Master 1 – Image et 3 Dimensions
Projet de Programmation Avancée
Rapport Final

Maxime Hohl

5 mai 2020

Table des matières

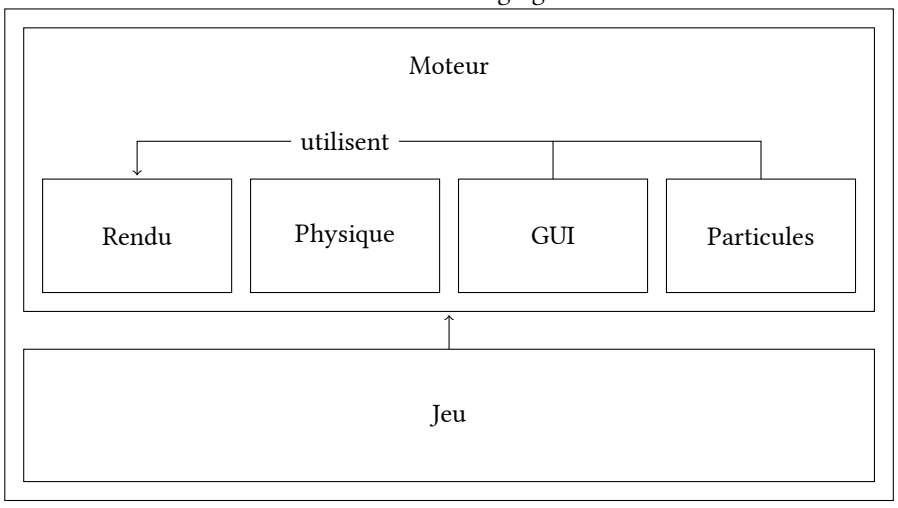
1. Design

Le jeu *Asteroids* est découpé en deux grandes parties, le moteur et le jeu. Le moteur est lui-même découpé en 3 modules :

- Le module de rendu : qui s'occupe de l'affichage de différents éléments à l'écran.
- Le module de physique : qui s'occupe des calculs physique, c'est à dire des déplacements et des collisions.
- Le module de GUI : qui s'occupe de l'affichage des éléments d'interface utilisateur.
- Le module de particules : Un système de particules qui ne fonctionne pas encore dans la version rendue, bien qu'il soit assez avancé d'un point de vue du développement.

Tout les éléments du jeu et du moteur interagissent ensemble grâce au *design pattern* Observateur. Voir la Figure 1.1.

Figure 1.1 – Design global



1.1 Le *design pattern* Observateur

Ce *design pattern* permet la communication entre les différents éléments du jeu. Il est composé d'éléments appelés *Observer* qui vont « recevoir » les messages envoyés par les éléments *Subject* qu'ils observent.

Cf. L'article Wikipedia pour plus d'informations sur le *design pattern* Observateur : `https://en.wikipedia.org/wiki/Observer_pattern`.

1.2 Le module de rendu

Le module de rendu s'occupe de l'affichage de tous les éléments à l'écran. Il est basé sur la bibliothèque *SDL2* mais est conçu pour pouvoir la remplacer très facilement par un autre système comme *OpenGL* ou *Vulkan*.

Il propose des fonctions de base permettant d'afficher tous les éléments simples suivants :

- Des Points
- Des Lignes
- Des Rectangles vides et pleins
- Des Polygones quelconques vides
- Des Cercles

Les modules de GUI et de particules se basent sur lui.

1.3 Le module de physique

Le module de physique s'occupe de tous les calculs physiques, c'est à dire, en l'état actuel des choses, des déplacements en utilisant accélération, vitesse et position ainsi que des collisions cercle-cercle, point-cercle et joueur-cercle, le joueur étant un polygone quelconque bien définis (Voir Figure 2.1).

1.4 Le module de GUI

Le module de GUI s'occupe de la gestion de tous les éléments d'interface graphique. En l'état actuel il permet de gérer :

- Des *Panels* : qui sont de simples rectangles.
- Des textes : qui, aussi surprenant que ça puisse paraitre, sont du texte.

Le module a été conçu pour permettre l'ajout de nouveaux éléments d'interface de manière assez facile.

1.5 Le module de système de particules

Ce module ne fonctionne pas encore, mais comme le gros est déjà développé et qu'il ressemble à quelque chose, il est présent dans le dossier **poc/particles** mais n'est pas encore utilisé dans le projet.

Il permet de gérer des particules de type point, avec une couleur spécifique.

1.6 Le jeu

Le jeu permet de gérer les astéroïdes au fur et à mesure du temps et en fonction du score du joueur. Le joueur gagne un point par astéroïde détruit.

Tous les éléments du jeu reposent sur les différents modules du moteur.

2. Implantation

L'implantation s'est faite de manière très directe à partir du design décrit précédemment. Le projet est programmé en *C++17* et repose sur la bibliothèque *SDL2* qui est en *C* ainsi que sur le système de compilation *CMake*. Les fichiers sont gérés grâce au système de contrôle de version *git* et à *GitLab* (`https://git.unistra.fr/m.hohl/projet-progav-maxime-hohl`).

2.1 Structure du projet

Le dossier du projet est découpé en plusieurs sous dossiers :

assets Contient les différents éléments graphiques et audio, c'est à dire uniquement l'icône du jeu dans le cas présent.

documents Contient le présent rapport ainsi que le sujet du projet

external Contient les bibliothèques externes essentielles au projet, uniquement *SDL2* actuellement.

include Contient tous les fichiers *headers* du projet.

poc Comme *Proof Of Concept*, contient des modules, morceaux de code ou expérimentations plus ou moins aboutis qui ne sont pas encore intégrés au projet.

src Contient tous les fichiers *source* du projet.

Les dossiers **include** et **src** sont eux-mêmes re-découpés en deux sous dossiers **Engine** et **Game** contenant respectivement les fichiers du moteur et ceux spécifiques au jeu.

2.2 Le *design pattern* Observateur

Toutes les classes qui veulent échanger des données doivent hériter de la classe `Subject` quand elles transmettent des données ou de la classe `Observer` quand elles reçoivent des données. Ces deux classes on comme paramètre de template le type de donné échangé.

2.3 Le module de rendu

Le module de rendu est constitué d'une classe appelée `Renderer` qui se base sur *SDL2* mais qui permet de changer aisément pour tout autre système en arrière-plan, comme *OpenGL*, *Vulkan*, *Metal*, etc.

2.4 Le module de physique

Le module de physique est axé sur deux classes principales, `PhysicEngine` et `PhysicEntity`. La première effectue tous les calculs liés à la physique et tous les objets qui doivent être inclus dans les calculs physiques doivent hériter de la seconde.

Le calcul des mouvements des objets se basent sur les trois formules simples suivantes :

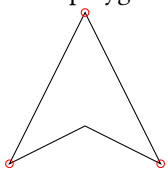
$$a_t = a_0 \tag{2.1}$$
$$v_t = v_{t-\Delta t} + a_t * \Delta t \tag{2.2}$$
$$r_t = r_{t-\Delta t} + v_t * \Delta t \tag{2.3}$$

Avec a l'accélération, v la vitesse et r la position, a_0, v_0 et r_0, respectivement l'accélération initiale, la vitesse initiale et la position initiale et Δt le temps écoulé depuis le dernier rafraichissement du système de physique.

Le système de collision ne permet actuellement que de détecter des collisions du type cercle-cercle, cercle-point et cercle-joueur.

Les collisions cercles/joueur sont gérés de manière assez particulières. Le joueur est un polygone concave quelconque (voir Figure 2.1), il faudrait donc utiliser un algorithme d'intersection entre cercle et polygone concave, mais ces algorithmes sont assez compliqués et complexes, et en observant bien la forme du polygone joueur et la différence de taille entre le joueur et les cercles (ici, les astéroïdes), on se rend compte que l'on peut tout simplement effectuer un test de collision entre le cercle et les trois points « extérieurs » (les points cerclés de rouge sur la Figure 2.1) du polygone et si un point est dans un cercle alors il y a collision entre cercle et joueur.

Figure 2.1 – Le polygone du joueur



2.5 Le module de GUI

Le module de GUI tourne autour des classes :

— `gui::Gui` qui est la classe de base de gestion des éléments d'interface. C'est elle qui gère le rendu et qui doit être utilisée pour créer tous les éléments d'interface.

— `gui::Entity` qui est la classe virtuelle pure dont tous les éléments d'interface doivent hériter.

Chaque entité est positionnée relativement à son parent et à son ancrage. Les entités sans parent sont placées par rapport à la fenêtre, toujours en fonction de l'ancrage (voir Figure 2.2).

Chaque entité va se dessiner en appelant des fonctions de la classe `GUIRenderer` puis appeler la fonction de rendu de ses enfants. Donc si l'on se représente les éléments d'interface via un arbre grâce à leur lien de parenté, le rendu se fait dans un ordre de

FIGURE 2.2 – Le positionnement des éléments d'interface. Chaque élément est représenté avec son point d'ancrage. Les éléments vert et rouge ont pour parent l'élément blue, et ce dernier a pour parent la fenêtre (en noir).

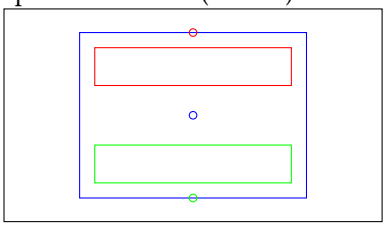
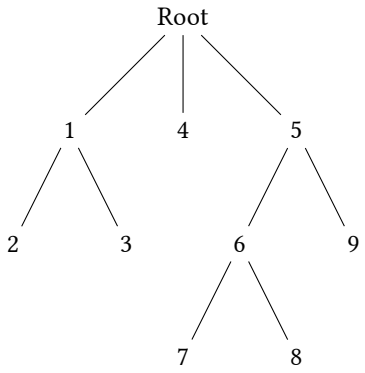


FIGURE 2.3 – Ordre de rendu des éléments graphiques (le numéro représentant l'ordre d'affichage)



parcoure en profondeur, ce qui permet de garantir que les parents sont toujours rendus avant leurs enfants (et sont donc affichés derrière). Voir Figure 2.3.

Ce module est le plus aboutit de tous, il ne dispose pas de beaucoup d'éléments différents, mais il est conçu de tel sorte qu'en ajouter un est extrêmement aisé. Il suffit de créer une classe qui hérite de `gui::Entity`

2.6 Le module de système de particules

Pour ce module je me suis fortement inspiré de système proposé sur la page suivante : `http://archive.gamedev.net/archive/reference/articles/article1982.html`. L'idée est de définir les particules et leur mise à jour grâce à des classes passées via les templates.

Il y a une classe `ParticleEmitter` qui permet d'émettre des particules et qui est paramétrée avec des classes héritants de `InitializationPolicy`, `UpdatePolicy` et `Particle`. Les classes `BaseXXXXXX` sont des implantations simples de ces classes de base et sont utilisées comme paramètre par défaut de `ParticleEmitter`.

En l'état actuel, le module n'est pas viable, les temps de calculs étant très aléatoires, allant d'un temps raisonnable à plusieurs secondes. D'autres problèmes semblent exister, mais faute de temps le module a été délaissé (bien que je compte le reprendre à titre personnel en me basant peut-être sur les *Compute Shader* d'*OpenGL*).

2.7 Le jeu

Le jeu en lui-même est axé autour des trois classes suivantes :

- `Game` : qui gère toute la logique du jeu, c'est-à-dire la boucle principale, les appels de rendu, la gestion des évènements la création des différents éléments du jeu, etc.
- `Player` : qui gère toute la logique derrière le joueur, c'est à dire ses déplacements, et ses tirs.
- `Asteroids` : qui gère tous les astéroïdes, de leur création au cours du temps et en fonction du score du joueur à leur destruction s'ils sont touchés par des tirs du joueur.

2.8 Optimisations

Toutes les fonctions sont conçues de la manière la plus optimisée possible, mais aucune passe d'optimisation n'a été effectuée. Cela n'était pas nécessaire, étant donné que le programme tourne à 77 images par seconde constantes sur *Debian 10* avec *KDE* et *X11* et à 60 images par seconde constantes sur *Windows 10*, les deux avec le système suivant :

- Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz, 6 coeurs
- 32Go de RAM DDR4
- NVIDIA GeForce GTX 1060 6GB

On remarquera cependant que les valeurs d'images par secondes ne sont sans doutes pas représentatives car étant donné les valeurs, elles doivent être limitées par une synchronisation verticale. Mais elles permettent quand même de se rendre compte qu'il n'y a pas de ralentissement dans les temps de rafraichissements et donc pas de forte nécessité d'optimisation.

3. Problèmes rencontrés et voies d'amélioration

3.1 Problèmes rencontrés

Au cour du développement du projet, de nombreux problèmes ont fait surface. Par exemple, le développement a commencé en considérant des astéroïdes de formes quelconques, mais après avoir eu beaucoup de difficultés (et pas assez de temps pour les surmonter) pour gérer les collions entre eux, ils sont devenus de simples cercles. Le module de particules m'a aussi donné pas mal de fil à retordre.

3.2 Bogues connus

En l'état actuel, le projet présente trois bogues connues :

- Le jeu « crash » quand un astéroïde avec une vitesse de $[0, 0]$ entre en collision avec le tir d'un joueur. Ce bogue n'est pas urgent à corriger étant donné que les astéroïdes ont une vitesse minimale supérieure à 0.
- Le jeu « crash » lors de la collision entre un astéroïde et un tir de joueur quand le projet est compilé avec *MSVC*.
- Il arrive que des astéroïdes acquièrent une vitesse beaucoup trop élevée, ce qui les rends quasiment impossible à éviter.

3.3 Voies d'amélioration

J'ai de très nombreuses idées de voies d'amélioration pour le projet, donc je ne détaillerai que les quatre plus importantes.

La première concernant le module de physique. Il devrait gérer de manière générique toutes les entités et non pas spécifiquement les astéroïdes, tirs et joueurs et devrait permettre de gérer les collisions entre polygones convexes, ce qui permettrait des astéroïdes avec des formes plus folklorique.

La seconde concerne le visuel. Un module de particules fonctionnel devrait permettre d'ajouter du dynamisme au jeu, par exemple en ajoutant des particules à la sortie du moteur du vaisseau du joueur ou lors de l'explosion d'un astéroïde. On peut aussi envisager de passer sur du rendu purement *OpenGL* pour permettre plus de flexibilité sur les effets visuels, comme des distorsions ou des tremblements de l'écran lors d'une explosion.

La troisième voie d'amélioration que j'ai envisagée est l'audio. L'ajout de musiques et d'effets sonores devrait ajouter un peu de vie dans le jeu.

Enfin, un vrai module de gestion des évènements permettrait de rentre le moteur réutilisable.

Conclusion

Le projet était très intéressant, développer un jeu de zéro (en dehors de *SDL2*) a été très instructif pour moi. J'ai axé le développement du projet plus sur le moteur que ce soit en utilisant la puissance du *C++17* ou en essayant de bien le concevoir que sur le jeu en lui même. C'est pourquoi je me retrouve avec un des modules que je trouve très aboutis, comme le module de GUI et d'autres bien pensés comme le module de Particules ou le *design pattern* Observateur (bien que les idées de bases ne soit pas de moi).

Si je devais exprimer une seule frustration ce serait de ne pas avoir pu passer plus de temps sur ce projet. Mais je ne me priverais pas d'améliorer le moteur de jeu en suivant les quelques idées que j'ai cité à la section 3.3.