

GRP1 LIVRABLE

SOMMAIRE

| | |
|--|-----------|
| 1/-MEMBRES DU GROUPE 1..... | 3 |
| 2/-INTRODUCTION..... | 3 |
| 3/-DIAGRAMMES..... | 5 |
| 4/-ENVIRONNEMENT DE DÉVELOPPEMENT..... | 8 |
| 5/-PROGRAMME..... | 13 |
| 6/-TEST UNITAIRE..... | 28 |
| 7/-POUR ALLER PLUS LOIN (HORS DE LA GRILLE D'ÉVALUATION)..... | 37 |
| 8/-CONCLUSION..... | 45 |

1/-MEMBRES DU GROUPE 1

-Benkherouf Abdennacer

-Labane Yanis

-Amokrane Leith Allah

2/-INTRODUCTION

Dans le cadre de l'UE de Programmation Orientée Objet, ce projet consiste à concevoir et implémenter en C++ une version complète du *Jeu de la Vie* de John Conway. L'objectif est de démontrer une maîtrise réelle des principes de la POO à travers une architecture claire, modulaire et extensible. Le programme doit charger un état initial depuis un fichier, simuler l'évolution de la grille selon les règles de l'automate, et proposer deux modes d'exécution : un mode console générant les sorties textuelles, et un mode graphique reposant sur la bibliothèque SFML pour l'affichage dynamique.

Une attention particulière est portée à la qualité de la conception orientée objet : organisation des classes, séparation des responsabilités, utilisation du polymorphisme et anticipation d'éventuelles évolutions (comme la gestion torique ou de nouveaux types de cellules). La phase de conception s'appuie sur la réalisation des diagrammes UML demandés : classes, séquence et activité.

Le projet inclut également l'intégration de tests unitaires pour vérifier la validité des itérations, ainsi qu'une présentation orale visant à démontrer la compréhension complète du fonctionnement et des choix techniques. Réalisé en binôme (*trinôme dans notre cas*), ce travail doit refléter une démarche rigoureuse et autonome, conformément aux exigences du module.

Lien github du projet- [GRP1_GAME-OF-LIFE_POO](#)

Rappel règles du jeux de la vie:

1. Naissance

Une cellule **morte** devient **vivante** si elle a **exactement 3 voisines vivantes**.

2. Survie

Une cellule **vivante** reste **vivante** si elle a **2 ou 3 voisines vivantes**.

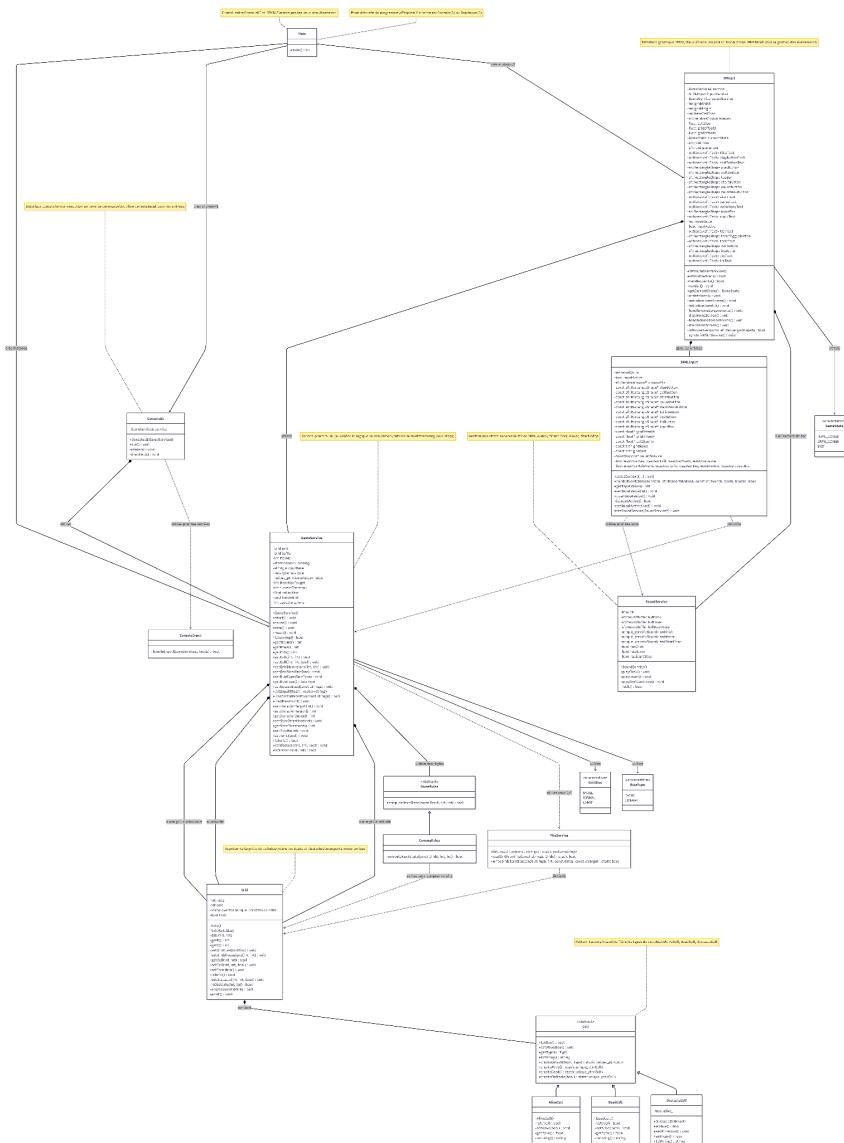
3. Mort

Une cellule **vivante** meurt :

- Si elle a **moins de 2 voisines vivantes** (*isolement*),
- Ou **plus de 3 voisines vivantes** (*surpopulation*).

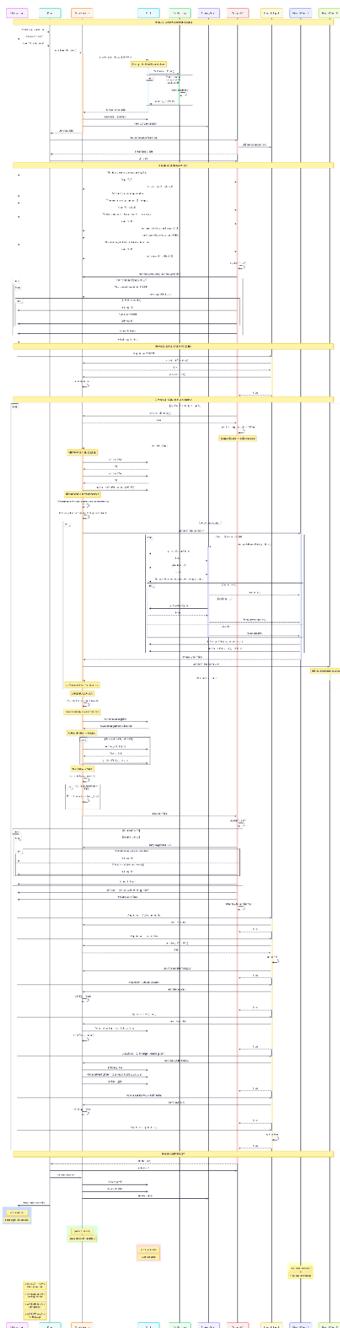
3/-DIAGRAMMES

Diagramme de Classes



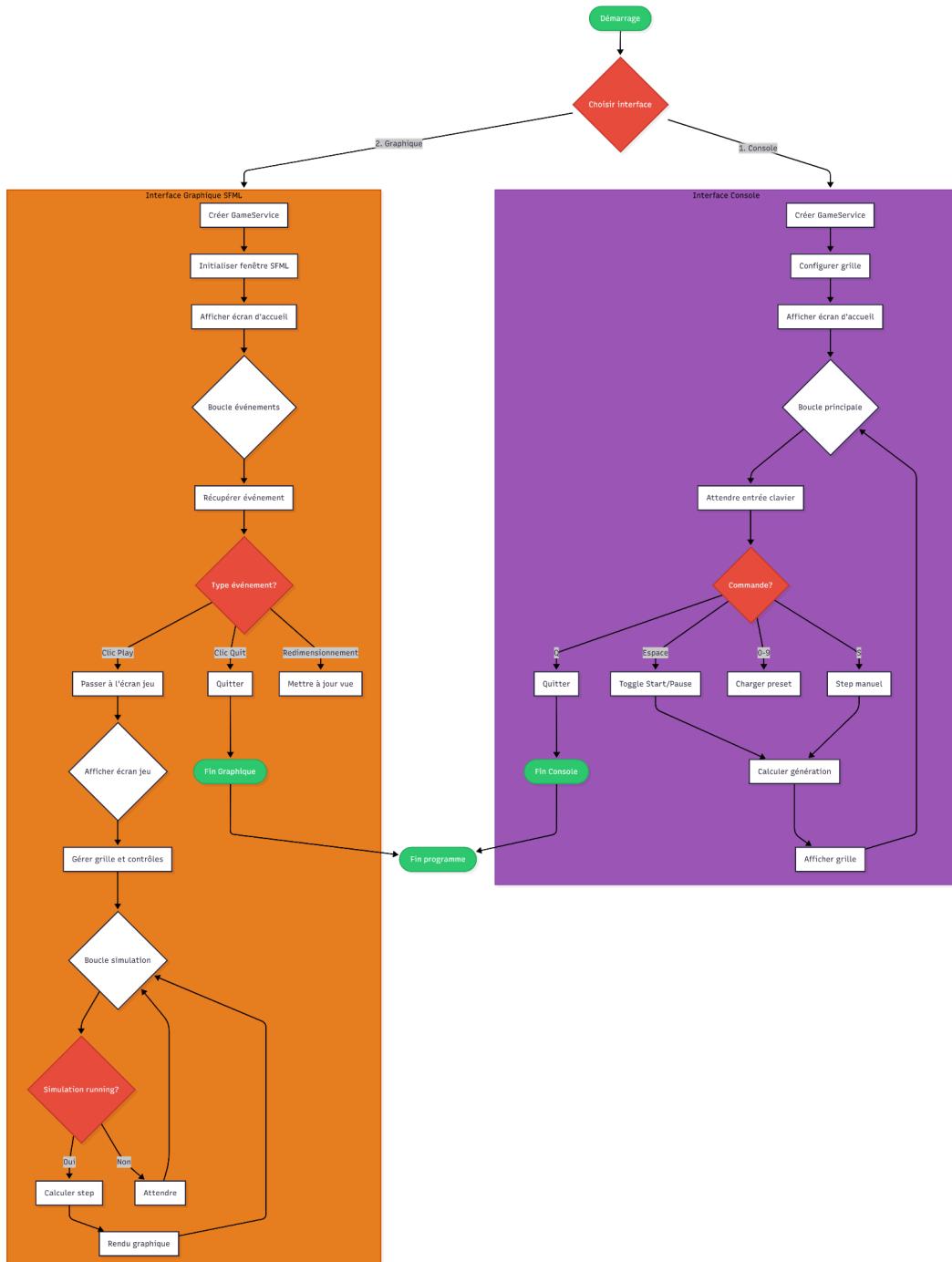
[Diagramme de Classes.svg](#)

Diagramme de Séquence



[Diagramme de Séquence.svg](#)

Diagramme d'Activité



[Diagramme d'Activité.svg](#)

4/-ENVIRONNEMENT DE DÉVELOPPEMENT

Ce projet repose sur plusieurs exigences techniques qui nécessitent la mise en place d'un environnement de développement adapté. Cette section présente les outils choisis ainsi que les étapes de configuration de l'environnement utilisé.

Principales exigences du projet :

1. Le programme doit être écrit en **langage C++**.
2. Le code doit respecter les principes de la **programmation orientée objet (POO)**.
3. L'application doit proposer **deux modes d'affichage** : un mode **console** et une **interface graphique** basée sur **SFML**.
4. Le développement doit être réalisé via un **repository GitHub**, afin de permettre le travail collaboratif et le suivi des versions.

Compte tenu de ces besoins, l'environnement de développement doit donc permettre :

- La compilation et l'exécution fiables de programmes **C++**.
- L'installation et l'intégration de la bibliothèque **SFML**.
- Une utilisation fluide de Git et GitHub pour le **travail en groupe**.

Notre choix : **Visual Studio Code**



Nous avons choisi **Visual Studio Code** comme environnement principal, car il répond pleinement à ces critères. Léger, extensible et compatible avec les outils nécessaires (*compilateurs C++, extensions Git, intégration SFML*), il offre une configuration flexible et adaptée à un projet orienté objet.

Les sections suivantes détaillent les différentes étapes de mise en place de l'environnement pour ce projet :

1 - Installation de Visual Studio Code

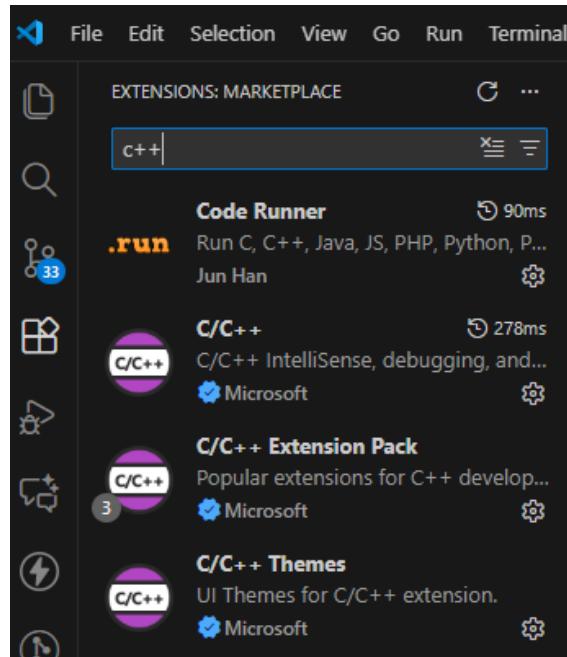
Télécharger l'éditeur depuis le site officiel :

<https://code.visualstudio.com/Download>

2 - Installation des extensions nécessaires pour C++ dans VS Code

Pour permettre la compilation, l'autocomplétion et le débogage en C++, il est essentiel d'installer les extensions suivantes :

- **C/C++** (Microsoft)
- **C/C++ Extension Pack** (optionnel mais recommandé)
- **CMake Tools** (si vous utilisez CMake)
- **GitLens** (pour faciliter le suivi du repository GitHub)



3 - Installation de SFML (version utilisée : 3.0.2)

Notre projet utilise **SFML 3.0.2**, il est donc indispensable de télécharger **la version exacte** pour garantir la compatibilité du code.

IMPORTANT :

- La syntaxe de SFML change fortement d'une version à l'autre. Utiliser la mauvaise version peut rendre le code incompatible.
- Chaque version de SFML est compilée avec une version spécifique du compilateur (MinGW, MSVC...).
- Pour éviter tout problème de compatibilité, il est recommandé de télécharger SFML et le compilateur associé directement depuis la page de téléchargement officielle.

⚠ The compiler versions have to match 100%!

If you want to use a MinGW package, it is *not* enough that the GCC versions seemingly match, you **have to** use one of the following matching compilers:

- WinLibs UCRT 14.2.0 (32-bit)
- WinLibs UCRT 14.2.0 (64-bit)

| 32-bit | 64-bit |
|--|--|
| Visual C++ 17 (2022) - Download 33.7 MB | Visual C++ 17 (2022) - Download 36.9 MB |
| GCC 14.2.0 MinGW (DW2) (UCRT) - Download 34.1 MB | GCC 14.2.0 MinGW (SEH) (UCRT) - Download 36.3 MB |

Lien de téléchargement SFML :

<https://www.sfml-dev.org/fr/download/>

4 - Configuration des chemin d'Include SFML dans VScode

Même si SFML et le compilateur compatible sont installés sur la machine, il est nécessaire d'indiquer à VS Code où trouver les fichiers d'en-tête et les bibliothèques SFML. Sans cette configuration, VS Code ne pourra ni compiler le projet, ni reconnaître correctement les fonctions de la bibliothèque.

Cette étape consiste à configurer les fichiers suivants dans le dossier `.vscode` du projet :

- `c_cpp_properties.json` permet d'ajouter les chemins `include` de SFML afin que VS Code reconnaisse les fichiers d'en-tête lors de l'écriture du code.
- `tasks.json` indique au compilateur où trouver les fichiers `.lib` ou `.a` de SFML pour la compilation et l'édition de liens.
- `launch.json` configure l'exécution et le débogage du programme.

En général, il faudra spécifier :

- Le chemin vers le dossier **SFML/include**
- Le chemin vers le dossier **SFML/lib**
- Et, au besoin, les bibliothèques SFML à lier (par exemple : `sfml-graphics`, `sfml-window`, `sfml-system`, etc., selon la version).

La configuration exacte dépend :

- Du compilateur utilisé (MinGW/GCC ou MSVC).
- De la manière dont les dossiers SFML ont été organisés dans le projet.
- Et de la plateforme (Windows / Linux / macOS).

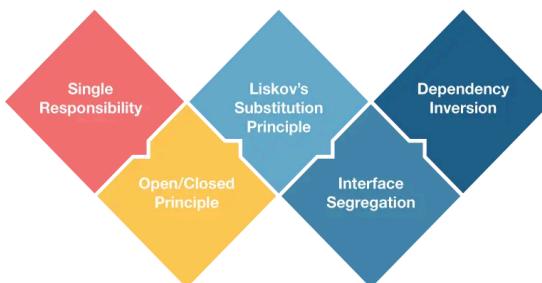
5/-PROGRAMME

Notre programme a été conçu pour répondre strictement à l'ensemble des exigences techniques et fonctionnelles spécifiées dans l'énoncé du projet. Il s'appuie sur une architecture claire, modulaire et orientée objet, garantissant robustesse, extensibilité et conformité aux principes SOLID.

Conformité aux spécifications techniques

Exploiter les concepts de POO et les principes SOLID

S.O.L.I.D.



Comment le programme y répond :

Le code est structuré en plusieurs classes ayant chacune une responsabilité unique :

- `Grid` gère la structure de la grille,
- `Cell` (et ses dérivées) modélise le comportement des cellules,
- `GameService` orchestre l'évolution du jeu,
- `GameRules` et `ConwayRules` appliquent les règles de transition,
- `SFMLUI` affiche l'état du jeu,
- `SFMLInput` gère les interactions utilisateur.

Cette organisation respecte notamment :

- **Le Single Responsibility Principle**

Le Single Responsibility Principle (SRP), ou Principe de Responsabilité Unique, stipule qu'une classe (ou un module, un service) ne devrait avoir qu'une seule raison de changer, signifiant qu'elle doit avoir une responsabilité unique et bien définie, regroupant des fonctionnalités étroitement liées à ce rôle unique.

- **L'Open/Closed Principle** (règles et types de cellules extensibles),

Le Principe Ouvert/Fermé (Open/Closed Principle - OCP) stipule qu'une entité logicielle (classe, module, fonction) doit être ouverte à l'extension mais fermée à la modification, permettant d'ajouter de nouvelles fonctionnalités sans changer le code existant grâce à l'abstraction et au polymorphisme (héritage/interfaces)

- **Le Dependency Inversion Principle** (interface GameRules, déconnexion logique UI).

Le Principe d'Inversion des Dépendances (DIP) est un principe de conception logicielle (partie des principes SOLID) qui stipule que les modules de haut niveau ne doivent pas dépendre des modules de bas niveau ; les deux doivent dépendre d'abstractions (interfaces ou classes abstraites), et les abstractions ne doivent pas dépendre des détails, mais les détails (implémentations concrètes) doivent dépendre des abstractions, ce qui rend le code plus modulaire, flexible et testable.

Pourquoi :

L'application devient plus simple à maintenir, tester et faire évoluer (nouvelles règles, nouveaux comportements de cellule, autre interface graphique éventuelle, etc.).

Jeu / Grille / Cellules en classes séparées

Comment :

- `src/Core/Grid.*` gère l'accès aux cellules, le voisinage et le stockage.
- `src/Core/Cell.*` contient une hiérarchie polymorphe (`AliveCell`, `DeadCell`, `ObstacleCell`).

```
class Cell {  
public:  
    enum class Type { Normal, Obstacle };  
    virtual ~Cell() = default;  
  
    // Abstract interface  
    virtual bool isAlive() const = 0;  
    virtual void setAlive(bool a) = 0;  
    virtual Type getType() const = 0;  
    virtual std::string toString() const = 0;  
  
    // Factories  
    static std::unique_ptr<Cell> createDefault(bool alive = false, Type t =  
Type::Normal);  
    static std::unique_ptr<Cell> createAlive();  
    static std::unique_ptr<Cell> createDead();  
    static std::unique_ptr<Cell> createObstacle(bool alive = false);  
};  
  
// Concrete AliveCell - always alive (declaration only)  
class AliveCell : public Cell {  
public:  
    AliveCell();  
    ~AliveCell() override;  
    bool isAlive() const override;  
    void setAlive(bool a) override;  
    Type getType() const override;  
    std::string toString() const override;  
};  
  
// Concrete DeadCell - always dead (declaration only)
```

```

class DeadCell : public Cell {
public:
    DeadCell();
    ~DeadCell() override;
    bool isAlive() const override;
    void setAlive(bool a) override;
    Type getType() const override;
    std::string toString() const override;
};

// Concrete ObstacleCell - can be alive or dead (declaration only)
class ObstacleCell : public Cell {
public:
    explicit ObstacleCell(bool alive = false);
    ~ObstacleCell() override;
    bool isAlive() const override;
    void setAlive(bool a) override;
    Type getType() const override;
    std::string toString() const override;
private:
    bool alive_;
};

```

- `src/Services/GameService.*` applique `GameRules` pour générer la nouvelle grille.

Pourquoi :

Séparer clairement la représentation (grid) de l'état (cell) et de la logique (service) permet un code propre, lisible et testable.

Hiérarchie polymorphe pour les cellules

Comment :

La grille stocke des `std::unique_ptr<Cell>` permettant d'héberger tout type de cellule. Les sous-classes redéfinissent leur comportement (ex. cellule obstacle non modifiable).

```
class Grid {  
  
private:  
  
    int rows;  
  
    int cols;  
  
    std::vector<std::vector<std::unique_ptr<Cell>>> cells;  
  
    // toric (wrap-around) behavior  
  
    bool toric = false;  
  
    // Méthode privée pour initialiser la grille  
  
    void initializeGrid();  
  
public:  
  
    // toric control  
  
    void setToric(bool t);  
  
    bool isToric() const;  
  
    // obstacle accessors  
  
    void setObstacle(int x, int y, bool obs);
```

```
bool isObstacle(int x, int y) const;

// deep equality check for stabilization detection

bool equals(const Grid &other) const;

public:

// Constructeur par défaut (grille NORMAL 20x20)

Grid();

// Constructeur avec taille prédefinie

Grid(GridSize size);

// Constructeur avec dimensions personnalisées

Grid(int r, int c);

~Grid();

int getR() const;

int getC() const;
```

```

void setR(int r);

void setC(int c);

// Méthode pour redimensionner la grille avec une taille prédefinie

void setGridSize(GridSize size);

// Méthode pour redimensionner la grille avec des dimensions précises

void setGridDimensions(int r, int c);

bool getCell(int x, int y) const;           // retourne l'état d'une cellule

void setCell(int x, int y, bool state); // modifie l'état d'une cellule

// copy semantics (deep copy)

Grid(const Grid &other);

Grid& operator=(const Grid &other);

void print() const;                      // affichage console pour test

};


```

Pourquoi :

Facilite l'ajout de nouveaux types sans modifier la structure existante respecte OCP.

Règles de transition isolées dans une hiérarchie indépendante

Comment :

- `GameRules` = interface
- `ConwayRules` = implémentation
`GameService` utilise un pointeur vers `GameRules` (Pattern Strategy).

Le pattern Strategy est un modèle de conception (design pattern) comportemental en programmation orientée objet (POO) qui permet de sélectionner un algorithme ou un comportement à exécuter au moment de l'exécution (runtime) plutôt qu'au moment de la compilation.

Pourquoi :

On peut remplacer les règles sans changer le moteur du jeu (ex : variantes du Jeu de la Vie).

Découplage métier / graphique

Comment :

Classes du dossier `Core` et `GameService` n'ont aucune dépendance à SFML.
`SFMLUI` se contente d'afficher ce que fournit `GameService`.

Pourquoi :

Permet un **mode console**, des **tests unitaires**, et une UI évolutive sans toucher au moteur.

Utilisation de C++ standard, STL et SFML

- C++17 pour bénéficier de RAI, `unique_ptr`, `vector`, etc.
- SFML pour la fenêtre, l'affichage et les entrées utilisateur.
- Code optimisé pour éviter fuites mémoire et duplication inutile.

Efficacité et robustesse

- Détection de stabilisation pour arrêter tôt.
- Double buffer pour éviter les corruptions.
- Gestion propre de la mémoire par `unique_ptr`.
- Vérification d'erreurs (fichier manquant, ressources).
- Possibilité de parallélisation selon la taille de la grille.

Conformité aux spécifications fonctionnelles

Lecture du fichier d'entrée

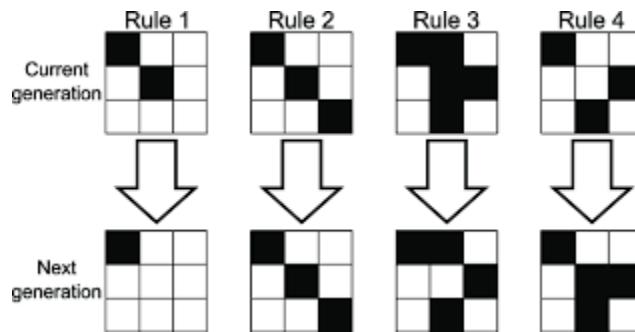
`FileService` lit et interprète le fichier initial, crée la grille et initialise chaque cellule.

Example:

```
5 10  
0 0 1 0 0 0 0 0 0 0  
0 0 0 1 0 0 0 0 0 0  
0 1 1 1 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0
```

Évolution à chaque itération

`GameService::step()` applique `GameRules` pour produire une nouvelle génération en suivant les règles de Conway.



Arrêt du programme

L'exécution s'arrête :

- Soit lorsque deux grilles successives sont identiques (État Stable).
- Soit lorsque le nombre max d'itérations est atteint.

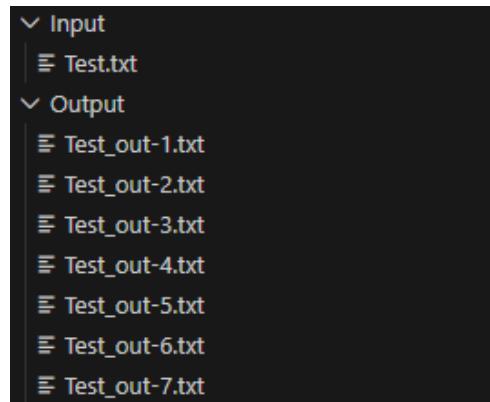
Mode Console

A screenshot of a terminal window within a code editor interface. The terminal tab is active at the top. The window displays the initial state of a Game of Life grid as binary digits (0s and 1s) and various control commands. The controls listed are: Space=start/pause, s=step, r=reset, 0-9=load preset, 1/2/3=size (S/N/L), c=toggle rule, +/-=speed, and q=quit. The tick interval is set to 200 ms, and the game is currently running.

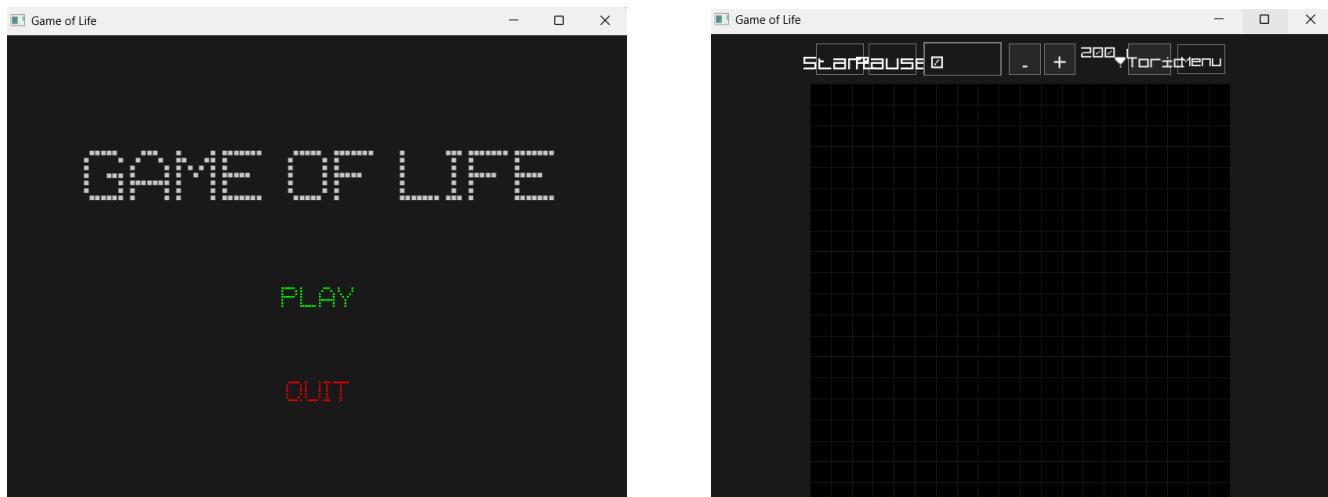
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

001000000
000100000
011000000
000000000
000000000
Controls: Space=start/pause  s=step  r=reset  0-9=load preset  1/2/3=size (S/N/L)  c=toggle rule  +/-=speed  q=quit
Tick(ms): 200  Running: No
[]
```

À chaque étape, l'état de la grille est exporté dans des fichiers structurés dans `<nomfichier>_out/`.



Mode Graphique (SFML)

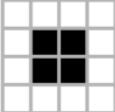
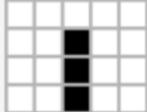
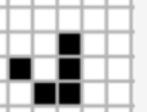
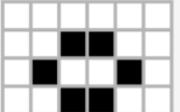
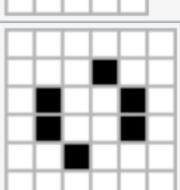
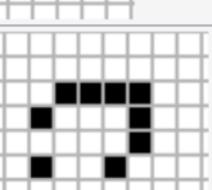
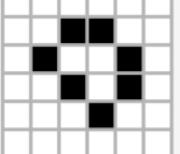
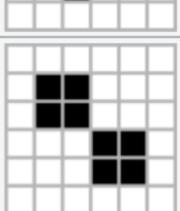
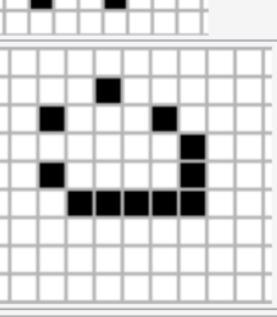
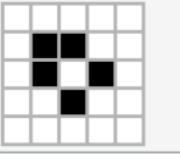
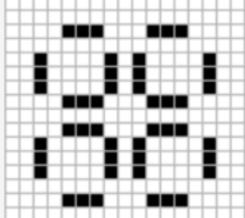
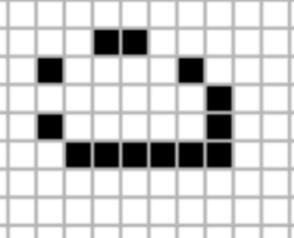
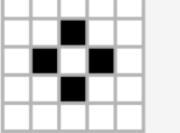
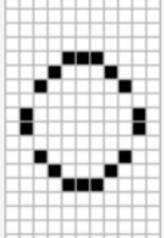


Affichage dynamique, grille animée, vitesse ajustable, interaction utilisateur.

Tests unitaires

Tests validant :

- Oscillateurs (blinker)
- Motifs stables (block)
- Comportement torique
- Obstacles
- Cohérence I/O.

| Still lifes | | Oscillators | | Spaceships | |
|-------------|--|--------------------------------|---|--------------------------------|--|
| Block |  | Blinker (period 2) |  | Glider |  |
| Bee-hive |  | Toad (period 2) |  | Light-weight spaceship (LWSS) |  |
| Loaf |  | Beacon (period 2) |  | Middle-weight spaceship (MWSS) |  |
| Boat |  | Pulsar (period 3) |  | Heavy-weight spaceship (HWSS) |  |
| Tub |  | Penta-decathlon (period 15) |  | | |

Extensions (bonus)

Grille torique

Indices de voisins gérés modulo la taille de la grille.

Cellules obstacles

Cellule dérivée immuable (ne change jamais d'état).

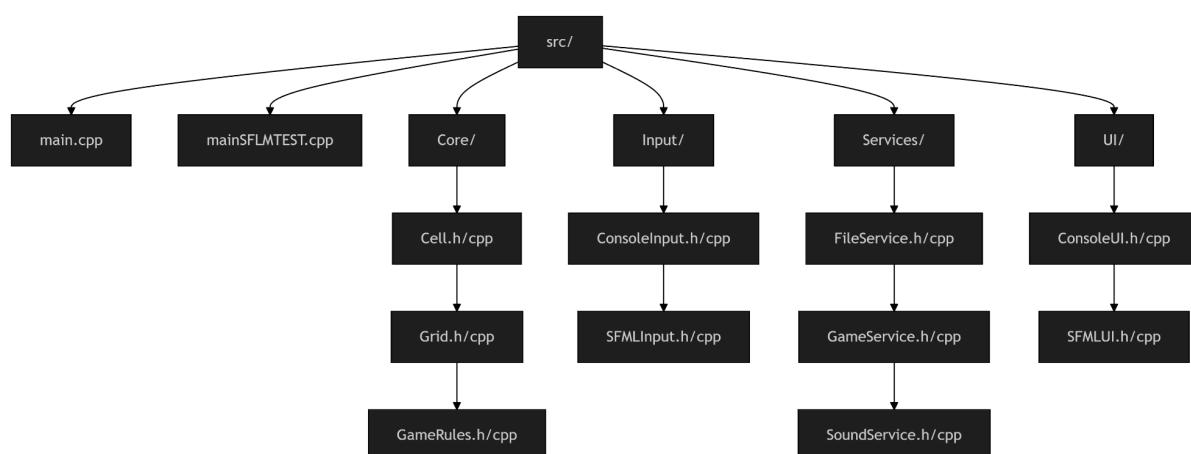
Motifs prédéfinis

Chargement de presets (glider, pulsar, etc.) via clavier / menu.

Parallélisation

Découpage par lignes et traitement en threads simultanés.

Structure du code source



Le projet est divisé en **couches** :

1. Core (Noyau métier)

- `Cell.*` — cellule polymorphe
- `Grid.*` — grille et voisins
- `GameRules.*` — règles du jeu

2. Input (Entrées utilisateur)

- `ConsoleInput.*`
- `SFMLInput.*`

3. Services (Logique applicative)

- `GameService.*` — moteur de simulation
- `FileService.*` — lecture/écriture fichiers

4. UI (Affichage)

- `ConsoleUI.*`
- `SFMLUI.*`

5. main.cpp

- Lit les paramètres.
- Initialise services et dépendances.
- Sélectionne le mode d'exécution.
- Lance la simulation.

6/-TEST UNITAIRE

Les tests unitaires constituent un élément essentiel du projet. Ils permettent de valider la conformité du moteur de simulation, de garantir l'absence de régressions et d'assurer que les fonctionnalités principales ainsi que les extensions respectent fidèlement le comportement attendu du Jeu de la Vie. Cette section décrit les outils utilisés, les scénarios testés et la méthodologie appliquée.

Démonstration:

```
.\bin\test_game.exe
[DEBUT] clignotant
Clignotant - etape 1 (attendu vs obtenu):
EXPECTED:
00000
00100
00100
00100
00000
ACTUAL:
00000
00100
```

```
00100
00100
00000
Clignotant - etape 2 (attendu vs obtenu):
EXPECTED:
00000
00000
01110
00000
00000
ACTUAL:
00000
00000
01110
00000
00000
[FIN] clignotant (4 ms)
[DEBUT] bloc_stable
Bloc stable (attendu vs obtenu):
EXPECTED:
0000
0110
0110
0000
ACTUAL:
0000
0110
0110
0000
[FIN] bloc_stable (1 ms)
[DEBUT] toric_wrap
Toric wrap (attendu vs obtenu):
EXPECTED:
000
000
001
ACTUAL:
000
```

```
000
001
[FIN] toric_wrap (1 ms)
[DEBUT] obstacle_preserved
Obstacle preserved (attendu vs obtenu):
EXPECTED:
00000
00100
00A00
00000
00000
ACTUAL:
00000
00100
00A00
00000
00000
[FIN] obstacle_preserved (3 ms)
[DEBUT] entree_sortie_fichier
File IO - attendu vs obtenu:
EXPECTED:
101
0A0
100
ACTUAL:
101
0A0
100
[FIN] entree_sortie_fichier (3 ms)
=====
=====
All tests passed (10 assertions in 5 test cases)
```

Framework utilisé

Choix du framework : Catch2

catchorg/Catch2

A modern, C++-native, test framework for unit-tests, TDD and BDD - using C++14, C++17 and later (C++11 support is in...)

389 Contributors 383 Issues 20k Stars 3k Forks

<https://catch2.org/> <https://github.com/catchorg/Catch2>

- Le projet utilise **Catch2**, intégré sous forme de single-header dans : `tests/catch.hpp`.
- **Pourquoi ce choix** : Catch2 est simple à intégrer (un seul fichier), lisible, moderne et idéal pour des tests unitaires C++ sans configuration complexe.
Il permet d'écrire des tests expressifs et fournit des rapports détaillés en cas d'échec.

Organisation et emplacement des tests

- L'ensemble des tests est regroupé dans :
`tests/test_game.cpp`
(avec le header Catch2 dans `tests/catch.hpp`).
- Le binaire de tests est généré automatiquement en :
`bin/test_game.exe`
via le script de compilation `Compile.bat`.

Objectifs des tests fournis

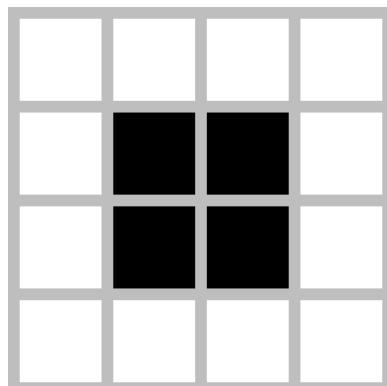
Les tests vérifient plusieurs comportements fondamentaux du Jeu de la Vie ainsi que des extensions optionnelles :

Motifs classiques

- **Blinker** : oscillateur à période 2 (évolution correcte sur deux étapes).

| | | | | | | | |
|-------------------|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 2 | 1 | 0 | 0 |
| 0 | 1 | 1 | 2 | 1 | 1 | 0 | 0 |
| 0 | 1 | 2 | 3 | 2 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Blinker (state 1) | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Blinker (state 2) | | | | | | | |

- **Block** : motif stable (ne doit jamais évoluer).



Fonctionnalités avancées

- **Voisinage torique** : les bords sont connectés (wrap-around).
- **Cellules obstacles** : immuables, ne changent jamais d'état.
- **Lecture / écriture (FileService)** : vérification d'un aller-retour (`load` → `save`) cohérent.

Chaque test compare un **EXPECTED** (résultat attendu) avec un **ACTUAL** (résultat produit par l'algorithme), et affiche les deux en cas d'erreur pour faciliter le diagnostic.

| Still lifes | | Oscillators | | Spaceships | |
|-------------|--|--------------------------------|--|--------------------------------|--|
| Block | | Blinker (period 2) | | Glider | |
| Bee-hive | | Toad (period 2) | | Light-weight spaceship (LWSS) | |
| Loaf | | Beacon (period 2) | | Middle-weight spaceship (MWSS) | |
| Boat | | Pulsar (period 3) | | Heavy-weight spaceship (HWSS) | |
| Tub | | Penta-decathlon (period 15) | | | |

Méthodologie d'écriture des tests

- Les tests se concentrent **exclusivement sur la logique métier** (`Grid`, `Cell`, `GameRules`, `GameService`).
- Aucun composant SFML n'est initialisé ; les tests sont indépendants de l'interface graphique.
- Utilisation de `REQUIRE()` et `CHECK()` pour les assertions.
- Les comparaisons sont faites via une **représentation textuelle** de la grille, facilitant la lecture EXPECTED vs ACTUAL.

Exécution des tests

Depuis la racine du projet, sous Windows PowerShell :

```
.\Compile.bat  
.\\bin\\test_game.exe > bin\\test_results.txt
```

- `Compile.bat` compile automatiquement les tests et l'exécutable principal.
- `bin/test_results.txt` contient l'intégralité des résultats, y compris les grilles textuelles comparées.

Format des résultats

Pour chaque scénario, le rapport inclut :

- Une description du test.
- **EXPECTED**: Suivi de la grille attendue.
- **ACTUAL** : Suivi du résultat réel.
- Un résumé Catch2 (tests passés / échoués).

Ce format facilite énormément l'analyse en cas d'échec.

Périmètre et isolation des tests

- Les tests couvrent **uniquement la logique métier**, garantissant portabilité et rapidité.
- Aucune dépendance à SFML ou à l'interface graphique.
- **FileService** est testé via des fichiers temporaires ou relatifs.

Comment ajouter un test

1. Ouvrir `tests/test_game.cpp`.
2. Ajouter un bloc :
`TEST_CASE("Nom du test", "[tag]").`
3. Initialiser une grille (**Grid**) ou un moteur (**GameService**).
4. Exécuter `step()` autant de fois que nécessaire.
5. Comparer EXPECTED vs ACTUAL avec `REQUIRE()`.
6. Recomplier et relancer les tests via `Compile.bat`.

Bonnes pratiques

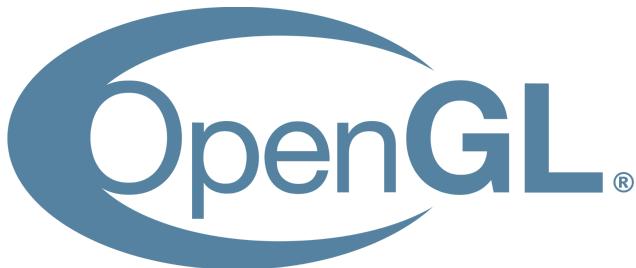
- Tester **un seul comportement à la fois.**
- Garder les tests **rapides** (simulation courte).
- Ne pas coupler les tests à l'interface graphique.
- Utiliser des utilitaires/fixtures pour générer des grilles plus facilement.
- En cas d'erreur, extraire un test minimal isolant le problème.

Débogage en cas d'échec

- Consulter **EXPECTED** vs **ACTUAL** pour comprendre la divergence.
- Vérifier si l'erreur concerne les règles, le torique, les obstacles ou la lecture fichier.
- Exécuter uniquement le test concerné via les tags Catch2 pour accélérer l'itération.

7/-POUR ALLER PLUS LOIN (HORS DE LA GRILLE D'ÉVALUATION)

Pour tester les limites de la faisabilité et surtout aller plus loin on peut créer un moteur de jeux 3D en combinant SFML avec OpenGL

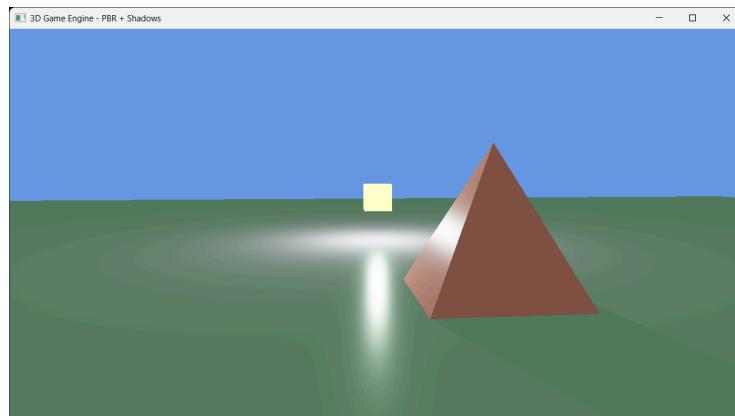


Projet — Moteur 3D expérimental (SFML + OpenGL)

Pour tester les limites de la faisabilité et surtout aller plus loin on peut créer un moteur de jeux 3D en combinant SFML avec OpenGL.

Voici la suite une présentation claire et synthétique du **projet général**, ses objectifs, son architecture, son usage et les points d'attention à garder en tête.

[3D-game-engine-demo](#)

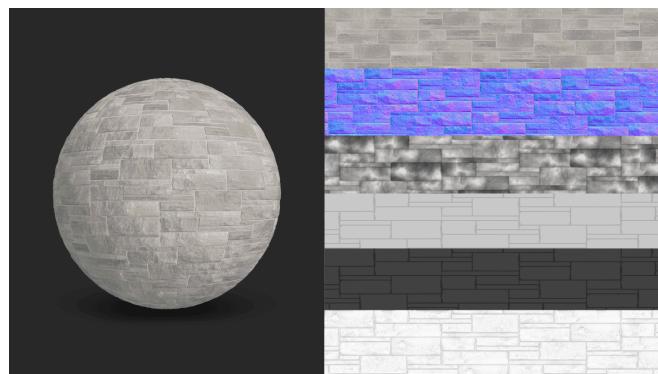


Résumé rapide

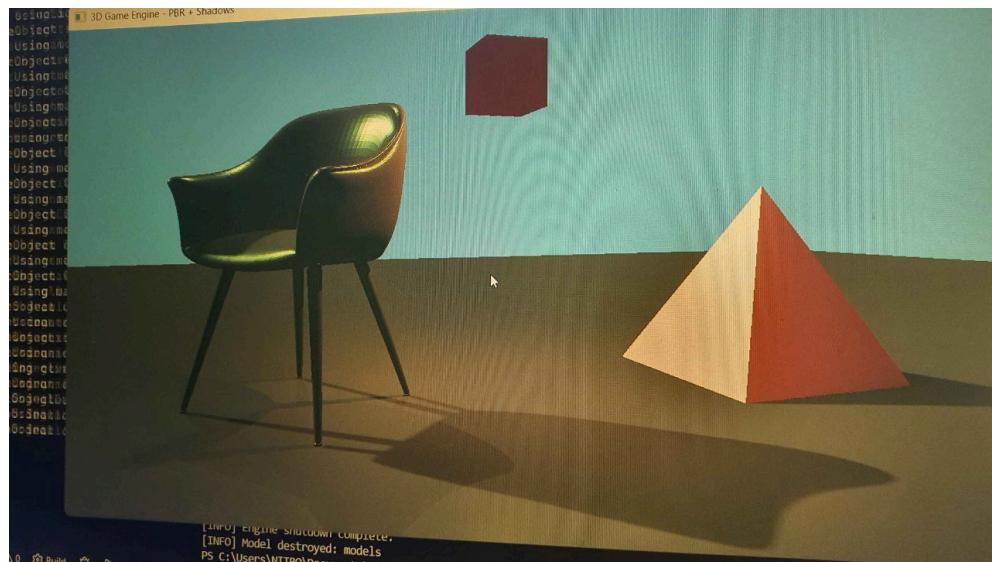
Ce projet est un moteur/mini-démo 3D en C++ moderne (C++17) destiné à l'apprentissage et à l'expérimentation. Il illustre une pipeline de rendu physically-based (PBR), la gestion d'ombres (directionnelle / spot / point), le chargement de modèles (Assimp / glTF), la gestion de textures (fichiers et textures embarquées), une structure de scène simple (GameObject / Scene) et des contrôles d'entrée/caméra basés sur SFML (création de contexte OpenGL + événements). L'objectif n'est pas d'être un produit commercial mais un banc d'essai pédagogique et technique.

Objectifs et usages

- **Apprendre** les bases et détails pratiques du rendu PBR et de l'intégration OpenGL dans une boucle d'application moderne.



- **Expérimenter** avec le shadow mapping (directionnel, spot, cubemap pour lights ponctuelles) et les ressources GPU (FBOs, cubemaps).



- **Valider** des techniques de chargement de modèle (Assimp / glTF), y compris la manipulation de textures embarquées et des transforms de textures.

assimp/ assimp_view

The Asset-Importer-Lib Viewer



3
Contributors

5
Issues

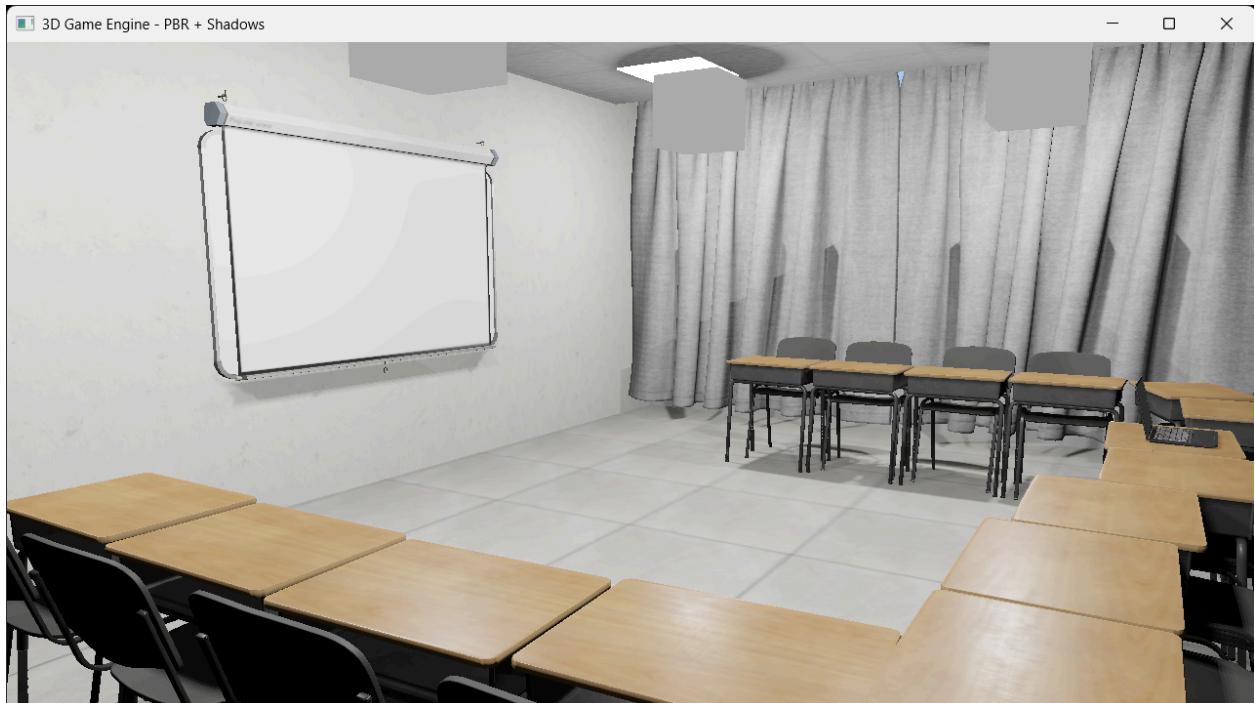
2
Discussions

56
Stars

18
Forks



- **Servir** de base pour prototypes (visualisation, rendu réaliste, debug shaders) et comme point de départ pour migration vers une build system plus robuste (CMake).



Fonctionnalités principales

- Pipeline PBR pour matériaux (albedo, metallic, roughness, ao, emission).
- Shadow mapping complet : directional, spot et point-light (cubemap).
- Chargement de modèles via Assimp (glTF/.glb support, embedded textures).
- TextureManager capable de décoder textures depuis la mémoire (stbi) et de gérer des placeholders sûrs en cas d'erreur.
- Structure de scène simple : `Scene`, `GameObject` (Model ou ManualMesh) et calculs de bounding pour culling/split d'ombres.

- Système d'input basé sur SFML : event loop, capture de la souris, style caméra première personne.
- Logging/diagnostic simple pour aider le débogage shader et chargement de modèles.

Architecture (haut niveau)

- **Core** : orchestration de l'application (création de fenêtre, boucle, gestion du temps, initialisation des sous-systèmes).
- **Graphics** : renderer (initialisation GL, passes de rendu), shader manager, texture manager, classes Model/Mesh, gestion des lights et des shadow maps.
- **Scene** : gestion des objets, création/chargement de modèles, calculs de bounds.
- **Input** : capture SFML, recentrage de souris, génération de deltas pour la caméra.
- **Resources** : shaders (pbr, shadow, point_shadow, debug quad...), modèles d'exemple et textures.
- **Utilitaires** : logger, gestion des bibliothèques tierces (glad, glm, stb_image).

Points techniques notables

- Export de variables pour forcer GPU discret sur laptops (NvOptimus / AmdPowerXpress).
- Shadow map directionnelle par défaut très grosse (ex. 8192) cela peut dépasser les limites GPU ; il est conseillé d'utiliser 2048-4096 pour la portabilité.
- Textures embarquées dans glb décodées depuis mémoire via stbi; fallback robuste en cas d'échec.

- Le Renderer lie des unités de texture réservées pour shadow maps pour éviter d'écraser les textures de matériaux.
- Beaucoup de logs/printfs présents : utiles pour debug mais à filtrer pour un usage production.

Prérequis et compilation (usage général)

- Compilateur supportant C++17 (g++, clang++ ou MSVC avec adaptation).
- OpenGL moderne (drivers GPU à jour).
- Bibliothèques : SFML (window/context + input), Assimp (chargement de modèles).
- Chargeur GL (glad) et utilitaires headers-only (GLM, stb_image).
- Un système de build (Makefile fourni ; pour un projet plus durable, migrer vers CMake est fortement recommandé).

Exécution et comportements à l'exécution

- L'application crée un contexte OpenGL via SFML et gère la boucle principale (update + render).
- Input : souris capturée et recentrée chaque frame ; déplacement caméra en "first-person" mais avec hauteur d'œil constante.
- Chargement de scène : modèles et meshes sont chargés à l'initialisation ; la scène calcule des bounding et configure des lights.
- Debug : shaders de debug (fullscreen quad) permettent d'inspecter albedo / textures / maps.

Limitations connues & risques

- **Taille des shadow maps** : valeurs trop élevées provoquent échec d'allocation ou OOM sur GPU modestes.
- **Chemins contenant des espaces** : peuvent casser scripts/makefiles éviter les dossiers avec espaces.
- **Linking statique (-static)** : complexifie l'environnement d'exécution sur Windows ; pour le développement préférez le linking dynamique.
- **Robustesse du loader** : modèles corrompus / textures énormes peuvent provoquer des erreurs les checks sont présents mais prudence nécessaire.
- **Verbosité des logs** : beaucoup de `std::cout` dans le code ; envisager un système de niveaux pour limiter le bruit.

Améliorations recommandées / évolutions possibles

- **Migration CMake** : meilleure portabilité, simplification du linking SFML/Assimp et intégration Visual Studio.
- **Réglages runtime** : exposer tailles d'ombre, niveaux de qualité et binding units via config ou interface debug.
- **Système de log** : implémenter niveaux (DEBUG/INFO/WARN/ERROR) et option d'écriture dans un fichier.
- **Tests automatisés** : petits tests d'intégration (chargement multiple de modèles, stress test textures) pour attraper fuites/mémoire.
- **Profiling GPU/CPU** : intégrer timers et compteurs pour identifier goulets d'étranglement (draw calls, passes shadow).

- **Pipeline d'assets** : outils pour normaliser l'échelle et les conventions de matériaux entre modèles tiers (plutôt que hacks ad-hoc).
- **UI debug** : overlay ImGui pour toggler lights, tailles d'ombre, visualiser textures et logs shaders en temps réel.

Cas d'usage pédagogiques

- Comprendre comment organiser un renderer moderne (passes, ressources, gestion d'états).
- Étudier le shadow mapping avancé (cubemap pour lights ponctuelles, gestion d'artefacts).
- Apprendre les pièges du chargement de modèles glTF (embedded textures, texture transforms).
- Expérimenter PBR et correct binding de textures multiples dans un shader.

Check-list rapide pour démarrer

1. Installer un compilateur moderne et drivers GPU à jour.
2. Installer SFML et Assimp, ou configurer via CMake s'il est ajouté.
3. Vérifier que les chemins de bibliothèques n'ont pas d'espaces.
4. Choisir des tailles d'ombre raisonnables (ex. 2048/4096) pour commencer.
5. Lancer l'exécutable depuis le dossier racine (les ressources doivent être relatives à la racine d'exécution).
6. Utiliser les shaders de debug pour valider que les textures / normales / maps sont chargées correctement.

8/-CONCLUSION

Ce projet nous a permis d'appliquer de manière concrète les fondamentaux de la programmation orientée objet à travers l'implémentation complète du Jeu de la Vie de Conway. La conception et le développement en C++ ont mis en évidence l'importance d'une architecture bien structurée, où la modularité, la clarté et l'extensibilité sont au cœur des décisions techniques. La gestion du chargement d'un état initial, l'évolution de la grille ainsi que la double interface console et graphique via la bibliothèque SFML ont constitué un terrain d'expérimentation riche pour mettre en pratique les principes de responsabilité unique, de polymorphisme et de séparation des rôles.

La production des diagrammes UML, la mise en place de tests unitaires et la préparation de la présentation orale ont contribué à renforcer notre compréhension globale du système, tout en garantissant la robustesse et la cohérence de notre solution. Réalisé en trinôme, ce travail a également été l'occasion de développer une approche collaborative rigoureuse et autonome. Au final, ce projet constitue une expérience formatrice qui consolide nos compétences en POO et nous prépare à aborder des développements plus complexes dans la suite du cursus.

FIN DU DOCUMENT