

# Compte Rendu du Projet C++

Noé Dubois et Matteo Garzella

January 2023



## Table des matières

<b>1 Description du jeu</b>	<b>3</b>
<b>2 Explication du code</b>	<b>6</b>
<b>3 Procédure d'Utilisation et d'Exécution</b>	<b>7</b>
<b>4 Fiertés et Commentaires</b>	<b>8</b>

# 1 Description du jeu

Torch est un jeu de réflexion sur l'univers des jeux olympiques antiques. Il se présente sous la forme d'une série de niveaux contenant 2 personnages, dont l'un tient la torche olympique. Le but est de réussir à faire sortir la torche du niveau pour s'approcher d'Athènes. Pour réaliser cette objectif, les joueurs peuvent appuyer sur des plaques de pressions liées à des portes. Pour éviter d'avoir besoin de se tenir sur les plaques, il est possible de poser la torche ou des jarres disponibles dans les niveaux sur celles-ci, ce qui les maintiens enfoncées.



FIGURE 1 – Ecran de sélection des niveaux

Quand le jeu s'allume, un écran de sélection permet de choisir un niveau

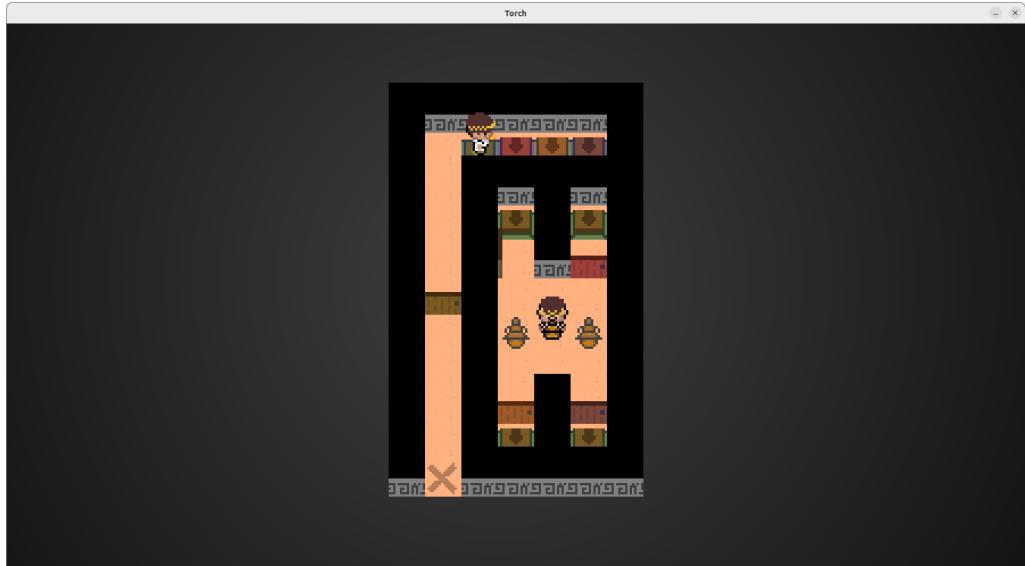


FIGURE 2 – Niveau V

Les niveaux contiennent 2 personnages, dont l'un tient une torche.

- Le personnage brun se déplace à l'aide des commandes Z Q S D.
  - Le personnage blond se déplace à l'aide des commandes O K L M.
- Quand un joueur tient un objet (torche ou jarre), il peut le lâcher ou le donner. Si le personnage est

sur une case adjacente à l'autre, et le regarde, presser la touche donnera l'objet à l'autre personnage, sinon, presser la touche permet de poser l'objet par terre.

- Le personnage brun lache/donne son objet à l'aide de la commande A.
- Le personnage blond lache/donne son objet à l'aide de la commande I.



FIGURE 3 – Passage de la torche par un mur percé

Certains niveaux contiennent des murs persés, à travers lesquels il est possible de se faire passer un objet. Le personnage donneur doit regarder le mur percé, et appuyer sur la touche de don alors que l'autre personnage se trouve de l'autre côté du mur.



FIGURE 4 – Récupération d'une jarre

Pour récupérer un objet (torche ou jarre) posé par terre, il suffit de se déplacer dessus avec les mains vides.

## 2 Explication du code

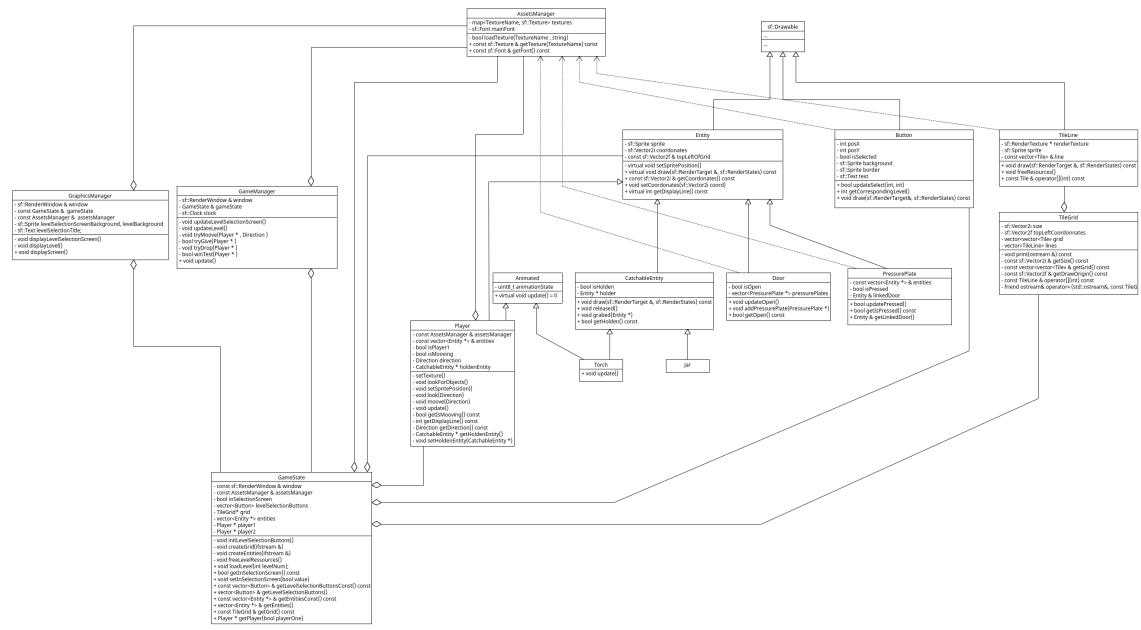


FIGURE 5 – diagramme UML complet

Le diagramme UML est disponible à la racine de l'archive git.  
Cette partie vise à décrire l'architecture générale du programme. Pour une vue plus précise de son fonctionnement, tous les fichiers sources sont commentés.

Une fois lancé, la fonction main du programme ne se charge que de quelques choses :

- Changer le répertoire de travail
- Ouvrir la fenêtre du jeu
- Instancier les objets uniques nécessaires au fonctionnement du jeu (AssetsManager, GameState, GameManager, GraphicsManager)
- Entretenir la boucle principale du jeu en appelant à tour de rôle GameManager.update() et GraphicsManager.displayScreen()

Les 4 objets uniques mentionnés ci-dessus sont importants. Leur classes sont décrites dans les fichiers .hpp du même nom.

- **AssetsManager** : Charge et garde en mémoire les différents assets du jeu (textures et police d'écriture)
- **GameState** : Contient l'état courant du jeu, ainsi que les méthodes nécessaires au chargement d'un niveau.
- **GameManager** : Met à jour l'objet Gamestate, en suivant l'avancée du temps, et les événements découlant de l'utilisateur.
- **GraphicsManager** : Affiche le rendu du jeu à l'écran, en lisant l'objet GameState.

L'objet GameState Contiens une série de champs regroupant toutes les informations sur le jeu :

- **levelSelectionButtons** : La liste des boutons de selection de niveau du menu principal
- **grid** : La grille de tuiles (Mur, mur percé, chemin, objectif) qui constitue la base d'un niveau.
- **entities** : La liste des entités (Joueur, porte, plaque de pression, jarre, torche) qui peuplent un niveau
- **player1 et player2** : Des liens directs vers les entités joueur, qui sont aussi contenus dans entities.

Tous les champs ci dessus correspondent à des objets dont les classes sont décrites dans les fichiers Button.hpp, Entity.hpp et Grid.hpp. Ces objets contiennent notamment des méthodes pour "simplifier

la vie" des objets GameManager et GraphicsManager. Par exemple, les entités héritent de la classe sf : :Drawable, ce qui permet de surcharger la méthode "draw" permettant d'afficher un objet sur une surface de rendu donnée. Ce tour général de l'organisation du programme donne une idée générale de la façon dont il fonctionne.

Les contraintes du projet ont toutes été respectées :

- **8 classes minimum** : On est largement bon
- **3 niveaux d'héritage minimum** : Sans compter les héritages faits sur la classe sf : :Drawable de la bibliothèque, le programme contient bien 3 niveaux de hiérarchie. En effet, la classe Entity a pour enfants Door, PressurePlate, Player et CatchableObject, qui elle-même a pour enfant Torch et Jar. Ces héritages sont justifiés, car, d'une part la classe mère Entity permet du polymorphisme au niveau du traitement de la liste globale des entités (ex : afficher toutes les entités, sans se soucier de leur type), et d'autre part, la classe CatchableObject permet du polymorphisme au niveau des objets que peuvent tenir les joueurs (donner une torche à l'autre joueur ou donner une jarre reviens au même).
- **2 fonctions virtuelles différentes** : Sans compter la surcharge de la fonction sf : :Drawable : :draw de la bibliothèque, la classe Entity contient 2 fonctions virtuelles. D'une part, getDisplayLine permet au GraphicsManager de savoir quand afficher une entité (la grille s'affiche ligne par ligne, pour garder une impression de perspective). La version de base de cette fonction renvoie juste la coordonnée y des entités, mais Player étant une entité qui peut se déplacer, et s'afficher entre 2 tuiles, la fonction doit être surchargé pour envoyer une valeur permettant de conserver un affichage propre. D'autre part, setSpritePosition permet de calculer les coordonnées dans la fenêtre où afficher l'entité, et mettre à jour le sprite de cette entité en conséquence. La version de base de cette méthode fait un calcul très simple à partir des coordonnées de l'entité, mais Player pouvant être affiché entre 2 tuiles lors d'un déplacement, cette fonction est surchargée pour permettre ce calcul.
- **2 surcharge d'opérateurs** : La classe TileGrid contient 2 opérateurs surchargés. L'opérateur [] permet de directement récupérer une ligne de la grille (représentée par un objet TileLine). L'opérateur « à » a été surchargé pour que la grille puisse se décrire en une chaîne de caractère. Ainsi, appeler cout « grid permet d'afficher la grille dans la console. Cette fonctionnalité n'est pas utile pour le jeu final, mais est très utile pour les debugs.
- **2 conteneurs STL** : La classe AssetsManager contient une map qui fait le lien entre un énumérateur (TextureName) et les textures chargées. La classe GameState contient un vecteur d'entités appelé entity qui contient toutes les entités d'un niveau.

### 3 Procédure d'Utilisation et d'Exécution

Pour démarrer ce projet, nous nous sommes basé sur le premier tutoriel officiel de SFML, qui fournit un template github d'un projet de base construit avec CMake (<https://www.sfml-dev.org/tutorials/2.6/start-cmake-fr.php>). Nous n'avons eu qu'à apporter quelques modifications au fichier CMakeLists.txt pour avoir une base de projet solide.

Pour compiler le projet, vous avez donc besoin d'installer la librairie SFML, par exemple, pour les distributions Linux basées sur Debian, en utilisant

```
sudo apt update
sudo apt install \
    libxrandr-dev \
    libxcursor-dev \
    libudev-dev \
    libopenal-dev \
    libflac-dev \
    libvorbis-dev \
    libgl1-mesa-dev \
    libegl1-mesa-dev \
    libdrm-dev \
    libgbm-dev
```

Puis, il faut utiliser CMake pour générer le projet (explications dans le tuto SFML si besoin).

Une fois cela fait, l'exécutable se trouve dans le répertoire buil/bin/ et s'appelle Torch. Il suffit de lancer le programme depuis un terminal, et le jeu est utilisable. (Remarque : vérifiez que l'executable se trouve biens dans le même répertoire que le dossier Assets, ou le jeu ne s'ouvrira pas).

## 4 Fiertés et Commentaires

Ce premier projet complet en C++ à été le théâtre de beaucoup de problèmes inattendus, d'expérimentations, et de comprehension sur les notions de POO. Au final, le temps étant la principale limite, nous avons produit un projet dont nous sommes fier, avec certaines parties très intéressantes, mais l'expérience acquise nous montre aussi que ce projet aurait pu être mieux construit. Dans cette partie, nous allons explorer ces 2 aspects.

Pour commencer, nous allons exposer la partie dont nous sommes le plus fier : l'affichage de la grille des tuiles (fichiers Grid.hpp et Gric.cpp). Une approche naïve de l'affichage de la grille serait d'entretenir un tableau de tuiles élémentaires, et d'afficher le sprite de chacune de ces tuiles à la bonne position à chaque itération de la boucle principale. Cette approche est fonctionnelle, mais nous avons trouvé une solution plus intéressante.

Pour conserver une notion de perspective, les éléments à l'écran doivent être affichés dans le bon ordre, et notamment, la grille doit être affichée ligne par ligne (pour que les joueurs passent "derrière" un mur, il faut afficher le joueur, puis le mur qui est devant lui). SFML fourni un objet sf::RenderTexture, qui permet de faire un prérendu, qui pourra plus tard être affiché. Cet objet permet au moment du chargement du niveau, de créer une RenderTexture pour chaque ligne de la grille, qui permet de générer un sprite pour toute la ligne, une fois pour toute, en faisant un prérendu de l'affichage de toutes les tuiles de la ligne. De cette façon, à chaque itération de la boucle principale, on affiche une série de gros sprites, plutôt qu'afficher une multitude de petits sprites, ce qui constitue une optimisation avec SFML (il faut réduire au maximum les appels à la fonction draw). Pour mieux comprendre ce système, le code commenté est la meilleure solution.

Quand nous avons eu l'idée de Torch, nous avons vraiment été très motivé pour le réaliser, et ce que nous vous rendons est plus proche pour nous d'un *proof of concept* que de ce que nous avons en tête. En effet, nous pensons que le concept de ce jeu est assez bon, et qu'il serait possible de le produire+ une version bien plus jolie graphiquement, et contenant beaucoup d'autres mécaniques (la torche permet de mettre le feu à certaines entités, présence d'eau dans laquelle on peut nager si on pas d'objet, présence de vent qui éteint la torche si on est pas à l'abris, etc...). Cependant, nous nous rendons compte qu'il serait difficile de développer tout cela sur la base du rendu que nous vous fournissions. Cela nous fait réaliser que notre architecture de programme n'est sûrement pas la meilleure. Cette expérience nous donne donc une meilleure idée de ce qu'est un projet développé avec les principes de la POO, et quelles difficultés l'on peut rencontrer en cours de route.