Identity Modular Engine

(I.D.E)

By team

Looking For Group

Filipe Antunes Da Silva, Lilian Gadeau and Gabriel Meloche

Overview

This Technical Design Document (TDD) is written for the expressed purpose of defining the basis of the work that have been undertaken to create a video game engine for Windows.

Our goal was to create an engine that is:

- User-friendly
- Optimised (uses as little resources as possible)
- Focused on FPS game-making

Our engine handles:

- Rendering, including shaders
- Audio with basic 2D and 3D functions
- Compatibility with Windows
- Object loading within a scene
- Physics and collisions between objects

Systems

We have coded our engine using mostly the KISS principle (Keep It Simple Stupid), meaning that the architecture of the code will be as clear as possible to our programmers as well as to outside observers. This extends to the way the user will interact with the engine as well; we want to make sure that the interface is intuitive, uncluttered and simple yet elegant.

We have an Entity Component System (ECS). Game objects will have components attached to them, such as meshes or transforms, and those components only contain pointers and/or references to their data in the memory; no raw data is held directly by the components.

Our architecture is modular, meaning that modules can be added to or removed from the project without breaking the program. For example, we could run the project while deactivating the audio module without any consequences on the other modules. This ensures that programmers working on different modules would not create errors for the other workers, thus saving a significant amount of time and headaches on debugging. Theoretically, it may also permit us to enable the user to change external dependencies to their liking, e.g. using a different physics library than the one included in our engine. However, this is not a priority, nor do we expect to implement this idea.

To make this possible, we have wrapped most of the external dependencies' functions, methods and variables in classes of our own so that the engine doesn't directly use the API. This makes it possible to swap some dependencies without compromising the rest of the engine's code.

Design

Naming conventions

General	Classes
Functions : PascalCase() *ex: DoSomething(...);*	**Namespaces** : PascalCase *ex: Core::Rendering*
Local Variables : camelCase *ex: newPosition*	**Basic Classes :** PascalCase *ex: InputManager*
Parameter variables : p_camelCase *ex: p_newPosition*	**Abstract Classes** : Capital 'A' before the name *ex: AClassName*
Static Variables : static m_ camelCase *ex: static m_numberOfInstances*	**Interfaces :** Capital 'I' before the name *ex: IClassName*
Pointers : type* m_camelCase *ex: char* m_playerName* NOTE : Prefer use of smart Pointers;	**Struct** : PascalCase *ex: Vector3*
Typedef : alllower *ex : typedef*	**Methods** : PascalCase *ex: DoSomething(...)*
Macros : ALLCAPS *ex: RENDERINGMODE*	**Properties**: PascalCase *ex: Property {get(){...}; set(){...};}* ***(might change)***
Templates : template<typename PascalCase> *ex*: *template<typename MyType>*	**Fields (Members)** : m_ camelCase *ex: m_memVar*
Enum : PascalCase { ALLCAPS, ALLCAPS... }; *ex: Color { RED, GREEN, BLUE };*	**Events :** PascalCase *ex: Start();*

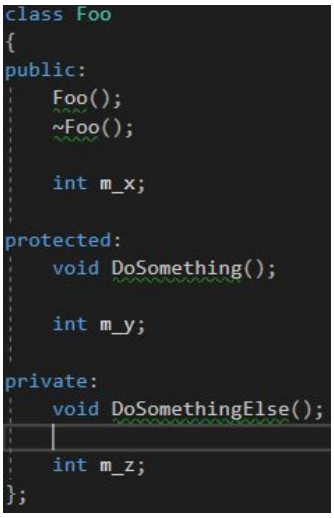
Norms

Language specification: c++17

Class and struct layout: Classes and structs are declared in this order:

- public : methods, then variables
- protected: methods, then variables
- private: methods, then variables

For example:



For primitive variable types, we always try to use a fixed width type that takes the least memory space possible while still being able to represent maximum and minimum values that we will need to use. For example, should an int variable's max used value be less than 127, we would declare it as an int8_t rather than an int.

Organization

File Organization

All C++ classes are a part of a general namespace relative to their common context (behaviour).

We had planned to install the engine on the user's machine, however, time limits have made us prefer the simpler approach of running it through an executable.

When adding a new external library you need to add it to its specific folder in the Dependencies folder.

Saved resources are located in Engine/Resources, and saved scenes and resource manager files are located in Editor/SaveFiles.

Code organisation

Our Visual Studio solution contains 2 projects:

1. Engine

- This project will be exported into a DLL to be used by the Editor project.
- There will be 2 main folders that hold the .h and .cpp files: include and src. These two folders will contain as many subfolders as there are namespaces used in our code and the subfolders will be hierarchised in the same fashion. For example, a class that is part of the namespace Rendering::Tools will have its .cpp file in ProjectDirectory->src->Rendering->Tools and its .h file in ProjectDirectory->include->Rendering->Tools.

2. Editor

- We use this project's main() to test our functionalities and to display the editor through ImGui, so the engine's DLL will need to be included into the project settings.
- Same file hierarchy as for the engine.

All external libraries need to be wrapped by classes that we will create so the user will only use the engine's functions directly instead of the libraries'. This also ensures that maintenance or changes of the dependencies will not need a refactoring of the whole code for the user program.

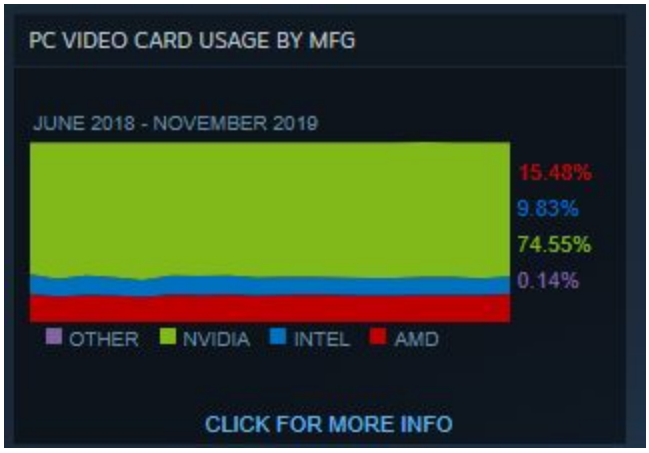
Tab indentations will consist of 4 spaces, not a full tab.

Includes will be declared using chevrons (<>), not quotes ("").

Dependencies

Rendering: DirectX

We chose DirectX because we have never worked with it, so it would be a nice experience. It's one of the most used APIs so it would definitely be a nice addition to our knowledge. It's made by Microsoft and it is much more adapted to Windows than OpenGL, and it usually gets all the updates and fixes first, then they get implemented on OpenGL.
We are building an Engine for Windows. The biggest manufacturer for video cards is currently Nvidia, and Nvidia cards work better with DirectX than AMD ones. DirectX is made for Windows. Therefore, by using DirectX, our engine would be adapted to the majority of our users.



One other reason to use DirectX is that it comes with a lot more developer options (such as debugging) by default. This comes due to the fact that more developers use DirectX, so there is a bigger community, which in turn means there is more documentation and more support for it.

Physics: Bullet

Given the smaller scale of our project, using Bullet would be wiser than using PhysX since Bullet generally requires less debugging. We would expect the implementation of PhysX to bog down production for a while since this has happened to a more experienced team of people we know working on a similar project before. Bullet's ease-of-use is a welcome attribute that facilitates production. However, given that Bullet's proper documentation has disappeared from the Internet since the middle of the project, it has proven to be difficult to debug for now.

Math Library: GPM

We use our homemade math library because we made it ourselves so we know exactly how it works and we can modify it whenever we want. We will use it as a static library since its code will never be modified at runtime and because accessing its functionalities will be faster and more efficient.

3D Loader: Assimp

We want to work with Assimp since it is efficient resource-wise and our team is already vaguely familiar with it. Our engine needs to be able to handle both .obj and .fbx files, which Assimp supports (among many other types). Furthermore, Assimp can "repair" or readapt broken models (ie triangulate quads, create normals if they're not present, flip the UVs...). Textures are also fairly easy to swap during run-time. It would take a long time for our team to develop such capabilities, thus using Assimp will allow us to direct fewer resources to mesh parsing (mostly debugging) and more resources to building other critical parts of the engine.

Scripting Language: [To be determined]

Audio: Irrklang

We use Irrklang for our engine's audio because it is very easy to implement and has

many low-level and high-level features. It uses its own decoders and is independent from the OS, saving us much time and effort while removing most of the potential hardware compatibility issues. Its documentation is very detailed and straight-forward. We would have liked to add it as a static library, but unfortunately that is only available for the pro version. Therefore, we will use it as a dynamic library.

Graphic Interface: ImGui

ImGui is much more efficient and easier to implement than our other alternative, Qt. ImGui is an API that is more suited to making widgets and interfaces, but more importantly, it does not run on its own update loop as opposed to Qt, to which we would have to adapt. We prefer an API that will adapt to our engine, and not the other way around.

Saving: Homemade Serialization

By the client's demand, we will save our user's projects and configurations using serialization that we will code ourselves.

Technical Risks

- Our engine will be event-driven and unfortunately, although the system works with functions that have no arguments, it still has issues with functions that take parameters, limiting the usefulness of the system.

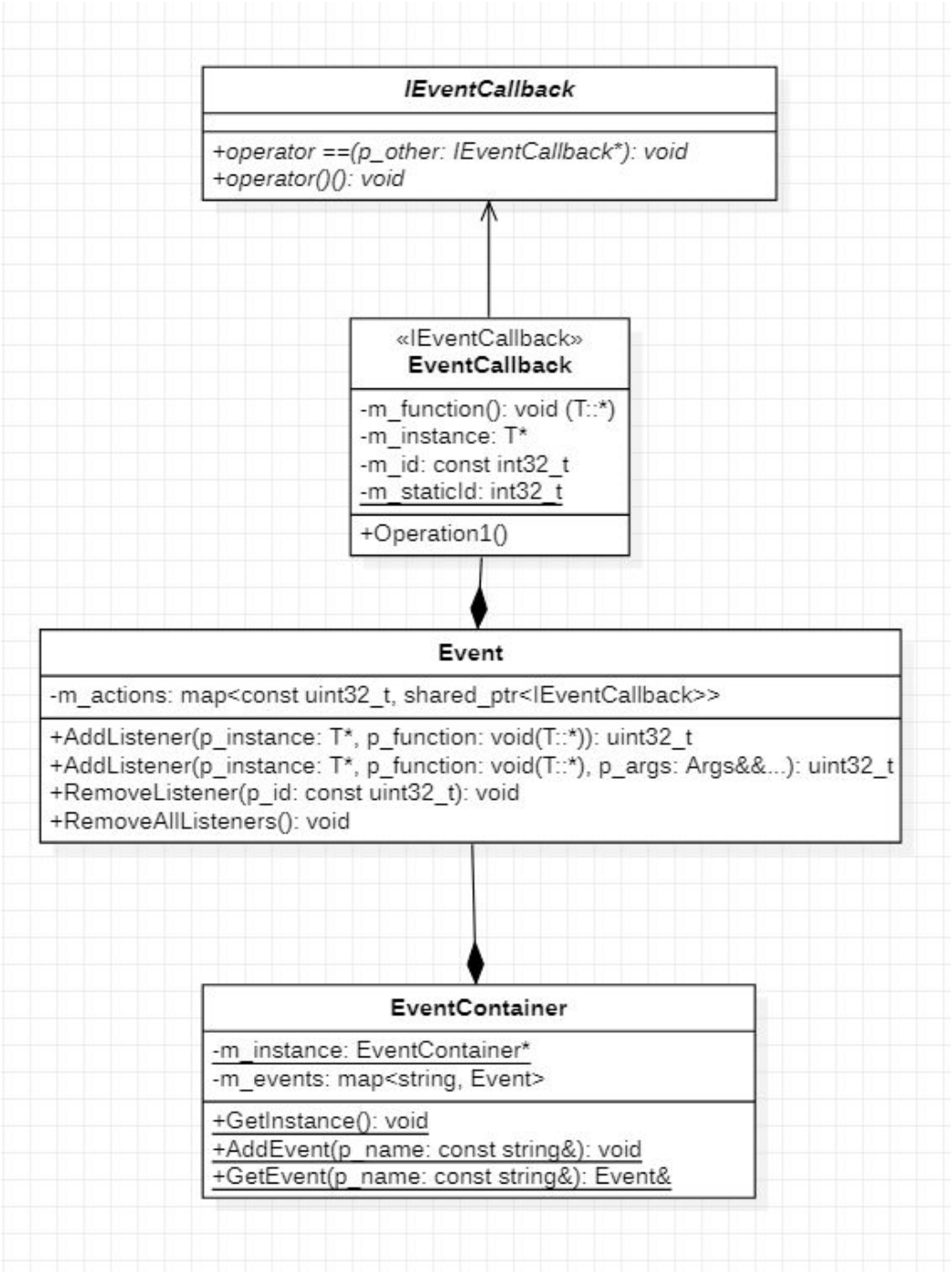
- Given the tight deadlines we had, most of the more recent code that has been written is a bit sloppy, unintuitive and bug-prone, and optimisation has been left aside in favor of implementing primary systems to make sure we delivered a functional engine on time. This leaves us with a lot of technical debt.

- DirectX's left-handed coordinate system caused us many issues relating to object rendering. It's very hard to debug, since we do not know what DirectX expects. Furthermore, our math library's matrices are in column-major while DirectX demands row-major matrices.

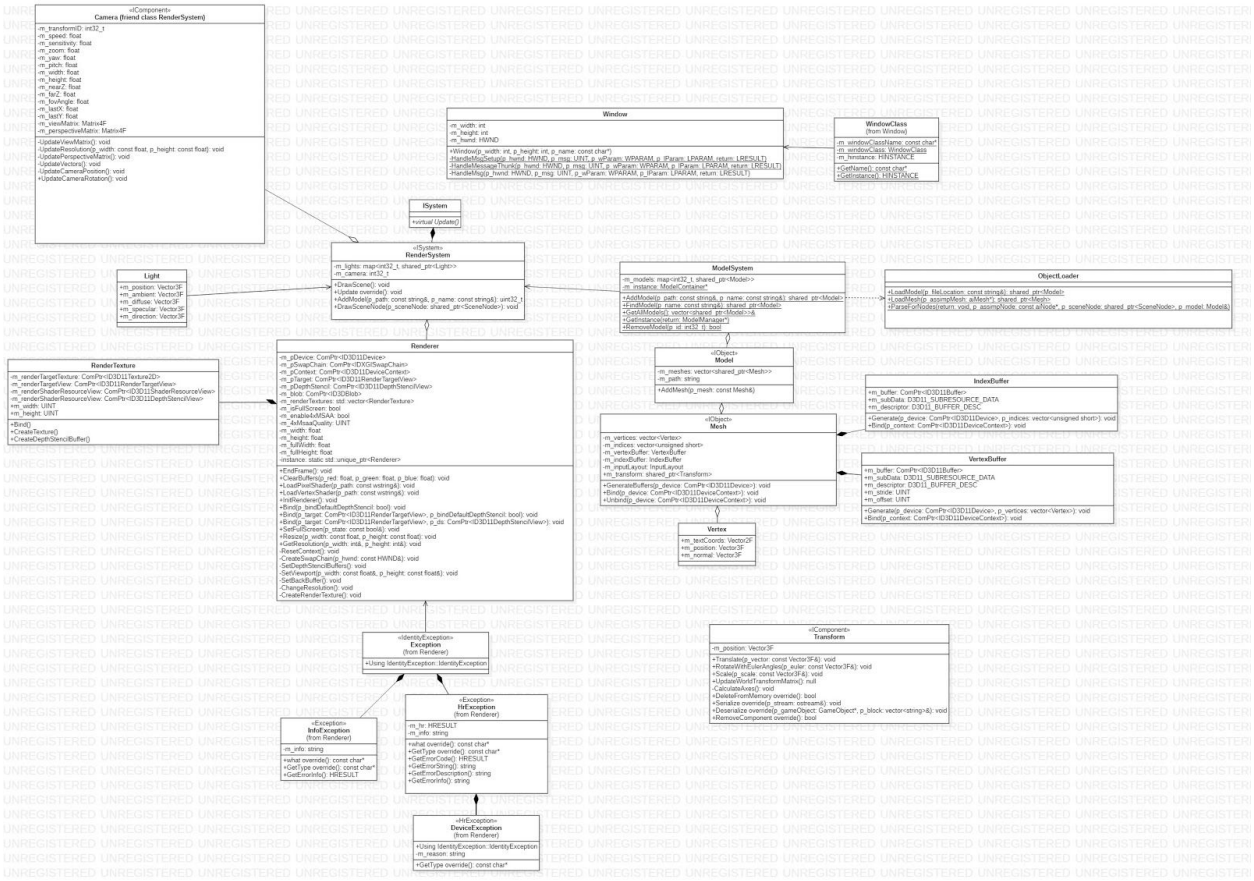
- The ECS architecture we want to implement has been hard to fully conceptualize; since we have only started learning about it, it took many versions and rollbacks before finally settling on a model that makes sense.

- Unfortunately we have not had the time to implement scripting in order to create a video game from our engine.

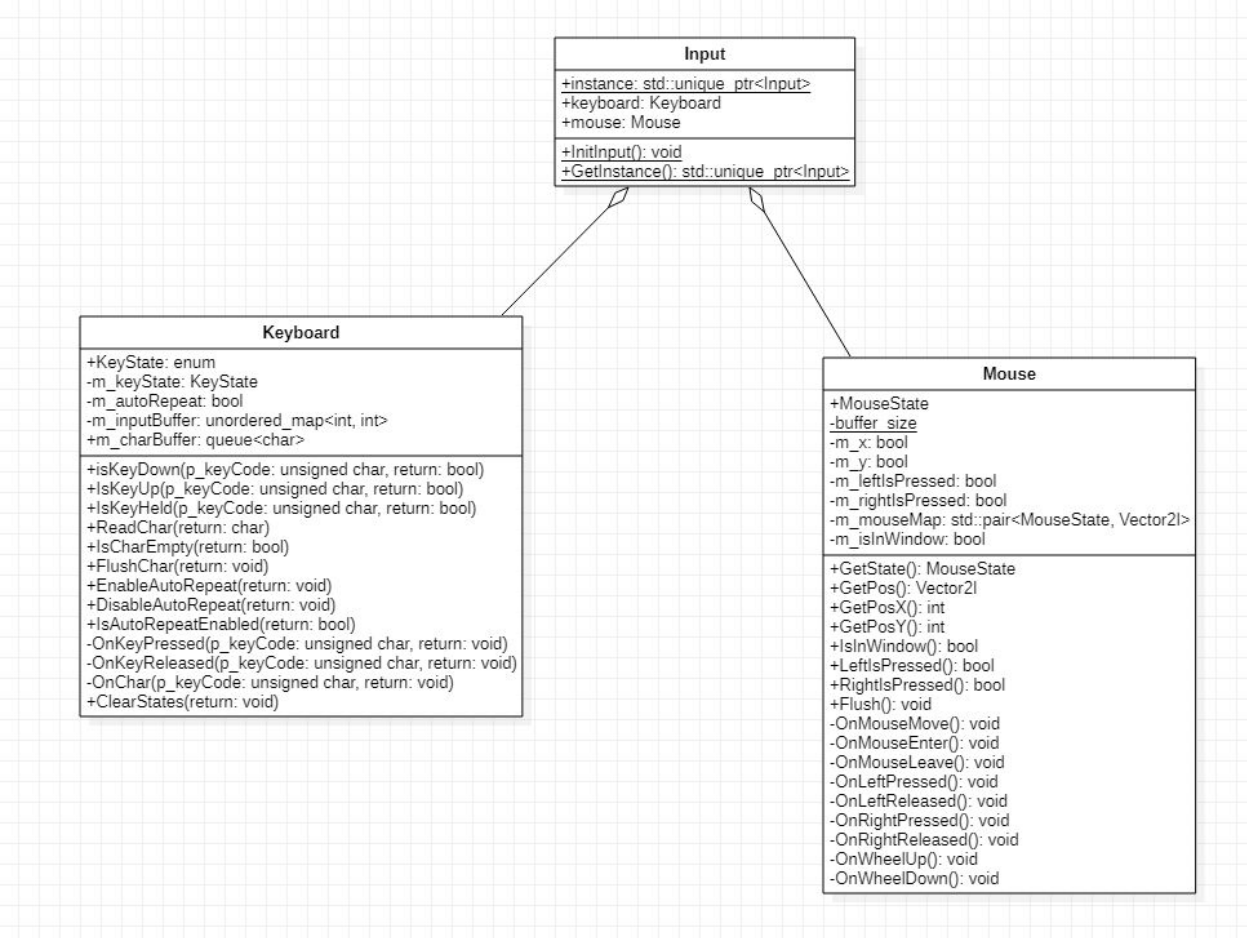
Event Diagram



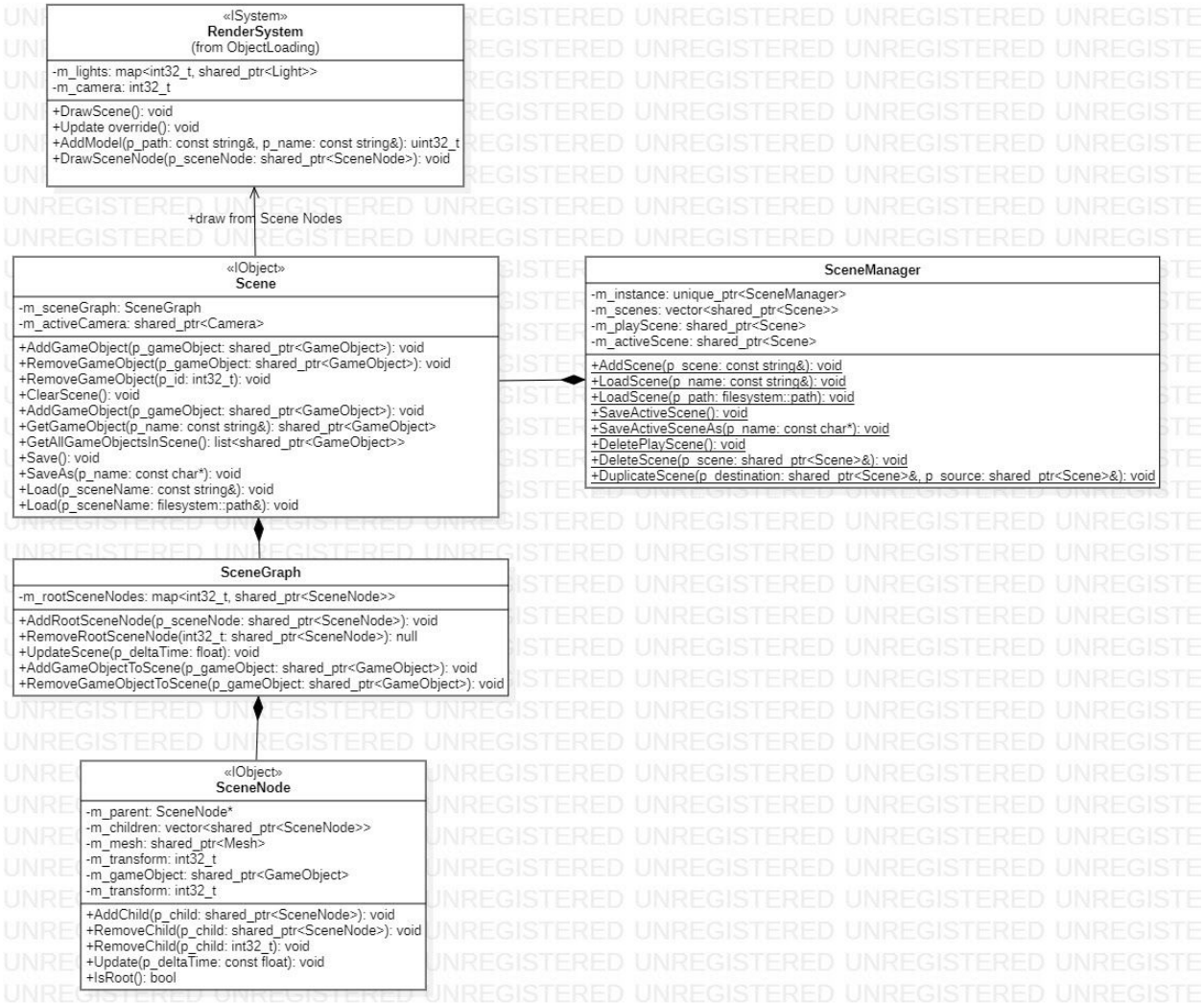
Rendering Diagram (better quality version available in git repository -> documentation)



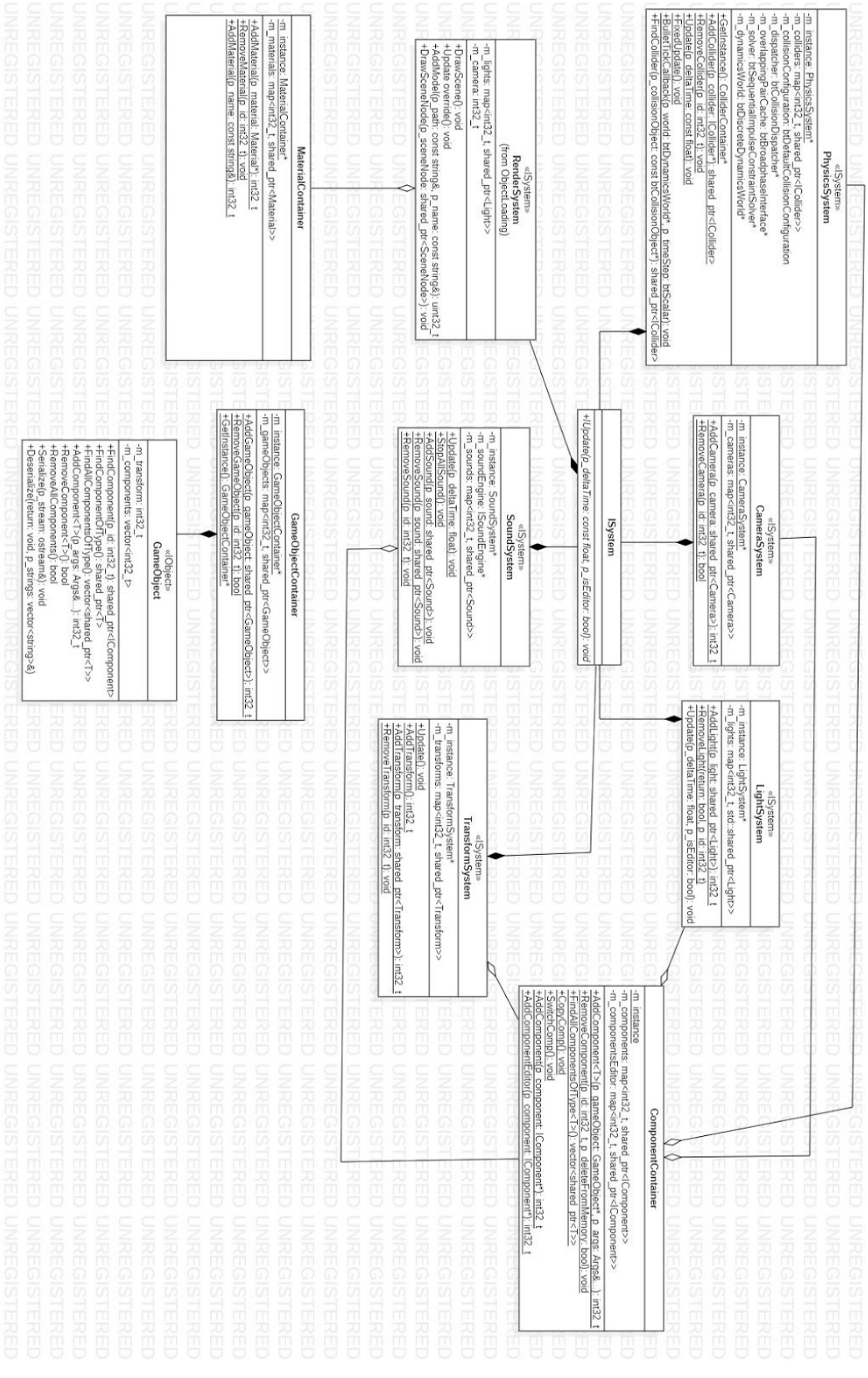
Input (better quality version available in git repository -> documentation)



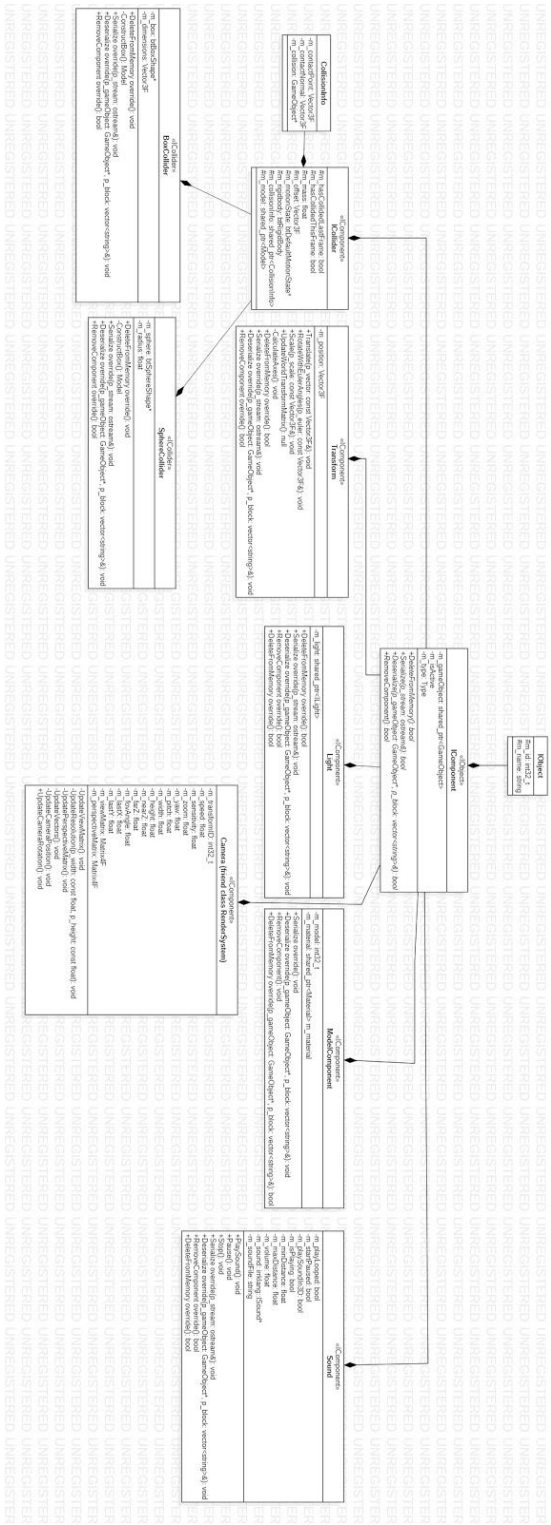
Scene Graph (better quality version available in git repository -> documentation)



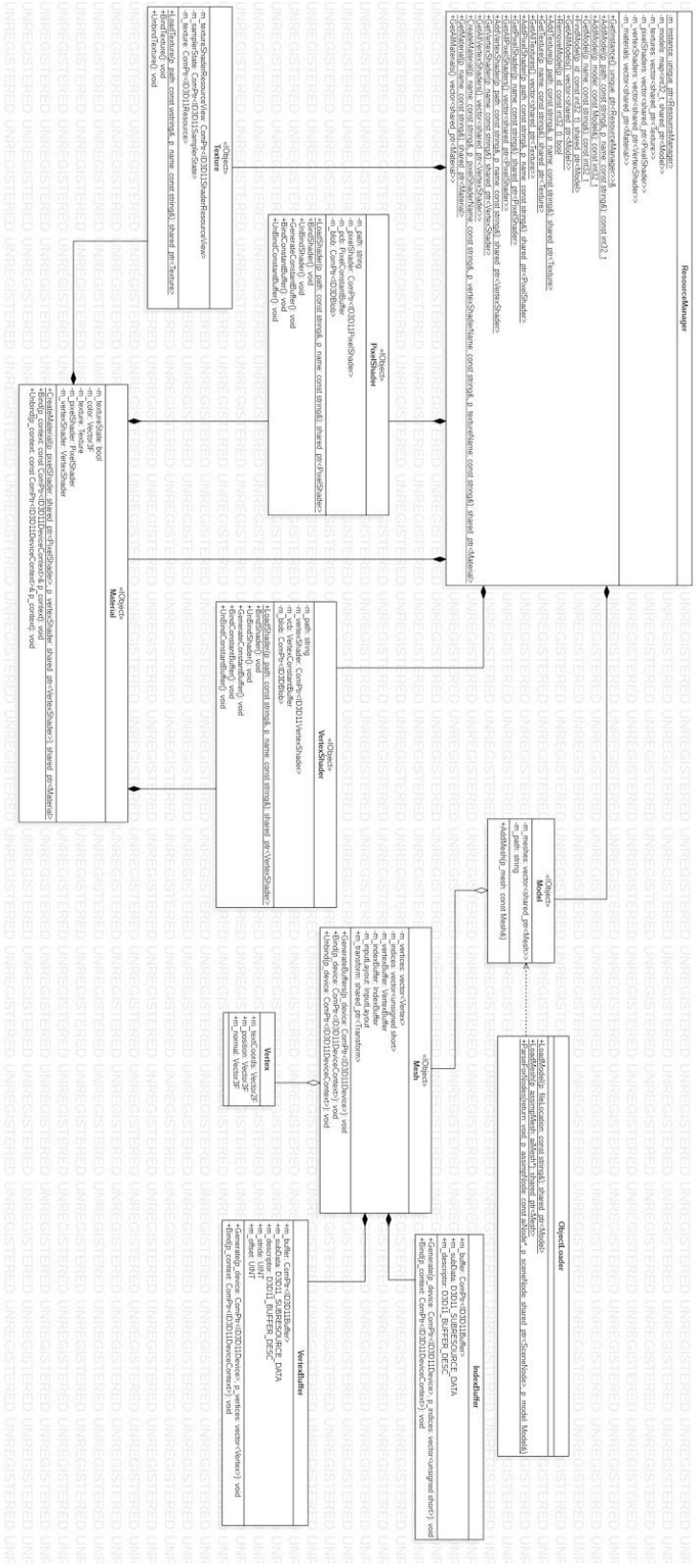
Systems (better quality version available in git repository -> documentation)



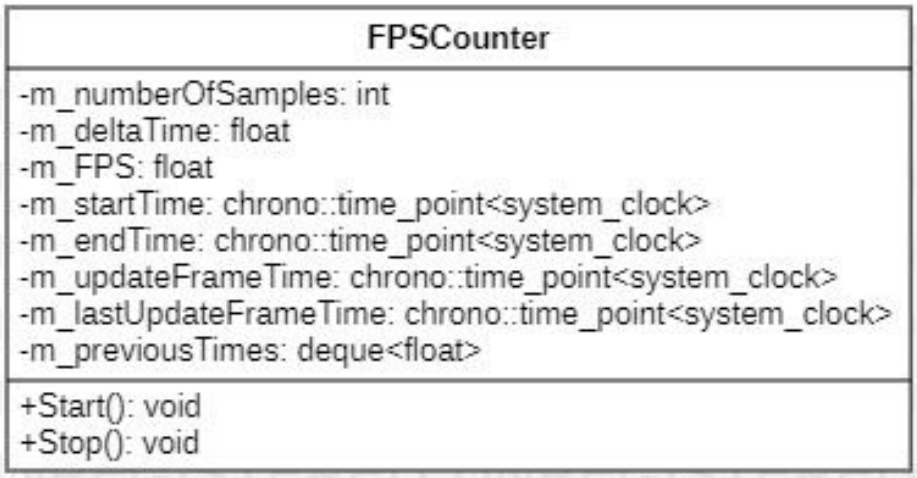
Components (better quality version available in git repository -> documentation)

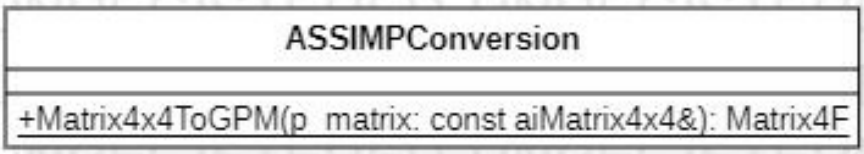


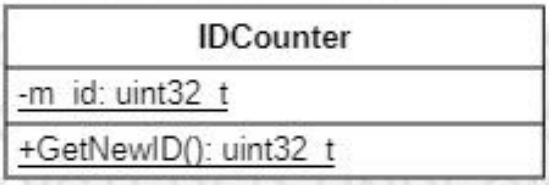
Resource Manager (better quality version available in git repository -> documentation)



Tools


FPSCounter
-m_numberOfSamples: int
-m_deltaTime: float
-m_FPS: float
-m_startTime: chrono::time_point<system_clock>
-m_endTime: chrono::time_point<system_clock>
-m_updateFrameTime: chrono::time_point<system_clock>
-m_lastUpdateFrameTime: chrono::time_point<system_clock>
-m_previousTimes: deque<float>
+Start(): void
+Stop(): void


ASSIMPConversion
+Matrix4x4ToGPM(p_matrix: const aiMatrix4x4&): Matrix4F


IDCounter
-m_id: uint32_t
+GetNewID(): uint32_t

UI (better quality version available in git repository -> documentation)