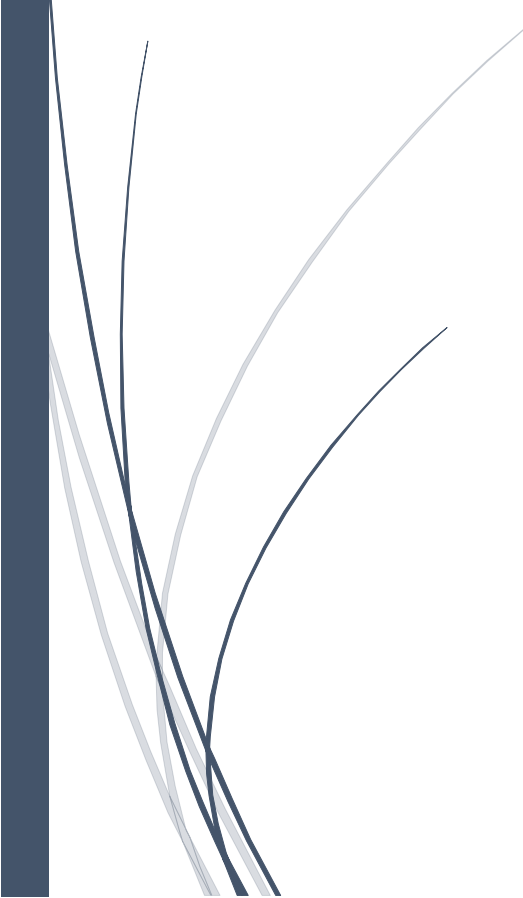


A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

3/15/2020

# Omega

Technical Design Document

Several thin, curved lines in dark blue and light gray originate from the bottom left and curve upwards and to the right.

Victor JORE - Thibaut PONCHON  
ACCELIT



## Table des matières

Table des figures (UML) .....	3
Introduction .....	4
Normes et organisation des fichiers .....	5
Normes.....	5
Typologie.....	5
Règles de nommages des Getters et Setters .....	6
Ponctuation.....	6
Template .....	7
Organisations des fichiers .....	8
Arborescence .....	8
Gitflow.....	9
Découpage des modules.....	10
Choix des API et technologies .....	11
Audio .....	11
Physique.....	11
Rendering.....	11
Chargement de modèles 3D .....	12
Chargement d'images .....	12
Éditeur.....	12
Avantages.....	12
Inconvénients.....	13
Scripting .....	13
Diagramme de classes : UML .....	14
Module audio.....	14
Module Rendering .....	15
Contexte.....	15
ResourceManager .....	16
Module Core .....	17
SceneGraph.....	17
ECS.....	18
Components du Core .....	19
Autres modules.....	20

## Table des figures (UML)

Figure 1: UML du module Audio .....	14
Figure 2: UML du contexte du module Rendering.....	15
Figure 3 Resource Manager .....	16
Figure 4 Core module : SceneGraph .....	17
Figure 5 Core module : ECS.....	18
Figure 6 Core module : Components .....	19

## Introduction

Le projet moteur Omega est un projet de fin d'études réalisé durant cette année 2020 en Game Programming à ISART Digital Montréal. Il est composé de 3 phases de développement séparant la conception du moteur et permettant de faciliter l'organisation du projet.

Plusieurs contraintes nous sont imposées, tant techniquement qu'humainement. On notera premièrement une main d'œuvre très limitée et inférieure aux autres groupes pour un mandat identique. Cela nous oblige à être très consciencieux de la gestion du temps et nous demandent d'être efficace pour rester dans les temps. L'autre contrainte imposée concerne l'utilisation de bibliothèques externes, qui nous sont limitées, nous obligeant donc à utiliser et développer nos propres bibliothèques pour le projet.

Certaines fonctionnalités devront nécessairement être développées par nous-mêmes comme un Serializer de scènes en XML. Par ailleurs nous sommes dans l'obligation d'utiliser notre propre bibliothèque de mathématiques écrite lors d'un ancien projet commun. Le temps de développement étant relativement court, certaines fonctionnalités n'ont pas pu être testées dans tous les scénarios possibles. Elle est donc une source potentielle d'erreur résultant d'une maintenance au cours du développement du projet Omega.

Nous avons choisi comme nom de moteur la dénomination Omega car elle représente la dernière lettre de l'alphabet grecque. Cette caractéristique symbolise pour nous l'une des facultés que nous souhaitons apporter à ce moteur qui se veut moderne et de dernière génération profitant ainsi des dernières technologies matérielles et logiciels dans le secteur du jeux vidéo. La philosophie de ce moteur se veut performant, ergonomique, scripting user-friendly, possibilité de rendu réaliste, simple mais avancé.

# Normes et organisation des fichiers

## Normes

### Typologie

Le projet se devra de respecter la typologie suivante :

- Classes : **XCamelCase** – X devant être présent en cas de classe abstraite ou interface et ainsi remplacé par **A** ou **I** respectivement.
  - Méthodes membres et statiques : **CamelCase**
  - Membres et statiques de classes : **m\_camelCase**
- Fonctions : **CamelCase**
- Paramètres : **p\_camelCase**
- Enum et Union : **ECamelCase** et **UCamelCase** respectivement
  - Membres Enum et Union : **UPPER\_CASE**
- Constantes / define : **UPPER\_CASE**
- Variables locales : **lowerCase**
- Variables globales : **UpperCase**
- Structures : **CamelCase**
  - Méthode membres de statiques : **CamelCase**
  - Membres et statiques de classes : **camelCase**

Chaque ligne devra respecter une délimitation maximale de 125 caractères.

L'indentation devra être respectée et consistante.

Chaque module fera partie du namespace global OgEngine.

### Exemple :

Module :

— Engine : *namespace OgEngine*

Sous modules du module Engine :

```
| Resources
| Buffers
| Etc...
```

Pour les membres de classes pouvant contenir de multiples éléments tel un tableau, on privilégiera l'utilisation du pluriel.

Exemple :

```
std::vector<OgEngine::VertexBuffer> m_vertexBuffers;
```

### Règles de nommages des Getters et Setters

Ce n'est pas parce qu'il y a un Get qu'il y a un nécessairement un Set et vice-versa. On ne mettra à disposition que ce qui peut être utile à l'utilisateur final et qui ne puisse pas compromettre l'intégrité des modules et sous-modules.

Les Getter ne doivent **PAS** avoir le préfixe Get devant le nom s'il ne fait que renvoyer une copie ou une référence constante. Ainsi seuls les Getter renvoyant un pointeur modifiable (non constant) disposeront du préfixe Get. Si l'on veut renvoyer (par un Getter) des données non primitives comme une structure de données tel qu'un Vector3F on essaiera autant que possible d'en renvoyer une référence constante.

En outre, le préfixe *Get* sur les Getters ne doit être appliqué que si l'on admet vouloir modifier l'objet retourné directement tel un pointeur.

### Exemples :

```
int Health(void) const;
const Vector3F& Position(void) const;
Window* GetWindow(void) const;
```

Exemple d'une possible corruption d'intégrité : Le getter *GetWindow* qui retournerait le contexte de la fenêtre via un pointeur non constant, la modification du pointeur serait alors possible et engendrerait ainsi le crash du programme en cas de perte de contexte. À éviter.

Les Setters quant à eux garderont le préfixe Set afin de les dissocier des Getters qui n'auront, eux, très majoritairement aucun préfixe.

### Ponctuation

Les accolades seront disposées sous les noms de leurs méthodes, boucles et branchements.

Si trop de paramètres sont nécessaires dans une méthode / fonction, on présentera chaque paramètre sous le nom de la méthode / fonction afin de garder l'information condensée (tous les paramètres) à un seul endroit visible. Bien que cette façon de faire augmente le nombre de ligne et peut paraître lourde elle améliore la lisibilité. Cela s'applique à la définition, l'implémentation et l'utilisation.

Exemple :

```
int ThisFunctionIsVeryLong(  
    const int p_paramOne,  
    const int p_paramTwo,  
    const int p_paramThree,  
    const int p_paramFour,  
    const int p_paramFive)  
{  
    // Code ...  
}
```

## Template

Les templates seront défini dans le .h et implémenté dans un .inl. On fera attention si cela est nécessaire de faire de la vérification de type et de n'activer le template via *std::enable\_if* que lorsque l'on voudra interdire certains types pouvant corrompre le bon comportement du programme. L'interdiction de type peut se faire via *static\_assert*.

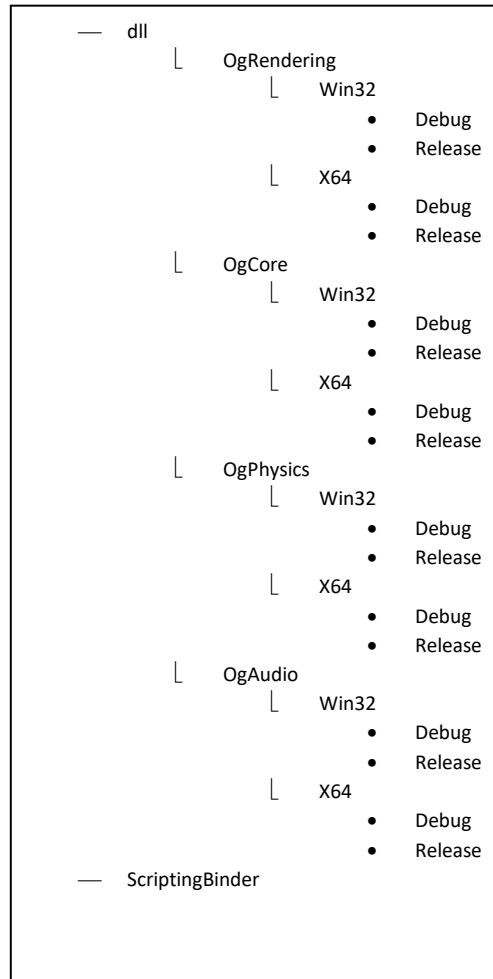
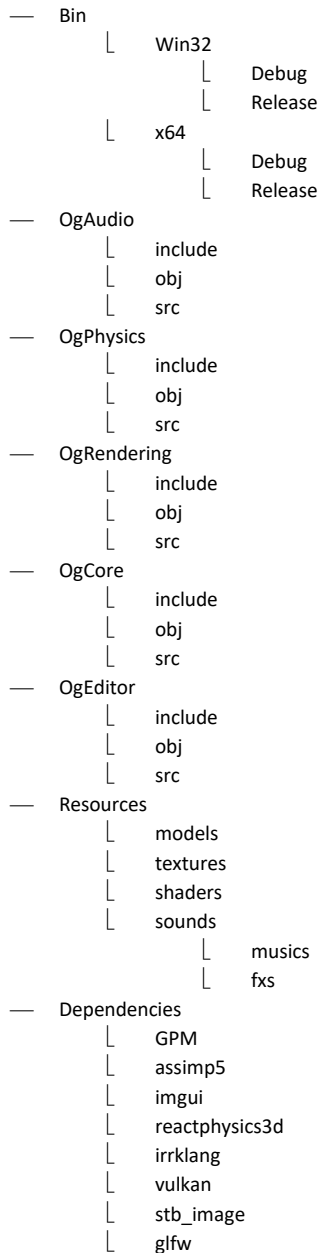
Exemple d'un template suivant les règles ci-dessus :

```
template <typename Floating,  
    std::enable_if_t<std::is_floating_point<Floating>::value, int> = 0>  
Floating ClampFloatValue(  
    const Floating p_value,  
    const Floating p_min,  
    const Floating p_max)  
{  
    return (p_value < p_min) ? p_min : ((p_value > p_max) ? p_max : p_value);  
}
```



# Organisations des fichiers

## Arborescence



Chaque projet contiendra ses fichiers intermédiaires .obj/.a dans le dossier obj dudit projet. La génération des projets en dll iront dans le dossier DLL a la racine de la solution et contiendra les .lib et .dll associées à chaque projet.

Ces derniers seront par la suite copiés automatiquement dans le dossier bin dans la bonne configuration afin d'obtenir l'exécutable avec ses dépendances, de même avec le dossier Ressources qui sera copié lui aussi vers l'exécutable.

## Gitflow

Le versionning sera composé de plusieurs branches afin de séparer le développement des fonctionnalités.

Voici donc la liste des branches :

- develop
  - features\_x (x = nom de la feature)
- master (dernier build fonctionnel)
- alpha (phase 1 du projet)
- beta (phase 2 du projet)
- release (phase 3 du projet)

On programmera chacun dans notre sous-branche develop et ne pusherons notre travail, fonctionnel uniquement, que dans master via un merge. Si master venait à ne pas compiler, on s'assurera de fix le problème avant de continuer le développement d'une autre fonctionnalité dans develop.

Lorsque du travail a été ajouté dans master on **devra** récupérer les modifications et les merger dans notre branche de développement afin de rester à jour et éviter des conflits lors du merge dans master.

Ainsi, on s'assure de pouvoir travailler chacun indépendamment des autres avec les dernières fonctionnalités de master.

À l'approche de chaque deadline on devra push la dernière version de master supposément fonctionnelle dans la branche associée : alpha pour la phase 1 du projet, beta pour la phase 2 du projet et release pour la phase 3 du projet.

## Découpage des modules

Les modules seront présents au nombre de 5. On y retrouvera l'audio, le rendu, l'éditeur, la physique et enfin le cœur.

Chaque module sera précédé par **Og** et donnera donc la liste suivante :

- **OgAudio**
- **OgCore**
- **OgEditor**
- **OgPhysics**
- **OgRendering**

Le module OgAudio aura pour rôle de jouer les sons et musiques du jeu ainsi que de contenir les fichiers audio prêt à être utilisé.

Le module OgCore sera le système de gameplay avec les appels aux Update, la logique, les entrées, gestion de scènes ou encore le système ECS.

Le module OgEditor sera principalement la présentation du module OgCore en offrant une interface graphique pouvant exposer toutes les possibilités offertes par le module OgCore en code.

Le module OgPhysics permettra de gérer la physique de chaque entité du module OgCore.

Le module OgRendering aura pour rôle de gérer l'affichage et le rendu de l'API choisie.

## Choix des API et technologies

### Audio

Deux grandes API viennent naturellement à l'idée afin d'implémenter un module audio, irrKlang et FMOD. irrKlang est bien connu pour son efficacité mais surtout pour sa facilité d'usage et en fait un très bon allié pour une itération rapide. FMOD est nettement plus complet et puissant mais est aussi beaucoup plus dur à intégrer et utiliser.

Par soucis de temps et de main d'œuvre et étant donné le choix des autres API utilisées pour les autres modules, nous avons choisi d'utiliser irrKlang afin d'obtenir un module audio rapidement et pratiquement plug-and-play dans notre moteur.

### Physique

Deux grandes API existent aussi pour réaliser de la physique, on a pensé notamment à PhysX ainsi qu'à Bullet. Deux défauts majeurs entrent en cause dans le choix de ces deux API, PhysX est complexe à utiliser et à mettre en place. D'un autre côté Bullet ne nous a pas convaincu à cause de son manque de stabilité et maturité, tout manque de clarté. Nous n'étions donc pas satisfaits envers aucune des deux technologies citées. Ainsi, après quelques recherches sur les autres API existantes mais moins connus, nous sommes tombés sur ReactPhysics3D. Il s'agit d'une API qui se veut très simple d'utilisation, certes cette API est moins puissante que PhysX et Bullet, mais nous ne souhaitons pas passer un temps conséquent à l'implémentation d'une physique extrêmement réaliste.

Une physique simple mais suffisante est ce que nous recherchons plutôt que de la réelle simulation, nous faisons ce choix afin de répondre à notre problématique de main d'œuvre et de temps afin de se concentrer sur l'essentiel que sont le gameplay et le rendu.

### Rendering

Pour le rendu nous avons pensé à plusieurs APIs que sont OpenGL, DirectX11, DirectX12 et Vulkan.

Nous connaissons déjà OpenGL et voulions nous diriger vers quelque chose de nouveau et technologiquement avantageux en performance et conceptuellement pour améliorer nos compétences.

Par la même occasion, nous disposons de cartes graphiques ayant une architecture permettant le Ray Tracing et souhaitons aussi utiliser le plein potentiel de ces nouvelles architectures en implémentant le Ray Tracing. De ce fait, l'utilisation du Ray Tracing nous empêche d'utiliser DirectX11 du fait que cette API ne propose pas de compatibilité pour le Hardware Accelerated Ray Tracing. Cela laisse donc le choix entre Vulkan et DirectX12.

Tous deux ont le défaut d'être extrêmement verbeux et punitif en cas d'erreur bien que le debug et rapports d'erreurs soit parti intégrante de ces APIs. L'avantage premier de Vulkan est d'être compatible avec toutes les plateformes, ce qui signifie une certaine abstraction ne nous forçant donc pas à adopter les syntaxes d'un système plutôt qu'un autre. Un autre point positif concerne les benchmarks qui ont su démontrer à plusieurs reprises un léger avantage de performance pour cette dernière, cela risque de

varier d'ici la démocratisation de ces deux API, encore trop peu utilisées et maîtrisées dans l'industrie, qui seront plus analysées, améliorées et « benchmarké » avec leurs évolutions et leurs intégrations dans l'industrie.

Finalement après tous ces comparatifs, nous avons décidé d'utiliser Vulkan pour tous les avantages décrit précédemment et avons pu réaliser avec succès un rendu en Rastérisation ainsi que le nouveau rendu en Ray Tracing dans des temps assez court.

### Chargement de modèles 3D

Pour cela nous avons fait un choix assez rapidement entre Assimp et FBX SDK. FBX SDK nous est inconnu et est la propriété d'Autodesk, il est extrêmement complet et permet bien plus que de charger des modèles 3D mais aussi complexe à utiliser, Assimp nous est bien connu et permet de charger beaucoup de modèles 3D très simplement et efficacement. Nous avons décidé d'aller au plus simple et d'utiliser Assimp étant donné que nous sommes à l'aise avec et savons parfaitement nous en servir.

### Chargement d'images

On utilisera pour cela une API très légère dénommée « stb\_image » qui fait parfaitement bien le travail demandé et est compatible Vulkan côté alignement mémoire.

### Éditeur

On pouvait soit choisir Dear ImGui qui est très connue et très utilisée ou bien Qt, encore plus connue et encore plus utilisée mais qui ne s'implémente pas aussi facilement que Dear ImGui. Dear ImGui a la particularité d'être très léger et intimement lié au contexte de rendu représentant moins d'overhead et donc de meilleures performances. Sa force réside dans sa simplicité et ne nécessite que très peu de code pour aboutir à une fenêtre de rendu complète. Qui plus est, Dear ImGui est nativement compatible avec OpenGL (en premier lieu) mais aussi avec sa succession qu'est Vulkan et même DirectX12.

L'intégration de ImGui dans Vulkan est bien expliquée et la prise en main est facile. ImGui s'intègre bien dans Vulkan s'il est bien architecturé est rendu modulaire. ImGui possède son propre pipeline de rendu à quelques détails près. Il possède sa propre render pass, ses propres framebuffers, ainsi que son propre draw call.

### Avantages

Les avantages d'ImGui font qu'il est très modulaire, comprends beaucoup d'extensions simple à mettre en place, ainsi qu'une utilisation et une intégration très simple pour pouvoir créer son éditeur.

Nous pouvons facilement intégrer des fenêtres diverses et variées ainsi que des modules déjà pré-fait par ImGui nous permettant par exemple d'implémenter des « Drag Float » ou encore des « Tree » nous permettant d'intégrer facilement notre « SceneGraph » dans l'éditeur, ce qui nous permet d'avoir une hiérarchie d'objets dans celui-ci. Grâce à cette hiérarchie modulaire, nous pouvons également sélectionner un GameObject et modifier ses composants grâce à une fenêtre inspecteur comprenant tous les composants de l'objet (Transform, Material etc...).

Grâce à ImGui, nous disposons également d'une barre de menus sur la partie supérieure de l'éditeur nous permettant d'avoir, pour le moment, des onglets comprenant l'ajout d'objets comme un cube, une sphère ou une surface, ainsi qu'un hot-reload des shaders en un clic. Cependant, nous avons rencontrés quelques problèmes techniques durant l'implémentation d'ImGui.

### Inconvénients

Durant l'implémentation d'ImGui, une limite technique nous a fait barrière concernant la « Game View » de l'éditeur. Pour afficher une image rendue depuis Vulkan, il faut la convertir dans un format de texture qu'ImGui peut lire et interpréter. Cependant, bien qu'ImGui disposait de fonctions permettant la conversion depuis presque toutes les API les plus connues, Vulkan n'en faisait pas parti, du moins l'implémentation était encore à l'état expérimental et donc indisponible dans la version stable d'ImGui. Pour résoudre ce problème, il nous a fallu arranger nous-même une image Vulkan pour qu'elle soit lisible par ImGui. La solution que nous avons trouvée est donc de créer un VkDescriptorSet comprenant une texture précédemment remplie depuis l'image brut que Vulkan fournissait pour la présentation à l'écran. Une fois ce VkDescriptorSet créé, il pouvait être traduit et lu par ImGui dans un format « ImTextureID » qui est un type pointeur sur data. Après avoir résolu ce problème, nous pouvions donc rendre notre scène Vulkan précédemment dessinée à l'écran dans une fenêtre ImGui. Pour le moment, il s'agit de la seule limitation technique que nous avons rencontrée.

### Scripting

Nous avons recherché des façons faciles d'exporter du code C++ dans un langage de scripting tel que Python, Lua, etc. Nous avons finalement retenu Python 3.8 pour son énorme popularité et ses très nombreux package facilitant l'export des fonctionnalités d'un langage. Pour Python nous avons retenu SWIG, la majorité des autres package étant réservé pour Linux. En créant simplement des fichiers .i permettant d'interfacer avec les fichiers .h du projet que l'on souhaite exporter, toutes les méthodes se retrouvent exportées via 2 lignes de commandes Python. L'export est donc très simple et le langage Python de-même, cela correspond parfaitement à la philosophie du moteur.

## Diagramme de classes : UML

### Module audio

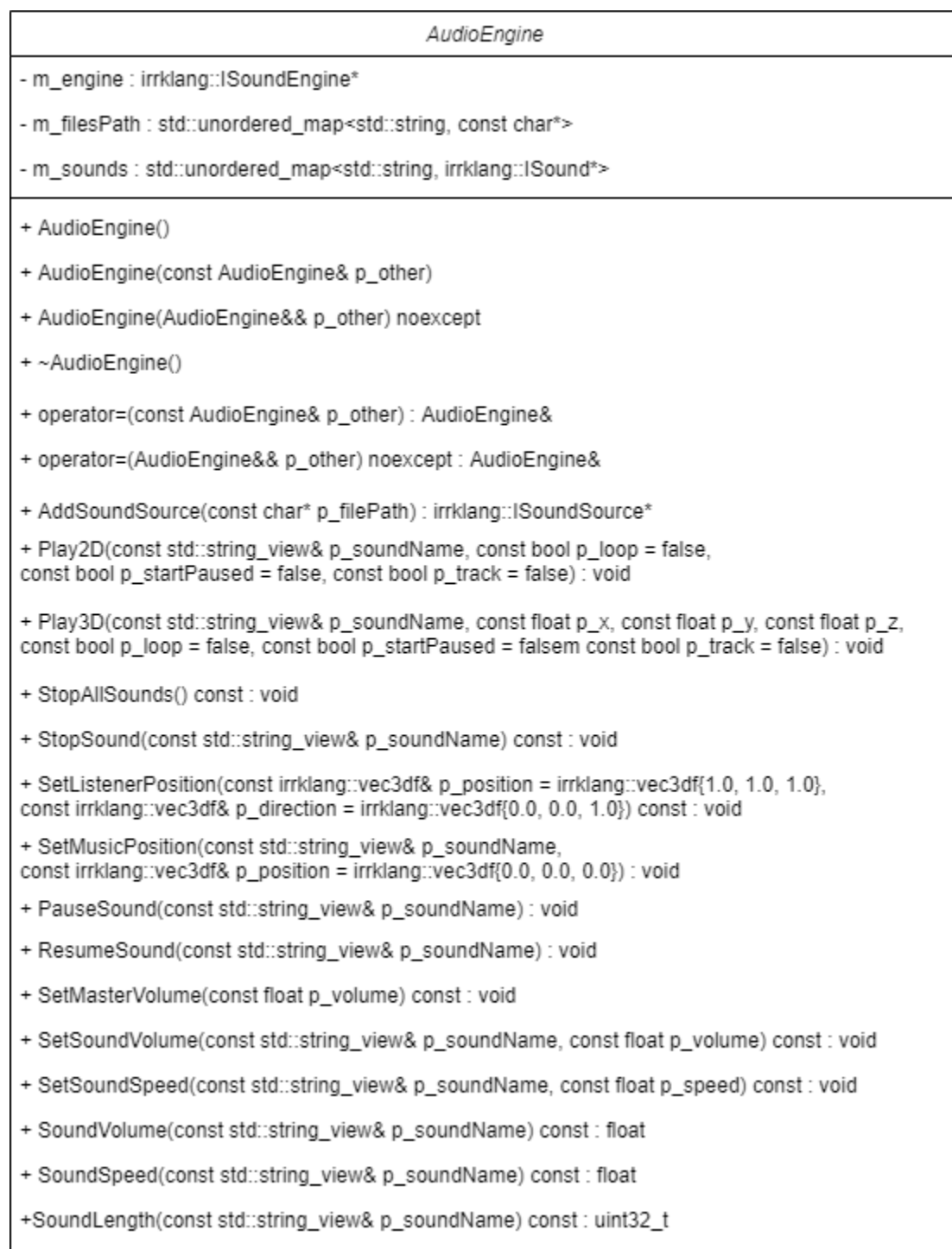


Figure 1: UML du module Audio

## Module Rendering

### Contexte

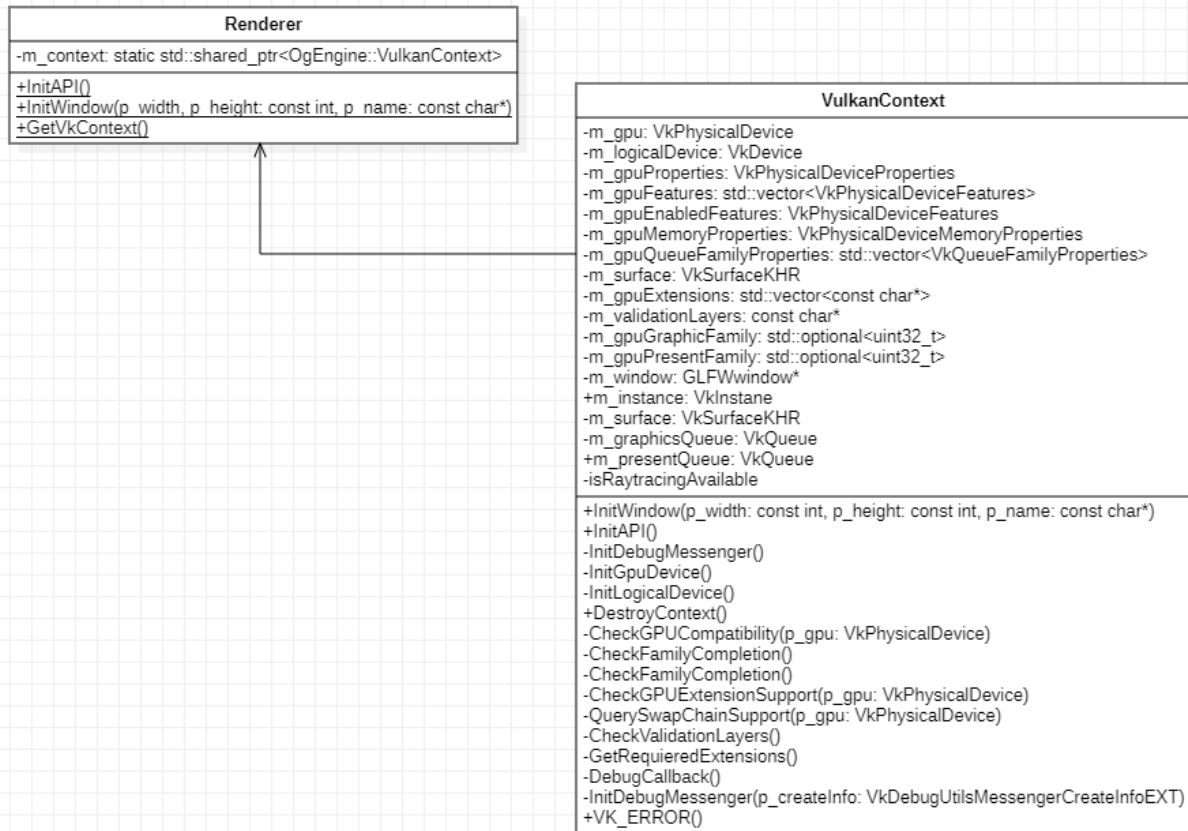


Figure 2: UML du contexte du module Rendering



## ResourceManager

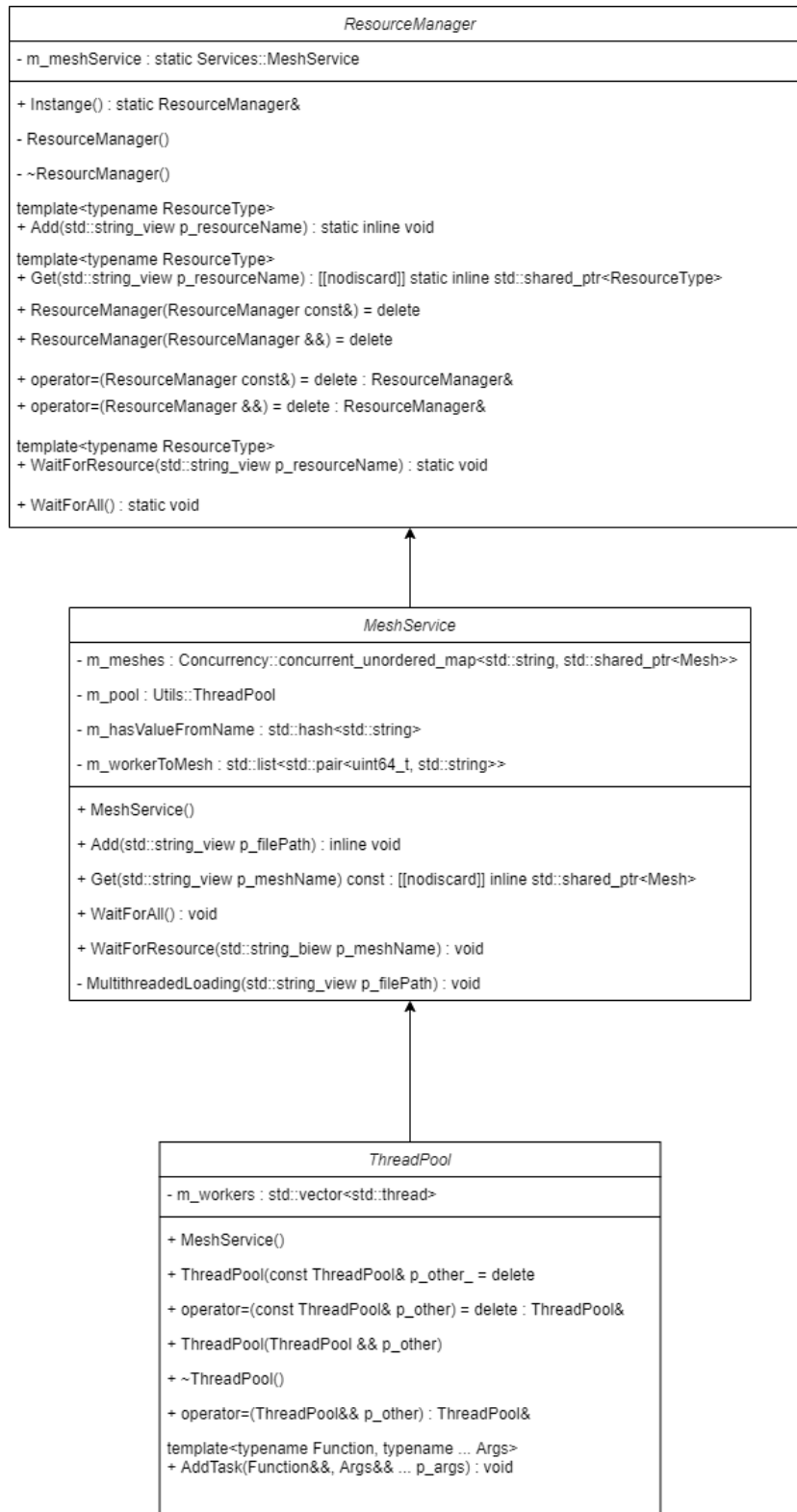


Figure 3 Resource Manager

## Module Core

### SceneGraph

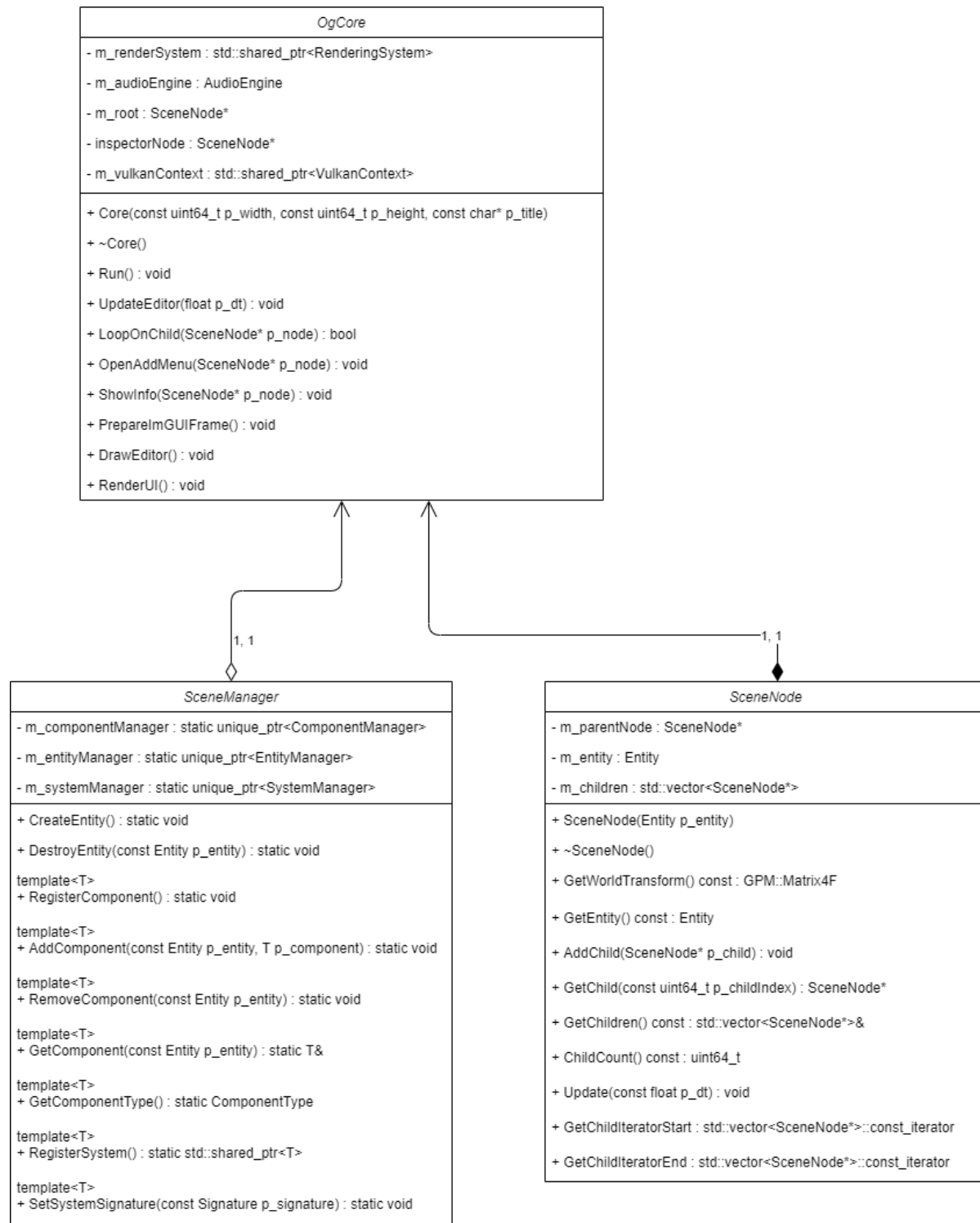


Figure 4 Core module : SceneGraph

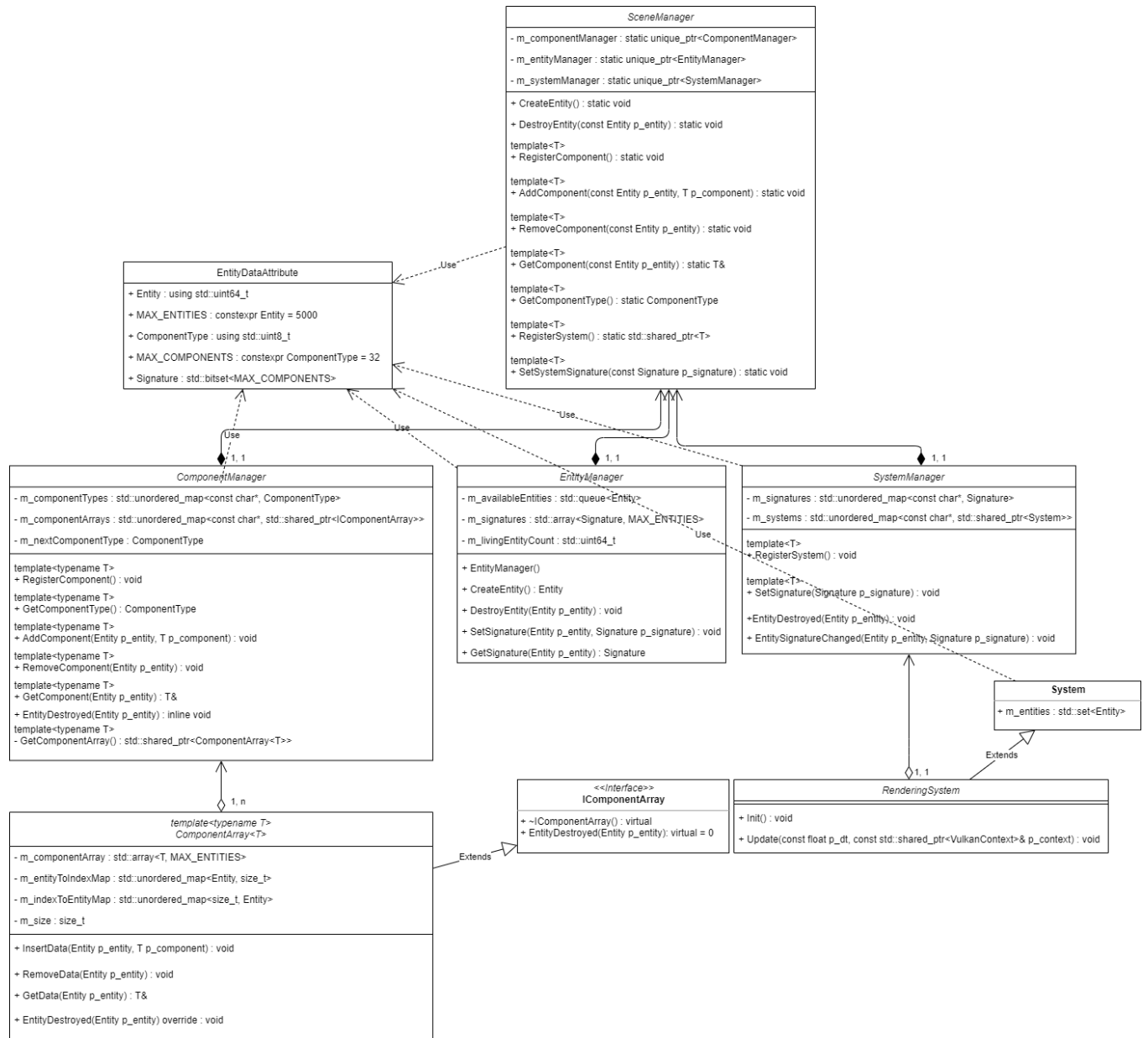


Figure 5 Core module : ECS

## Components du Core

Model	Material	Transform
<ul style="list-style-type: none"> <li>- m_mesh : std::shared_ptr&lt;Mesh&gt;</li> <li>- m_material : Material</li> <li>- m_meshTransform : Transform*</li> </ul>	<ul style="list-style-type: none"> <li>+ color : const Vector4F&amp;</li> <li>+ metallic : const float&amp;</li> <li>+ reflectance : const float&amp;</li> <li>+ roughness : const float&amp;</li> <li>+ type : const int&amp;</li> <li>- _color : Vector4F</li> <li>- _metallic : float</li> <li>- _reflectance : float</li> <li>- _roughness : float</li> <li>- _materialType : int</li> </ul>	<ul style="list-style-type: none"> <li>+ worldMatrix : const Matrix4F&amp;</li> <li>+ localMatrix : const Matrix4F&amp;</li> <li>+ position : const Vector3F&amp;</li> <li>+ localPosition : const Vector3F&amp;</li> <li>+ scale : const Vector3F&amp;</li> <li>+ localScale : const Vector3F&amp;</li> <li>+ rotation : const Quaternion&amp;</li> <li>+ localRotation : const Quaternion&amp;</li> <li>+ rotation : const Quaternion&amp;</li> <li>+ localRotation : const Quaternion&amp;</li> <li>- _parent : Transform*</li> <li>- _rotation : Quaternion</li> <li>- _localRotation : Quaternion</li> <li>- _scale : Vector3F</li> <li>- _localScale : Vector3F</li> <li>- _localPosition : Vector3F</li> <li>- _position : Vector3F</li> <li>- _worldMatrix : Matrix4F</li> <li>- _localMatrix : Matrix4F</li> </ul>
LightSource		
<ul style="list-style-type: none"> <li>- m_lightTransform : Transform*</li> <li>+ position : Vector3F</li> <li>+ ambient : Vector4F</li> <li>+ diffuse : Vector4F</li> <li>+ specular : Vector4F</li> </ul>	<ul style="list-style-type: none"> <li>+ Material()</li> <li>+ Material(const Material&amp; p_material)</li> <li>+ Material(Material&amp;&amp; p_material)</li> <li>+ ~Material()</li> <li>+ SetColor(const Vector3F p_color) : void</li> <li>+ SetRoughness(const float p_roughness) : void</li> <li>+ SetMetallic(const float p_metallic) : void</li> <li>+ SetReflectance(const float p_reflectance) : void</li> <li>+ SetType(const int p_type) : void</li> </ul>	<ul style="list-style-type: none"> <li>+ Transform()</li> <li>+ GetTransform() : const Transform*</li> <li>+ Transform(const Transform&amp; p_transform)</li> <li>+ Transform(const Matrix4F&amp; p_matrix)</li> <li>+ Transform(Transform&amp;&amp; p_transform)</li> <li>+ ~Transform()</li> <li>+ Translate(const Vector3F&amp; p_translation) : void</li> <li>+ Scale(const Vector3F&amp; p_scale) : void</li> <li>+ Rotate(const Quaternion&amp; p_rotation) : void</li> <li>+ SetPosition(const Vector3F&amp; p_position) : void</li> <li>+ SetScale(const Vector3F&amp; p_scale) : void</li> <li>+ SetRotation(const Quaternion&amp; p_rotation) : void</li> <li>+ SetParent(Transform* p_parent) : void</li> <li>+ SetWorldMatrix(const Matrix4F&amp; p_worldMatrix) : void</li> <li>+ WorldForward() : Vector3F</li> <li>+ WorldUp() : Vector3F</li> <li>+ WorldRight() : Vector3F</li> <li>+ LocalForward() : Vector3F</li> <li>+ LocalUp() : Vector3F</li> <li>+ LocalRight() : Vector3F</li> </ul>
<ul style="list-style-type: none"> <li>+ Model()</li> <li>+ Model(std::string_view p_meshName)</li> <li>+ Model(std::shared_ptr&lt;Mesh&gt; p_mesh)</li> <li>+ Model(const Model&amp; p_model)</li> <li>+ Model(Model&amp;&amp; p_model)</li> <li>+ SetMesh(const std::shared_ptr&lt;Mesh&gt;&amp; p_mesh) : void</li> <li>+ SetMesh(std::string_view p_meshName) : void</li> <li>+ SetMaterial(const Material&amp; p_material) : void</li> <li>+ Mesh() const: const std::shared_ptr&lt;Mesh&gt;&amp;</li> <li>+ Material() : Material&amp;</li> <li>+ ModelMatrix() const : Matrix4F</li> <li>+ SetLocalTransform(Transform&amp; p_transform) : void</li> </ul>		

Figure 6 Core module : Components

## Autres modules

L'UML des autres modules sont encore en phase de conception.