SCHNEBERGER Maxime – M1 CMI Image

Projet Arkanoïd

Schneberger Maxime & Roth Nathan

Durant le développement de ce projet Arkanoïd en C++, nous nous sommes répartis les parties de l'implémentation de la manière suivante :

Roth Nathan :

- Architecture générale (dépendance des classes, héritage, etc....)
- Réalisation des fonction de « dessins » (assigner une texture à un objet, prendre une certaine partie des fichiers bmp pour rogner l'image, etc.…)
    - Sprite statique (sans animation)
    - Sprite dynamique (avec animation)
- Gestion des niveaux du jeu (transition, affichage, fenêtre win/lose, menu)
- Gestion du rendu

Schneberger Maxime :

- Mise en place du gameplay général :
    - Mouvement du vaus, de la balle
    - Gestion des collisions entre la balle et le vaus, ainsi que les briques, ou encore entre les bonus et la vaus
    - Système de bonus (tout les bonus sont présents et utilisables en « cohabitation », ils respectent le principe de 1 seul powerup à la fois)
    - Gestion des briques cassables/incassables
    - Variables liées au joueur : vie, score, bonus actif, niveau en cours
- Réalisation des niveaux par fichiers textes
- Implémentation des différents mouvements (balle, vaus, bonus) en coordonnées réelles et affichage à l'écran en coordonnées entières

Nous nous sommes assez bien répartis la charge de travail tout le long du projet, ce qui nous a permis d'implémenter ce qui était requis dans les temps. Nous communiquions très régulièrement pour nous assurer de l'avancement respectif de chacun d'entre nous ainsi que pour avoir l'assurance de développer ce qu'il fallait à l'instant t.

Pour rentrer un peu plus dans les détails techniques, par rapport tout d'abord au parsing des niveaux (plus de détails dans le rapport de Nathan), nous avons voulu avoir un fichier texte de ce type :

```
//LevelX.txt
<width> <height> //en nombre de lignes + colonnes
<plateau cylindrique ? 0/1 >
<id du background>
<id des briques> //width fois dans ce sens
.
.
<id des briques>
// height fois dans ce sens => rectangle de width * height d'id pour les briques
```

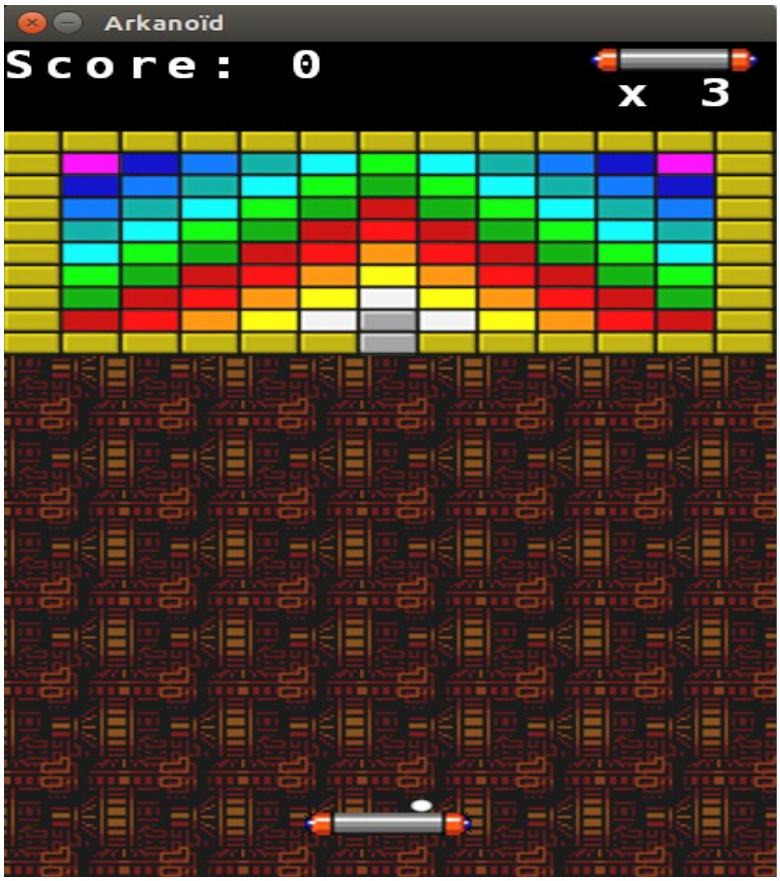
Nous avons borné width à 13 et height à 25 pour éviter que la taille de la fenêtre ne soit excessive,

mais il est tout de même possible de réduire la taille jusqu'à 1 * 1

Exemple d'un fichier texte de niveau :

```
13 10
0
3
14  14  14  14  14  14  14  14  14  14  14  14  14
14  9   5   8   12  3   4   3   12  8   5   9   14
14  5   8   12  3   4   6   4   3   12  8   5   14
14  8   12  3   4   6   11  6   4   3   12  8   14
14  12  3   4   6   11  7   11  6   4   3   12  14
14  3   4   6   11  7   2   7   11  6   4   3   14
14  4   6   11  7   2   10  2   7   11  6   4   14
14  6   11  7   2   10  1   10  2   7   11  6   14
14  11  7   2   10  1   13  1   10  2   7   11  14
14  14  14  14  14  14  13  14  14  14  14  14  14
```

Qui donne à l'écran :



Concernant les déplacements à l'écran, comme énoncé précédemment, les déplacement de la balle se font en coordonnées réelles et sont convertit en coordonnées entières au moment de l'affichage. La balle conserve un attribut de « fpos » à savoir position en flottant, qui est incrémenté/décrémenté au niveau des fonctions de déplacement.

Afin d'assurer la direction de la balle ainsi que sa vitesse, j'ai associé un vecteur de force à la balle ainsi qu'une variable de vitesse, le vecteur de force indiqué la direction tel un vecteur normé

auquel on multiplie la variable de vitesse (speed). Cette dernière pouvant donc être incrémenté à chaque tape sur la vaus pour accélérer progressivement la balle tout au long d'un niveau, et pouvant donc être aussi réduite dans le cas de l'obtention d'un bonus S (slow).

Les collisions sont gérées avec des box collider (xmin/max, ymin/max) et on peut donc simplement faire -1*force en x ou en y suivant le point de contact pour inverser simplement la direction de la balle sans avoir à recalculer la direction.

Le déplacement du vaus est géré par un entier direction, qui lorsqu'il est égal à -1 fait bouger le vaus à gauche et inversement pour 1. Le déplacement se fait dans la boucle Gameloop pour éviter de dépendre du « stutter » sur les événement clavier successifs. (Si l'on reste appuyé sur une touche de clavier, on a d'abord une impulsion, puis un temps de silence puis des imulsions régulières successives, ce qui produit une sensation de « stutter » : saccadé). On évite donc cela en appliquant un déplacement régulier et en changeant juste la valeur de cet entier direction en fonction du dernier input, ce qui produit un déplacement naturel et non saccadé.

La gestion des briques cassables/incassables et simplement géré par un vecteur de coup restants pour détruire la brique, donc pour les briques cassables, cet valeur est égalé à 2 + (int) ((level-1)/8) pour que l'on passe à 3 coups à partir du level 9, etc.…. Les briques incassables quant à elle, sont initialisées à -1 et passe donc par une condition sur de la collision pour ne faire jouer que l'animation de la brique sans plus.

Le score est géré selon l'id de chaque brique (les id étant répertoriés dans les fichiers texte, voir page précédente) comme l’exige les règles de base.

La gestion des vies est également assez simple : Toute balle touchant le bord inférieur de l'écran fait décrémenter le compteur de vies de 1 et fait reset la position du vaus et de la balle (fonction resetPosition()). Lorsque le joueur n'a plus de vie, un écran de transition apparaît pour faire revenir le joueur soit au menu soit le faire quitter le jeu.

Dans le cas d'un bonus « divide » actif, tant qu'il y a au moins une balle sur le terrain, alors les balles touchant le bord inférieur n’entraînent ni la perte d'une vie ni un reset (trivialement), elle sont simplement supprimé du jeu. Une fois 2 balles sur 3 perdu le bonus s'efface et donc retour au conditions normales de perte de vie.

Pour finir sur la partie technique, il nous reste à parler des bonus :
Le joueur transporte des variables liées au bonus sous forme de booléens (expand, divide, laser, magnet et slow) qui sont initialisées à false sauf pour magnet en début de level (pour que la balle colle au vaus au début du niveau) et une fois la balle lancée, celui-ci passe aussi à false. Par contre lorsqu'un bonus est touché par la vaus, on passe dans une fonction setBonus, qui assigne la valeur du bonus à true pour le joueur et met à false tout les autres, pour que l'on supprime les effets d'un éventuellement ancien bonus actif. Les effets des bonus sont ensuite gérés au cas par cas (slow => réduction de la variable speed ; magnet => met force à 0,0 quand la balle touche le vaus ; laser => fait spawn des mini-balles au clic sur les bords du vaus qui se désintègre au contact d'une brique ou d'un mur ; divide : crée 2 balles et les ajoutent dans le vector m_balls du jeu (game.cpp) avec des force différentes de la balle originale ; expand => change l'affichage du vaus en celui du vaus en dessous (dans la spritesheet))

Pour conclure, je dirais que ce projet a été sympathique à réaliser, on se prend très vite au développement d'un jeu comme celui-ci même si le temps à y consacrer peut être important. Je n'avais que les bases fondamentales du C++ au début de ce semestre, je me sens désormais plus apte à développer dans ce langage.