Projet Multijoueur en C++ : The New Pong

Auteurs: Dounia Bakalem, Yanis Sadoun, Vasileios Filippos Skarleas

L'objectif

Puisqu'il n'y a rien de plus amusant pour découvrir un langage que de créer son propre jeu, nous vous présentons **The New Pong**, un jeu multijoueur développé dans le cadre du module de programmation en langage objet pour la spécialité Robotique à Polytech Sorbonne.

Dans ce projet, il nous était demandé de choisir un jeu à programmer en C++, afin de mettre en pratique les notions étudiées en cours tels que:

- L'héritage
- Le polymorphisme
- L'abstraction
- Les foncteurs
- Les fonctions lambdas
- CMake
- Les fonctions virtuelles

Nous avons donc opté pour un grand classique: Pong. Préparez-vous à renvoyer la balle, tout en perfectionnant vos compétences en C++ !

Le jeu

Afin de revisiter l'expérience Pong, l'un des tout premiers jeux vidéo d'arcade et pionnier des jeux de sport, nous avons décidé d'en développer notre propre version. Au-delà d'un simple hommage, nous y avons ajouté de nouvelles fonctionnalités pour rendre ce Pong encore plus captivant que l'original. Pour cela, quatre modes de jeu distincts ont été introduits :

1. **AI mode**
2. **Classic**
3. **Storytime mode**
4. **Fun mode**

Toutes les instructions relatives à ces modes et leurs spécificités sont détaillées dans la section: **Les différents modes**. Bonne lecture et bon amusement !

Compilation

Nous avons intégré un fichier **CMakeLists.txt** afin de faciliter la compilation du projet sur les principaux systèmes d'exploitation tels que macOS et Linux. Toutefois, l'interface graphique nécessite plusieurs dépendances spécifiques.

Pour simplifier cette étape, nous avons également créé un script Bash qui vérifie automatiquement si toutes les dépendances sont installées. Si ce n'est pas le cas, il se charge de télécharger et d'installer ce qui manque. Vous trouverez la liste exhaustive de ces dépendances dans la section **Dépendances** .

Instructions de lancement

Pour démarrer le programme en mode automatique, suivez les étapes suivantes :

1. Faire un clone du projet
2. Ouvrez un terminal

3. Saisissez `chmod 777 play.sh` dans le terminal, puis lancez le script avec `bash play.sh`.

Notes

> Sur Linux, le script télécharge et installe automatiquement les bibliothèques SDL nécessaires. Toutefois, si l'une d'entre elles requiert une autre dépendance spécifique, vous devrez l'installer manuellement. Une fois la dépendance installée, relancez simplement le script avec la commande `bash play.sh`.

Documentation

Pour plus de détails sur la structure du projet et les commentaires (classes, fonctions, etc.), rendez-vous sur : https://pong.madebyvasilis.site

Dépendances

Voici la liste des dépendances indispensables au bon fonctionnement du programme :

- **SDL2** : Bibliothèque principale pour la gestion de la fenêtre et des événements
- **SDL2_ttf** : Bibliothèque pour le rendu du texte (polices TrueType)
- **SDL2_mixer** : Bibliothèque pour la gestion du son et de la musique
- **SDL2_image** : Bibliothèque pour le chargement d'images (formats multiples)
- **SDL2_net** : Bibliothèque pour les fonctionnalités réseau

(Assurez-vous que ces bibliothèques sont installées ou que le script les télécharge correctement)

Les différents modes

Classic

Le concept originel de Pong s'apparente à un simulateur de ping-pong minimaliste : une balle se déplace de part et d'autre de l'écran en rebondissant sur les bords supérieur et inférieur. Chaque joueur contrôle une raquette coulissant verticalement le long du bord de l'écran. La balle rebondit différemment selon la partie de la raquette qu'elle touche.

- **Fonctionnalités incluses** :
  - **High Score**
  - **Game Save**

Dans notre version, il n'y a pas de score maximum prédéfini ; les joueurs peuvent simplement s'entendre oralement sur un objectif à atteindre. Lorsqu'ils souhaitent arrêter, il suffit de choisir « End the game ». Ici, la motivation ultime est : ***qui fera exploser le compteur du high score et revendiquera le titre de meilleur pongiste ?***

AI mode

Ce mode reprend les règles du **Classic** , à la différence qu'il ne peut être joué que par un seul joueur : la raquette adverse est contrôlée par l'ordinateur. **Préparez-vous à affronter une IA tenace. Arriverez-vous à la battre, ou rejoindrez-vous la longue liste de ses victimes ?**

Storytime mode

Dans ce mode, deux joueurs s'affrontent sur **3 tours.** Le vainqueur est celui qui remporte le plus de tours . Chaque tour se compose de **8 points**, et c'est le premier joueur à atteindre 8 points qui gagne le tour.

Une nouveauté pimentera votre partie : des lettres tombent depuis le haut de l'écran. En les touchant, vous obtenez un point supplémentaire et vous contribuez à former un mot caché, révélant peu à peu une phrase secrète.

Fun mode

Ce mode s'inspire des règles du **Storytime Mode** , avec un format de **3 parties** où l'objectif est d'atteindre 5 points pour remporter chaque partie. Toutefois, nous y avons glissé plusieurs surprises et easter eggs destinés à dynamiser la compétition.

Puisque nous sommes de futurs roboticiens, nous ne pouvions pas résister à ajouter une petite touche de robotique : vous verrez ainsi de mystérieux robots apparaître au cours de la partie. En les touchant, vous déclencherez des effets inédits :

- Raquette géante : votre raquette gagne temporairement en taille.
- Balle invisible : la balle disparaît momentanément, rendant la partie plus chaotique.
- Contrôles inversés : les touches de votre adversaire se retrouvent soudainement inversées.

Saurez-vous exploiter ces bonus (et pièges) pour devenir le champion incontesté du Fun Mode ?

High Score

Cette fonctionnalité est disponible uniquement en mode Classic . Le jeu vérifie en permanence si un joueur atteint un score supérieur au record actuel. Lorsque c'est le cas, le record est immédiatement mis à jour.

La sauvegarde est effectuée dans un fichier nommé `game_pong-highscore_849216.txt`, dont le contenu est chiffré afin de garantir l'intégrité des données et d'empêcher toute modification non autorisée. Ce fichier contient uniquement le dernier high score ainsi que le nom du joueur correspondant.

Voici l'algorithme qui détermine si quelqu'un a fait un nouveau highscore:

```
// High score logic
        if (player1->get_user_score() >= last_highscore || player2-
>get_user_score() >= last_highscore)
        {
            last_highscore = (player1->get_user_score() >= player2-
>get_user_score()) ? player1->get_user_score() : player2-
>get_user_score();

            strncpy(last_highscore_name, ((player1->get_user_score() >=
player2->get_user_score()) ? player1->get_user_name() : player2-
>get_user_name()).c_str(), 19);
            last_highscore_name[19] = '\0';
        }
```

Game Save

Envie de faire une pause et de retenter de battre le record un peu plus tard ? Avec la fonctionnalité de Game Save , vous pouvez sauvegarder l'état de votre partie et la reprendre quand vous le souhaitez. Là encore, le chiffrement est appliqué pour garantir l'intégrité des données.

Game save logic

```
SaveState saveState;
saveState.score1 = player1->get_user_score();
saveState.score2 = player2->get_user_score();
saveState.paddle1_y = racket1->get_pos_y();
saveState.paddle2_y = racket2->get_pos_y();
saveState.ball_x = mBall->get_pos_x();
saveState.ball_y = mBall->get_pos_y();
saveState.ball_vel_x = mBall->get_vel_x();
saveState.ball_vel_y = mBall->get_vel_y();
saveState.ball_type = mMiddleMenu->get_selected_option();

strncpy(saveState.player1_name, player1->get_user_name().c_str(), 19);
saveState.player1_name[19] = '\0'; // Ensuring that the name ends to \0
that is standar for string types
strncpy(saveState.player2_name, player2->get_user_name().c_str(), 19);
saveState.player2_name[19] = '\0';

if (Saving::save_game(saveState))
{
  SDL_Log("Game saved successfully");
  mMenu->set_saved_file_exists();
  mNoticeMenu->set_notice_id(GAME_SAVED);
  mGameState = game_state::Notice_Menu; // We go back to the main menu
}
else
{
   SDL_Log("Failed to save game");
   mIsRunning = false;
}
```

> La sauvegarde du jeu est réalisée dans un fichier nommé `game_pong-save_849374.txt`. Ce fichier reste disponible jusqu'à ce que le joueur reprenne la partie sauvegardée ou choisisse de démarrer une nouvelle partie, auquel cas il sera automatiquement supprimé. Ainsi, votre progression est préservée même après avoir quitté le jeu.

Game retrieve logic

```
SaveState savedState;
if (Saving::load_game(savedState))
{
    player1->set_user_score(savedState.score1);
    player2->set_user_score(savedState.score2);
```

```
    player1->set_user_name(savedState.player1_name);
    player2->set_user_name(savedState.player2_name);

    racket1->set_pos_y(savedState.paddle1_y);
    racket2->set_pos_y(savedState.paddle2_y);
    ball_creation(savedState.ball_type);
    mBall->set_position(savedState.ball_x, savedState.ball_y);
    mBall->set_velocity(savedState.ball_vel_x, savedState.ball_vel_y);
    update_background_color();

    Saving::delete_save(); // Delete the saved game file once we have
loaded the game state

    mGameState = game_state::Playing;

    SoundEffects::change_music_track(mBackgroundMusic);
}
```

Choisir le type de la balle

Par défaut, la balle du Pong est de forme circulaire, mais pourquoi ne pas la personnaliser ? À chaque début de partie, vous pouvez sélectionner l'une des 3 formes proposées :

1. Cercle : avec une image graphique pour la détection de collision [SDL forme utilisée pour détecter les collisions].
2. Triangle
3. Carré

Ce n'est qu'une preuve de concept : rien ne vous empêche d'imaginer et d'intégrer des formes plus originales dans l'interface graphique.

Changement de la musique

Grâce à la bibliothèque SDL Mixer, nous pouvons gérer différents effets sonores et musiques avec des fonctions de fade-in et fade-out. Chaque mode peut ainsi avoir sa propre ambiance sonore, pour rendre l'expérience de jeu encore plus immersive.

Voici l'implémentation:

```
void SoundEffects::change_music_track(Mix_Music *music_file,
                                      int fade_out_duration,
                                      int fade_in_duration,
                                      int volume)
{
    Mix_FadeOutMusic(fade_out_duration);
    // SDL_Delay(5);
    Mix_FadeInMusic(music_file, -1, fade_in_duration);
    Mix_VolumeMusic(volume);
}
```

Chiffrement des données

La sauvegarde des données utilise un système de chiffrement XOR simple avec une clé rotative:

```
class SavingEncryption {
private:
    static const std::vector<uint8_t> KEY;

    static void encryptData(std::vector<uint8_t>& data) {
        for (size_t i = 0; i < data.size(); ++i) {
            data[i] ^= KEY[i % KEY.size()];
        }
    }
};
```

Les données sont chiffrées avant l'écriture sur le disque et déchiffrées lors de la lecture, assurant une protection basique des sauvegardes.

> Inspiré de https://www.101computing.net/xor-encryption-algorithm/ L'utilisation de XOR permet à la même opération de chiffrer et de déchiffrer

Les objets

Dans ce projet, toutes les fonctionnalités ont été implémentées sous la forme d'objets, garantissant ainsi la modularité, la flexibilité et une organisation claire du code. Chaque élément du jeu Pong est représenté par une classe spécifique, ce qui permet une maintenance aisée et une évolutivité simplifiée du programme.

Voici les différentes classes que nous avons définies :

Class	Description	Fichier
AI	Intelligence artificielle pour contrôler une raquette automatiquement	ai.cpp
BallBase	Classe de base abstraite pour tous les types de balles dans le jeu car nous proposons différents types de balles à choisir avant de lancer le jeu	ball_base.pp
ClassicBall	Implémentation classique de balle circulaire héritant de BallBase	classic_ball.cpp
Game	Contient tous les paramètres principaux, surtout les références de tous les autres objets mentionnés dans cette liste	game.cpp
GameOver	Gère l'écran de fin de partie lorsqu'une partie est terminée ou si on choisit de terminer manuellement une partie	game_over.cpp
GUI	Classe utilitaire fournissant des fonctionnalités d'interface utilisateur (donner notre prénom via SDL))	gui.cpp
HighScore [structure]	Structure représentant un record de score . Il gère la sauvegarde de ces données spécifiques	game_save.cpp

Class	Description	Fichier
InvisiblePower	Rend la balle temporairement invisible. Il hérite de la classe Power	invisible_power.cpp
Power	Représente les éléments de power-up qui affectent le gameplay comme le changement de la taille de la raquette, ou rendre la balle invisible	power.cpp
Letter	Représente une lettre dans le mode de jeu Storytime. Contient toute la fonctionnalité pour gérer les mots dans ce mode Storytime	letter.cpp
Paddle	Représente une raquette (paddle) de joueur	paddle.cpp
SaveState [structure]	Structure représentant l'état complet du jeu pour la sauvegarde/le chargement	game_save.cpp
Saving	Classe utilitaire de sauvegarde pour gérer la sauvegarde de la partie et la fonctionnalité de score élevé	game_save.cpp
SoundEffects	Classe pour gérer les effets sonores et la musique dans le jeu	sound_effects.cpp
SquareBall	Implémentation de la balle en forme de carré héritant de BallBase	square_ball.cpp
TriangleBall	Implémentation de la balle en forme de triangle héritant de BallBase	triangle_ball.cpp
User	Représente un joueur dans le jeu avec son nom et le suivi du score	user.cpp
page_2b_1t	Écran d'avis avec 2 boutons et 1 titre	page_2b_1t.cpp
page_3b	Menu de pause avec 3 boutons	page_3b.cpp
page_3b_0t	Classe de menu principal avec 3 boutons et aucun titre	page_3b_0t.cpp
page_3b_1t	Classe de menu intermédiaire avec 3 boutons et 1 titre	page_3b_1t.cpp
page_4b_1t	Définit le menu de sélection de mode avec 4 boutons et 1 titre	page_4b_1t.cpp

Les foncteurs

Foncteur	Descriptions	Fichier
triangle_renderer	Foncteur pour le rendu des formes triangulaires	renderers.cpp
square_renderer	Foncteur de rendu de formes carrées	renderers.cpp

L'utilisation de foncteurs nous permet d'ajouter facilement de nouveaux types de formes et de les tester individuellement.

> Cette approche nous a permis d'accélérer le développement en permettant des tests isolés des différents SDL renderers.

Structure des pages

Afin de garantir une interface utilisateur claire, fluide et facilement adaptable, nous avons défini plusieurs structures prédéfinies pour l'affichage des différentes pages du jeu. Chaque modèle est conçu pour répondre à des besoins spécifiques et assurer une navigation intuitive.

Voici les spécifications précises de chaque modèle :

- `page_3b_1t` : Trois boutons centrés verticalement, accompagnés d'un titre en gras en haut de la page (utilisé pour les menus principaux).
- `page_2b_1t` : Deux boutons et une large section dédiée à un texte explicatif (idéal pour l'affichage d'avis ou d'instructions détaillées).
- `page_4b_1t` : Quatre boutons répartis sur la page, avec un titre en gras en haut (utilisé pour la sélection des modes de jeu).
- `page_3b_0t` : Trois boutons répartis de manière spécifique : deux placés en haut et un troisième positionné vers le bas de la page (permettant de mettre en avant une option particulière).
- `page_3b` : Trois boutons alignés verticalement et centrés au milieu de l'écran (structure utilisée pour le menu pause).

Ces structures offrent une navigation cohérente, garantissant une meilleure expérience utilisateur tout au long du jeu.

La logique du jeu

L'interface graphique

Maintenant que nous avons une vue d'ensemble des différentes pages et des éléments interactifs du jeu, intéressons-nous à la façon dont l'interface graphique est conçue et gérée.

Nous utilisons **SDL** pour afficher et rendre toutes les formes et objets du jeu dans une fenêtre aux dimensions prédéfinies dans le fichier `macros.hpp` (plus de détails dans la section ***Pourquoi macros.hpp***).

Le programme principal repose sur la classe `Game` , qui orchestre l'ensemble du jeu à travers trois méthodes clés :

1. `initialise()` – Initialise tous les paramètres et variables nécessaires au jeu.
2. `loop()` – Gère la boucle principale du jeu.
3. `close()` – Libère les ressources et termine proprement l'exécution.

La méthode `loop()` constitue le cœur du jeu : il s'agit d'une boucle while qui tourne en continu tant que le jeu est actif. Cette boucle s'arrête uniquement si la variable booléenne `mIsRunning` est définie sur `false`, soit lorsque le joueur ferme la fenêtre SDL, soit lorsqu'il sélectionne "**Exit Game**"

```
void Game::loop()
{
    while (mIsRunning) // set to false when we either tap on the X to
close the SDL window or when we tap on the Exit game button
    {
        game_logic();
```

```
            game();
            output();
        }
    }
```

Loop

Dans cette boucle, trois fonctions essentielles assurent le bon déroulement du jeu :

- `game_logic()` : Gère la logique principale et décide des transitions entre les pages, menus et événements du jeu.
- `game()` : Met à jour l'état du jeu en fonction des actions du joueur, détermine si une partie est terminée et applique les règles.
- `output()` : Génère et affiche les éléments visuels sur l'interface SDL en fonction des paramètres définis par la logique du jeu.

Ces trois fonctions fonctionnent en synergie pour offrir une expérience fluide et dynamique, assurant que le jeu réagit de manière cohérente aux interactions du joueur.

Héritage

L'héritage est largement utilisé pour étendre la fonctionnalité des classes de base. Les trois types de balles (`ClassicBall`, `SquareBall` et `TriangleBall`) héritent tous de la classe abstraite `BallBase`. Par exemple, dans `classic_ball.hpp`, nous voyons :

```
class ClassicBall : public BallBase {
public:
    ClassicBall() : BallBase(24.0f) {}
    void render_object(SDL_Renderer *renderer) override;
    // ...
};
```

Dans le domaine des power-ups, nous avons également une hiérarchie d'héritage. Les classes `InvisiblePower` et `InversePower` héritent de la classe `Power`, comme on peut le voir dans `invisible_power.hpp` et `inverse_power.hpp`. Cela permet de partager le comportement commun tout en spécialisant certaines fonctionnalités

Polymorphisme

Le polymorphisme est implémenté à travers l'utilisation de méthodes virtuelles et leur redéfinition dans les classes dérivées. Un exemple clair se trouve dans la hiérarchie des balles, où la méthode `render_object()` est définie différemment dans chaque type de balle :

- Dans `classic_ball.cpp`, elle dessine un cercle.
- Dans `square_ball.cpp`, elle dessine un carré.
- Dans `triangle_ball.cpp`, elle dessine un triangle.

Le jeu peut manipuler n'importe quel objet dérivé de `BallBase` de manière uniforme, en appelant `mBall->render_object(renderer)` dans `game.cpp`, sans se soucier du type spécifique de balle utilisé.

De même, les power-ups démontrent le polymorphisme avec leurs méthodes `update()` et `render()` qui sont appelées de manière générique mais exécutent un comportement spécifique à chaque type de power-up.

Abstraction

L'abstraction est implémentée principalement à travers les classes abstraites du projet. La classe `BallBase` est un excellent exemple d'abstraction. Dans le fichier `ball_base.hpp`, nous définissons une interface commune pour tous les types de balles avec des méthodes abstraites comme `render_object()`. Cette méthode est déclarée virtuelle pure (`= 0`), obligeant toutes les classes dérivées à fournir leur propre implémentation.

```
class BallBase {
public:
    // ...
    virtual void render_object(SDL_Renderer *renderer) = 0;
    // ...
};
```

De même, la classe `Power` dans `power.hpp` fournit une abstraction pour les différents types de power-ups du jeu, avec des méthodes virtuelles qui peuvent être redéfinies par les classes dérivées comme `InvisiblePower` et `InversePower`.

Fonctions lambda

Nous utilisons des fonctions lambda pour contrôler les limites physiques de la raquette (paddle). Cette approche nous permet d'obtenir un code modulaire, facilitant la mise à jour des fonctionnalités liées au déplacement et aux contraintes de position de la raquette.

Les lambdas sont particulièrement adaptées à notre cas, car elles nous permettent de définir des fonctions anonymes tout en bénéficiant d’un typage automatique, simplifiant ainsi l'écriture du code.

Les deux fonctions lambda utilisées sont:

- `auto move_paddle = [this](float delta, float time)` utiliser pour bouger l'objet paddle (la raquette)
- `auto adjust_boundaries = [this]()` responsable de vérifier et ajuster les limites de la raquette

```
auto move_paddle = [this](float delta, float time)
    {
        this->set_pos_y(this->get_pos_y() + delta * this-
>get_racket_speed() * time);
    };
```

```
auto adjust_boundaries = [this]()
    {
        if (this->get_pos_y() < this->get_racket_height() / 2.0f)
        {
            this->set_pos_y(this->get_racket_height() / 2.0f);
        }
        else if (this->get_pos_y() > 600.0f - this->get_racket_height() /
2.0f)
        {
            this->set_pos_y(600.0f - this->get_racket_height() / 2.0f);
        }
    };
```

Encapsulation

L'encapsulation est présente dans presque toutes les classes du projet, avec une distinction claire entre les interfaces publiques et les détails d'implémentation privés. Par exemple, la classe `User` dans `user.hpp` encapsule les données relatives au joueur :

```
class User {
private:
    std::string name;  // Le nom de l'utilisateur
    int score;         // Le score actuel de l'utilisateur
    int round;         // Le numéro de round actuel

public:
    // Méthodes pour interagir avec les données encapsulées
    void increment_score();
    void reset_score();
    int get_user_score() const;
    // ...
};
```

Un autre exemple d'encapsulation se trouve dans le fichier `game_save.cpp`, où un namespace anonyme est utilisé. Tous les détails sont disponibles ci-dessous.

Namespace

Dans le fichier game_save.cpp, nous utilisons un namespace anonyme afin d'encapsuler les constantes sensibles (comme la clé de chiffrement XOR) ainsi que les fonctions utilitaires dédiées au codage et au décodage des données de sauvegarde.

Ce choix présente plusieurs avantages :

1. **Sécurité renforcée** : empêche tout accès externe aux mécanismes internes du système de sauvegarde.
2. **Prévention des conflits** : réduit les risques de collision de noms avec d'autres parties du code.

3. **Organisation claire** : regroupe logiquement les éléments qui interagissent ensemble, améliorant ainsi la lisibilité et la maintenabilité du code.

L'utilisation d'un namespace anonyme garantit donc une encapsulation stricte et protège les données critiques du jeu contre toute manipulation involontaire ou non autorisée. Cette approche garantit que ces éléments ne sont accessibles que depuis ce fichier, renforçant ainsi la sécurité du mécanisme de sauvegarde.

```
namespace
{
    unsigned char codec_byte(unsigned char byte)
    {
        return byte ^ KEY; // XOR operation
    }

    void codec_float(float &value)
    {
        unsigned char *bytes = reinterpret_cast<unsigned char *>(&value);
// casting to return a float value to byte representation
        for (size_t i = 0; i < sizeof(float); ++i)
        {
            bytes[i] = codec_byte(bytes[i]);
        }
    }

    void codec_int(int &value)
    {
        unsigned char *bytes = reinterpret_cast<unsigned char *>(&value);
        for (size_t i = 0; i < sizeof(int); ++i)
        {
            bytes[i] = codec_byte(bytes[i]);
        }
    }

    void codec_string(char *str, size_t length)
    {
        for (size_t i = 0; i < length; ++i)
        {
            str[i] = codec_byte(str[i]);
        }
    }
}
```

Cette organisation du code illustre parfaitement le principe d'encapsulation, un pilier fondamental de la programmation orientée objet. En limitant l'accès aux éléments internes du système de sauvegarde, nous renforçons la sécurité, l'isolation et la robustesse globale du jeu.

> Afin de réduire la longueur du rapport, nous avons retiré les commentaires détaillés des différentes fonctions. Pour une explication complète et une vue d'ensemble du code, vous pouvez consulter directement game_save.cpp.

Autres aspects du développement

Tout au long du projet, nous avons soigneusement choisi les niveaux de visibilité des variables dans nos classes, en décidant de les déclarer `private` ou `public` en fonction de leur usage. De plus, nous avons veillé à utiliser de manière appropriée les mots-clés `static` et `virtual`, garantissant ainsi une encapsulation efficace et une meilleure organisation du code.

Les tests

En cours, nous avons vu le principe d'utiliser CTest pour organiser la vérification des tests unitaires. Dans ce cadre-là, et vue que nous avons déjà intégré un fichier `CMakeLists.txt`, nous avons décidé de procéder avec cette méthode pour tester les fonctionnalités et les méthodes crées pour les différentes classes dans notre projet.

Vous allez trouver un répertoire nommé `tests`. Il a son propre `CMakeLists.txt` qui permet de créer les exécutables de nos programmes tests qui vérifient la bonne fonctionnalité de nos méthodes. Chaque ficher *_test.cpp de base est un programme individuel qui peut se comporter comme un programme main Independent qui va retourner la valeur 0 lorsque le test est effectué avec succès et renvoie une autre information dans le cas contraire.

La realisation des tests unitaires participe d'une démarche d'intégration continue et permet de valider la non régression du code au cours du development de notre jeu. Voici les differents choses que nous avons testé:

- **balls_test :** Toutes les fonctionnalités de la classe abstraite BallBase et ses classes hérités SquareBall, TriangleBall et ClassicBall inclus les constructeurs, les setters et getters ainsi que les méthodes responsables pour le rendering sur SDL.
- **user_test :** Les méthodes de création et mise à jour d'un joueur d'User class
- **paddle_test :** On est également la création de nos deux rackets (paddles) pour le jeu. On vérifie s'ils ont bien une distinction de positionnement (gauche ou droit). En plus, on vérifie la mise à jour de leurs tailles par les méthodes spécifiques, le rendering de ces objets ainsi que le control/communication de positionnement
- **letter_test.cpp :** Pour le Storytime mode, qu'on utilise particulièrement la classe Letter, on test la bonne création via ses constructeurs. On vérifie le comportement des méthodes responsables de détecter des collisions entre la balle et les lettres, ainsi que la validation des méthodes pour mettre le score des joueurs.

Comment on test ?

Pour produire un résultat de test, nous utilisons les méthodes statiques de la classe Assert pour tester les résultats réels par rapport aux résultats attendus. Si besoin par la classe qu'on test, on commence par l'initialisation de l'environnement SDL nécessaire au rendu graphique des éléments de jeu, même si cette visualisation reste invisible durant l'exécution des tests.

Chaque appel à `assert` évalue une expression booléenne qui représente une condition que le programme doit satisfaire pour être considéré comme correct. Si cette condition est vraie, l'exécution du programme se poursuit normalement, permettant ainsi de vérifier des conditions supplémentaires. En revanche, si l'expression s'avère fausse, le programme s'interrompt immédiatement avec un message d'erreur précisant

le fichier source. Au final, si jamais notre main d'un test retourne 0, alors le test est bien terminé sans des erreurs.

Pourquoi macros.hpp

Le fichier macros.hpp joue un rôle central dans notre projet en servant de référentiel unique pour toutes les constantes globales du jeu. Il permet de centraliser et de faciliter la gestion des paramètres essentiels, tels que :

Les dimensions de la fenêtre du jeu Les identifiants des modes de jeu (ex. : mode IA, mode 2 joueurs) Les constantes associées à la navigation dans les menus Les niveaux de difficulté

Grâce à ce fichier, nous avons assuré une meilleure lisibilité et une maintenance simplifiée, en évitant la dispersion des constantes dans l'ensemble du code.

Pour aller plus loin

Initialement, nous avons tenté d'implémenter un mode multijoueur en réseau via TCP avec une architecture client-serveur. Cependant, nous nous sommes rapidement heurtés à la complexité de cette intégration.

En effet, cette fonctionnalité aurait dû être pensée dès le début du projet afin d'être intégrée naturellement dans l'architecture existante. L'ajout tardif d'un mode réseau implique de lourdes modifications sur la structure actuelle du code, ce qui s'avère être un défi technique conséquent.

Malgré ces difficultés, nous avons commencé le développement de cette partie dans les fichiers `network.cpp` et `network.hpp`, en nous concentrant sur les aspects suivants :

- Le contrôle des raquettes à distance
- La réception et la synchronisation des positions des différents éléments (balle, raquettes, etc.)

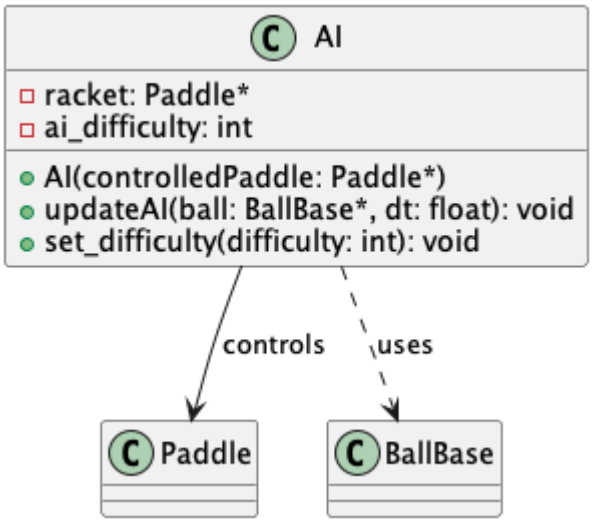
UML – Modélisation des classes du jeu

Pour mieux structurer notre projet et assurer une architecture claire et maintenable, nous avons modélisé les principales classes du jeu sous forme de diagrammes UML. Ces diagrammes UML permettent de visualiser l'architecture du projet et les interactions entre les classes. Cette structuration facilite la compréhension du code, son évolutivité et sa maintenance.

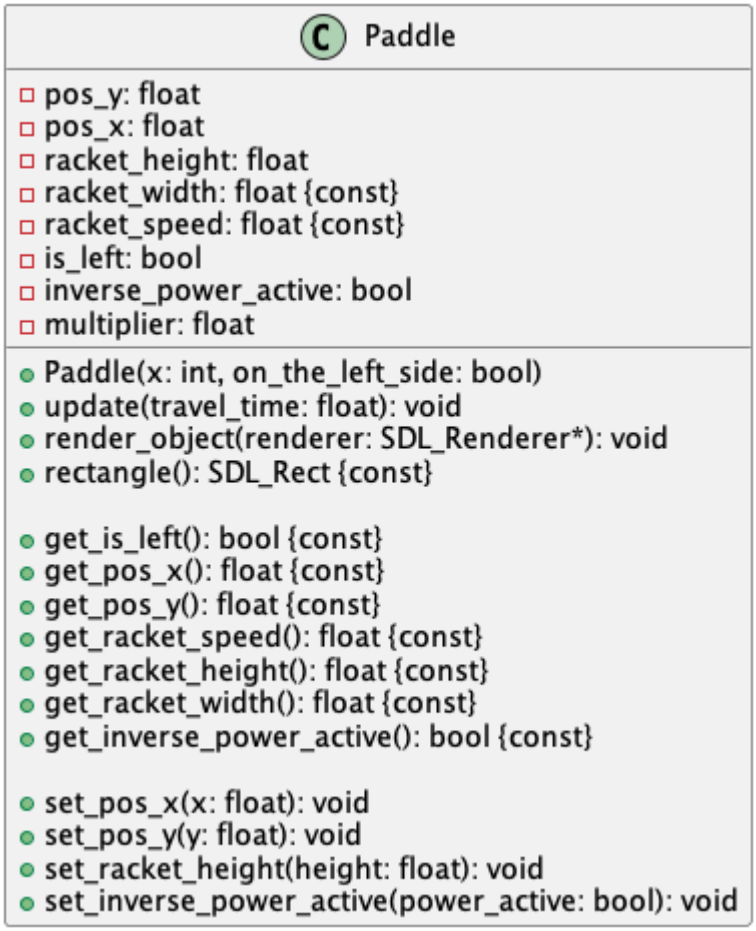
Class	UML

Class UML

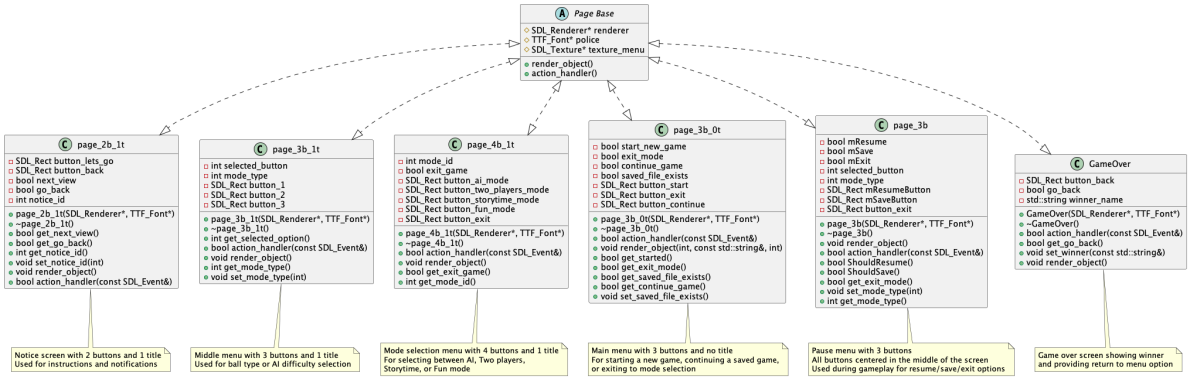
AI



Paddle

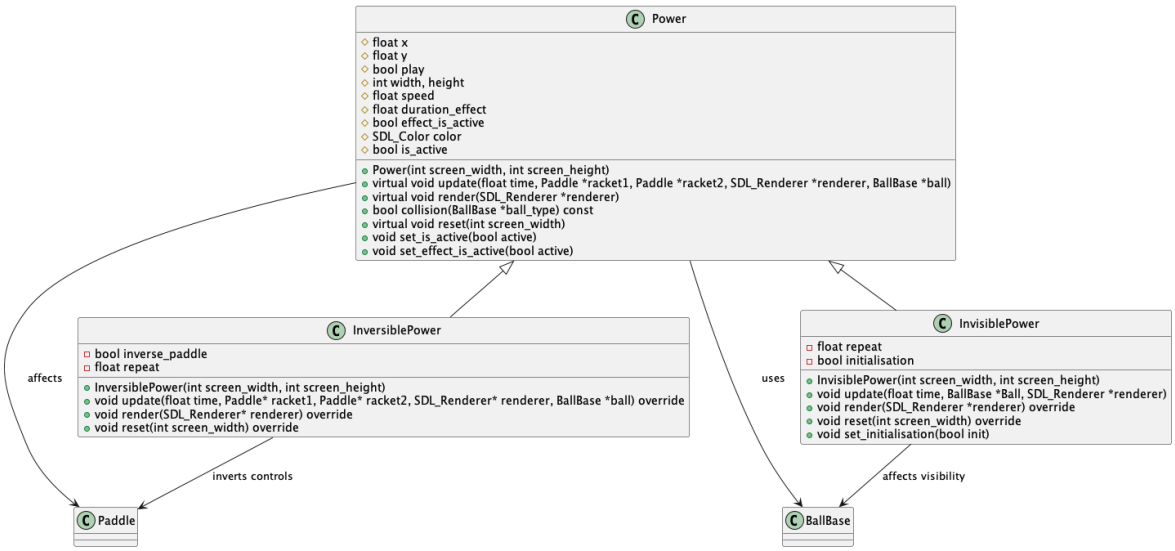


Pages (all)

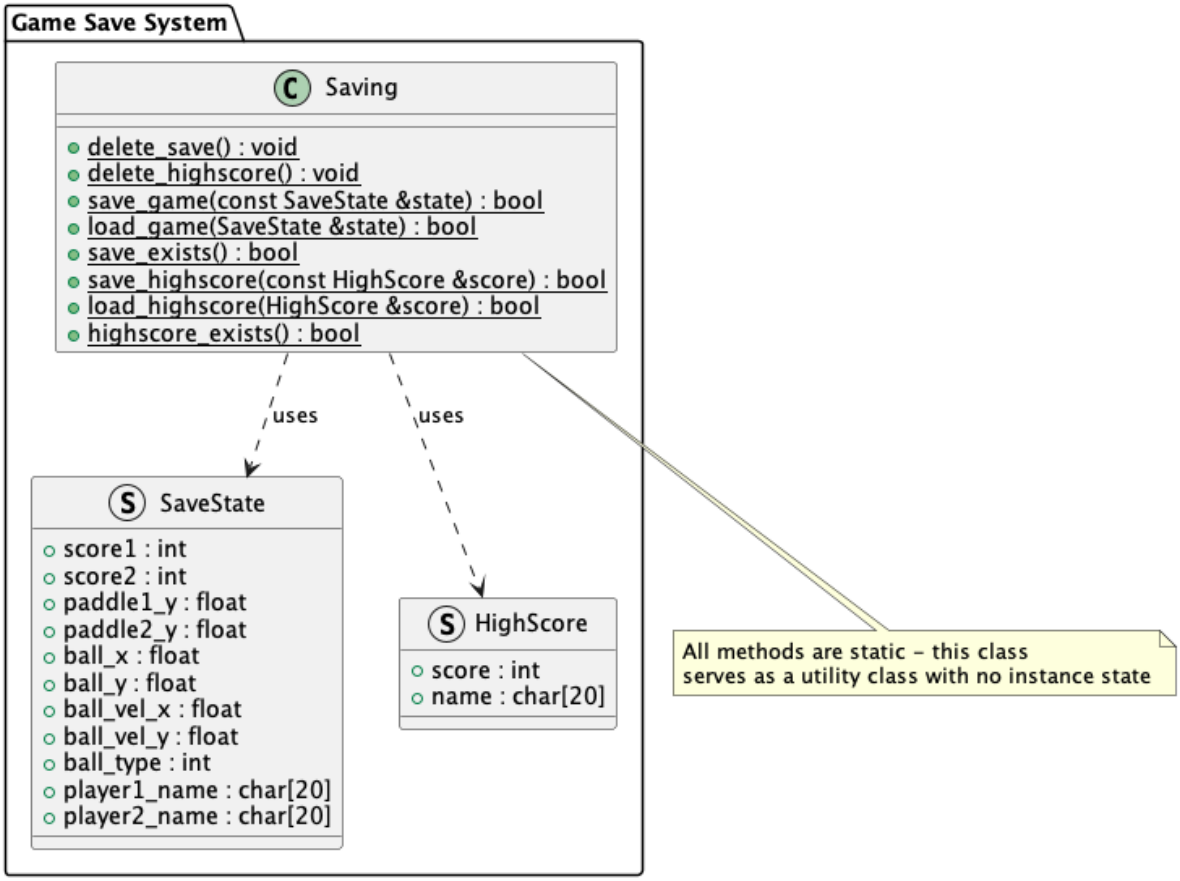


Class UML

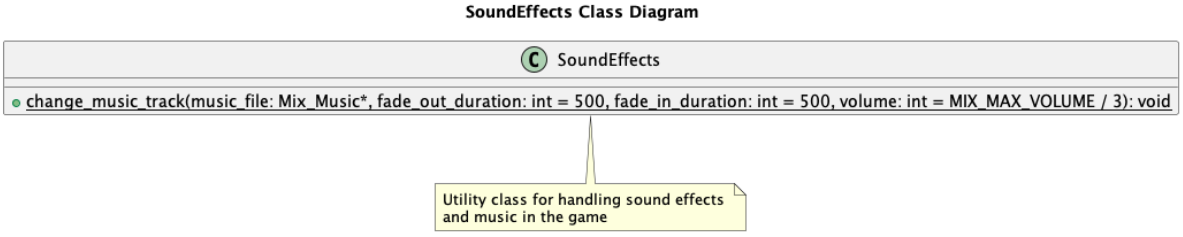
Powers (all)

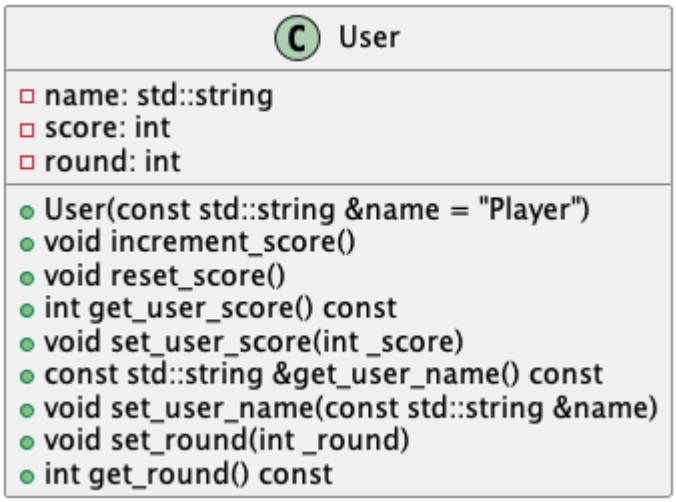


Save

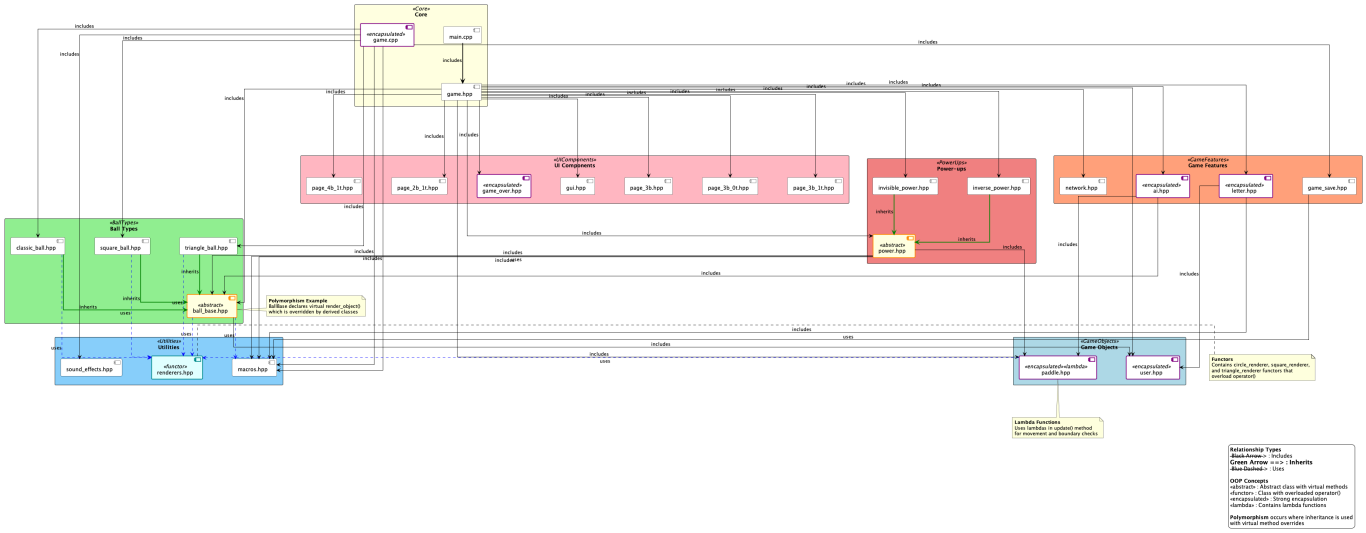


Sound



Class	UML
User	

Avec cette approche, chaque élément du jeu remplit un rôle bien défini et reste modulaire, ce qui permet d'ajouter de nouvelles fonctionnalités (comme le mode réseau) sans perturber l'ensemble du projet. Nous avons créé un diagramme UML pour les différentes dépendances entre les classes. De plus, les différentes notions de programmation utilisées dans chaque partie y sont indiquées.



Conclusion

Pong, mais en mieux ! Notre projet revisite ce grand classique du jeu vidéo en exploitant pleinement les principes de la programmation orientée objet, nous permettant de créer un code modulaire, extensible et maintenable.

Grâce à l'abstraction et à l'héritage, nous avons structuré notre jeu avec des interfaces claires et des hiérarchies logiques. Le polymorphisme nous a permis de manipuler différents objets de manière uniforme, tandis que l'encapsulation a assuré la protection et l'intégrité des données. Nous avons également tiré parti des foncteurs et des fonctions lambda pour encapsuler des comportements spécifiques, rendant notre implémentation plus souple et efficace.

Mais ce projet ne se limite pas à un simple exercice de programmation ! Nous avons voulu pousser l'expérience plus loin, en intégrant plusieurs modes de jeu inédits, un système de sauvegarde sécurisé, une

interface graphique fluide avec SDL, et même une tentative d'implémentation du multijoueur en réseau.

Le résultat ? 🎾 Un jeu fun, dynamique et personnalisable, qui vous permet de revivre l'expérience du Pong… mais avec une touche de modernité !

👉 **Prêt à relever le défi et à battre le high score ? Jouez, et montrez-nous qui est le véritable maître du Pong !**

Versions

Le versioning est un élément clé en programmation, assurant la cohérence des modifications et facilitant la collaboration. Il est aussi primordial pour la récupération de données en cas de perte ou corruption. Au fil du projet, nous avons créé différentes versions de notre code, chacune marquant une étape importante de son évolution. Cela nous a permis de suivre les progrès, d'intégrer de nouvelles fonctionnalités et d'effectuer des corrections de manière structurée.

- V4.2.2 Saving the demi correct views
- V4.2.3 The views logic has been completed
- V5.0.1: **Added lambda functions** on the paddle.cpp. The reason why we do this is found on the paddle.cpp file and added multiple notices support
- V5.1.1: **Functors added for the different ball shaped renders on the renderers.cpp file** (it includes also the explanation why we use functors [purposes])
- V5.1.2: Added getter and setter for the notices (it will be needed on the game's main logic to showcase the correct notices) + on Makefile added the mode_menu implementation
- V5.1.3: Prepared notices so that we can return back to the mode menu
- V5.1.4: Added logic for showing the pause button or not on the game
- V6.0.1: We added the change views functionality successfully
- V6.1.0: Added updates pages structures and logic for correct AI and balls selection to their respective modes
- V6.1.1: Changes GameState to game_state for normalisation reasons
- V6.1.2: Renamed files according to the pages structure below for clarity reasons. The classes on those files has not been updated yet
- V6.1.3: Updates class names according to the pages structures below and AI logic has been implemented. Needs to be added on the game's global logic
- V7.0.1: Added the AI on the global game's logic
- V7.1.2: Added game over page and Cmake structure. Changed some function names on the game.cpp and we added an automatic installer and handler of packages.
- V8.1.0: Added user class, started network codebase, added special effects support for game's actions and buttons click actions, updated saving state to include players names and updated the game's logic to support the definition of the player's names.
- V8.1.1: Added support for showing player's names when we play
- V9.0.1: Added high score functionality on two players mode. The game now ends when the users ask for it, so this is what makes a high score. Also added support to add user names on SDL interface
- V9.1.0: Updated the gui.cpp for better results and cohesion
- V9.2.0: Storymode added. We need to update the instructions
- V9.2.1: Removed setup.cpp and setup.hpp dependencies
- V9.2.2: Removed final dependencies of setup.cpp

- V10.8.23 : Implemented fun mode with rackets size changing and ball invisibility. Macros were extended, updated games logic to respond to the new criteria and fixed some bugs on the source code (+ 8.5 hours for it to work after the modifications of Dounia and Yanis)
- V10.8.24: Old makefile removed and gitignore updated. You will avoir à performer le cmake to compile. You can use directly the play.sh script
- V10.9.1: play.sh installer was updated
- V11.0.1: Added documentation structure. Fixed some issues on the fun mode. We need to reset the paddle height at the end of the fun mode manually if the button end game is tapped otherwise we risk to have different size paddles on other game modes
- V11.0.2: Added more comments and included inverse mode there is a segmentation fault issue
- V11.1.2: Resolved segmentation fault issue. Corrected some logic bugs and updated the documentation
- V12.0.1: Game has been completed. Some comments are missing on the inversible power file
- V13.1.5: Tests correction in oder to use CTest functionality. Documentation updated and added more details regarding the tests on the compte rendu (README). Also fixed some bugs on the letter class methodes.
