Hadrien **BEAUFILS**
Benoît **BELZ**
David **IMBS**
Hubert **LACÔTE**
Anthony **PAPROKI**
Guillaume **STUBER**

IN55

RAPPORT DE PROJET

SUJET #1
Moteur 3D pour le jeu vidéo Delirium2

Responsable de l'UV : Fabrice Lauri
P2010

SOMMAIRE

Introduction

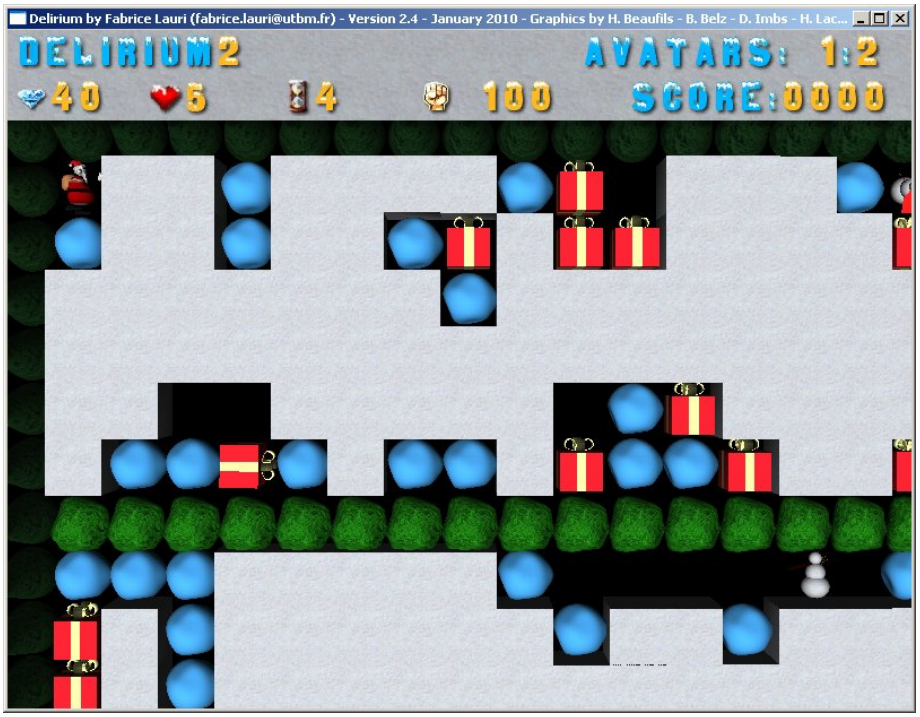
Pour ce projet d'IN55, nous avons choisi de traiter le sujet n°1, la conception et le développement d'un moteur 3D pour le jeu Delirium2 nous plaisait par différents aspects. En effet, il nous permettrait dans un premier temps de passer à l'application des connaissances du cours, cela sans toutefois s'occuper des mécanismes de jeux, nous focalisant seulement sur l'aspect graphique. Dans un deuxième temps, le projet s'effectuant avec un groupe de six personnes, il nous a permis de bien saisir les enjeux et les difficultés du travail en groupe.

Le but de ce jeu est de déplacer un mineur et de lui faire ramasser des diamants en évitant qu'il ne se fasse tuer par des ennemis ou encore par des chutes de pierres. Une fois un nombre suffisant de diamants ramassés, il peut se rendre à la sortie et passer au niveau suivant.

Nous avons donc dû nous greffer à cette mécanique de jeu, récréant les modèles 3D, les structures et classes permettant de les lire, de les stocker pour finalement utiliser leurs informations pour l'affichage du jeu. Nous avons pour ce projet tenté de créer une base modulaire, base que nous pourrons alors utiliser pour d'autre projet.

I. Présentation du moteur original et du nouveau moteur

Le jeu original



Le jeu avec le nouveau moteur

II. Création de librairies indépendantes de Delirium2

Notre volonté dès le départ était de faire de ce projet un projet modulaire, association de librairies efficaces et « indépendantes », c'est-à-dire que nous pourrions réutiliser. Ne pouvant pas réutiliser de librairies existantes, nous avons donc décidé de créer plusieurs librairies indépendantes :

- Une librairie Importer3ds, permettant d'extraire la structure d'un fichier 3DS ;
- Une librairie LSG « Light Scene Graph », permettant de de disposer des fonctionnalités d'un graphe de scène dans notre projet.

II.1. Importer 3ds

Pour importer des modèles 3d dans notre scène, nous avions besoin d'un importeur. Notre choix s'est porté vers le format 3ds pour différentes raisons. Tout d'abord, ce format est celui utilisé par le jeu original. Les modèles 3D originaux sont donc dans ce format. Nous pouvions ainsi faire nos tests sans attendre la fin de la modélisation de tous les objets du jeu. Ensuite il s'agit d'un des formats les plus répandus, et il est implémenté dans un nombre conséquent de logiciel. Et contrairement au .obj, format également très répandu, il gère la notion de hiérarchie. Il a par contre le désavantage d'avoir des spécifications non dévoilées au grand public, ainsi il faut se reposer sur des spécifications faites à base de tests, ce qui complique la tâche.

Nous avons fait le choix de créer une librairie indépendante Importer3ds contenant plusieurs classes : en premier lieu il y a la classe Parser, qui comme son nom l'indique va s'occuper de la lecture et de l'extraction des informations du fichier concerné. Un fichier 3ds est composé de blocs ou de « chunk » possédant une définition, une longueur et possiblement des blocs fils. Chacun de ces « chunks » comporte des informations sur des points précis du modèle, et plus on descend dans la hiérarchie, plus on arrive aux informations les plus précises (comme les définitions des vertex, des matériaux, etc...). La classe Chunk est utilisée par la classe Parser pour lire ces blocs.

L'importeur a sa propre structure de stockage interne, qui la rend complètement indépendante des autres éléments du moteur. Il suffit alors de traduire cette structure dans la structure que nous voulons finalement avoir. Cette structure interne est définie par les classes Node, Entity et Material. La première définit très simplement l'équivalent d'un nœud dans un scène-graph et nous permet de construire une hiérarchie. Elle peut contenir la seconde classe, à savoir Entity, qui représente un maillage. Enfin cette classe contient la classe Material qui stocke les infos des matériaux associés aux faces du maillage. Le diagramme de classe de l'importeur est fourni en annexe. **(Annexe 1)**

Cet importer peut évidemment être sujet à évolution car il ne gère pas certains points du format. En effet, il ne lit pas les animations, se contentant de lire la première KeyFrame pour des besoins de placement du modèle. Il résout avec plus ou moins de succès des problèmes qui peuvent survenir quand on déplace le pivot d'un maillage manuellement dans le logiciel de modélisation.

II.2. Création du graphe de scène LSG

Il nous semblait très important dans notre projet de disposer d'un graphe de scène. Le graphe de scène que nous avons créé est très facile d'utilisation et est très évolutif de par sa conception. Nous nous sommes inspirés du graphe de scène libre OSG (Open Scene Graph) pour créer un graphe de scène léger, efficace et adapté à nos besoins : Light Scene Graph.

II.2.1. Structure générale de la librairie

La librairie créée est orientée autour de deux composantes principales :

- La structure du graphe de scène (LSG) ;
- Un module permettant l'affichage de graphes de scène (LSGViewer).

Le rôle du graphe de scène est de fournir une structure pour stocker des données chargées depuis des modèles 3D tout en permettant de les positionner. Il fournit une interface permettant de parcourir les nœuds : l'objet NodeVisitor qui peut être facilement dérivé. Et il permet aussi de récupérer des informations sur les modèles 3D, comme les boîtes englobantes des nœuds par exemple. La structure de la librairie LSG est présentée en Annexe à l'aide d'un diagramme de classes UML. **(Annexe 2)**

Le rôle de l'afficheur de graphes de scènes est de permettre de visualiser plusieurs scènes et des HUDs (Head Up Display). L'afficheur permet donc d'afficher des scènes ou des HUDs et a donc pour mission de charger les composantes du graphe de scène au moment opportun. La structure de la librairie LSGViewer est présentée en Annexe à l'aide d'un diagramme de classes UML. **(Annexe 3)**

II.2.2. Fonctionnement du graphe de scène

Noeuds

Notre graphe de scène est très modulaire et permet ainsi qu'un nœud puisse avoir plusieurs parents. Les modèles 3D sont ainsi chargés dans un nœud, et le nœud résultant peut être positionné – à l'aide d'une Transformation - à plusieurs endroits de la hiérarchie sans avoir à dupliquer le noeud. Nous utilisons pour cela des pointeurs partagés qui assurent d'ailleurs la suppression automatique des nœuds qui ne sont plus utilisés.

Il possède une liste de pointeurs partagés de géométries – donc un objet géométrie peut être partagé par plusieurs noeuds – les géométries permettent de contenir un maillage. Enfin un noeud contient une boîte englobante (correspondant à la boîte englobante de ses fils et des géométries qu'il contient) permettant de savoir où il se trouve localement ou dans le monde.

Géométries

La structure des géométries permet de stocker le maillage d'un modèle 3D. Etant donné qu'une géométrie ne possède qu'un seul matériel, un modèle 3D est donc généralement composé de plusieurs géométries. Les géométries partagent entres elles une liste de sommets (position 3D, normale, coordonnées de textures 2D), et possède chacune

une liste d'indices permettant de définir quels sommets sont utilisés pour constituer une facette de la géométrie.

Matériel

La classe Material contient les informations qui vont permettent d'éclairer les pixels d'une géométrie. Elle contient des vecteurs 4D pour chacune des composantes de lumière (diffuse, émissive, spéculaire, ambiante) ainsi qu'un chemin vers une texture (la texture de la géométrie) qui pourra être appliquée sur les facettes selon les coordonnées de mapping définies par les sommets la composant. La structure proposée ne permet pas de gérer le multitexturing.

Transformations

La classe Transformation sert à positionner déplacer nos objets dans l'espace local ou selon le nœud parent. Elle contient le panel nécessaire d'éléments mathématiques (positon en 3D, vecteur d'échelle, quaternion d'orientation) afin que l'on puisse orienter et positionner les noeuds. Il est à noter que la matrice 4x4 homogène n'est générée que quand nécessaire.

Boîtes englobantes

Les boîtes englobantes de notre graphe de scène sont des AABB (Aligned Axis Bounding Box). Elles sont calculées automatiquement lors de la création ou de la mise à jour du graphe de scène.

II.2.3. Fonctionnement de l'afficheur de graphes de scène

Fonctionnement général de l'afficheur

L'afficheur contient plusieurs scènes et plusieurs HUDs. Son rôle va être de dessiner les scènes puis de dessiner les HUDs.

L'afficheur se charge de positionner les lumières et de préparer les objets présents dans le graphe de scène de façon par exemple à ce que les textures soient chargées et invalidées au moment opportun (il est possible de voir dans la console le moment où les textures sont chargées et décharger à l'aide du Logger mis en place).

L'afficheur supporte les textures 2D au format TGA. Le chargement de textures est un point important car celles-ci sont utilisées dans tout le projet (scènes pour les objets et HUD pour les symboles). Une classe a été créée afin d'importer les textures au format TGA, cet importeur supporte les formats suivants :

- Les TGA standards en 8, 16, 24 ou 32 bits ;
- Les TGA en niveaux de gris de 8 ou 16 bits ;
- Les TGA compressées en RLE de 8, 16, 24 ou 32 bits ;
- Les TGA en niveaux de gris, compressées en RLE de 8 ou 16 bits.

Scènes et caméras

Chaque scène est dotée d'une caméra et référence un graphe de scène qui peut éventuellement être partagé par plusieurs scènes. Ainsi, il est par exemple possible de créer un graphe de scène et de le visualiser à partir de plusieurs caméras tout comme il est possible de visualiser plusieurs graphes de scènes différents.

Chaque scène est un Viewport, ce qui permet de la positionner sur l'écran et de lui donner une dimension qui lui est propre. Il est tout à fait possible de dessiner plusieurs scènes dans un ordre défini, étant donné que l'afficheur permet de gérer l'ordre d'affichage des scènes.

Les caméras de chaque scène sont accessibles depuis l'extérieur, ce qui permet de leur faire subir des transformations.

HUDs

Un autre aspect important d'un moteur 3D se place au niveau de l'affichage d'informations textuelles et/ou de symboles (de type icônes notamment). C'est dans cette optique que la classe HUD (Head Up Display) a été crée.

Un HUD est un Viewport particulier qui permet d'afficher des informations, souvent textuelles et au dessus d'une scène.

Cette classe a pour particularité de représenter une vue de type orthogonale, un ratio d'aspect de 1, et de ne pas tenir compte de la profondeur des objets dessinés. Ainsi, l'arrière plan sera dessiné en premier, puis ensuite viendront les symboles.

Le HUD contient en effet des tableaux de symboles (qui sont complétés par une position sur le HUD et un facteur de dilatation). Chacun de ces tableaux est identifié par une chaîne de caractères. L'association des deux, forme des objets de type SymbolStringType. Ainsi, on peut créer un objet SymbolString d'identifiant `"Force"` associé au tableau `["Strength","","1","0","0"]` qui aura pour effet l'affichage suivant :

Le HUD permet aussi la gestion d'un ensemble de tableaux de symboles de manière simple et efficace.

III. Utilisation de LSG pour la création du moteur 3D de Delirium

III.1. Présentation du fonctionnement global du moteur 3D

Après avoir créé les librairies Importer3ds et LSG, il ne nous restait plus qu'à les utiliser en fonction des données du jeu. Un diagramme d'activité UML reprend en annexe le fonctionnement global du moteur 3D. **(Annexe 5)**

Nous utilisons la librairie LSG créée pour créer un graphe de scène qui va contenir tous les modèles 3D chargés. Le graphe de scène est ensuite mis à jour selon le déplacement des entités. Il peut y avoir jusqu'à deux scènes et quatre HUDs selon le nombre de joueurs et l'affichage ou non des caractéristiques de la caméra et du jeu.

Le diagramme UML du moteur 3D se trouve en annexe. **(Annexe 4)**

III.2. Construction et mise à jour du graphe de scène

Le rafraîchissement du graphe de scène a été confié à la classe GraphUpdater. Le rafraîchissement du graphe était surtout nécessaire pour s'adapter au fonctionnement du jeu. Il s'agit ici de mettre à jour le graphe de scène à chaque frame. Le code réalisé a pour rôle de modifier, supprimer ou d'ajouter des nœuds au graphe de scène.

Afin d'éviter tout problème de fuites mémoires et d'optimiser les itérations (comprendre ne faire qu'un parcours de la carte), nous avons utilisé une sorte de garbage collection qui va supprimer tous les nœuds des uid n'étant plus présent dans le jeu.

III.3. Gestion de la caméra

La camera de notre projet utilise bien entendu la camera de LSG. Nous faisons l'interaction avec celle-ci via les classes CameraController et CameraAvatarFollower.

III.3.1. CameraController

Cette classe est utilisée pour interagir avec le moteur jeu (elle hérite donc de AbstractCamera). Elle manipule la caméra de la scène – dans le cas de deux joueurs, la caméra du deuxième joueur – et donc possède une référence vers celle-ci. C'est un pointeur vers ce contrôleur de caméra que l'on envoie au moteur du jeu pour que ses fonctions soient appelées lorsque l'on appui sur les touches.

III.3.1. CameraAvatarFollower

Cette caméra possède aussi une référence vers la camera LSG de la scène. Elle permet de suivre automatiquement l'avatar lors de ses déplacements. Les modifications de cette camera ne sont faites que lorsque l'avatar bouge. Dans le cas où il bouge, nous ajoutons le mouvement à effectuer à un vecteur stockant la somme des mouvements encore à faire et nous démarrons la translation. Le mouvement se fait progressivement afin d'avoir une camera fluide lors de ses déplacements.

A noter que dans le cas ou nous avons deux avatars (deux joueurs), nous instancions deux CameraAvatarFollower, chaque instance possédant une référence vers l'une des deux caméras.

III.4. Gestion des HUDs

La classe HUD de l'afficheur est utilisée pour afficher les différentes informations du jeu.

Le rendu obtenu est le suivant pour l'affichage du haut :

Pour ce faire, le HUD contient un arrière plan transparent et plusieurs chaînes de symboles. Chaque symbole est définit de cette façon :

Plusieurs HUD sont créés et mis à jour dans le jeu. Lorsqu'il y a deux joueurs, deux HUD sont créés en haut de l'écran pour afficher les informations spécifiques à chaque joueur.

Voici le rendu final avec tous les HUDs activés sauf celui des informations de la caméra dans le mode deux joueurs :

III.5. Modélisation

Etape indispensable à la mise en valeur du moteur de rendu, la modélisation a consisté à créer une quarantaine de modèles 3D destinés à être affichés par le moteur.

Format utilisé

Le format d'objet 3D que nous avons choisi est le format 3DS du logiciel de modélisation 3ds Max. Nous avons préféré ce format à d'autres plus utilisés (comme le .obj) parce que c'est le format qui était utilisé dans le moteur de rendu de Delirium déjà existant. Ainsi nous avons pu tester nos propres modèles 3D avec un moteur de rendu autre que le notre.

Outils utilisé

Le format utilisé étant le .3DS, il est évident que nous avons utilisé le logiciel d'Autodesk, 3ds Max 2009.

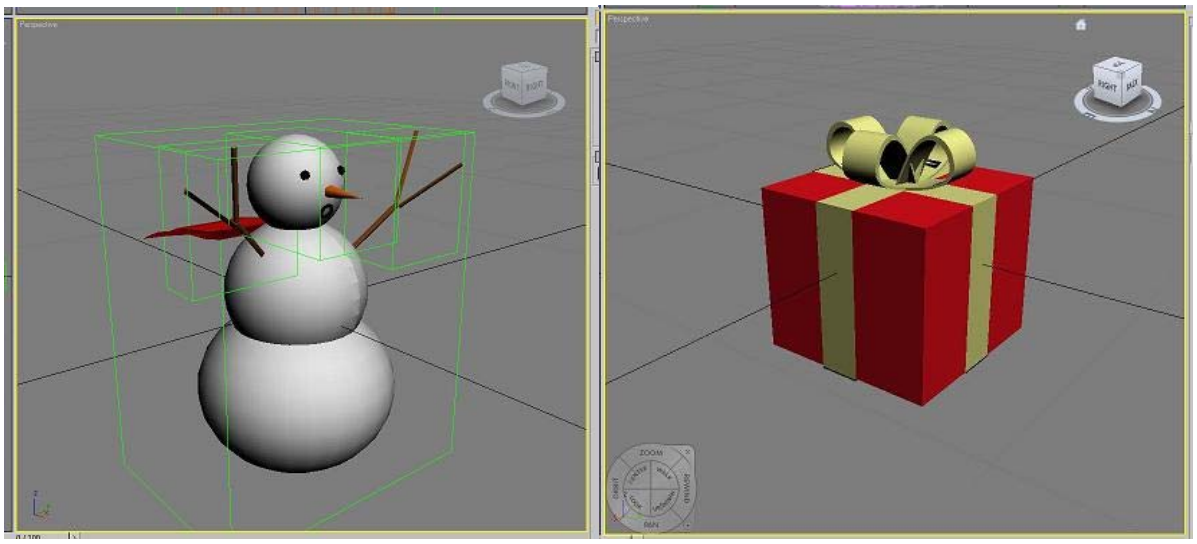
Les autres outils utilisés sont les logiciels GIMP pour la création de texture, et XnView pour la conversion d'images au format TGA utilisé pour les textures appliquées au modèle.

Travail réalisé

La première étape a été de se documenter sur la modélisation 3D en général et sur l'utilisation du logiciel 3ds Max en particulier, car la modélisation n'est pas abordée dans le cadre de l'UV. Les connaissances apprises en cours nous on tout de même permis d'appréhender très facilement les différents concepts de la modélisation 3D.

Une fois les bases acquises, le reste du travail a consisté à appliquer ces connaissances en réalisant l'ensemble des modèles 3D.

Nous avons ici choisi un thème différent de celui du jeu original (mineur ramassant des diamants dans une galerie). Nous avons opté pour un thème en relation avec Noël, avec l'utilisation de père Noël, de neige, de cadeaux, de bonhommes de neige, etc.



Les modèles réalisés sont plus complexes que les modèles du jeu de base. Nous avons toutefois veillé à modéliser les objets de façon optimisée, en réduisant au maximum le nombre de facettes utilisées. Voici par exemple le modèle de l'avatar que dirige le joueur, avec les facettes visibles :

IV. Points particuliers du projet

IV.1. Difficultés rencontrées

La plus grande difficulté à laquelle nous nous sommes heurtée a été d'organiser le projet de manière à ce qu'il soit bien conçu, performant et évolutif sans que personne n'ait les bases d'OpenGL prélables. Il a ainsi été nécessaire d'en apprendre très rapidement le fonctionnement tout en mettant en relation les différents éléments afin de créer un graphe de scène. Nous nous sommes pour cela inspiré du graphe de scène libre Open Scene Graph, ce qui nous a permis de créer un graphe de scène léger, efficace et adapté à nos besoins : Light Scene Graph.

Créer un projet très modulaire a apporté des difficultés supplémentaires. Nous avons dû ainsi créer une structure ne supposant rien sur Delirium puis utiliser les informations données par le jeu pour mettre à jour cette structure. Le jeu ne donnant pas suffisamment d'informations sur les événements se produisant, nous avons eu des difficultés à mettre à jour le graphe de scène et cela est très coûteux.

IV.2. Points remarquables

Conception

Les librairies créées pourraient être utilisées dans d'autres projets. Ainsi, la librairie Importeur3ds peut être utilisée séparément de LSG.

Le graphe de scène créé peut être très facilement étendu en ajoutant différents types de nœuds (nœuds de lumières, nœuds d'animation, …). Il suffirait ainsi d'adapter la classe NodeVisitor de manière à ce qu'elle permette l'accès à ces différents types de nœuds, ce qui a été prévu au départ, donc qui ne demandera que peu de modifications.

Nous avons été amené à utiliser différents Design Pattern nous ayant permis de créer un projet modulaire (Visitor, Singleton pour les classes utilitaires, …).

Nous avons respecté les conventions de codage imposées et nous avons même créé une documentation Doxygen de notre code consultable à cette adresse :

http://luffy.luck.free.fr/doc/in55/

Apprentissage

Nous avons été amené à mettre en relation toutes les connaissances vues ce semestre pour créer un projet performant et facilement extensible.

Plutôt que d'utiliser une librairie de chargement de fichiers 3ds existante telle que lib3ds, nous avons choisi de re-développer entièrement l'import de fichiers dans un but pédagogique. Cela nous a permis de bien comprendre la structure d'un fichier 3D.

Projet robuste et fiable

Nous utilisons dans notre projet des bibliothèques C++ performantes et éprouvées (STL, Boost et CML une bibliothèque Mathématiques), ce est une assurance de performance.

Une gestion des erreurs est mise en place à l'aide d'un système de Log, tous les affichages ont ainsi été redirigées vers une classe Logger ce qui permettrait par exemple de créer un fichier contenant les différents logs plutôt que de les afficher directement dans la console.

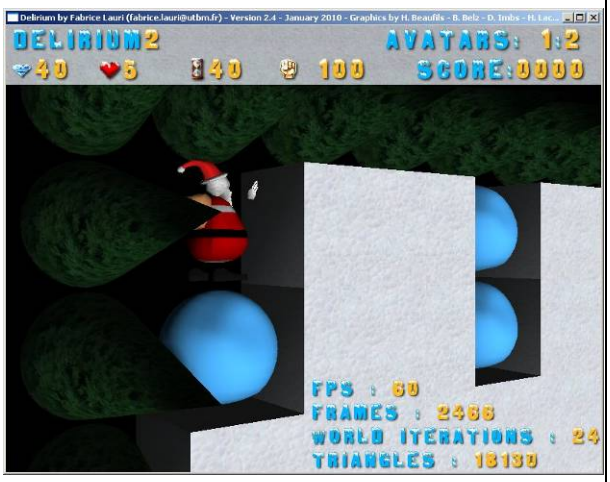
Des tests sont effectués pour savoir si il faut activer ou non certaines extensions (VBO, mipmapping) qui pourraient ne pas être supportées par tous les ordinateurs.

Optimisations

Le point le plus critique de notre programme réside dans l'actualisation du graphe de scène puisque nous sommes obligés de comparer à chaque tour, les objets qui se sont déplacés ou non. Une classe Profiler a été créée pour tester ce point critique.

Nous avons optimisé l'affichage en utilisant les VBO si l'extension est supportée et en n'affichant que les objets se trouvant dans le champs de vision à l'aide des boîtes englobantes de chaque nœud.

Le graphe de scène créé ne supposant rien sur Delirium, nous n'avons pas ajouté d'optimisations spécifiques à ce jeu et il sera performant dans des contextes très différents. Ainsi, la scène est par exemple rendue en n'affichant que les objets visibles dans le champs de vision quelle que soit l'orientation de la caméra contrairement au jeu d'origine qui affiche tous les triangles de la scène :

	
Moteur 3D original lancé avec nos modèles 3D affichant tous les objets de la scène dès que la caméra est rotée. (353114 triangles affichés ici)	Moteur 3D réalisé filtrant les objets n'étant pas dans le champs de vision quelle que soit l'orientation de la caméra. (18130 triangles affichés ici)

Cette optimisation permet de doubler le nombre de frames affichées par seconde.

Lors du chargement des modèles 3DS, nous avons fait le choix d'appliquer directement les transformations sur les points. Cela permet de gagner en performance puisqu'il y a moins de matrices à passer à OpenGL.

IV.3. Amélioration possible

Shaders

Une possible utilisation des shaders serait à envisager. En effet, nous avons ajouté au projet les classes nécessaires au chargement, la compilation et à l'utilisation de shaders. Ceci laisse place à une potentielle amélioration du rendu de la scène telle que :

- Modèle d'illumination géré par les shaders ;

- Modèle d'ombrage ;

- Cell-Shading ;

- Applications d'effets graphiques haute-qualité.

Système de particules

L'intégration d'un système de particules dans le projet serait une grande amélioration. En effet nous pourrions par exemple afficher des effets graphiques utilisant ce système pour :

- Afficher une animation lorsque l'avatar meurt ;

- Faire sortir de la fumée de la cheminée tant que l'avatar ne possède pas assez de diamants ;

- Améliorer les effets graphiques liés à la transformation des monstres ;

- Améliorer la qualité des animations de déplacements.

Gestion de la lumière dans le graphe de scène

Une idée que nous avons eue mais n'avons pas eue le temps de concrétiser est la gestion des lumières de la scène par le graphe de scène. Il s'agirait d'un autre type de nœud, et ainsi nous pourrions gérer la position de cette lumière grâce aux noeuds et aux transformations. Cela permettrait par exemple de positionner une lumière sur le personnage principal comme si il avait une lampe de poche.

Gestion d'autres formats de fichiers que TGA et 3DS

Actuellement, notre système ne peut gérer que le format d'images TGA, et le format 3DS pour les fichiers 3D. La gestion d'autres formats pourrait aussi être un bon complément au projet actuel. Cela permettrait une plus grande flexibilité d'utilisation et enlèverait certaines limites de la version actuelle.

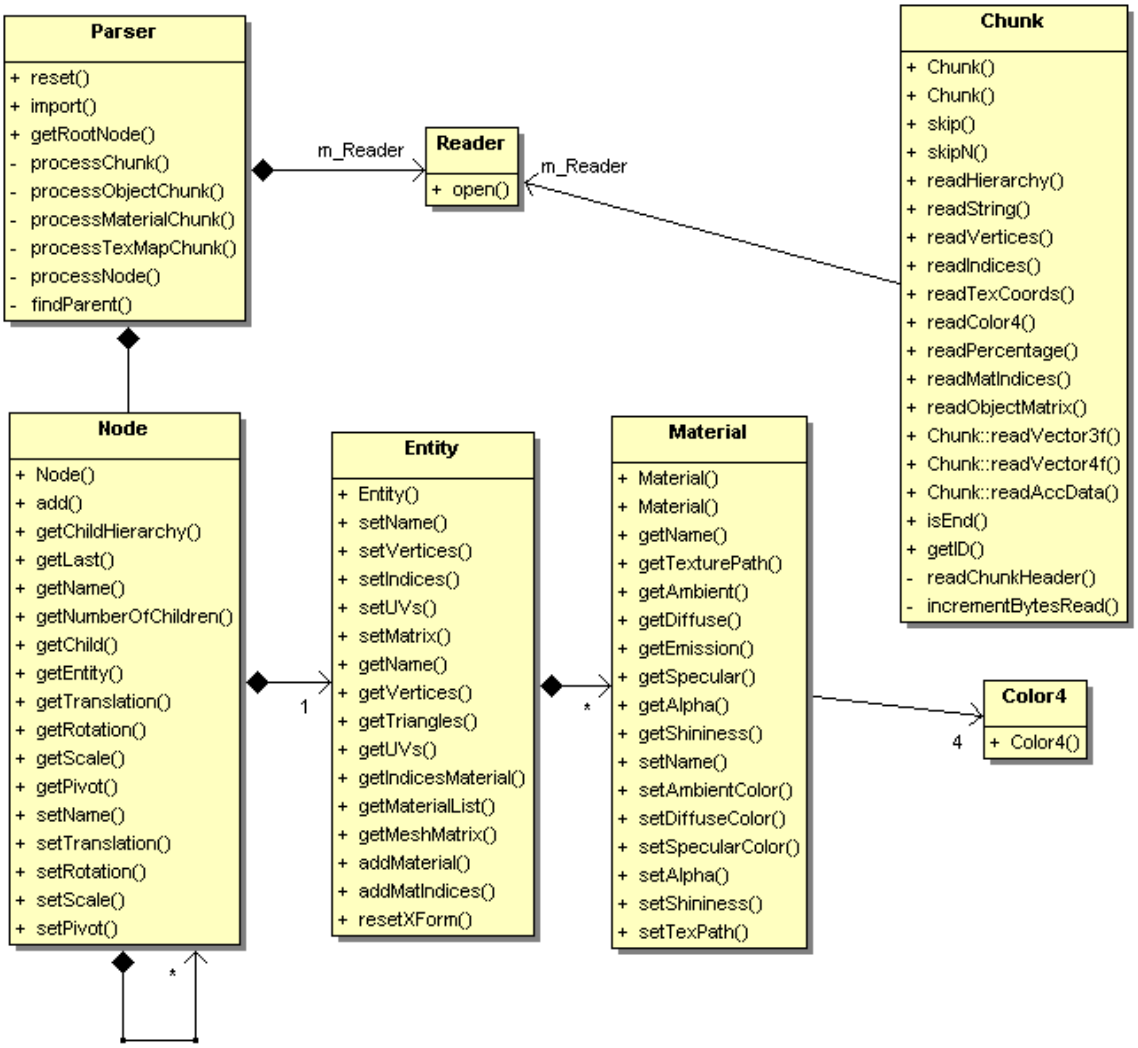
Conclusion

Ce projet a été très intéressant, et nous a notamment permis de mettre en application les connaissances acquises dans cette UV avec un projet concret. Cela a été très intéressant de mettre en relation toutes les connaissances apprises tout en réussissant à créer du code modulaire, performant et fiable.

Etant donné que nous étions six pour réaliser le projet, réussir à organiser le projet de façon à ce qu'il soit modulaire tout en répartissant les tâches entre les différentes personnes a été un vrai défi. Nous avons ainsi été amené à trouver de bons moyens pour communiquer et pour partager notre travail.

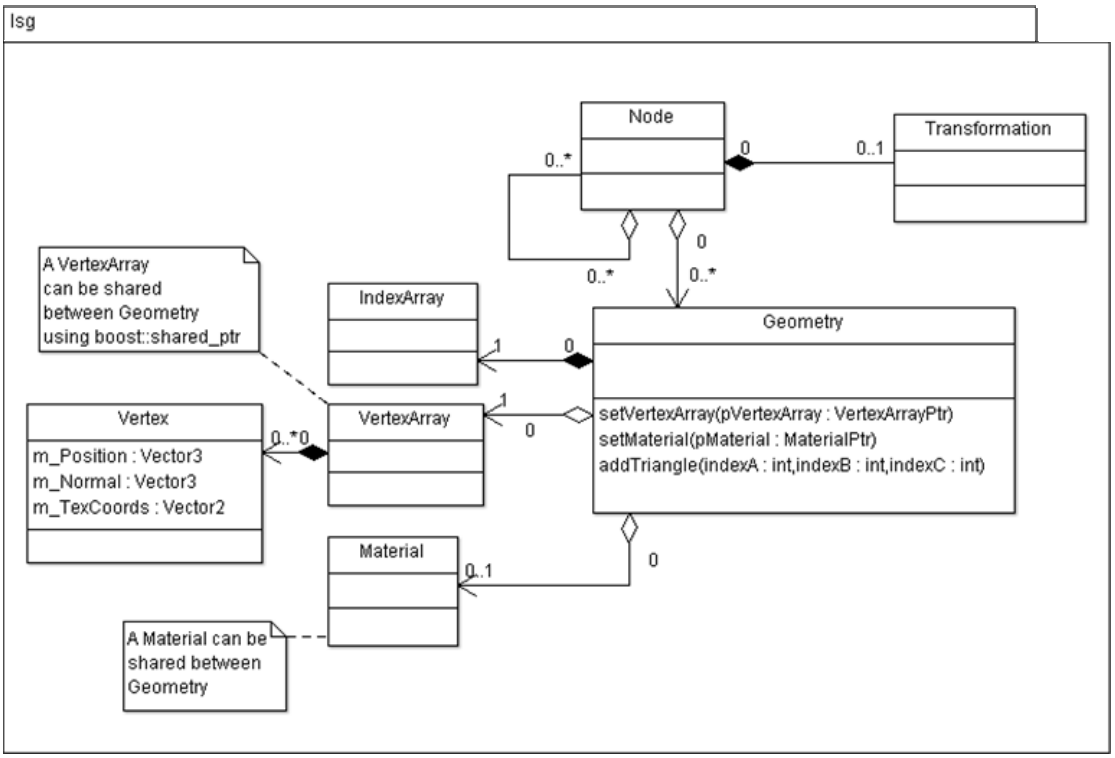
ANNEXE 1

Diagramme de classes de la librairie Importer3ds



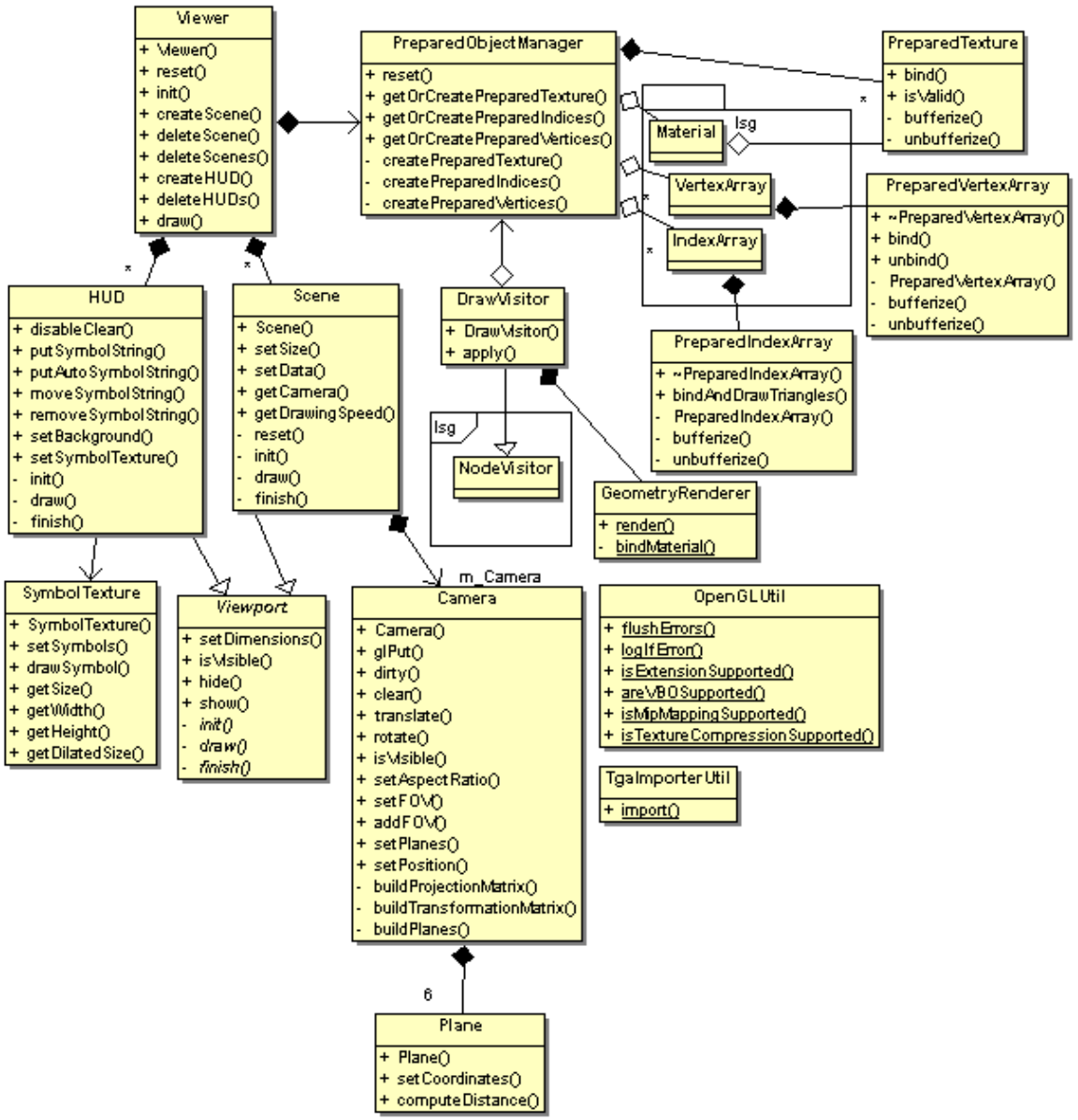
ANNEXE 2

Diagramme de classes de la librairie LSG



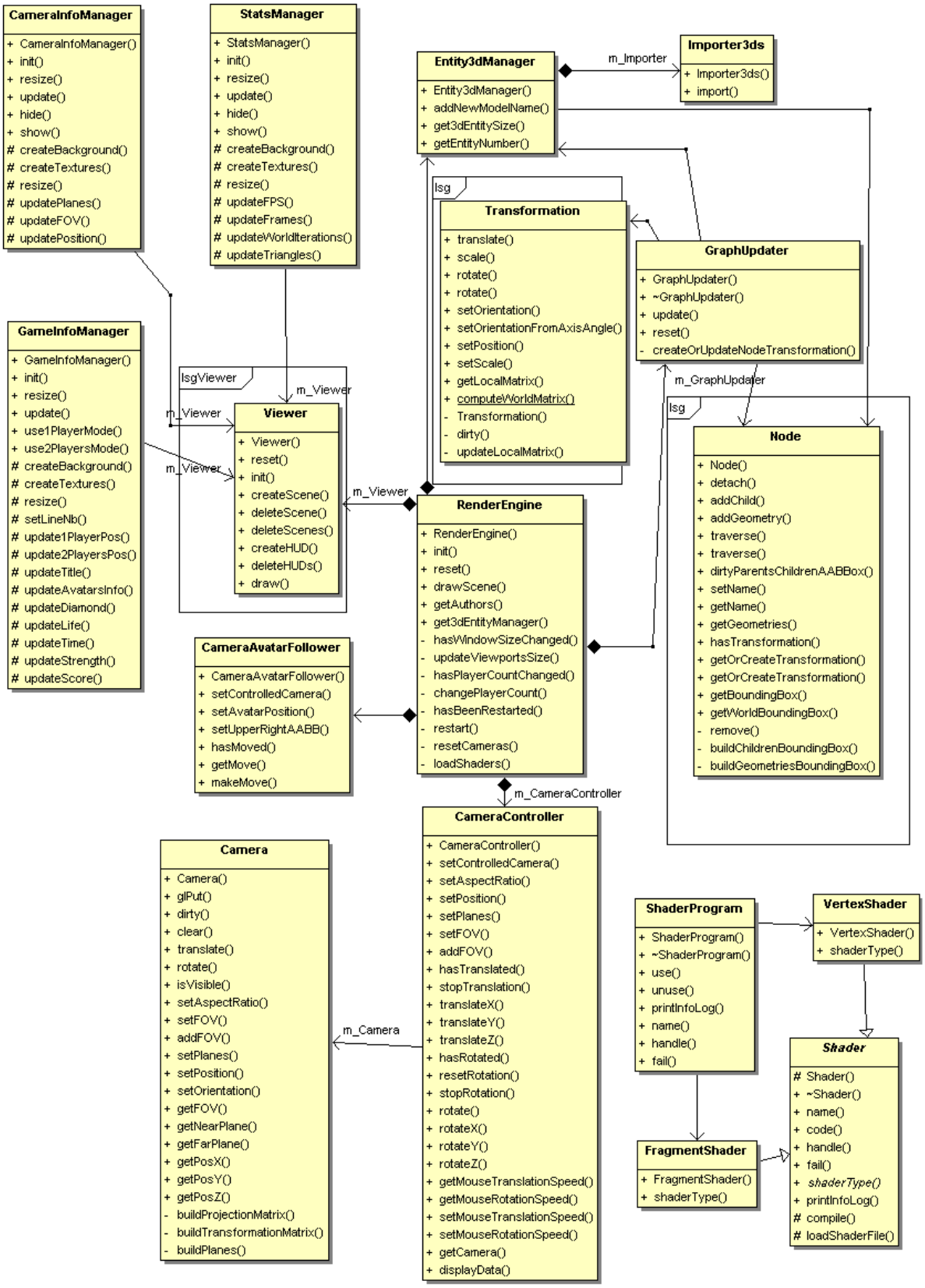
ANNEXE 3

Diagramme de classes de la librairie LSGViewer



ANNEXE 4

Diagramme de classes du moteur 3D



ANNEXE 4

Diagramme d'activité du moteur 3D