib
Lights	Using light data to all shaders	Sending multiple light sources	For now the engine supports only one light source. In the next iteration it will support up to 4 of the closest lights
<p>The ECS was finished after the rendering was done. Few Adjustments had to be taken.</p> <p>Working with ECS was gymnastic for my brain in the beginning, thinking about component data and systems that manage them behind.</p>			
UniformRendering System	A system responsible for updating uniform rendering data so that all shaders connected to it receive updated	Wrapping my head around ECS and how it functions	Talking with Charly who implemented the ECS

	data		
--	------	--	--

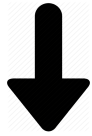
7.2 Core:

Part	Description	What Problems I encountered	How I did overcome them
------	-------------	-----------------------------	-------------------------

Resources Manager	Load and keep every resources loaded	How to avoid code duplication	The resource manager is a template and the resources have to define a load function
-------------------	--------------------------------------	-------------------------------	---

I learned how to use typeid to detect a type name and use it to automatically create a path to the resources

MemoryAllocation	memory allocation and management	there is a lot of case of allocation and as much allocator to do	The memory allocation implementation was to time consuming so we didn't used it, if we have time later, we can finish the it.
------------------	----------------------------------	--	---

Entity Component System	create and manage entities, components and systems	It was my first ECS implementation so i had to deeply understand how it works; Instantiate in the application entities and components defined in engine	A lot of reading; 
-------------------------	--	--	--

statics are instantiate for each application/DLL which means that you have differents value depending on where you are. To overcome this i had to create an instance and don't have it static. Also, static member of template type couldn't be deducted when instantiated in the application but declared in the DLL, so we had to specify for each types how to calculate the static member.

7.3 Tools:

Part	Description	What Problems I encountered	How I did overcome them
CloudMath	Mathematic library	quaternion correction and optimization, switch matrices from row major to column major	A lot of drawing and brain gymnastic
File Control	File control is necessary to handle the use of the library <Filesystem> allowing us to manage the creation of files and directories	Depending the version of Visual Studio, the library <Filesystem> is under the label <experimental> or not. Due the lack of use of this library, it was hard to notify the problem at first.	I used a #Define following the version of Visual Studio the user is on.
File Control & Qt		It looks like Qt uses a personalized version of <Filesystem> making the use of it impossible with the #define i used in the previous problem i encountered.	I decided to remove the #define for now and use only the library under the label <experimental> (the one Qt is using).
Profiler	The profiler is used to compare, benchmark, calculate times or iteration between two points in the code.	I had the idea of how to do it but I was unable to pull it out.	i decided to remove it from the project for now but i surely come back at it later.
Qt	The Editor.	First, I had an issue with the dependencies linked & the linkers in Visual Studio. It looks like when we changed manually the linkers in visual studio, Qt was not recognized anymore.	I let Qt & Visual Studio managed it by themselves and didn't changed it manually.

Qt		Manipulate the widgets in Qt. I couldn't find out a decent way to use the widget allowing us to move/resize it.	I created a Container of widget allowing us to set any widget in it and move/resize it at will.
----	--	--	---

References:

- [Geospatial System Requirements Specification template](#)
- [Game Engine Architecture 3rd Edition](#)
- <https://wprock.fr/blog/conventions-nommage-programmation/>
- https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/NamingBasics.html#//apple_ref/doc/uid/20001281-1001751-BBCFE_CGB
- <https://google.github.io/styleguide/cppguide.html#Naming>
- https://www.reddit.com/r/cpp_questions/comments/6vhfet/modern_c_naming_conventions/
- <https://datasift.github.io/gitflow/IntroducingGitFlow.html>
- <https://fr.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>
- <https://www.opengl.org/>
- [https://en.wikipedia.org/wiki/Bullet_\(software\)](https://en.wikipedia.org/wiki/Bullet_(software))
- <https://forum.unity.com/threads/why-the-heck-is-almost-every-game-engine-trying-to-switch-over-to-bullet-physics.72731/>
- <https://www.ambiera.com/irrklang/>
- <https://www.qt.io/>
- <http://www.assimp.org/>
- https://en.wikipedia.org/wiki/Flyweight_pattern
- <http://www.mountingoatsoftware.com/agile/scrum>
- <https://en.wikipedia.org/wiki/Entity%E2%80%93component%E2%80%93system>
- <https://github.com/tobias-stein/EntityComponentSystem>
- <https://stackoverflow.com/questions/2746076/how-do-i-create-undo-in-c>
- <http://www.drdoobs.com/an-efficient-undoredo-algorithm/184404861>
- <https://stackoverflow.com/questions/42160538/undo-redo-memento-pattern-c-sharp>
- <http://gernotklingler.com/blog/implementing-undoredo-with-the-command-pattern>
- <https://learnopengl.com>
- https://en.wikipedia.org/wiki/Application_programming_interface