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

Maxime Hohl

5 mai 2020

Table des matières

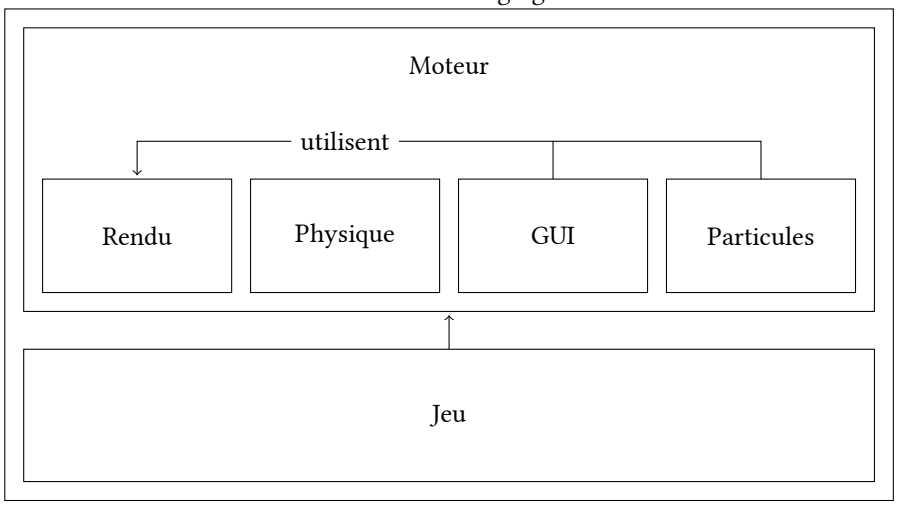
1. Design

Le jeu *Asteroids* est découpé en deux grandes parties, le moteur et le jeu lui même. 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 tout 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 tout les éléments simples suivants :

— Des Points

— Des Lignes

— Des Rectangles vides et pleins

— Des polygones quelconques vide

— Des Cercle

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

1.3 Le module de physique

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

1.4 Le module de GUI

Le module de GUI s'occupe de la gestion de tout 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 à était 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, je l'ai quand même ajouté au projet, bien qu'il ne soit actuellement pas utilisé.

Il permet de gérer des particules de type point, avec un 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.

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

2. Implantation

L'implantation se 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*.

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 essentiels au projet, uniquement *SDL2* actuellement.

include Contient tout les fichiers *headers* du projet.

src Contient tout les fichiers *source* du projet.

Les dossiers **include** et **src** sont eu-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 *class* qui veulent échanger des donnés doivent hériter de la *class* `Subject` quand elles transmettent des donnés ou de la *class* `Observer` quand elles reçoivent des donnés. Ces deux *class* on comme paramètre de template le type de donné échangé.

2.3 Le module de rendu

Le module de rendu est constitué d'une *class* 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 *class* principales, `PhysicEngine` et `PhysicEntity`. La première effectue tout les calculs liés à la physique et tous les objets qui doivent êtres inclus dans les calculs physiques doivent hériter de la seconde.

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

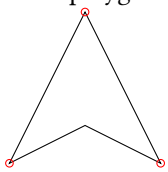
$$a_t = a_0 \tag{2.1}$$
$$v_t = v_{t-\Delta t} + a_t * \Delta t \tag{2.2}$$
$$r_t = r_{t-\Delta t} + v_t * \Delta t \tag{2.3}$$

Avec a l'accélération, v la vitesse et r la position, a_0, v_0 et r_0, respectivement l'accélération initial, la vitesse initial et la position initial 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 do 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 *class* :

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

— `gui::Entity` qui est la *class* virtuelle pure dont tout 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 *class* `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 parcoure en profondeur, ce qui permet de garantir que les parents sont toujours rendus avant leur 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 conçu de tel sorte qu'en ajouter un est extrêmement aisé. Il suffit de créer une *class* qui hérite de `gui::Entity`

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 à pour parent la fenêtre (en noir).

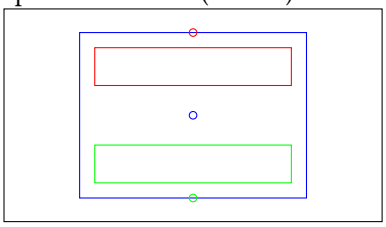
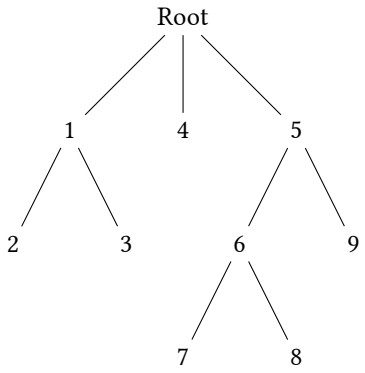


FIGURE 2.3 – Ordre de rendu des éléments graphiques (le numéro représentant l'ordre d'affichage)



2.6 Le module de système de particules

A faire

2.7 Le jeu

Le jeu en lui même est axé autour des trois *class* 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 créations 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 tout les astéroïdes, de leur création au cours du temps et en fonction du score du joueur à leur destruction si ils sont touchés par des tirs du joueur.

3. Problèmes rencontrés et voies d'amélioration

3.1 Problèmes rencontrés

A faire

3.2 Bogues connus

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

- Le jeu « crash » quand un astéroïde avec un 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 minimal supérieur 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 impossibles à éviter.

3.3 Voies d'amélioration

J'ai de très nombreuses (trop nombreuses ?) idées de voies d'amélioration pour le projet.

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 folkloriques.

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 peut de vie dans le jeu.

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