INFO-F202 - Langages de Programmation 2 - Rapport de Projet

El Muhur Nabil
000590273

Laghmouchi Yassir
000594326

5 Janvier 2025

Contents

1 Introduction

1.1 Objectif

L'objectif de ce projet de Langages de Programmation 2 était de coder un jeu casse-briques inspiré d'Arkanoid en faisant usage d'éléments de programmation orientée objet. Ce jeu devait posséder une multitude de fonctionnalités et d'éléments qui rendent sa jouabilité plus agréable qu'un simple prototype (score, vies, niveaux, etc...). Pour ce faire, nous avons eu recours à diverses ressources détaillées dans le point suivant.

1.2 Moyens et Pré-requis

Le langage imposé pour ce projet est le C++. La compréhension des sections suivantes nécessite donc des connaissances de base en C++ et de ses spécificités. Nous devions aussi faire usage de la librairie **Allegro**, qui dispose de méthodes offrant une multitude de possibilités (affichage d'une fenêtre, création d'un *timer*, etc...); et de sa documentation officielle. Nous nous sommes également basés sur le wiki non-officiel d'Arkanoid pour élaborer les détails du fonctionnement du jeu. Ayant une expérience acquise lors de précédents projets de programmation, notamment dans le domaine des jeux, nous espérons alors qu'elle se reflète dans la qualité de ce travail.

2 Structure du Projet

Nous détaillerons par la suite les rouages du projet, en commençant par les tâches imposées.

2.1 Tâches

Chaque tâche représente une fonctionnalité du jeu. L'intégralité des tâches imposées fut implémentée. C'est-à-dire:

- La raquette du joueur
- Des briques de différentes couleurs et bonus de points
- Des briques argentées et dorées
- Un score et un meilleur score
- Les 3 vies du joueur
- Un système de victoire / défaite
- Plusieurs niveaux avec possibilité d'avancer ou reculer d'un niveau
- La possibilité de contrôler la raquette à la souris / au trackpad
- Des bonus (laser, agrandir, ralentissement, etc...)

Une fois que les bases du code furent posées, le reste du travail fut grandement facilité. Les sections qui suivent apporteront plus de précision sur ces tâches, mais aussi sur la façon dont nous avons tiré parti de la programmation orienté objet pour assurer la modularité de notre code.

2.2 Classes

Le projet se divise en plusieurs classes, chacune étant catégorisée en fonction de son rôle. Voici donc un **diagramme UML** qui illustre bien la structure du projet. L'utilisation de l'héritage permet une cohérence dans la logique de programmation et une modularité du code qui ouvre la porte à un grand nombre de possibilités.

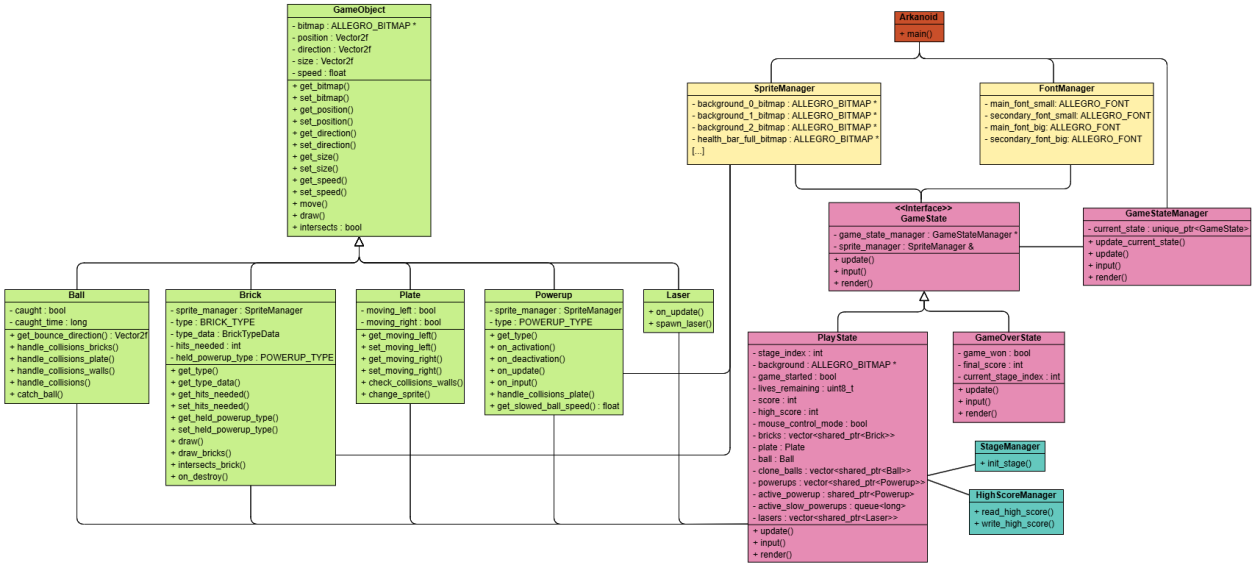

Figure 1: Diagramme UML du projet

2.2.1 Gestion des Ressources

Les classes `SpriteManager` et `FontManager` servent toutes deux à initialiser les ressources utilisées par le programme pour afficher des éléments à l'écran. Elles possèdent chacune des attributs de classe `ALLEGRO_BITMAP *` ou `ALLEGRO_FONT` qui restent inchangés tout le long de l'exécution du programme et qui nécessitent d'être chargés qu'une fois par **Allegro** au lancement de celui-ci afin d'éviter de surcharger la mémoire.

2.2.1.1 SpriteManager

Cette classe charge tous les sprites présents dans le dossier `res/sprites/` à l'aide de la fonction `al_load_bitmap()` d'**Allegro**. Une instance `SpriteManager` est créée une unique fois avant le début de la boucle du programme dans le fichier `main` `Arkanoid` et est utilisée comme paramètre pour les classes qui nécessitent d'avoir accès aux images du jeu. Parmis cesdites classes on retrouve `GameState`, et certaines classes enfants de `GameObject`. *Sprite* est le terme anglais employé dans ce domaine pour désigner les images 2D qui constituent les ressources d'un jeu.

2.2.1.2 FontManager

Cette classe charge toutes les polices d'écriture présentes dans le dossier `res/fonts/` à l'aide de la fonction `al_load_font()` d'**Allegro**. De la même manière que pour `SpriteManager`, on crée qu'une seule instance de cette classe avant la boucle principale. Elle est utilisée uniquement comme paramètre de la fonction `render` de `GameState`.

2.2.2 Objets

2.2.2.1 GameObject

Chaque élément dynamique, c'est-à-dire tout élément susceptible de se déplacer, d'apparaître, de disparaître ou de se multiplier, est représenté par la classe parent `GameObject` (c.f. diagramme UML). Cette classe encapsule tous les attributs essentiels d'un élément de jeu, notamment sa position, sa taille, sa vitesse et sa direction. De plus, elle fournit une méthode, `intersects()`, pour vérifier si cet élément entre en collision avec un autre. Cette méthode utilise l'algorithme d'intersection des **AABB** (*Axis-Aligned Bounding Boxes*), qui modélise les éléments sous forme de rectangles définis par leur position et leur taille. Si ces rectangles se superposent, il y a collision, sinon, il n'y en a pas. Elle fournit également une fonction `move()` qui utilise l'équation `position += vitesse * direction` pour calculer le mouvement de l'objet et une fonction `draw()` qui dessine le *sprite* de l'élément à l'écran à l'aide de la fonction `al_draw_bitmap()` d'**Allegro**.

2.2.2.2 Plate

Enfant de `GameObject`. Cette classe représente la raquette du joueur, elle ajoute deux attributs `moving_left` et `moving_right` qui facilitent la gestion des *inputs* du joueur, si les deux valeurs sont `false`, alors la direction de la raquette est nulle, autrement elle correspond à la direction de mouvement. Sa fonction `check_collisions_walls()` vérifie les potentielles intersections de la raquette avec les bords de l'arène et sa fonction `change_sprite()` qui nous sert à changer le *sprite* de la raquette dans le cas où le *power-up* d'agrandissement est récupéré par le joueur.

2.2.2.3 Ball

Enfant de `GameObject`. Les deux seuls attributs spécifiques à cette classe sont `caught` et `caught_time` qui indiquent si la balle a été attrapée par la raquette et à quel instant en millisecondes lorsque le *power-up* d'attrapage est actif. Nous avons décidé d'y placer les fonctions servant à vérifier les collisions de la balle, par souci d'organisation. Chacune de ces fonctions vérifie les collisions entre la balle et un élément spécifique tel que les briques, les murs ou la raquette. Une fonction `handle_collisions()` réunit et appelle les trois. Il serait moins propre de devoir appeler les trois pour vérifier les collisions de la balle principale ET celles des balles clones provoquées par le *power-up* d'interruption. Enfin, la fonction `get_bounce_direction()` calcule la direction de rebondissement de la balle en fonction de la direction incidente et de la normale de l'intersection à l'aide de l'équation de réflexion.

2.2.2.4 Brick

Enfant de `GameObject`. Chaque brique a un certain type qui possède des caractéristiques. Pour cela nous avons écrit une classe **enum** `BRICK_TYPE` et un **struct** `BrickTypeData` qui contient sa couleur `color`, le nombre de points qu'elle rapporte `points_bonus` et le nombre de coups nécessaires pour la détruire `hits_needed`; qui servent à définir les attributs de la classe. Une brique peut tenir un *power-up*, par conséquent il a convenu d'ajouter un attribut `held_powerup_type` indiquant le type de *power-up* que tient la brique. Une fonction `intersects_bricks()` vérifie les intersections d'un objet spécifié avec une des briques du niveau (provenant du vecteur de briques de la classe `PlayState`). La fonction `on_destroy()` définit ce qui se passe lors de la destruction d'une brique.

2.2.2.5 Powerup

Enfant de `GameObject`. Lorsque le joueur détruit une brique détenant un bonus, celui-ci tombe et peut-être récupéré en touchant la raquette du joueur. Il fut donc naturel d'en faire un élément comme les autres, à quelques détails près. Effectivement, il faut pouvoir gérer divers comportements en fonction du type de *power-up* récupéré. La classe **enum** `POWERUP_TYPE` répertorie les types de bonus, on peut donc ajouter un attribut `type` et l'utiliser pour déterminer le comportement que le jeu doit adopter au moment d'activer ce bonus. Les fonctions `on_activation()`, `on_deactivation()`, `on_update()` et `on_input()` permettent donc de définir ledit comportement. Pour finir, la fonction `handle_collisions_plate()` vérifie les collisions du *power-up* avec la raquette du joueur.

2.2.2.6 Laser

Enfant de `GameObject`. Les instances de cette classe sont uniquement créées lorsque le `Powerup` de type laser est actif. La classe ajoute une fonction `on_update()` qui décide de ce qui se passe à chaque appel de la fonction `update()` du `GameState` actuel (mouvements, collisions). Une autre fonction `spawn_laser()` qui, elle, est `static`, crée une nouvelle instance de la classe et la place au centre de la raquette du joueur.

2.2.3 États

Le processus d'un jeu-vidéo peut-être segmenté en plusieurs états (menu, jeu, pause, *game over*, etc...). Chacun de ses états pourrait nécessiter un traitement propre à celui-ci. La création d'une classe qui généralise les états de jeu fut un bon moyen de poser des bases communes.

2.2.3.1 GameState

En effet, chaque état de jeu doit pouvoir mettre à jour des informations, afficher des choses à l'écran et gérer les *inputs* du joueur. C'est ce que les fonctions `update()`, `input()` et `render()` font respectivement. Chaque état de jeu va hériter de ces fonctions et les définir pour convenir au traitement que l'on souhaiterait accomplir lorsque cet état est actif.

2.2.3.2 GameStateManager

Cette classe contient un attribut `current_state` de l'état actuel du jeu. Nous nous en servons pour appeler `update()`, `input()` et `render()` dans la boucle principale, de sorte à ce que la définition de ces fonctions dépendra du type de `GameState`.

2.2.3.3 PlayState

Enfant de `GameState`, correspond à l'état de jeu. Toute la magie se passe dans cette classe. Nous verrons son fonctionnement plus en détails dans la section 2.3.

2.2.3.4 GameOverState

Enfant de `GameState`, correspond à l'état de *game over*. Nous affichons un texte "VICTORY" au centre de l'écran si la partie fut gagnée, "DEFEAT" autrement. Le score du joueur ainsi qu'un texte "PRESS ANY KEY TO PLAY" sont affichés en-dessous. Le joueur peut appuyer sur n'importe quelle touche pour re-créer une nouvelle instance de `PlayState` assignée à `GameStateManager::current_state` et relancer la partie. Si c'est une victoire, on passe au niveau suivant, sinon on revient au premier niveau.

2.3 Logique de Jeu

Au lancement du programme, les modules d'**Allegro** nécessaires au bon fonctionnement du jeu sont initialisés. Nous créons une instance de `GameStateManager` avec comme état initial `PlayState`, ainsi qu'une instance de `SpriteManager` et de `FontManager` pour charger les *assets* du jeu avec **Allegro**. Nous lançons ensuite le timer d'**Allegro** suivi de la boucle principale. Dans celle-ci, nous appelons d'abord la fonction bloquante `al_wait_for_event()` d'**Allegro** qui attend le signal d'un événement provoqué par le timer ou par un *input* quelconque pour continuer l'exécution du code de la boucle; puis nous appelons les fonctions `update()`, `input()` et `render()` de l'état actuel traqué par `GameStateManager`.

Tel que mentionné plus tôt, l'état initial de jeu est `PlayState`. À la construction de cette classe, on lit le fichier texte qui contient les caractéristiques du niveau spécifié dans `res/stages/` afin de charger les briques ainsi que l'arrière-plan du niveau. La raquette et la balle sont aussi créées et positionnées en bas au centre de l'écran, et le meilleur score est lu depuis le fichier `res/player_data/high_score.txt` puis assigné à la variable `PlayState::high_score`. À ce moment-là, le jeu n'est pas encore considéré comme commencé. C'est lorsque le joueur bouge la raquette la première fois qu'il commence et que la balle se met à bouger. En vérifiant les inputs dans la fonction `PlayState::input()` et en assignant `true` à la variable `PlayState::game_started` si les touches appuyées correspondent à celles qui font bouger la raquette. Si cette variable est `false`, la fonction `PlayState::update()` retourne directement et n'exécute pas son contenu.

Une fois le jeu commencé, `PlayState::update()` peut s'exécuter entièrement et faire bouger les éléments du jeu (la balle, la raquette, les potentiels bonus, les potentiels lasers). Elle gère également les collisions entre les différents éléments à l'aide des fonctions de collisions citées précédemment. Si une collision se déroule entre la raquette et un *power-up*, on assigne celui-ci à la variable `active_powerup`. En conséquence, les fonctions propres à la classe `Powerup` s'exécuteront pour le type du *power-up* attrapé. Enfin, elle vérifie si la partie est terminée (si toutes les briques sauf les briques dorées ont été détruites ou si le joueur n'a plus de vies) et bascule vers l'état `GameOverState` si c'est le cas; puis elle assigne `score` à `high_score` si le score dépasse le meilleur score.

En même temps, `PlayState::input()` s'exécute pour gérer les *inputs* du joueur et faire bouger la raquette que ça soit au clavier ou à la souris / au trackpad. Elle gère aussi les touches qui permettent de réinitialiser le score, passer au niveau suivant ou reculer d'un niveau. Les bonus d'attrapage et de ralentissement ajoutent chacun une fonctionnalité différente à la barre espace, nous faisons alors appel à la fonction `Powerup::on_input()`.

La fonction `PlayState::render()` affiche à chaque instant l'arrière-plan et les éléments du jeu (la balle, la raquette, les potentiels bonus, les potentiels lasers); la barre de vie en bas de l'écran; le score actuel, le meilleur score et le niveau actuel en haut de l'écran.

Lorsque la balle touche une brique, si il lui restait un coup pour être détruite, celle-ci est sans surprise détruite puis supprimée du vecteur de briques. Si il lui restait plus d'un coup pour être détruite, son nombre de coups nécessaires est décrémenté et le programme continue. Au moment de sa destruction, le bonus de points est ajouté au score et le *power-up* que tenait la brique (si elle en tenait un) tombe. On crée une nouvelle instance de `Powerup` pour l'ajouter aux vecteurs de *power-ups*. Ce que nous venons de décrire provient intégralement de la fonction `Brick::on_destroy()`.

2.4 Modèle-Vue-Contrôleur

Nous utilisons bien le **Modèle-Vue-Contrôleur** dans notre projet. Les classes dérivées de `GameState` servent ici de modèles, leur fonction `GameState::render()` de vue et leurs fonctions `GameState::update()` et `GameState::input()` de contrôleurs. De cette manière, nous pouvons clairement faire la distinction entre les différents comportements que le programme doit adopter en fonction du modèle (de l'état de jeu). Voici ci-dessous la déclaration de la classe `GameState` pour illustrer davantage la séparation entre les différents aspects du **MVC**.

```cpp
// Modele
class GameState {
protected:
    GameStateManager *game_state_manager;
    const SpriteManager &sprite_manager;

public:
    explicit GameState(GameStateManager *_game_state_manager, const SpriteManager &
    _sprite_manager) :
            game_state_manager(_game_state_manager), sprite_manager(_sprite_manager) {
    }

    virtual ~GameState() = default;

    virtual void update() = 0; // Controleur

    virtual void input(const ALLEGRO_MOUSE_STATE &, ALLEGRO_EVENT_TYPE, int) = 0; // Controleur

    virtual void render(const FontManager &) const = 0; // Vue
};
```

Listing 1: Déclaration de la classe `GameState`

Veuillez consulter le **diagramme UML** pour plus de détails.

3 Difficultés Rencontrées

Tout projet n'est pas sans son lot de difficultés. Le plus grand *challenge* pour nous fut d'apprendre à manier **Allegro** et à gérer ses lacunes (*memory leaks*, *segfaults*, etc...). Initialement, nous n'avons pas été prudents et les *assets* du jeu étaient chargées à chaque itération de la boucle principale. Sans grande surprise la mémoire se remplissait et faisait *crash* le programme, il nous est donc venu à l'idée de les charger qu'une seule fois au lancement. Par la même occasion, nous avons pu aiguiser nos compétences d'écriture, pour éviter les erreurs liées aux pointeurs principalement, mais aussi pour mieux manier les modificateurs de visibilité et de portée (`static`, `const`); afin d'optimiser la gestion de la mémoire et d'éviter de ré-interpréter des valeurs constantes au moment du *runtime*.

4 Conclusion

Pour conclure ce rapport, il est bon de dire que cet exercice de programmation orientée objet fut une manière pertinente de pratiquer et de surmonter des problèmes d'organisation ou d'optimisation qui peuvent survenir lors de la conception d'un projet de cette envergure. Il fut également fructifiant de pouvoir travailler en binôme, du fait qu'il faille non seulement savoir programmer correctement mais aussi communiquer ses idées de façon claire et complète. Peut-être serait-ce une porte d'entrée vers un intérêt plus prononcé pour le développement de jeux-vidéos ?