OS - PROJET

I. Introduction décrivant le jeu et ses fonctionnalités

Introduction

Notre OS est une reproduction du jeu *Smash Bros* d'une manière plus simplifiée avec moins de fonctionnalités. L'objectif principal était de mettre en oeuvre les notions vus en cours, notamment:

- gestion des threads
- synchronisation
- accès au matériel graphique et au clavier
- développement d'une application complexe (notre jeu) dans un environnement sans système d'exploitation hôte (autre que le nôtre).

Présentation du jeu

Notre jeu propose les fonctionnalités suivantes:

- Un menu principal permettant de démarrer ou quitter le jeu
- Une sélection de personnages parmi 5 personnages différents
- Un combat entre 2 joueurs distincts, chacun contrôlé par un clavier (le même) où chaque joueur possède plusieurs actions à sa disposition
- Un affichage graphique en mode VGA
- Un HUD affichant les informations essentielles du joueur (pourcentage de dégâts)

Personnages

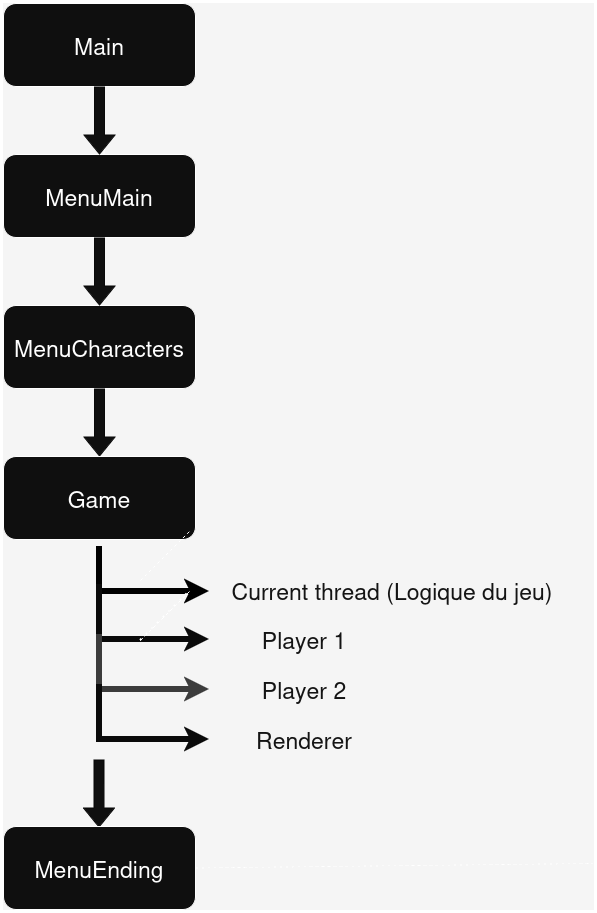
Chaque personnage dispose de caractéristiques propres (sprite, animations), et peut effectuer les actions suivantes:

- Se déplacer horizontalement
- Attaquer
- Parer
- Sauter

Les personnages sont implémentés de manière générique afin de faciliter l'ajout de nouveaux combattants.

II. Architecture de l'application avec schéma

L'application est organisée en plusieurs modules principaux afin de faciliter l'ajout de nouveaux combattants.



Main

Le module Main initialise l'environnement d'exécution avant de lancer le jeu.
Il est notamment responsable de:

- Initialisation du matériel
- Gestion des interruptions
- Initialisation mémoire
- Initialisation de l'ordonnancement

- Lancement de l’application (Affichage du menu principal, sélection des personnages et lancement de la partie

MenuMain

C’est l’écran d'accueil. Il permet soit de lancer une partie soit de quitter le jeu.

MenuCharacters

Pour la sélection des personnages, le Joueur 1 choisit en premier et le Joueur 2 choisit ensuite parmi les personnages restants.

Game

C’est le cœur de l’application. Il est responsable de :

- la création et la gestion des threads
- la logique de jeu
- le rendu graphique
- la synchronisation entre les différentes activités

MenuCharacters

C’est le menu qui se lance à la fin et qui affiche qui a gagné.

III. Détails et justifications d'implémentation sur la manière de gérer les activités, la synchronisation et la mémoire

Gestion des activités (threads)

Lorsque les 2 joueurs ont choisi leur personnage, la méthode run() de Game se lance. Celle-ci s’exécute dans le thread principal et est responsable de la logique globale (règles du jeu et combat). A l’intérieur de cette méthode, plusieurs threads se lancent:

- Thread Player 1 : met à jour l’état du joueur 1
- Thread Player 2 : met à jour l’état du joueur 2

- Thread Render : responsable du rendu graphique

→ Cette organisation nous permet d'avoir une bonne séparation des responsabilités et d'éviter des *race conditions* car chaque objet Player est responsable de son propre état et de ses déplacements et les états de player1 et de player2 coexistent en parallèle, sont indépendants et sont modifiés en même temps. Ensuite, le *render* est un élément indépendant qui ne fait qu'afficher l'état du monde à un instant t. Enfin, le thread principal est chargé de la logique globale du jeu. Il coordonne les interactions entre les joueurs et applique les règles communes, assurant ainsi une cohérence globale du système tout en s'appuyant sur l'exécution concurrente des autres threads.
Cette organisation permet de limiter les risques de *race conditions* en isolant les mises à jour locales des joueurs dans des threads distincts, tout en confiant la logique globale à un thread central. Nous verrons dans la partie suivante comment nous avons protégé ce risque.

Synchronisation

Comme mentionné précédemment, plusieurs threads accèdent à des données partagées (position des joueurs, état et données du jeu), une synchronisation est donc indispensable.

Choix de synchronisation
La synchronisation est assurée à l'aide d'un SpinLock (on a utilisé celui déjà présent dans le code fourni) créé en tant qu'attribut dans la classe Game
Avant toute modification ou lecture de l'état partagé :

- le thread acquiert le verrou
- effectue son traitement
- puis libère le verrou

A la fin de ces opérations, chaque thread fait un appel à *thread_yield()* pour donner la main aux autres threads.

Gestion de la mémoire

Le projet s'exécute dans un environnement sans runtime C++ standard:

- pas de pile (pour le stockage dynamique)

- pas de new / delete
- pas d'allocations dynamiques implicites

Ainsi, tous nos objets sont alloués:

- soit statiquement
- soit par valeur

Gravité

Chaque joueur est modélisé à l'aide des grandeurs physiques standards :

- une position
- une vitesse
- une accélération

Ce modèle permet de représenter le mouvement des personnages. Des forces peuvent être appliquées aux joueurs, notamment la gravité qui agit en permanence sur l'axe vertical.
Lorsqu'une touche de déplacement est pressée, une variation de vitesse est appliquée au joueur. Le mouvement horizontal et vertical est ensuite calculé à partir des équations cinématiques, en tenant compte de l'intervalle de temps entre deux mises à jour.
Afin d'éviter des déplacements irréalistes ou instantanés, un mécanisme de friction est introduit. Il consiste à réduire progressivement la vitesse du joueur à chaque frame, ce qui permet d'obtenir un mouvement plus naturel et contrôlé.

Collisions

Il n'y a que des collisions d'un player avec les sols et plateformes. La zone de collision d'un personnage est située principalement au niveau de la base du sprite, ce qui correspond au point de contact avec le sol.
Lorsqu'un déplacement est calculé, une nouvelle position "candidate" est déterminée. Une détection de collision est ensuite effectuée:

- si le déplacement en x provoque une pénétration dans l'objet, le déplacement en x est annulé
- si le déplacement en y provoque une pénétration dans l'objet, le déplacement en y est annulé

- si les deux axes sont concernés, dans la plupart des cas, les déplacements dans les deux axes sont annulés (sauf dans des cas spécifiques de collisions en diagonale)

Hitbox

Les personnages et les plateformes sont modélisés par des hitboxes rectangulaires définies par 4 coordonnées. Pour tester les collisions, on teste simplement l'intersection de 2 rectangles.
Pour stocker ces hitbox en mémoire, on le fait dans un tableau statique. Si nous avons n objets, nous avons donc n hitbox à stocker et ce tableau contiendra 4*n valeurs.
Nous avons essayé d'utiliser des 'struct' mais nous avons beaucoup de problèmes au runtime lors du passage ou du retour de structure en argument. On n'est pas sûr des raisons de ces problèmes au vu des erreurs compliquées à lire et à comprendre mais on juge que l'environnement restreint au runtime par rapport au C++ standard n'a pas aidé.

IV. Conclusion

Ce projet nous a permis d'appliquer concrètement les notions fondamentales vues en cours, notamment la gestion des threads, la synchronisation et l'accès direct au matériel, dans un environnement sans système d'exploitation hôte. L'architecture modulaire et concurrente du jeu assure une bonne séparation des responsabilités tout en limitant les risques de *race conditions*. Malgré les contraintes liées à l'absence de runtime C++ standard, nous avons obtenu un jeu fonctionnel et extensible, illustrant efficacement les enjeux du développement bas niveau.