Projet de IN55
Un système de particules

Sibel BEDIR Johann NOVAK RABILLER Donatien

4 juin 2015

Introduction

Ce rapport a été réalisé dans le cadre de l'UV IN55 concernant un projet de rendu graphique 3D à réaliser à l'aide des librairies OpenGL sous l'environnement graphique QT. Parmi le sujets proposés, nous avons choisi le système de particule car c'est un thème très intéressant. Nous ne nous rendons pas forcément compte mais beaucoup de phénomènes physiques nous entourant peuvent être assimilés à un système de particule, plus ou moins complexe. Que ce soit le feu, la fumée, les nuages ou même la poussière environnante, tout peut-être représenté par un tel système.

Ce projet nous a donné alors l'opportunité de créer un moteur 3D temps réel générique capable d'afficher n'importe quel objet affichable sous OpenGL avec des utilitaires robustes. Un soin particulier a été donné sur son architecture pour qu'il soit réutilisable pour n'importe quelle autre application. Nous avons décidé de partager ce rapport en trois parties. Tout d'abord nous aborderons le gros du sujet qui concerne l'architecture de notre moteur 3D. Ensuite dans une deuxième partie, nous présenterons deux scènes où nous décrirons pour chacun les différentes spécificités. Finalement dans une troisième partie, nous ferons un bilan du projet avec les performances du moteur, les difficultés rencontrées et les améliorations possibles.

Table des matières

1 L'architecture du moteur 3D **3**
- 1.1 Un concept de "Scène" . . . 3
  - 1.1.1 Définition . . . 3
  - 1.1.2 Diagramme de classe . . . 3
  - 1.1.3 Avantages . . . 3
- 1.2 Les systèmes de particules . . . 3
  - 1.2.1 Définition d'un point de vue physique . . . 3
  - 1.2.2 Diagramme de classe . . . 4
- 1.3 La boucle de simulation . . . 4
  - 1.3.1 Situation initiale . . . 5
  - 1.3.2 Mise à jour des objets . . . 5
  - 1.3.3 Rendering à l'écran . . . 6
- 1.4 Des utilitaires très pratiques . . . 6
  - 1.4.1 Un importateur d'objets créés sous 3ds Max . . . 6
  - 1.4.2 Une gestion générique des inputs utilisateurs . . . 6
  - 1.4.3 Un système de log avancé . . . 6

2 Rendu dans une scène **8**
- 2.1 Appartement Feng-Shui . . . 8
  - 2.1.1 Quelques impressions écrans . . . 8
  - 2.1.2 Objets présents dans la scène . . . 8
  - 2.1.3 Shaders utilisés . . . 8
- 2.2 Tempête de sable au Sahara . . . 8
  - 2.2.1 Quelques impressions écrans . . . 8
  - 2.2.2 Objets présents dans la scène . . . 8
  - 2.2.3 Shaders utilisés . . . 8

3 Bilan du projet **9**
- 3.1 Benchmark . . . 9
- 3.2 Difficultés rencontrées . . . 9
- 3.3 Améliorations possibles . . . 9

Annexes **10**

Chapitre 1

L'architecture du moteur 3D

Nous voulions faire un moteur 3D temps réel avec une **architecture générique**. De ce fait, il serait alors simple plus tard de le réutiliser pour une autre application 3D. Nous avons alors réfléchis sur la question : comment gérer les objets (et leur matrices projectives) dans un monde complexe? Le concept de "*Scène*" en est ressorti. Ensuite, la création du système de particule ne fut pas difficile une fois le concept compris. Finalement, pour lier le tout et faciliter à la fois le travail et les phases de debug, des utilitaires auxiliaires ont été créés.

1.1 Un concept de "Scène"

1.1.1 Définition

Une scène n'est rien de plus qu'un arbre composé d'un noeud racine[1] qui lui est composé de zéro à plusieurs autres noeuds, etc... Ces noeuds sont appelés dans notre projet des SceneNode[2]. Chaque SceneNode possède comme attribut sa position locale et ses matrices projectives locales. De plus, à un noeud est associé un Object3D[3] unique qui est dessiné lorsque le noeud est dessiné.

Évidemment, tous les objets à dessiner doivent être ajoutés à un noeud

1.1.2 Diagramme de classe

1.1.3 Avantages

1.2 Les systèmes de particules

1.2.1 Définition d'un point de vue physique

Un système de particule d'un point de vue physique peut être résumé à un ensemble de caractéristiques. Le premier ensemble est celui du système en lui-même, il doit posséder :

→ un **point de l'espace** à partir duquel émettre des particules;

1. On appellera dans le rapport le noeud racine, root node.
2. **SceneNode** : Noeud de scène
3. Nom de la classe représentant n'importe quel objet ayant une représentation 3D sous OpenGL.

→ le nombre de particule qu'il émettra au total dans sa vie. Ce dernier peut être égal à l'infini bien entendu.

Cependant les particules émises par ce systèmes sont un peu plus complexes, elles sont reliées par :

→ leur durée de vie (de leur émission à leur disparition de l'écran) appelé "Time To Live"[4] ou **TTL**;

→ le moment à laquelle elles ont été créées, appelé **spawn time**[5];

→ un vecteur vitesse à trois composantes $\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$ propre à chacune qui défini leur **trajectoire** tout au long de leur vie;

→ un vecteur à trois composantes $\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$ qui défini leur **accélération**;

→ un vecteur $\begin{pmatrix} r \\ g \\ b \end{pmatrix}$ qui défini sa **couleur**[6].

A partir de ces informations, coder le système de particule est simple. Mais alors comment le rendre générique ?

1.2.2 Diagramme de classe

FIG. 1.1 – *Diagramme de classe de l'architecture du moteur.*

Comme le schéma ci-dessus le montre, nous avons une classe mère abstraite *AbstractParticleSystem* qui s'occupe des matrices projectives et du shader à utiliser.

Remarque Nous avons choisi de pas implémenter 'Particule' en tant que classe mais en tant que structure car cette dernière n'est qu'au final un tableau encapsulé. Ainsi il est possible d'accéder à chaque attribut de la particule via l'opérateur '[]'. Ce choix nous est utile lorsque nous transmettons les attributs de la particule au shader, nous utilisons le paramètre de 'stride' de la fonction *glVertexAttribPointer* qui permet de préciser la taille à parcourir lors de chaque appel.

1.3 La boucle de simulation

Notre simulation possède quatre états :

1) la situation initial avant le lancement de la simulation

4. **Time To Live** : temps restant à vivre.
5. **Spawn time** : temps d'apparition.
6. **RGB** : **R**ed **G**reen **B**lue, Rouge Vert Bleu, trois composantes de couleur nécessaires pour dessiner un pixel à l'écran.

2) dès que la boucle de jeu est lancée, il faut mettre à jour tous les objets de la scène;

3) après mise à jour, il faut les dessiner à l'écran. (La simulation passe ainsi de l'état 2 à l'état 3 indéfiniment)

4) lorsque que la simulation est arrêtée, toutes les données sont détruites et les shaders sont déchargés.

FIG. 1.2 – *Diagramme de transition de la simulation.*

Nous avons utilisé le framework utilisé en TP, c'est-à-dire les classes *GLWindow* et *AbstractFramework* (cependant des légères modifications ont été apportées). C'est la classe Game qui s'occupe de gérer l'initialisation de tous les objets lors de la situation initial, et lorsque l'application est lancée, l'update et le draw. Le timer qu'utilise QT est paramétré à 16 millisecondes, ce qui rend un affichage qui tourne à 60 FPS[7] Nous ne parlerons pas de la destruction de notre programme car cela ne concerne pas la simulation en elle-même.

1.3.1 Situation initiale

Lorsque que la simulation est lancée, le framework s'occupe d'initialiser les librairies OpenGL tandis que notre classe principale elle se charge d'initialiser tous les objets de la scène : elle les créer, les paramètre si besoin est et les ajoute dans un nouveau SceneNode qui est ajouté à la scène. De plus elle charge les shader qui sont nécessaire au bon fonctionnement du programme. Les fichiers sont lus par pair[8], et sont testés pour voir s'ils ne contiennent aucune erreur (qui peut survenir du code shader lui-même ou lorsque le code est associé[9] avec le programme).

1.3.2 Mise à jour des objets

C'est pendant cette étape que sont lues les inputs utilisateurs, que la matrice Vue est mise à jour ainsi que tous les objets de la scène. Un paramètre est requis cependant pour mener à bien cette tâche, il est nécessaire d'avoir l'intervalle de temps qui sépare l'update courante avec la dernière update. Grâce à cette durée, la position de chaque objets de la scène est recalculée dynamiquement.

Concernant les systèmes de particules, la mise à jour sert principalement à incrémenter la durée depuis laquelle ils ont été créés. Ce qui permet alors de calculer pour chaque particule le temps qui leur reste à vivre (et s'il dépasse leur TTL, il faut les supprimer).

7. **FPS** : **F**rame **P**er **Second**, image par seconde.

8. Un shader est composé d'un 'Vertex shader' traitant les problèmes de positionnement et d'un 'Fragment shader' se chargeant de la couleur des pixels à afficher.

9. Ou aussi dit "linké".

1.3.3 Rendering à l'écran

Une fois que chaque objet à sa position et ses matrices projectives mis à jour, il faut les dessiner à l'écran. C'est lors de cette étape que sont activés les shaders à utiliser, que les variables des shaders sont transmises et que les vertices de chaque objets sont envoyés pour permettre leur affichage sur l'écran.

1.4 Des utilitaires très pratiques

Nous avons créé des utilitaires qui ne suivent pas la continuité du projet mais qui en revanche nous ont faciliter la réalisation de certaines tâches ou l'accès à certaines informations (utile pour les phases de debug).

1.4.1 Un importateur d'objets créés sous 3ds Max

1.4.2 Une gestion générique des inputs utilisateurs

Chaque librairie graphique permet, selon une certaine façon, de gérer des inputs utilisateurs. Nous avons alors pensé à réaliser un singleton *InputManager* qui resterait générique et qui permettrait aux autres classes du programme de connaître à n'importe quel moment quelles inputs sont en train d'être réalisées. Elle définit ses propre énumérations 'KeyID' et 'MouseButtonId', et des classes filles propres aux applications sont censées transformer les inputs propres aux applications (par exemple les QKeyEvent ou les QMouseEvent) en 'KeyId' et en 'MouseButtonId'.

FIG. 1.3 – *Diagramme de classe de l'InputManager.*

Remplacer Qt pour la SDL par exemple ne nécessite pas de changer le code lors de tests d'inputs utilisateurs. Il suffit juste de créer une classe qui hérite de *InputManager* pour transformer les événements d'input utilisateurs SDL dans nos propres énumérations.

1.4.3 Un système de log avancé

L'affichage de l'information est primordial pour à la fois connaître l'état des variables du programme mais également pour connaître s'il y a eu un bug où il est apparu. L'afficha en console reste limité ce qui nous a incité à créer notre propre système de log. Voici le diagramme de classe associé à ce système :

FIG. 1.4 – *Diagramme de classe du LogManager.*

Ce système est composé d'un *LogManager* qui possède une liste de *LogEventHandler* à qui il renvoie chaque *LogEvent* qui lui est envoyé. Les Handlers peuvent logger l'événement à condition :

→ qu'il ait la possibilité de logger un tel *LogEventType*;

→ qu'il ait la bonne priorité de log (*LogLevel*) dans ses attributs.

L'utilité de la classe *LogEventHandler* repose sur le fait qu'elle soit générique et qu'il est donc possible de créer autant de Handler que possible avec chacun leur spécificités. Nous avons créé par exemple un *ConsoleLogEventHandler* qui ne log que dans la console ou bien un *QTLogEventHandler* qui log dans un widget QT.

Chapitre 2

Rendu dans une scène

2.1 Appartement Feng-Shui

2.1.1 Quelques impressions écrans

2.1.2 Objets présents dans la scène

2.1.3 Shaders utilisés

2.2 Tempête de sable au Sahara

2.2.1 Quelques impressions écrans

2.2.2 Objets présents dans la scène

2.2.3 Shaders utilisés

Chapitre 3

Bilan du projet

3.1 Benchmark

3.2 Difficultés rencontrées

3.3 Améliorations possibles

Conclusion

Nous pouvons donc conclure que ce projet fut un tant soit peu difficile à réaliser notamment par le fait que nous voulions réaliser un moteur générique. Ce fut également une très bonne occasion pour manier les shader qui étaient alors inconnus pour certains avant l'UV IN55.
Nous avons également eu l'occasion de créer des utilitaires qui pourront nous resservir, que ce soit pour ,les inputs utilisateurs ou bien le système de log.

Annexes

Tableau d'inputs utilisateurs possibles

Entrée utilisateur	**Influence sur la simulation**
ECHAP	Arrêt de la simulation.
R	Repositionne la caméra à son emplacement initial.
Z, Flèche avant	Translation positive de la caméra sur l'axe des x.
S, Flèche arrière	Translation négative de la caméra sur l'axe des x.
D, Flèche droite	Translation positive de la caméra sur l'axe des y.
Q, Flèche gauche	Translation négative de la caméra sur l'axe des y.
E	Translation positive de la caméra sur l'axe des z.
A	Translation négative de la caméra sur l'axe des z.
P	Rotation positive autour de l'axe des z.
O	Rotation négative autour de l'axe des z.

Sources

→ Logiciel utilisé pour créer le projet : QtCreator
https://www.qt.io/download-open-source/
→ Logiciel utilisé pour écrire le rapport en LaTeX : ShareLatex
www.sharelatex.com
→ Logiciel utilisé pour faire de la modélisation 3D : 3ds Max
http://www.autodesk.fr/products/3ds-max/overview
→ Librairies utilisées pour le rendu graphique 3D : OpenGL
https://www.opengl.org/
→ Librairie d'aide pour calculs mathématiques : glm
http://glm.g-truc.net/0.9.6/index.html
→ Logiciel utilisé pour faire du versioning : Git
https://git-scm.com/