ANNÉE 2021 - 2022

2281.1 Projet P2 SA – Rapport technique – ISC2il-b

Poutine Style

Créer un jeu s'inspirant de Wolfenstein 3D, en utilisant le principe du Raycasting pour simuler une impression de 3D, à l'aide de la librairie graphique SFML.

Étudiants participant à ce travail :

Alessio Comi, ISC2il-b
Nicolas Aubert, ISC2il-b
Théo Vuilliomenet, ISC2il-b

Présenté à :

Benoit Le Callennec
Julien Senn
Aïcha Rizzotti

Restitution du rapport : **25.01.2022**

Période : **2021 – 2022**

École : **HE-Arc, Neuchâtel**

haute école arc ingénierie
neuchâtel berne jura www.he-arc.ch

1 - Abstract

PoutineStyle, un projet, un jeu, une volonté, une révolution.

Nonobstant l'enjeu que peuvent apporter les nouvelles technologies dans le monde du jeu vidéo, nous allons ici nous concentrer sur du rétrogaming.

Pionniers dans le domaine, les premiers Wolfenstein et Doom 3D sont devenus des incontournables, dictant les bases du FPS (jeu de tir à la première personne) moderne si bien que, pendant de nombreuses années, ce terme fut remplacé par celui de « DoomLike ». Ici, nous allons détailler toutes les étapes de création de notre projet, qui, malgré l'utilisation d'un environnement de développement contemporain, tente de refléter les techniques du début des années 1990.

Table des matières

ANNÉE 2021 - 2022

2 - Glossaire

Discord	Réseau social, adapté à la communication textuelle.
FPS (Jeu)	*First Person Shooter*, jeu de tir à la première personne
FPS / IPS	*Frames Per Second*, Images Par Seconde. Nombre de fois que l'affichage est rafraîchi en l'espace d'une seconde.
Frame	Une itération de la gameLoop. Si une seule itération est effectuée en l'espace d'une seconde, le nombre de FPS est à 1.
GameLoop	Boucle principale de notre jeu.
Git	Gestionnaire de version.
Loot	Récompenses qu'un ennemi vaincu laisse à terre, ou les objets que le joueur obtient en ouvrant un coffre.
Parser	Parcourir le contenu d'un fichier texte pour en extraire des éléments.
Raycasting	Technique permettant un rendu 3D à partir d'un monde en 2D.
Sound design	L'adaptation sonore.
Sprite	Partie d'une texture qui sera affichée.

3 - Introduction

Ce rapport va vous présenter les diverses étapes de création du projet Poutinestyle, de son origine jusqu'à sa réalisation.

Ledit projet a pris place dans le cadre du P2 et aura duré le premier semestre de deuxième année à la HE-Arc de Neuchâtel, à raison de 4 périodes par semaine. Nous avons eu pour but de réaliser un projet complet, de thème libre, comprenant la création du cahier des charges, mais aussi sa conception, sa planification ainsi que sa réalisation.

La décision de faire un jeu a été prise dans deux buts, premièrement afin d'approfondir nos connaissances dans le monde du jeu vidéo et notamment son commencement. De ce fait, nous nous sommes imposé diverses contraintes, dans le but de répondre au mieux à nos interrogations, tel que la création de la gameLoop, de créer un rendu 3D en utilisant la technique du raycasting et donc pas de moteur de jeu préconstruit.

Un autre de nos objectifs, plus secondaire ce coup-ci, était de construire un univers graphique et sonore cohérent ainsi qu'une narration.

Ce document n'a pas pour but de passer en détail chaque ligne de code, mais d'accompagner le lecteur dans notre démarche de création. Au travers de ce dernier, nous présenterons l'analyse de notre cahier des charges.

Dans un second temps, nous détaillerons conception et planification, suivie de la réalisation à proprement parler. Pour finir, nous conclurons en vous présentant les résultats obtenus ainsi que les limitations et perspectives de ce projet.

4 - Analyse

4.1 - Spécifications détaillées

4.1.1 -Les menus (Accueil et GameOver)

Nous allons créer une fenêtre d'accueil dans l'ambiance et la même ligne graphique que notre jeu. Elle contiendra un grand bouton central "Jouer" et deux autres boutons pour "Quitter" et "Option".

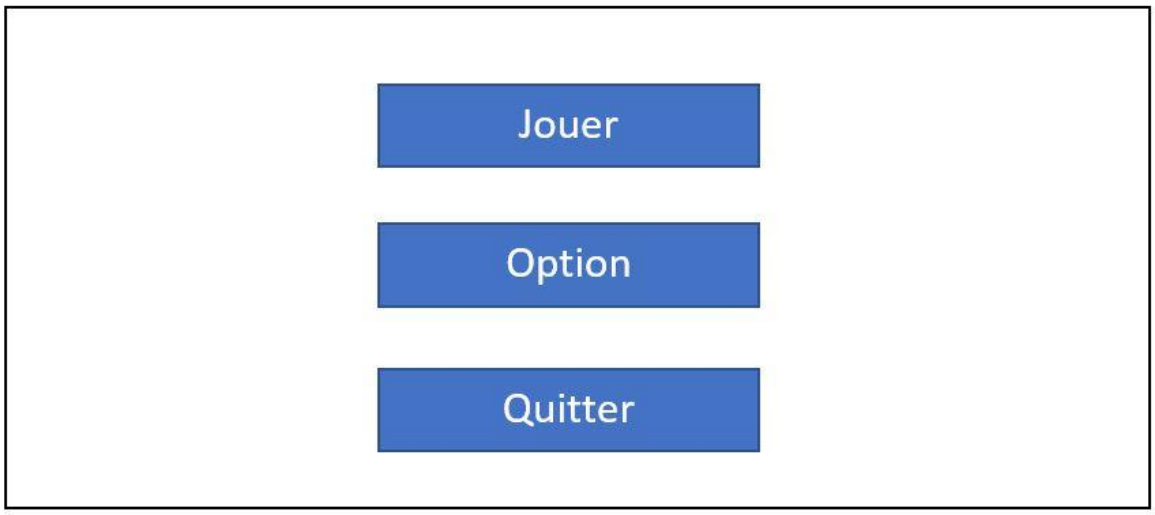


Figure 1 : Maquette du menu principal

Le bouton "Option" permettra les configurations suivantes :

- Le choix de la map
- Le choix de la langue du jeu (Russe ou Russe traduit en Français)
- Le choix de la difficulté du jeu (sans talent ou fait moi mal)

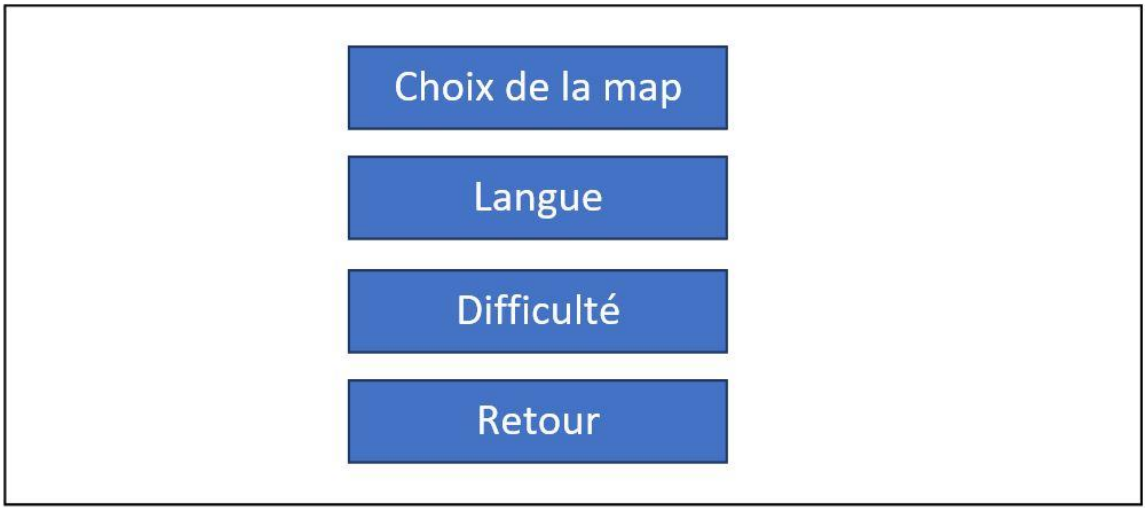


Figure 2 : Maquette du menu paramètres

Pour ce qui est du menu GameOver, nous allons simplement indiquer au joueur s'il a accompli sa mission ou s'il a malheureusement échoué avec aussi la possibilité de rejouer ou de quitter le jeu.

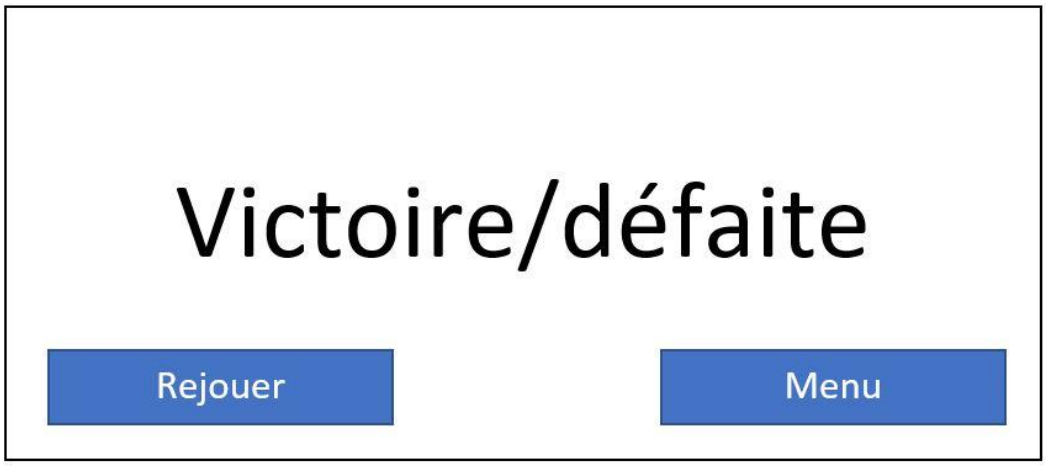


Figure 3 : Maquette du menu de victoire / défaite

4.1.2 -Modifications dans les menus

Le premier menu qui fait office de launcher de notre jeu n'a pas changé. Cependant, nous avons modifié les options.

Elles permettent dorénavant les réglages suivants :

- Le choix de la map
- L'affichage des métadonnées telles que le nombre de FPS
- La sensibilité de la souris
- La difficulté du niveau
- Le volume sonore

Voici un aperçu de ce à quoi les options ressemblent :

Figure 4 : Menu des paramètres (actuel)

4.1.3 - La carte

Nous pensons créer une carte 2D qui ressemblerait à un labyrinthe avec des salles plus grandes dans lesquelles le joueur serait confronté à des ennemis. Les murs auront des textures différentes et des entités seront ajoutées à la map. Voici, ci-dessous un exemple de carte 2D :

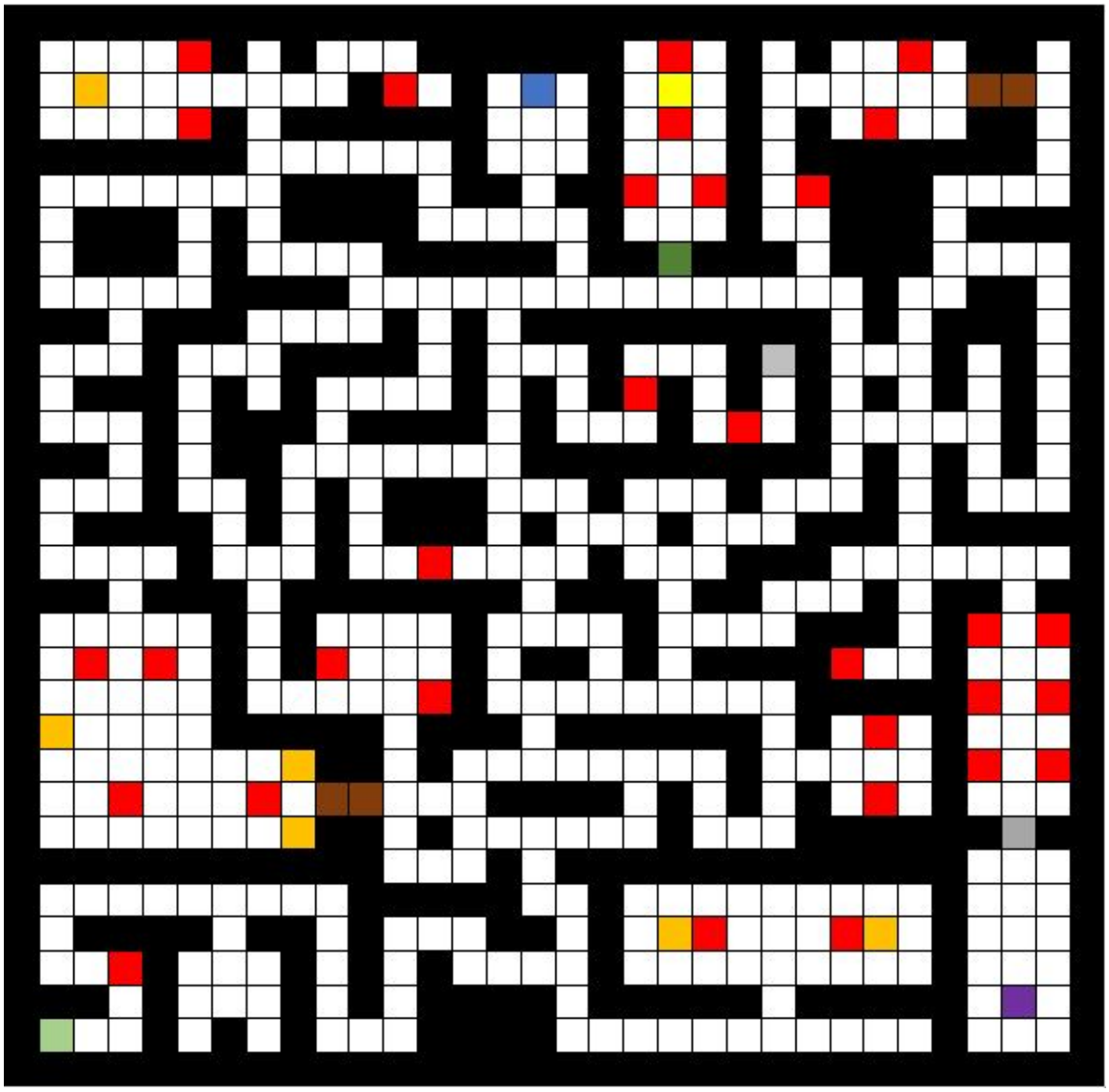


Figure 5 : Schéma d'une carte

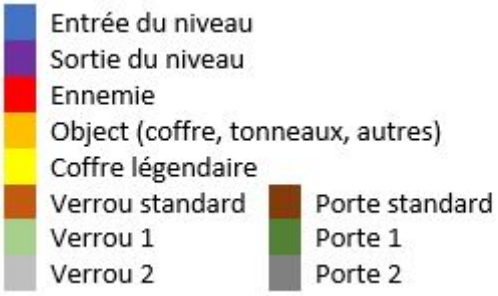


Figure 6 : Légendes d'une carte

4.1.4 -Modification dans la carte

Nous avons développé une grande quantité de textures (murales) ainsi que diverses entités ce qui nous a donc amenés à revoir notre façon d'encoder les map.

4.1.5 - Le gameplay

Le joueur aura comme mission d'aller délivrer un membre du KGB. Pour ce faire, il devra parcourir le labyrinthe à la recherche de sa cible tout en se défendant face aux Américains qui voudront le tuer.

Figure 7 : Maquette du jeu

Interactions dans le jeu :

- Déverrouiller des portes à distance pour débloquer des parties du labyrinthe (les paires verrou-porte sur la carte 2D)
- Déverrouiller des portes à sens unique (les doubles portes brunes sur la carte 2D)
- Viser et potentiellement tuer les ennemis que vous rencontrerez (les points rouges dans la carte 2D)
- Les ennemis pourront aussi vous tirer dessus et faire baisser votre barre de vie.
- Récupérer le super-loot qui vous redonne des munitions et de la vie (le point jaune dans la carte 2D)
- Terminer le jeu en parlant un PNJ final (point violet sur la carte 2D)

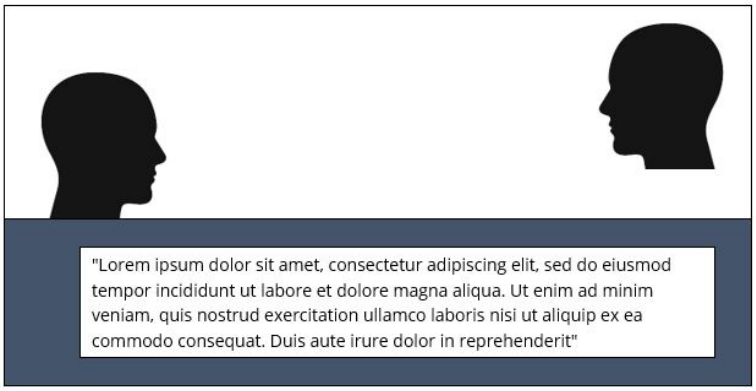


Figure 8 : Maquette narration

4.1.6 - Modifications dans le gameplay

Les interactions ont été revues pour améliorer l'expérience utilisateur. Voici la nouvelle liste des interactions possibles dans notre jeu.

- Récupérer des clés, les stocker dans un inventaire et être en mesure de déverrouiller la porte associée
- Pas de double porte (ça n'apportait pas grand-chose au gameplay)
- On peut tirer et tuer les ennemis que l'on rencontre
- Les ennemis peuvent aussi vous tirer dessus et descendre votre vie
- Interagir avec diverses entités (armes, soin, munitions, clés ...) et les récupérer
- Passer au niveau suivant ou terminer le jeu en prenant un ascenseur

4.1.7 - Les commandes

Notre jeu sera une application Windows (.exe) et sera jouable uniquement avec clavier et souris. Le jeu n'étant encore qu'à ses débuts, nous ne pouvions pas donner les commandes précises pour chaque interaction dans le jeu.

Évidemment, les touches **W, A, S, D** permettront de se déplacer et la souris permettra de viser.

Certaines commandes ont été rajoutées. Elles seront explicitées dans la section guide utilisateur.

4.1.8 - L'affichage "3D" et la vue du joueur

Pour visualiser en "3D" notre carte 2D, nous utiliserons le raycasting. Le joueur aura donc l'impression de se déplacer en 3D dans un labyrinthe.

Le HUD du joueur comportera la vie du joueur, ses munitions ... (voir maquette jeu)

Aucune modification…

4.1.9 - Objectifs secondaires

L'univers soviétique

Comme le veut notre pitch, nous avons voulu mettre en place de nombreuses références à l'URSS. Nous allons par exemple mettre des textures patriotiques ou des ennemis ressemblant à des Américains. Les menus et l'histoire seront aussi très marqués par l'empreinte soviétique.

Le menu pause

Nous souhaitons que le joueur puisse à tout moment mettre le jeu en pause et le reprendre. Il pourra aussi quitter la partie en retournant au menu principal.

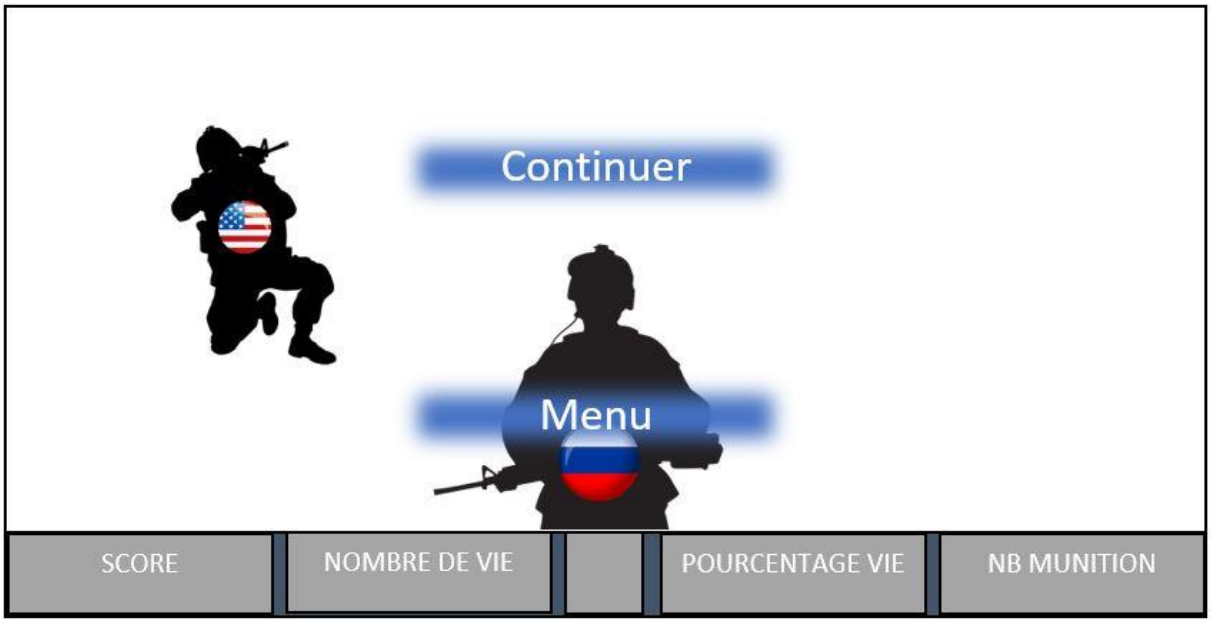


Figure 9 : Maquette menu pause

Les effets sonores

Nous désirions ajouter des effets sonores lors des tirs, des passages des portes et tout autre son pertinent dans le jeu. Et pourquoi ne pas raconter l'histoire du jeu par une bande-son audio lors de discussion avec le PNJ de fin.

L'histoire

L'intérêt de notre jeu ne serait que plus grand si l'intrigue se dévoilait au fur et à mesure des différents tableaux lui conférant ainsi un contexte et une plus-value. De cette façon nous souhaitions faire apparaître des bulles de texte lors des interactions avec les PNJs présents au début et à la fin des niveaux.

Carte

En appuyant sur une touche définie ("M"), une carte avec notre position et la configuration du labyrinthe apparaîtra à l'écran. Cette action ne mettra pas la partie en pause.

Plusieurs armes

Il sera possible de ramasser d'autres armes dans le labyrinthe pour améliorer sa cadence de tir, le nombre de munitions et même les dégâts des balles.

Animations

Nous souhaitons donner un aspect plus lissé et réel à notre jeu par des animations plus propres et moins saccadées telles que la mort d'un soldat américain.

4.1.10 - Modification dans les objectifs secondaires

Au fil de nos premières implémentations et de nos premiers tests, nous avons senti le besoin de proposer une aide supplémentaire au joueur afin qu'il puisse mieux se repérer dans les labyrinthes que nous avons conçus. C'est pourquoi, en plus de la carte complète accessible avec la touche **M**, nous avons décidé de rajouter une mini-map en haut à gauche de la fenêtre de jeu.

Cette fonctionnalité n'étant pas anecdotique en termes d'implémentation, nous l'avons rajoutée à nos objectifs secondaires.

4.1.11 - Technologies utilisées

- Langage de programmation : C++
- IDE : Visual Studio 2019
- Librairie : SFML
- Rendu « 3D » : Raycasting (notre implémentation)

5 - Conception

5.1 - Flow graph

Flow graph du menu principal :

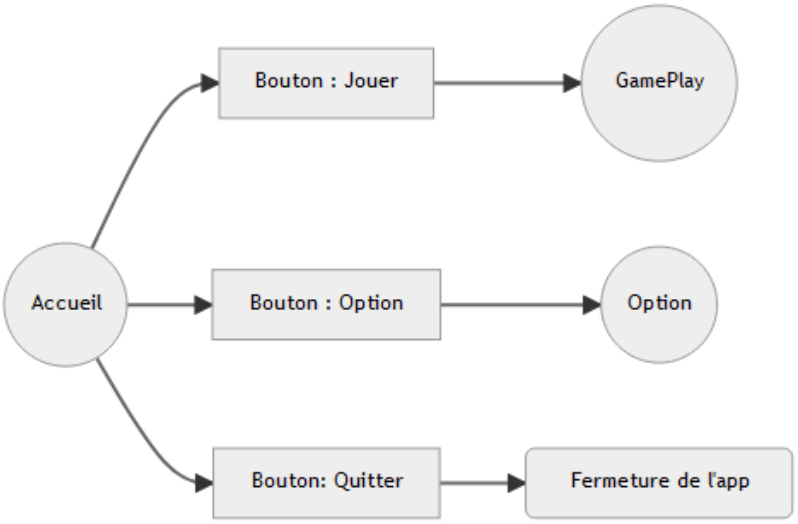


Figure 10 : Flow graph du menu principal

Flow graph du menu pause :

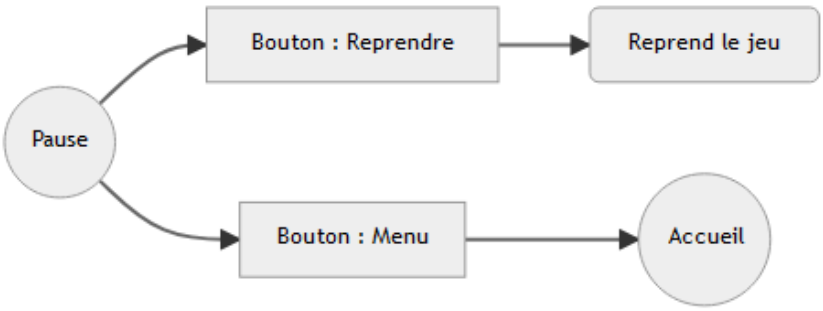


Figure 11 : Flow graph du menu pause

Flow graph du menu option :

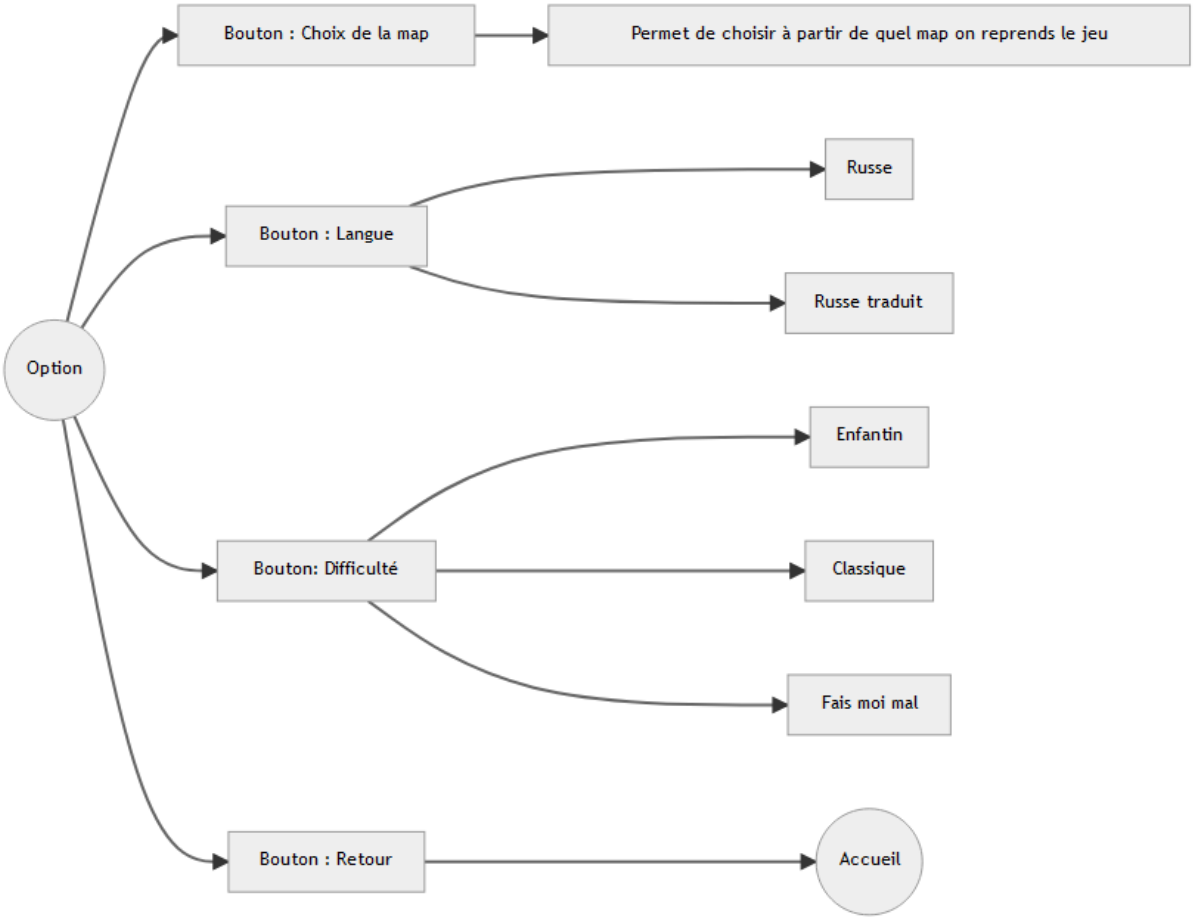


Figure 12 : Flow graph du menu option

Flow graph du menu game over :

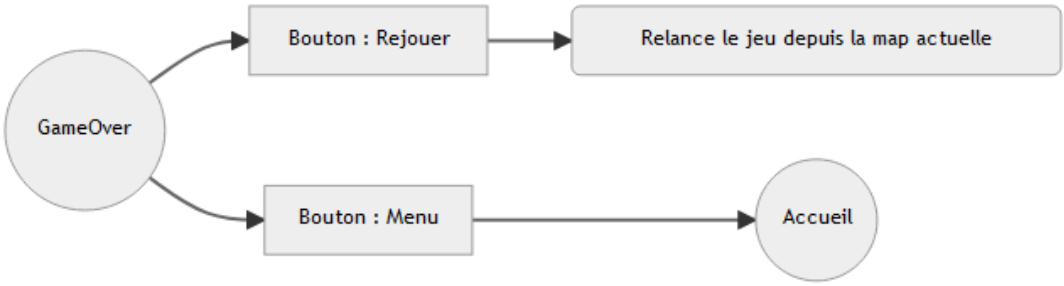


Figure 13 : Flow graph du menu game over

Flow graph du gameplay :

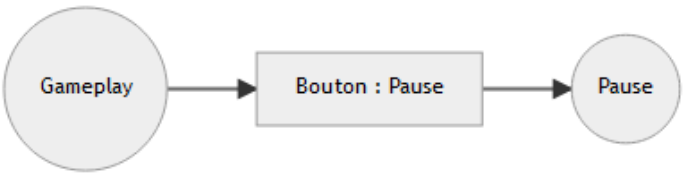


Figure 14 : Flow graph du gameplay

Flow graph du menu principal : aucune modification

Flow graph du menu pause : ajout de la branche quitter.

Flow graph du menu option : plusieurs modifications comme explicitées dans la partie analyse concernant les options.

- Le choix de la map a été implémenté
- Le choix de la langue n'a pas été implémenté
- Il n'y a que deux niveaux de difficulté (sans talent et fais-moi mal)
- La modification de la sensibilité de la souris
- L'affichage des métadonnées (FPS)
- Le réglage du volume sonore

Flow graph du menu game over : aucune modification

Flow graph du gameplay : aucune modification

5.2 - Diagramme des cas d'utilisation

Première version :

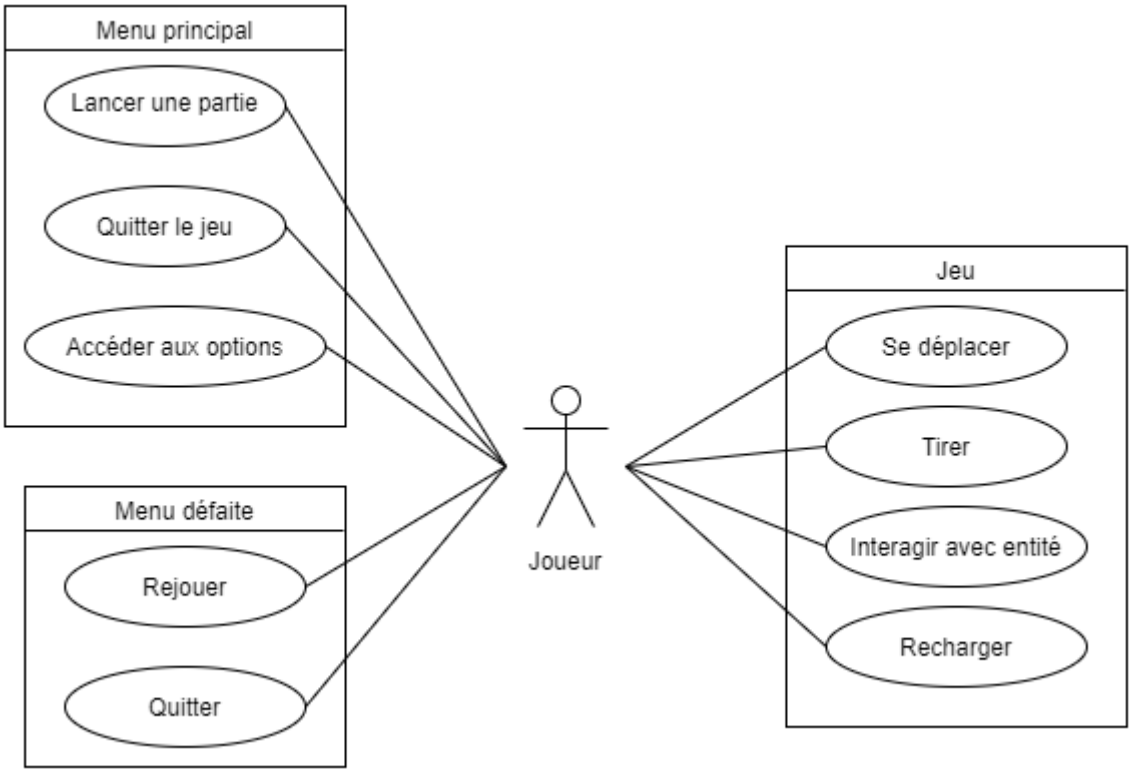


Figure 15 : Diagramme des cas d'utilisation

Toutes les utilisations représentées sur le diagramme ci-dessus ont été implémentées. De plus, le menu pause a été ajouté et il contient les options suivantes :

- Reprendre
- Commandes
- Menu principal
- Paramètres
- Quitter

5.3 - Diagramme de séquence

Notre diagramme de séquence étant très long et fastidieux à adapter pour notre document, nous préférons vous laisser le consulter sur notre wiki via le lien ci-dessous.

Lien sur notre diagramme de séquence

Nous n'avons pas apporté de modifications significatives au diagramme de séquence. Notre premier diagramme reflète bien les interactions entre le joueur et le jeu (système).

5.4 - Tableau Use case

Tout comme pour le diagramme de séquence, il nous semble plus logique de vous laisser consulter notre tableau des uses cases via un lien sur notre wiki.

Lien sur notre tableau des uses cases

Nous n'avons pas apporté de changements significatifs.

5.5 - Diagramme de classe UML

Pour ce qui est de notre diagramme de classe, il s'est bien étoffé. Cependant, les classes principales qui représentent la structure de notre jeu n'ont pas changé. Nous n'avons pas trouvé nécessaire de mettre à jour notre diagramme de classe, car il possède désormais 28 classes avec chacune une grande quantité d'attributs et de méthodes. Nous pensons en effet qu'une telle quantité d'informations ne permet plus de comprendre le fonctionnement de notre jeu. Cependant, les ajouts significatifs et majeures de notre diagramme sont listés et détaillés ci-dessous :

- L'ajout de la classe entité permet de regrouper un certain nombre de propriétés communes à certaines classes telles que celle du coffre, du pistolet ou encore du kit de soin. Cette nouvelle classe est aussi indispensable pour le polymorphisme que nous avons réussi à mettre en place et ce notamment lors de la création de listes d'entités d'une map, le polymorphisme que nous avons mis en place nous facilite la tâche. Cette classe est agrégée à la classe *StatePlayGame*, car c'est uniquement dans cet état que l'on utilise des entités.
- La spécialisation de la classe *Weapon* en cinq classes qui en hérite
    - *Knife* : un simple couteau qui sert d'arme de base
    - *Pistol* : un pistolet classique
    - *Uzi* : un fusil mitrailleur
    - *Shotgun* : un fusil à pompe qui fait beaucoup de dégâts
    - *GrenadeLauncher* : un lanceur de grenades
- Des classes utilitaires telles qu'*Animation* et *AnimatedVertexArray* qui nous permettent de visualiser, avec l'effet 3D du raycasting, des entités animées. Ces classes sont agrégées aux entités, car seules les entités ont besoin d'être animées.
- La classe Bullet que nous avions oubliée lors de notre premier diagramme de classe, mais qui est bien évidemment très importante pour tout l'aspect tir et visualisation de la trajectoire de la balle.

Une des premières versions de notre diagramme de classe ainsi qu'une version intermédiaire générée avec Visual Studio 2019 se trouve sur notre wiki. Vous pouvez les consulter via le lien ci-dessous :

Lien sur notre diagramme de classe

6 - Réalisation

6.1 - Gestion de projet

Concernant la gestion de notre projet, nous avons utilisé l'outil Git. Malheureusement, à nos débuts, nous ne pouvions pas l'exploiter pleinement, nous n'avions pas les connaissances requises. Celles-ci sont venues via notre cours de génie logiciel et ont été prises en compte aussitôt qu'apprises.

Malgré tout, nous avons pu utiliser une partie de son potentiel plus ou moins à bon escient tels que les branches, que nous avons décidé de ne pas fermer pour en garder une trace, les labels, ou encore les issues. Cependant, ces dernières ont été vues comme une obligation et pas un avantage dans les premiers temps. Pour pallier à ceci, nous avons énormément communiqué via Discord et ce notamment pour la planification des tâches.

6.2 - Structure du projet

6.2.1 -GameState

Pour permettre les interactions entre le menu principal, le jeu, le menu pause et le menu de fin, nous nous sommes inspirés de l'idée suivante : Game State Manager.

Le principe est de créer une classe générique *GameState* représentant un état du jeu.

Bien évidemment, au vu de sa fonctionnalité, cette classe est virtuelle pure. Il est aussi très important que chaque *GameState* aient accès à l'instance de **RenderWindow** (**window**) du *GameManager* afin d'obtenir les informations liées aux saisies utilisateur et aussi pour dessiner dans l'instance.

Par la suite, cette classe a été dérivée en quatre états différents :

- Menu principal (*StateMainMenu*),
- Jeu (*StatePlayGame*),
- Menu pause (*StatePauseMenu*),
- Menu de défaite et de victoire (*StateGameOverMenu*).

Les classes qui héritent de *GameState* se devront d'implémenter les trois méthodes ci-dessous :

- **handleInput**

  C'est dans cette fonction qu'on gérera toutes les entrées de l'utilisateur par le biais d'outils mis à disposition par la librairie SFML. Nous utiliserons notamment la classe *Event* et la méthode **window**.pollEvent(event) pour récupérer toutes les informations concernant les saisies clavier et les déplacements de la souris.

- **update**

  Après avoir géré les entrées utilisateur, il faut les traiter et les appliquer à notre jeu. C'est dans cette méthode qu'on s'occupera de le faire. Nous mettons notamment à jour la position du joueur lorsque l'utilisateur souhaite avancer en appuyant sur la touche **W.**

- **draw**

  Finalement, il ne nous reste plus qu'à tout redessiner dans notre instance de **RenderWindow** pour que **window**.display() puisse prendre en compte les modifications.

Ces différents états sont par la suite manipulés par le cerveau de ce projet : le Game Manager.

6.2.2 -GameManager

Le but de notre GameManager est de coordonner tout ce qui se passe dans notre jeu. Pour ce faire, il possède une pile contenant des GameStates. L'état se trouvant au sommet de cette pile est l'état « actif », autrement dit celui dans lequel se trouve le jeu.

Ayant fait le choix de se lancer dans la conception d'un jeu vidéo sans Framework particulier autre que la librairie SFML (pratique pour l'affichage graphique d'éléments 2D dans une fenêtre), nous avons dû créer notre gameLoop nous-mêmes.

Une gameLoop est une boucle infinie qui effectue une série d'actions encore et encore, jusqu'à ce que l'utilisateur quitte le programme. Lors de chaque itération (aussi appelée Frame), le GameManager va appeler les trois fonctions *handleInput*, *update* et *draw* de l'état se trouvant au sommet de la pile.

La durée (temps) séparant deux itérations sera appelée ***deltaTime*** dans le reste de ce rapport afin de simplifier son écriture et sa lecture. Si le jeu tourne avec 60 FPS (Frame Per Second, IPS : Image Par Seconde), cela signifie que la gameLoop effectuera 60 itérations par seconde, donc l'affichage est rafraîchi soixante fois par seconde. Le deltaTime vaut alors 1/60 = 0.01666 seconde. Cette valeur est indispensable lors de l'interaction avec l'utilisateur ainsi que la mise à jour de l'environnement. Pour comprendre pourquoi, prenons un exemple concret :

Un jeu vidéo en 2D (vu du dessous), où l'utilisateur peut déplacer une personne. Cette personne se déplace de 1 pixel par frame (par itération de gameLoop). Si le jeu tourne à 50 FPS, cette personne se déplacera de 50 pixels par seconde. Or, si le jeu tourne à 100 FPS, elle se déplacera de 100 pixels par seconde. Cela pose un problème : nous voulons que peu importe le framerate de notre jeu, la personne se déplace de 50 pixels par seconde.

Pour faire pour cela, nous devons utiliser le deltaTime vu précédemment. Il suffit de simplement multiplier la distance de déplacement par cette durée. À présent, si le jeu tourne à 50 FPS (deltaTime = 1/50 = 0.02), le joueur se déplacera de 50*0.02 = 1 pixel par frame. Comme il y a 50 frames par seconde, il se déplacera à 50 pixels par seconde.

Dans le cas où le jeu tourne à 100 FPS, le principe est le même : le deltaTime vaut 1/100 = 0.01. Le joueur se déplacera alors de 50*0.01 = 0.5 pixel par frame. Comme il y a 100 frames par seconde, il se déplacera de 100 * 0.5 = 50 pixels par seconde.

En conclusion, nous pouvons utiliser le temps entre chaque itération afin que, peu importe le framerate, le joueur se déplace de 50 pixels par seconde. Ce principe est utilisé dans notre jeu.

6.2.3 - Dessiner avec SFML

Une instance de la classe SFML RenderWindow (RenderWindow *window) est déclarée et instanciée. C'est dans cette instance de RenderWindow que nous pouvons afficher et dessiner des formes qui seront ensuite reproduites dans une fenêtre classique.

Nous avons utilisé la fonction fournie par SFML window.isOpen() comme condition principale à notre gameLoop. En effet, tant que nous affichons quelque chose dans notre fenêtre (window) il est nécessaire de passer par la gameLoop.

Lors de chaque passage dans la gameLoop il faut :

1. *window.clear()* : supprimer tout ce qui se trouve dans la fenêtre (la nettoyer)
2. *handleInput()*, *update()* et *draw()* de l'état actif : gère tous les évènements et dessine dans window
3. *window.display()* : affiche tout ce qui a été dessiné dans window, dans une fenêtre

6.3 - Map

6.3.1 -Création d'une map

La décision a été prise de créer des maps en dur et de ne pouvoir jouer que ces dernières. Celle-ci nous offre une rapidité de lecture et donc d'exécution extrêmement grande. Pour respecter les contraintes liées au raycasting, les niveaux se devaient d'être carrés et avec des murs extérieurs, afin d'empêcher le joueur de sortir de la carte et donc de faire crasher le programme.

Tous les niveaux ont été conçus, créés et testés par nos soins. Les cartes ont vu le jour sur Notepad et respectent une règle simple : labyrinthique. En effet, nous souhaitions donner une dimension supplémentaire à notre jeu pour le corser quelque peu, mais aussi pour créer un sentiment d'accomplissement chez le joueur lorsqu'il réussit à sortir du niveau. Cette dimension nous permet par ailleurs de jauger la difficulté en créant des cheminements de plus en plus complets et complexes.

Les différentes cartes ont été construites comme suit dans la figure suivante, sur laquelle on peut observer une grande occurrence de "0", correspondant au sol, passage que peux emprunter le joueur.

Les autres symboles représentent diverses textures, ennemis ou possibilités de trouver du matériel, et sont énumérés dans l'exemple de carte ci-dessous.

```
698a66666i6jI666
60T0668E00000006
60w06E6660606606
66W6600L6000f006
600A606660666066
6066600b00L6000J
60E0006660660606
6000600060660006
6E0E66606066060H
6606606000000060i
6E006000666A6606
6660660660C000006
6000000066X66606
i066H66060006606
600xhG0060006E0h
6666666666m666I6
```

6.3.2 - Parsing d'une map

Liste des codes utilisés pour parser une map :

Code(s)	Description
T	Position initiale du joueur
E	Gardien (ennemi standard)
C	Coffre
m	Ascenseur (fin du niveau)
L	Loot aléatoire (soin, arme, …)
b	Passage secret (mur qui peut se dérober)
[1-9] + {f, F, h, H, i, I, j, J, k, M, n, N, o, O}	Murs (textures différentes)
{V, W, X, Y, Z}	Portes (textures différentes)
{v, w, x, y, z}	Clés pour déverrouiller les portes

6.4 - Textures

La création des textures murales a été réalisée en deux étapes, chacune contenant divers critères. Premièrement, il a fallu récupérer les images de base.

En ce qui concerne les fonds muraux, quatre ont été retenus : un fond de pierre et de mousse, un mur en pierre, un mur en bois ainsi qu'une bibliothèque.

La nécessité d'obtenir un visuel fluide impliquait des fonds "seamless", ceux-ci ont la particularité d'être continus lorsqu'ils sont placés côte à côte.

Concernant l'ornemental, nous nous étions fixé un thème que nous nous devions de respecter, pour ce faire nous avons utilisé des affiches de propagande de l'URSS.

Sur ce choix, une certaine laxité a été acceptée. En effet, pour satisfaire un besoin purement visuel, nous avons sélectionné des affiches de divers pays appartenant à l'union, mais également de différentes temporalités. La décision a également été prise, dans un but de diversité, mais également pour rendre hommage à quelques tableaux dits du réalisme socialiste.

Une fois la sélection réalisée, il a fallu adapter ces images à notre univers ; nous avons par exemple dégradé la qualité de certaines affiches afin de les vieillir pour mieux correspondre à l'ambiance « cave » de la première carte. Nous avons également mis les tableaux dans un cadre, donné une inclinaison aux affiches pour obtenir un rendu plus réaliste et doux.

Figure 16 : Adaptation d'une image pour une texture

Nos sprites consistent en une suite d'images, de tailles et d'espacement régulier disposée sur une même ligne. Pour le choix de ceux-ci, nous avons retravaillé des bases existantes afin de les faire correspondre à nos attentes, telles que l'espacement entre chaque image, ajout de détails, mais également créer certains de toute pièce, telle que l'animation des torches.

Figure 17 : Sprite d'un ennemi

Concernant le sol, nous avons décidé de garder un affichage simple composé de dalles bicolores, noir et gris, afin d'optimiser le programme autant que possible.

Dans la même démarche d'optimisation, le plafond n'existe pas réellement et est simplement constitué d'un fond noir.

6.5 - Déplacement du joueur

6.5.1 -Vecteur direction

La direction dans laquelle le joueur regarde est stockée dans un vecteur de float *sf::Vector2f*. Elle est mise à jour lorsque la souris se déplace sur l'axe horizontal.

6.5.2 -Vecteur position

La position du joueur est stockée dans un vecteur de float *sf::Vector2f*. Elle est mise à jour lorsque le joueur appuie sur une des touches suivantes : **W**, **A**, **S** et **D**.

Si la touche **W** est pressée, le nouveau vecteur position est le vecteur position actuelle auquel est additionné le vecteur direction, multiplié par le deltaTime.

Nouvelle Position = Position Actuelle + Vecteur Direction * DeltaTime

Si la touche **S** est pressée, le nouveau vecteur position est le vecteur position actuelle auquel est soustrait le vecteur direction, multiplié par le deltaTime.

Nouvelle Position = Position Actuelle + Vecteur Direction * DeltaTime

Le même principe est utilisé pour les déplacements latéraux (touches **A** et **D**).

6.5.3 -Gestion des collisions avec les murs

Première façon :

Après avoir récupéré la nouvelle position du joueur, il faut contrôler que celle-ci ne se trouve pas sur l'emplacement d'un mur (ou tout autre bloc sur lequel le joueur ne peut pas aller). Si ce n'est pas un mur, le joueur peut alors se déplacer, si c'en est un, la position n'est pas mise à jour.

Cette façon comporte des limitations : par exemple, si les FPS sont bas, le joueur fera de plus grands pas (pour qu'il se déplace toujours à la même vitesse). Si un grand pas est plus grand que la largeur d'un mur, l'utilisateur passera à travers celui-ci.

Il existe une autre manière de contrôler les collisions avec les murs qui permet d'éviter ce problème :

Le principe consiste à lancer un rayon dans la direction du joueur (celui-ci s'arrêtera une fois un mur touché) et récupérer la longueur de ce rayon (qui représente la distance entre le joueur est le mur le plus proche se trouvant dans la direction du joueur). Il suffit alors de contrôler que la distance de déplacement du joueur soit plus petite dans la longueur de ce rayon. Si c'est le cas, la position du joueur peut être mise à jour.

6.6 - Raycasting des murs

6.6.1 -Principe du raycasting

Le raycasting est une technique de rendu permettant de créer des perspectives en trois dimensions à partir d'une carte (tableau) en deux dimensions.

Le principe est assez simple : on lance un rayon partant du joueur. Celui-ci se déplace dans la direction du joueur, en contrôlant sur la carte 2D s'il ne percute pas un mur (représenté souvent par le chiffre 1). Si un mur est rencontré, la distance entre le joueur et ce mur est alors calculée. Cette longueur permet par la suite de déterminer la hauteur du mur. Plus celle-ci est grande, plus le mur est éloigné du joueur, donc plus il est visuellement petit. Inversement, si la distance est faible, le joueur est proche du mur, celui-ci est donc plus grand. Le schéma ci-dessous illustre ce principe.

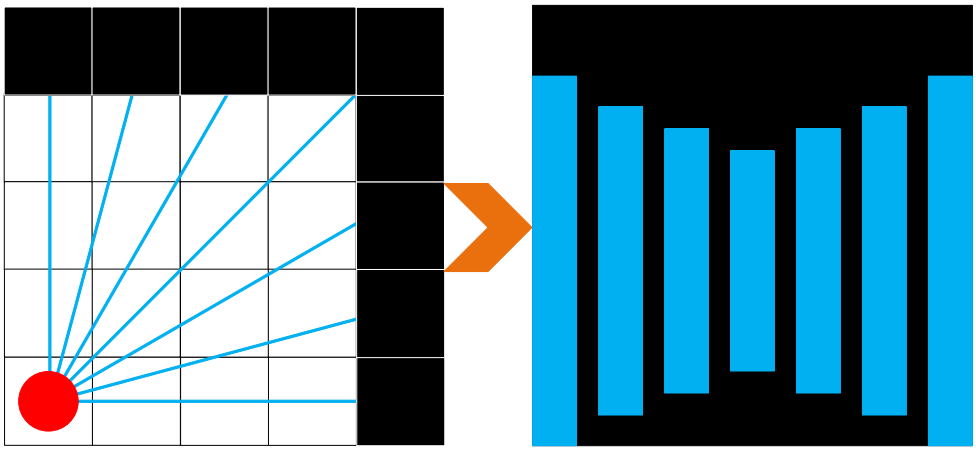


Figure 18 : Schéma illustrant le principe du raycasting

Il convient alors de lancer un rayon par pixel horizontal de l'écran. La coordonnée X représente le décalage d'angle par rapport à la direction du joueur, tout en restant dans le FOV (Field Of View, champ de vision en français) du joueur.

En dessinant ensuite une ligne verticale pour chaque rayon, l'illusion d'un monde en trois dimensions est créée, comme représentée sur l'exemple ci-dessous.

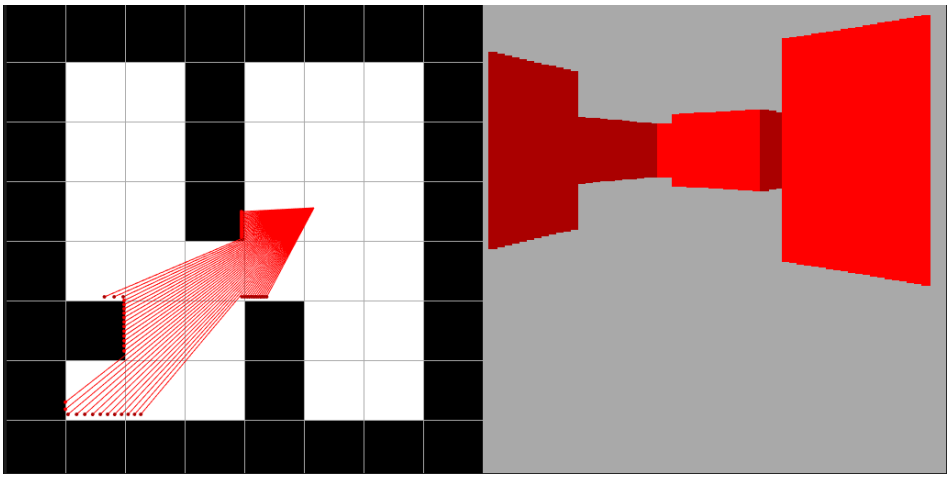


Figure 19 : Schéma illustrant un raycaster simple

Si chaque rayon devait avancer pixel par pixel en contrôlant s'il se trouve à l'intersection avec un mur, cela demanderait énormément de ressources. C'est pourquoi nous utilisons l'algorithme DDA (Digital Differential Analysis).

6.6.2 -Algorithme DDA

Cet algorithme permet à un rayon de contrôler uniquement les endroits où peut se trouver un mur : l'intersection entre deux blocs, comme illustrée par les points rouges sur ce schéma.

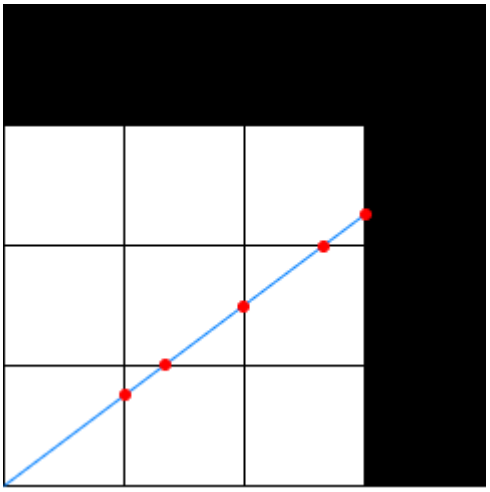


Figure 20 : Schéma illustrant l'algorithme DDA

En plus de limiter grandement le nombre de contrôles à effectuer par le rayon, cet algorithme est simple à comprendre et à mettre en place, puisqu'il s'agit uniquement de trigonométrie triviale. Nous n'expliquons pas en détail son fonctionnement, mais nous vous renvoyons au tutoriel de *Lode Vandevenne* qui est très bien écrit et facile à comprendre : https://lodev.org/cgtutor/raycasting.html#The_Basic_Idea_.

6.7 - Raycasting des sols et plafonds

Le raycasting des sols et du plafond fonctionne de la même manière que pour les murs. Cependant, le raycasting des murs demande X Vertex (dans notre cas, un Vertex est une ligne verticale), où X représente la largeur de la fenêtre en pixel. Pour le raycasting des sols et du plafond, le nombre de Vertex n'est pas fixe. En effet, si le joueur fait face à un mur très proche, le sol n'est pas visible. En revanche, si le joueur se trouve dans une grande pièce, énormément de Vertex doivent être utilisés pour pouvoir afficher le sol. Cela affecte les performances de façon non négligeable ; c'est pourquoi nous avons décidé d'afficher uniquement le sol.

6.8 - Raycasting avec textures

Le raycasting permettant d'afficher des murs texturés fonctionne avec le même principe que le raycasting standard, auquel nous ajoutons quelques calculs afin de déterminer la coordonnée X du mur touché par le rayon. Il convient alors, au lieu d'afficher une ligne verticale de couleur, d'afficher la ligne verticale de la texture correspondante à la coordonnée X.

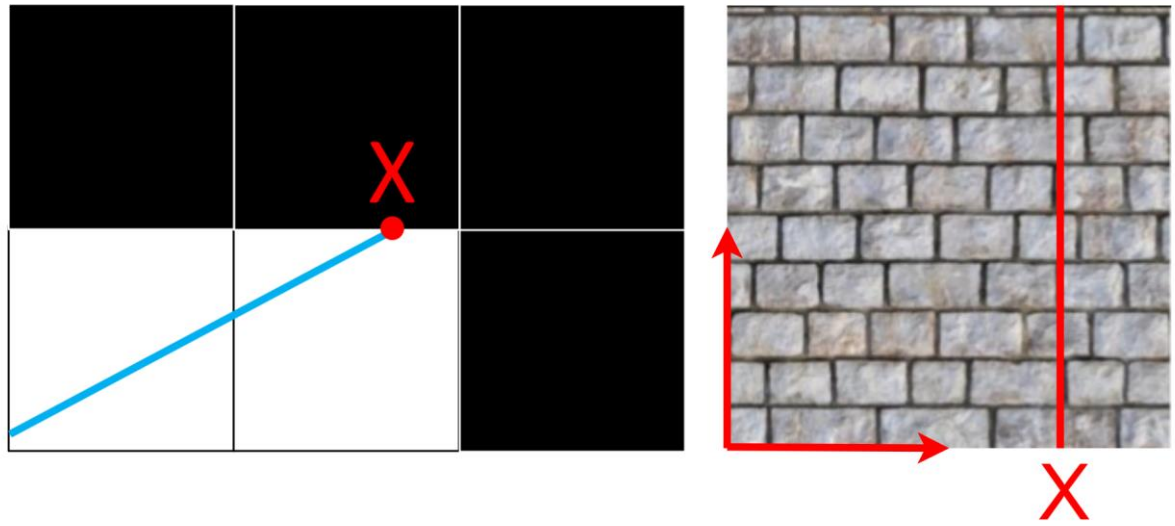


Figure 21 : Récupération de la coordonnée X d'une texture

Pour ce faire, la librairie SFML met à des dispositions des *Vertex* sur lesquels nous pouvons simplement mapper une texture. Il suffit alors de créer un *Vertex* par ligne verticale (par pixel horizontal sur la fenêtre), de calculer la fameuse coordonnée X, de rechercher la texture correspondante et de mapper celle-ci sur le bon *Vertex*.

Tous les Vertex sont stockés dans un tableau de type *sf::VertexArray*, qui peut simplement être affiché avec SFML comme tout autre élément.

6.9 - Raycasting des entités (texturées)

Avant tout, nous avons besoin de stocker la longueur de chaque rayon dans un tableau nommé *ZBuffer*.

Les entités n'ont pas le même comportement visuel que les murs. En effet, celles-ci font toujours face au joueur, contrairement aux murs, qui eux restent fixent.

En opposition aux murs qui peuvent être affichés avec un seul VertexArray, chaque entité nécessite son propre VertexArray. Celui-ci se construit en effectuant une démarche similaire à celle présentée précédemment pour afficher les murs. Nous ne détaillerons pas les calculs (assez longs et compliqués) de ce procédé, mais nous vous renvoyons à nouveau au tutoriel de *Lode Vandevenne*.

Une fois la longueur du rayon connue et avant de dessiner la ligne verticale, il est nécessaire de contrôler si la ligne verticale de l'entité se trouve derrière un mur. En effet, si c'est le cas, il n'y a pas besoin de la dessiner. Ce contrôle se fait grâce au ZBuffer rempli en amont. Si la longueur du rayon allant du joueur au mur est plus grande que la distance allant du joueur à l'entité, cette dernière est plus proche que le mur et doit donc être dessinée.

6.9.1 -Optimisation(s) à prendre en compte

Afin d'éviter de devoir effectuer les calculs pour les entités qui ne sont pas entièrement visibles par le joueur, nous avons eu recourt à une simple optimisation consistant à ajouter un booléen *toDraw* à la classe *Entity*. Lors du raycasting des murs, si un rayon rencontre une entité (autre qu'un mur) avant d'atteindre un mur, le booléen de cette entité est mis à *True*. Il est donc maintenant possible d'effectuer les calculs de raycasting uniquement les entités visibles par le joueur.

6.10 - Animation des entités

6.10.1 - Classe Animation

Cette classe permet d'afficher et d'animer un élément qui n'a pas besoin d'être raycasté, comme l'arme que le joueur brandit devant lui.

Lors de la création d'un objet *Animation*, il nous faut spécifier quelles textures doivent être utilisées pour l'animation, le nombre d'images (sprites) composant l'animation, ainsi que la durée durant laquelle une frame (un sprite) doit être affichée à l'écran.

Pour afficher l'entité, il suffit de faire appel à la méthode *draw*. Celle-ci sera affichée en fonction des paramètres saisis (position, taille …).

Pour animer un objet, il faut appeler la méthode *update* de cette classe, en précisant le temps écoulé depuis le dernier appel (à nouveau via le deltaTime). Cela est fait dans la gameLoop. Son fonctionnement est simple à comprendre et à mettre en place.

Prenons un exemple où une animation comporte quatre frames, qui doivent chacune être affichées pendant 0.1 seconde. Lors de l'appel de la méthode *update*, le deltaTime vaut 0.05 seconde : la durée d'affichage d'une frame n'a pas été dépassée, donc la frame actuelle doit encore être affichée. Lors du second appel, le deltaTime vaut 0.06 seconde, auquel on additionne la 0.05 seconde précédente, ce qui fait 0.11 seconde : la durée est dépassée, il faut donc passer à la frame suivante.

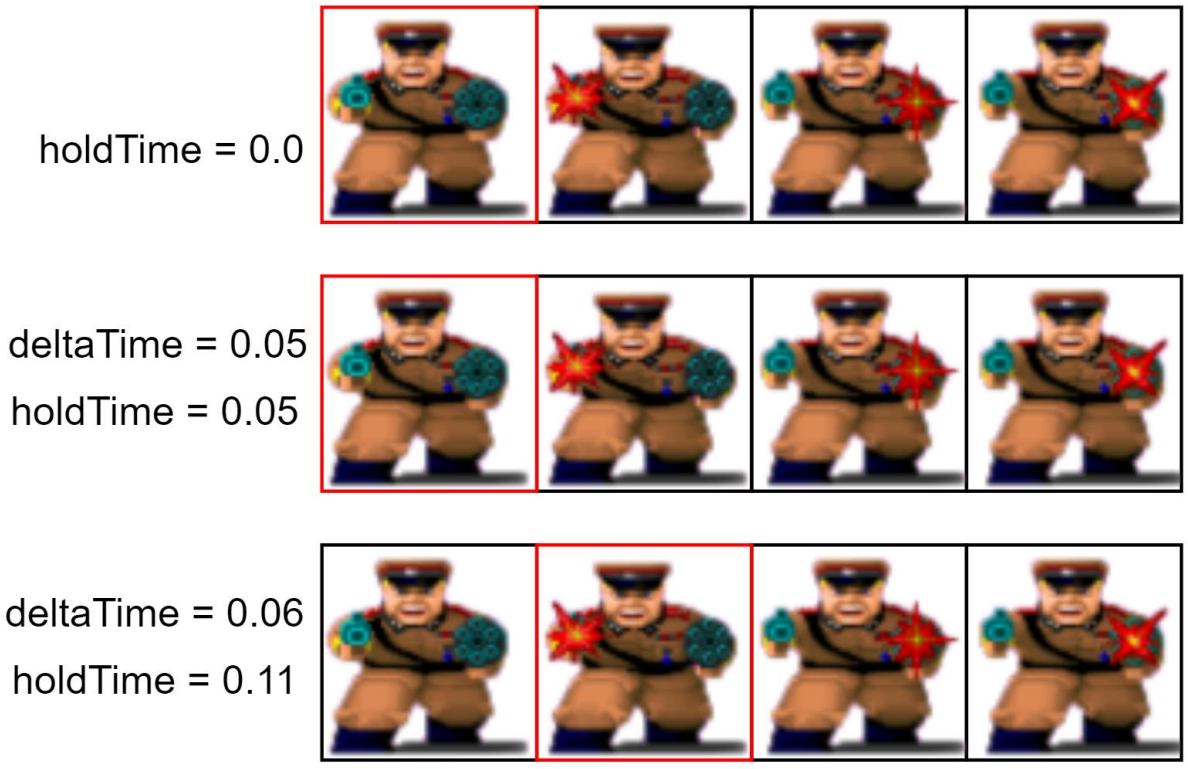


Figure 22 : Schéma illustrant le fonctionnement d'une animation

Lorsque la dernière frame est atteinte et que la durée est dépassée, il suffit simplement de recommencer à partir de la première frame.

6.10.2 - Classe AnimatedVertexArray

Cette classe permet d'afficher et d'animer toutes les entités qui doivent être raycastées, et qui n'ont pas le comportement d'un mur, c'est-à-dire toutes les entités qui regardent constamment le joueur. Le fonctionnement de l'affichage (raycasting) a été détaillé précédemment. L'animation d'une entité raycastée fonctionne comme dans la classe *Animation*.

6.10.3 - Classe Entity

La classe Entity est la classe mère de tout ce qui peut être affiché sur la carte (Murs, ennemis, armes …). Elle est par la suite spécifiée en d'autres sous-classes (classes filles), telles que les classes *Mur*, *Ennemy*, *Weapon*, etc.

Chacune de ces classes possède un ou plusieurs attributs de type *AnimatedVertexArray* représentant chacune des animations différentes que peut avoir l'entité. Par exemple, la classe *Ennemy* possède deux animations :

- *ShootingAnimation*, qui correspond à l'animation de tire d'un ennemi,
- Et *DyingAnimation*, qui correspond à l'animation de mort d'un ennemi.

L'animation affichée dépend de l'état de l'ennemi : si celui-ci est en train de mourir, l'animation de mort sera affichée. Cependant, si l'ennemi tire, c'est l'animation de tire qui sera affichée.

6.10.4 - Éviter l'overlapping des entités

Afin d'éviter qu'une entité éloignée soit dessinée par-dessus une entité proche, il est nécessaire de les dessiner dans l'ordre décroissant de la distance les séparant du joueur. Dans notre cas, toutes les entités sont stockées dans une liste, il convient alors de calculer la distance les séparer du joueur, et d'effectuer un tri décroissant par rapport à cette variable.

6.11 - Interactions

6.11.1 - Interactions joueur – entités

Pour que notre jeu prenne vie, nous avons donné la possibilité au joueur d'interagir avec des entités.

Exemple : le joueur aperçoit un coffre. Il doit être en mesure de l'ouvrir, de voir ce qu'il en sort et de potentiellement récupérer son contenu.

Pour se faire, lorsque le joueur appuie sur la touche **E** de son clavier, une méthode qui permet de déterminer si une entité se trouve sur la case juste en face de nous se lance. Cette dernière consulte la position actuelle du joueur dans la map et l'orientation du joueur dans la map (vecteur direction). Avec ces informations, la méthode va donc trouver les indices correspondant à la position de la case en face du joueur et consulter un tableau à deux dimensions comportant l'ensemble des entités de notre map.

Code associé au paragraphe ci-dessus :

```
1.  Entity* StatePlayGame::getInteractedEntity()
2.  {
3.      int playerDirectionPosX = int(floor(player->position.x + 1.0 * player->direction.x));
4.      int playerDirectionPosY = int(floor(player->position.y + 1.0 * player->direction.y));
5.      if (playerDirectionPosX > 0 && playerDirectionPosX < mapSize &&
6.          playerDirectionPosY > 0 && playerDirectionPosY < mapSize)
7.      {
8.          if (this->entityMap[playerDirectionPosX][playerDirectionPosY] != nullptr)
9.              return entityMap[playerDirectionPosX][playerDirectionPosY];
10.     }
11.     return nullptr;
12. }
```

Après quoi, un pointeur sur l'entité avec laquelle on souhaite interagir sera retourné et les interactions pourront avoir lieu.

Un exemple intéressant est celui du coffre. Lorsque le joueur interagit avec celui-ci l'animation d'ouverture de ce dernier se mets en marche, le coffre consulte l'entité qu'il doit faire apparaître, puis disparait des entités à afficher et finalement, laisse, à sa place dans le tableau bidimensionnel des entités, l'entité qu'il devait faire apparaître. Cette dernière est alors comme sortie du coffre et maintenant disponible.

6.11.2 - Interactions joueur – ennemis

Un joueur doit être capable de tirer sur un ennemi et pour que ce dernier puisse être touché et que de la même manière un ennemi puisse toucher le joueur. Il a fallu s'occuper des collisions entre les balles et les ennemis ainsi qu'entre les balles et le joueur.

Lorsqu'un joueur ou un ennemi tire, une balle se crée et est ajoutée à la liste des entités de la map. Par la suite, lorsque l'on parcourra la liste des entités et que l'on itérera sur une balle il faudra gérer ses collisions. En effet, si la balle rencontre un mur il faudra qu'elle explose.

Dans le cas où elle n'entre pas en collision avec une entité, il faudra modifier sa position dans le sens de sa trajectoire pour qu'elle continue d'avancer.

Le cas le plus intéressant est celui où la balle rencontre un ennemi, car il faudra premièrement récupérer le pointeur sur l'ennemi touché afin de réduire sa vie en conséquence. Une fois cette opération réalisée, la balle peut exploser et disparaitre des entités à afficher.

En ce qui concerne le tir des ennemis, dans notre première implémentation, ceux-ci tiraient uniquement lorsqu'ils étaient visibles par le joueur. Cela posait évidemment des problèmes de jouabilité, puisqu'il suffisait de tourner le dos aux ennemis pour ne pas se faire tirer dessus.

Notre deuxième (et définitive) approche consiste à lancer un rayon de l'ennemi jusqu'au joueur : si celui-ci rencontre un mur avant le joueur, cela signifie que le joueur n'est pas visible par l'ennemi, il ne tire donc pas. Inversement, si le rayon rencontre le joueur avant un mur, il est donc visible par l'ennemi et peut se faire tirer dessus.

6.12 - Menus

6.12.1 - Classe Button

Malheureusement pour nous, SFML ne contenait pas de classes utilitaires comme des boutons, des labels ou des combo-box. Nous avons donc dû coder nous-mêmes la classe bouton qui est indispensable dans un jeu comme le nôtre.

Pour se faire, nous avons utilisé les formes géométriques fournies par SFML afin de dessiner un bouton à l'aide de rectangles. Quatre rectangles ont été utilisés pour les bords de notre bouton et un rectangle pour le centre. Une zone de texte a également été ajoutée à notre bouton pour y écrire quelque chose à l'intérieur. Nous avons aussi mis en place les méthodes suivantes :

- **draw** : très utile pour afficher le bouton dans la fenêtre active de SFML
- **isClicked** : permet de détecter le clic sur notre bouton
- **selected** : change la couleur des bords de notre bouton pour monter qu'il est sélectionné
- **isHovering** : qui détecte et change la couleur du centre du bouton si la souris survole le bouton

6.12.2 - Gestion des saisies de l'utilisateur (clics)

Pour savoir si un clic a été effectué sur un bouton, il convient simplement de contrôler la position de la souris lors d'un clic gauche. Si cette position se trouve dans le rectangle formant le bouton, alors celui-ci a été cliqué.

Ceci est implémenté dans la fonction *isClicked* de la classe *Button*, qui retourne un booléen. Il suffit de, dans la fonction *HandleInput* du *GameState* contenant le bouton, contrôler si cette fonction retourne la valeur True et agir en conséquence (par exemple lancer le jeu lors d'un clic sur le bouton « Jouer » du *GameState StateMainMenu*).

6.12.3 - Les différents menus

Nous avons mis en place trois menus différents qui correspondent à trois classes différentes pour permettre un affichage et des fonctionnalités différentes dans chaque menu.

Menu d'accueil (StateMainMenu) :

Dans le constructeur de notre menu, nous déclarons tous les boutons dont nous aurons besoin par la suite ainsi que l'image qui fera office de fond d'écran. Le menu principal contient aussi une instance de la classe Settings que nous avons créée pour enregistrer toutes les préférences de l'utilisateur afin de passer ces configurations à l'état *StatePlayGame* lors de sa construction. Comme chaque *GameState* la classe de ce menu se doit d'implémenter certaines méthodes afin d'être compatible avec notre gameLoop.

Dans la méthode *HandleInput*, nous récupérons les clics de l'utilisateur afin de les traiter plus tard.

Dans la méthode update, nous utilisons les informations transmises par la méthode *handleInput* afin d'agir en conséquence. Nous mettons par exemple nos paramètres par le biais de l'instance de Settings ou encore les focus des boutons pour visualiser nos modifications dans les paramètres.

Dans la méthode *draw*, il suffit de dessiner le fond d'écran qui est un *Sprite* ainsi que les boutons.

Si la méthode *handleInput* détecte que nous souhaitons lancer une partie, alors la méthode *update* va créer une instance de la classe *StatePlayGame* et modifier l'état actif de notre *GameManager* pour afficher l'état de jeu.

```
1.  if (playButton.isClicked())
2.  {
3.    StatePlayGame* statePlayGame = new StatePlayGame(this->gameManager, settings);
4.    this->gameManager->changeState(statePlayGame);
5.    return;
6.  }
```

Menu de pause (StatePauseMenu) :

Le menu pause est instancié et placé en haut de la pile des états lorsque le *StatePlayGame* détecte que la touche "ESC" du clavier a été pressée.

Contrairement au menu d'accueil, ce menu reçoit, par le biais de son constructeur, une instance de *Settings* pour permettre au joueur de modifier quelques-uns des paramètres en cours de jeu. Cependant, il se doit aussi de créer ses propres boutons et son propre fond d'écran.

Le menu pause offre plusieurs possibilités qui ne seront pas détaillées, car le principe de fonctionnement est le même que celui du menu d'accueil.

Menu de fin de partie (StateGameOverMenu) :

Pour gérer le cas de victoire du joueur et celui de défaite dans le même état, nous avons simplement demandé un paramètre booléen à la création du *StateGameOverMenu*. Ce dernier permettra d'afficher une image de victoire, ou à l'inverse, de défaite. Les mécanismes de changements d'un état à un autre sont toujours les mêmes et aucune fonctionnalité particulièrement intéressante n'a été implémentée dans ce menu particulièrement à un autre. C'est donc pour cela qu'il ne sera pas plus détaillé.

6.12.4 - Les options (classes Settings)

Afin de personnaliser l'expérience du joueur, nous avons mis en place un certain nombre d'options. Les options ont été regroupées dans une classe Settings. Ce qui nous permet facilement de configurer le jeu en se passant des objets Settings qui comportent toutes les informations relatives aux options. Toutes les options disponibles sont configurables dans le menu d'accueil, les voici :

Map :

Ce paramètre est un entier de qui peut prendre les valeurs 0 à 5. La valeur 0 signifie qu'on laisse le jeu passer d'un niveau à l'autre automatiquement alors que les valeurs 1 à 5 précisent quel niveau l'utilisateur souhaite jouer. Si nous précisons le niveau que nous souhaitons jouer, au bout de celui-ci le jeu s'arrête.

Afficher Meta Data :

Ce paramètre est un booléen qui afficher ou non quelques informations du jeu.

Sensibilité :

Ce paramètre est un entier qui peut prendre les valeurs 3, 5 et 10. Ces valeurs ne sont autres que le facteur par lequel nous multiplions la vitesse de rotation du joueur. Ceci afin de simplifier la paramétrisation lorsque l'on récupère l'objet Settings.

Difficulté :

Ce paramètre est un entier qui, pour l'instant, peut prendre les valeurs 1 et 2. Ce choix est certes binaire, mais il permet de rajouter plus facilement un niveau de difficulté 3.

Volume :

Ce paramètre est un entier qui peut prendre les valeurs 0, 25, 50, 75 et 100. Ces 5 valeurs correspondent aux différents niveaux de volume sonore que nous pouvons immédiatement utiliser pour régler notre volume sonore.

6.13 - Musiques et sons

SFML travaille, pour la gestion sonore, avec la librairie OpenAl32.dll, pour des questions de praticité nous avons pris le pli de compiler avec les mêmes standards. La création de l'environnement sonore s'est fait sur deux tableaux, premièrement celui de la musique. Pour ce faire nous avons utilisé dans le jeu les morceaux Rip and Tears ainsi que BFG Division de Mick Gordon, créateur officiel de la bande son de Doom Eternal.

Pour le menu, nous avons choisi d'utiliser, à l'accoutumée, l'hymne de l'URSS revisité façon métal par les musiciens Richard M. Lauw et Andreas Bamberger pour le compte de REELTRACKS - music for your eyes®.

En ce qui concerne le sound design, nous avons utilisé différentes bibliothèques sonores libres de droits afin d'habiller le jeu. De même que pour le choix de nos textures, nous voulions donner un corps solide, varié et cohérent à notre univers sonore. De ce fait, les bruitages sélectionnés l'ont été sur trois critères : leur pertinence, leur qualité, mais également leur puissance en Db. Un pré-équilibrage sonore ainsi qu'une adaptation des durées ont été réalisés via le logiciel Audicity.

6.14 - Narration

PoutineStyle : un nom, un jeu, une histoire. 13 octobre 1962, alors que la tension entre les États-Unis d'Amérique et la resplendissante Union soviétique est à son comble, l'armée américaine réussi l'exploit d'infiltrer la base militaire ultra secrète de Cuba. Moscou, dans un état de panique totale, se doit d'agir fort et vite afin de ne pas perdre ce si précieux pied-à-terre cubain.

Elle décide d'envoyer son meilleur élément. C'est ainsi qu'entre en jeu Vladimir Vladimirovitch Poutine, jeune et brillant agent du KGB qui se voit dépêcher d'urgence sur place. Sa mission est simple : éradiquer la vermine américaine et assurer que la livraison des missiles puisse se dérouler sans accroche. Armé de son courage et de son couteau, l'inégalable Poutine se rend dans la base en passant par le sous-sol. Sa connaissance parfaite des lieux l'amène aux ascenseurs pour remonter jusqu'au dernier étage, où, selon les informations du KGB, se trouve le commandant Moore qui retiendrait Fidel Castro en otage.

7 - Résultats

7.1 - Menus

Globalement, les menus sont visuellement basic, néanmoins ils fonctionnent tous comme souhaité. Cela est dû principalement au fait que SFML ne propose d'éléments d'interface graphique (bouton, slider …). Nous avons donc dû les implémenter nous-mêmes, et comme nous ne voulions pas perdre trop de temps là-dessus, nous avons décidé d'implémenter uniquement un simple bouton.

7.1.1 -Menu principal

Figure 23 : Menu principal final

Le bouton « Jouer » permet de lancer une partie.

Le bouton « Paramètres » permet d'accéder aux paramètres du jeu.

Le bouton « Quitter » permet de quitter l'application.

7.1.2 -Menu des paramètres

Figure 24 : Menu paramètres final

Sur la première ligne, le joueur peut choisir de jouer en automatique (commencer par la carte 1, puis la deux une fois la première terminée …) ou dans un niveau en particulier.

Sur la deuxième ligne, le joueur peut choisir d'afficher ou non les métadonnées (FPS).

Sur la troisième ligne, le joueur peut choisir la sensibilité de la caméra pendant le jeu.

Sur la quatrième ligne, le joueur peut choisir la difficulté du jeu.

Sur la dernière ligne, le joueur peut choisir le volume du jeu (musique, effets sonores …).

En appuyant sur le bouton « Enregistrer », les paramètres sont enregistrés et seront pris en compte lors de la prochaine partie. L'utilisateur est ensuite redirigé sur le menu principal.

7.1.3 -Menu pause

Figure 25 : Menu pause final

Depuis le menu pause, le joueur peut à nouveau modifier certains paramètres, retourner au menu principal, quitter le jeu ou reprendre sa partie en cours.

7.1.4 -Menu de victoire

Figure 26 : Menu de victoire (final)

À partir du menu de victoire (et de défaite), le joueur peut recommencer une partie, ou quitter le jeu.

7.1.5 -Menu de défaite

Figure 27 : Menu de défaite (final)

7.2 - Jeu

7.2.1 -Différents murs

Un mur peut prendre 33 textures différentes ; toutes ces textures sont affichées ci-dessous.

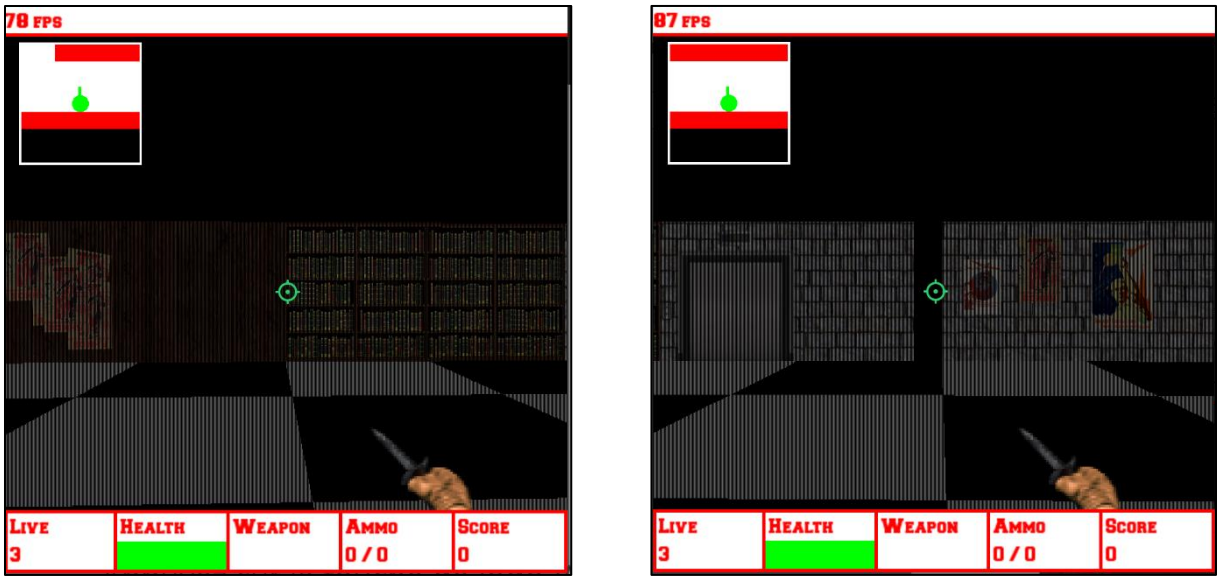


Figure 28 : Exemple de murs 1

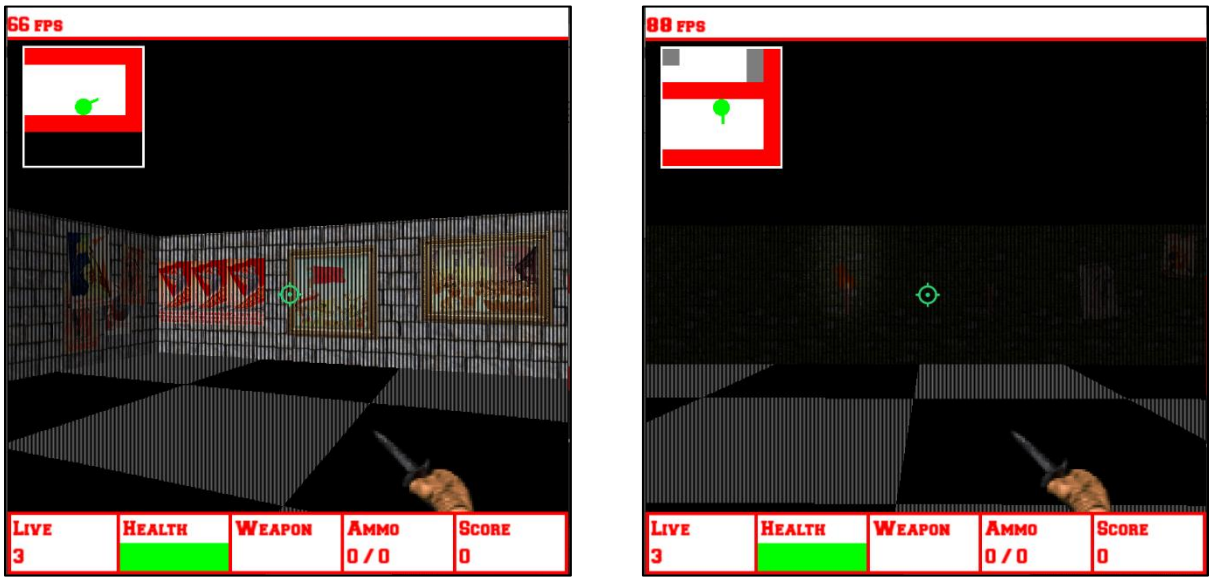


Figure 29 : Exemple de murs 2

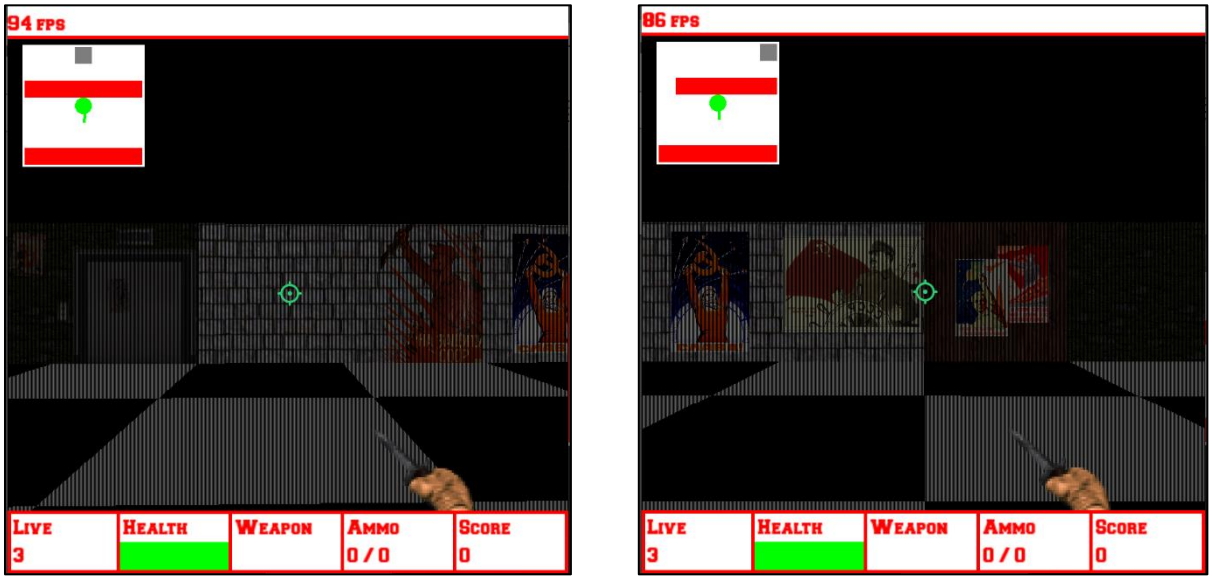


Figure 30 : Exemple de murs 3

7.2.2 -Différentes armes

L'arme de base, que le joueur possède toujours sur lui en tant qu'arme par défaut (et arme secondaire) est le couteau. Cette arme inflige des dégâts (faibles) aux ennemis se trouvant proches du joueur. Il n'utilise pas de munitions.

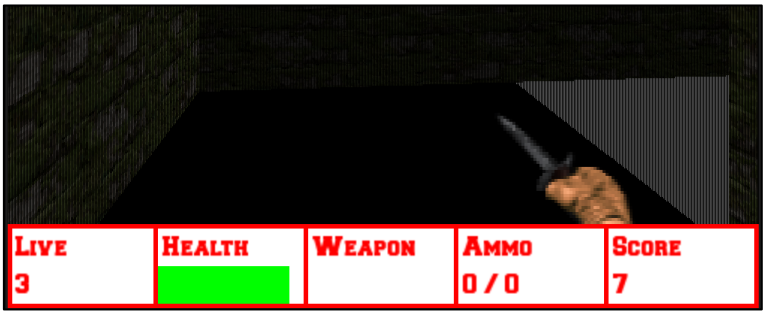


Figure 31 : Arme - couteau

Le pistolet est l'arme à feu la plus basique du jeu. Le joueur peut tirer au coup par coup, vidant à chaque fois une munition du chargeur qui en comporte douze au total. Ses dégâts ainsi que sa cadence de tir sont relativement faibles.

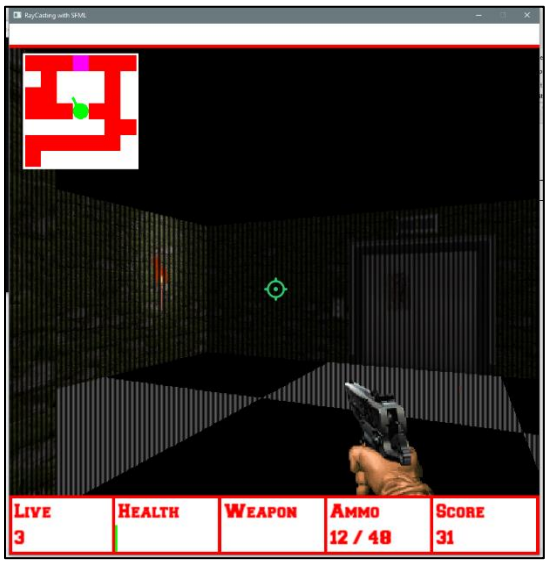


Figure 32 : Arme - Pistolet

Le fusil à pompe est une arme commune, facilement récupérable, tirant deux munitions par coup. Il inflige beaucoup de dégâts, mais a une faible cadence de tir et peu de munitions dans son chargeur.

Figure 33 : Arme - Fusil à pompes 1

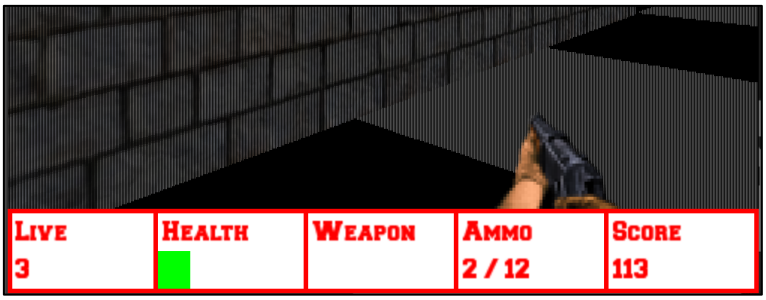


Figure 34 : Arme - Fusil à pompes 2

L'uzi est une arme relativement rare. Cette arme a deux modes de tir : coup par coup similaire au pistolet, et automatique avec une cadence de tir extrêmement élevée. Ses dégâts sont faibles.

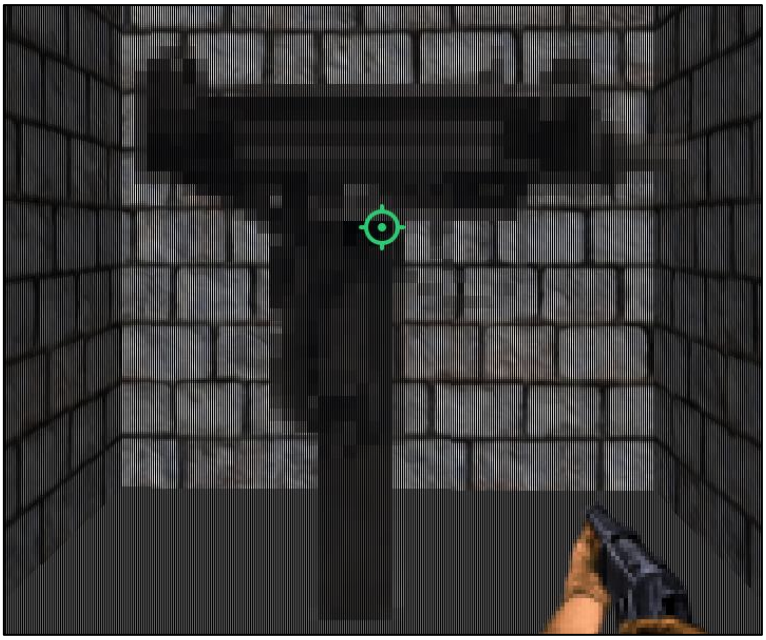


Figure 35 : Arme - Uzi 1

Figure 36 : Arme - Uzi 2

Le lance-grenades est très rare. Il tire des grenades ; celles-ci peuvent détruire certains murs (principalement les murs en bois) et infligent énormément de dégâts.

Figure 37 : Arme - Lance grenades 1

Figure 38 : Arme - Lance grenades 2

7.2.3 -Différents objets

Les coffres sont disséminés au travers du niveau et contiennent des armes ou des munitions. Lorsque l'on interagit avec ceux-ci, ils vont s'ouvrir en laissant place à ce qu'il contenait.

Figure 39 : Coffre

Les munitions sont génériques : toutes les armes possèdent les mêmes munitions. L'interaction avec une boîte de munitions rajoute un chargeur à l'arme que le joueur tient actuellement dans ses mains.

Figure 40 : Munitions

Les trousses de soins redonnent une vie supplémentaire au joueur.

Figure 41 : Trousse de soins

Il y a cinq clés différentes, qui peuvent chacune ouvrir une porte spécifique. Lorsque le joueur possède une ou plusieurs clés, celles-ci sont affichées en haut à droite de l'écran. Si le joueur utilise une clé pour ouvrir une porte, celle-ci disparait.

Figure 42 : Clé

Figure 43 : Porte

7.2.4 -Différents ennemis

Par manque temps, nous n'avons implémenté que deux types d'ennemis différents. Premièrement, les **Gardiens** : Ce sont les ennemis de base : faibles points de vie, faible cadence de tir et faibles dégâts.

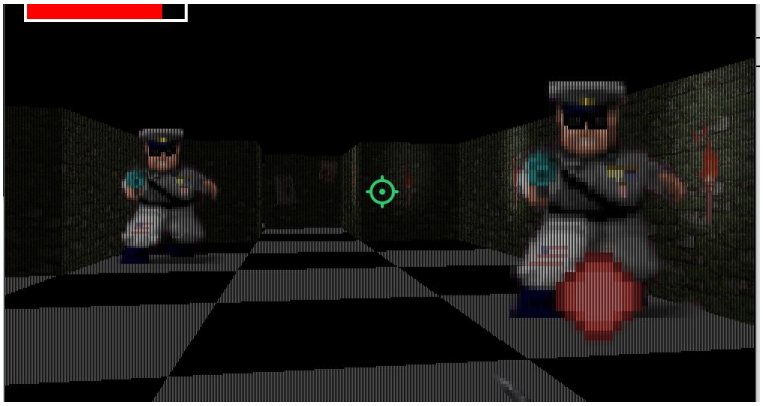


Figure 44 : Ennemi - Garde

Ensuite, les **Généraux** : Plus de points de vie et une meilleure cadence de tir.

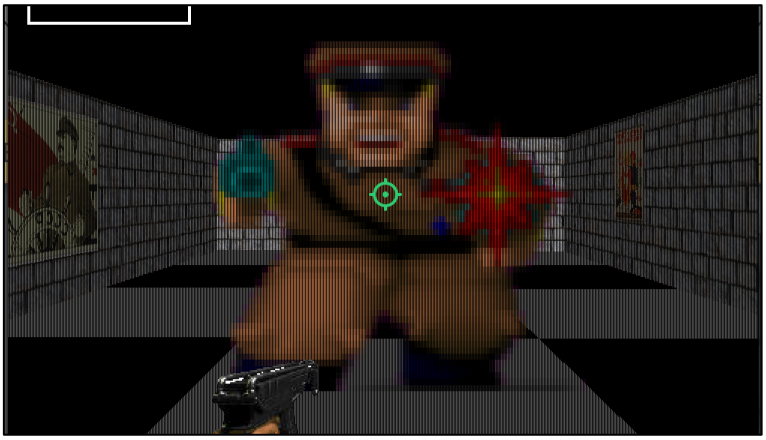


Figure 45 : Ennemi - Général

Lorsqu'un ennemi meurt, celui-ci peut potentiellement laisser tomber un objet.

7.3 - Tests

7.3.1 -Test du menu Accueil

Description du test	**Résultat : OK ou pas OK**
Lors du démarrage du jeu, on arrive sur le menu d'accueil et le son approprié se lance.	OK
Clic sur « Jouer », lance une partie.	OK
Clic sur « Paramètres », affiche les paramètres actuels.	OK
Si l'on modifie un paramètre, il est pris en compte.	OK
Si l'on clique sur « Enregistrer », on retourne au menu d'accueil.	OK
Si l'on clique sur « Quitter », le jeu se ferme	OK

7.3.2 -Test du menu Pause

Description du test	**Résultat : OK ou pas OK**
Lors d'une partie, appuyer sur ESC affiche le menu pause.	OK
Si l'on clique sur « Reprendre », on retourne au jeu.	OK
Si l'on clique sur « Commandes », les différentes commandes s'affichent	OK
Si l'on clique sur « Menu principal », on retourne au menu d'accueil.	OK
Si l'on clique sur « Paramètres », les paramètres actuels s'affichent.	OK
Si l'on clique sur « Quitter », le jeu se ferme.	OK

7.3.3 -Test du menu de fin de partie

Description du test	**Résultat : OK ou pas OK**
Si le joueur meurt dans le jeu, on atterrit dans le menu de fin de partie avec une image de défaite.	OK
Si le joueur termine le jeu, il atterrit dans le menu de fin de partie avec une image de victoire.	OK
Si l'on clique sur « Rejouer », on retourne dans le menu d'accueil.	OK
Si l'on clique sur « Quitter », le jeu se ferme.	OK

7.3.4 -Test du gameplay

Description du test	**Résultat : OK ou pas OK**
Le joueur réagit correctement à chaque commande (W → il avance …).	OK
Le joueur ne peut plus tirer s'il n'a plus de munitions.	OK
Les sons correspondent aux actions du joueur.	OK
Les ennemis tirent sur le joueur lorsqu'ils le voient. L'animation de tir lors d'un tir est lancée.	OK
On peut interagir avec toutes les entités du jeu.	OK
La mini-map suit les déplacements du joueur.	OK
L'interaction avec une entité effectue l'action correspondante.	OK
Lorsqu'un ennemi se fait tirer dessus (touché par une balle), il perd des points en fonction des dégâts de l'arme.	OK
Un ennemi qui n'a plus de vie meurt : Animation de mort, disparait et laisse potentiellement du loot.	OK

8 - Limitations et perspectives

8.1 - Optimisation des performances

8.1.1 -Multithreading

Un raycaster peut être parallélisé sur plusieurs threads, car le cast d'un rayon peut être effectué complètement indépendamment des autres. Malheureusement, nous n'avons aucune connaissance dans ce domaine (qui semble compliqué de prime abord), c'est pourquoi nous n'avons pas intégré cela dans notre projet.

8.2 - Ajouts possibles

Avec plus de temps, nous aurions pu ajouter divers éléments à notre jeu, tels que :

- Une leninade : Bonus redonnant de la vie (barre de vie, et non pas nombre de vies),
- Déplacer les ennemis à l'aide d'un algorithme (AStar par exemple),
- Ajouter un contexte par le biais de narration (avec un ou plusieurs PNJ),
- Améliorer visuellement les passages secrets,
- Passer le jeu en plein écran,
- Faire tester le jeu à d'autres personnes, et modifier l'équilibrage en conséquence,
- Ajouter plus d'animations (rechargement des armes entre autres),
- Équilibrer les différents effets sonores,

8.3 - Gestion de projet

Une meilleure utilisation des capacités fournit par git nous aurait permis de gagner beaucoup de temps, ainsi que d'avoir un workflow mieux structuré, et donc plus simple d'accès.

9 - Conclusion

Tous les objectifs principaux que nous avions fixés dans notre cahier des charges ont été réalisés. Pour ce qui est des objectifs secondaires, seuls, les dialogues et l'aspect histoire de notre jeu ont été mis de côté. Tous les autres objectifs secondaires ont aussi été réalisés. De plus, nous avons réalisé nous-mêmes notre gameLoop, et des sons différents pour chaque interaction ont été rajoutés.

Concernant le raycasting, nous avons aussi réussi à implémenter notre propre version adaptée à la librairie SFML. Nous nous sommes évidemment basés sur certaines sources notamment pour le principe général et pour les calculs à effectuer. Cependant, l'implémentation reste la nôtre.

Nous avons aussi imaginé et codé les cinq map disponibles dans le jeu en imaginant un système de valeurs qui représentent chacune un élément bien précis de notre jeu.

Finalement, toutes les textures présentes dans le jeu ont été adaptées et travaillées par nos soins.

10 - Annexes

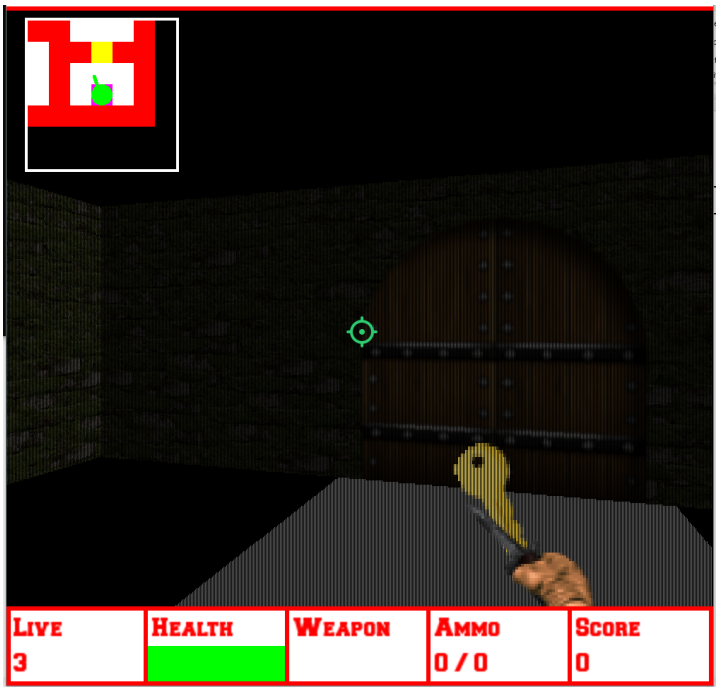
10.1 - Guide utilisateur

Lorsque vous démarrez le jeu, vous arrivez sur le menu d'accueil :

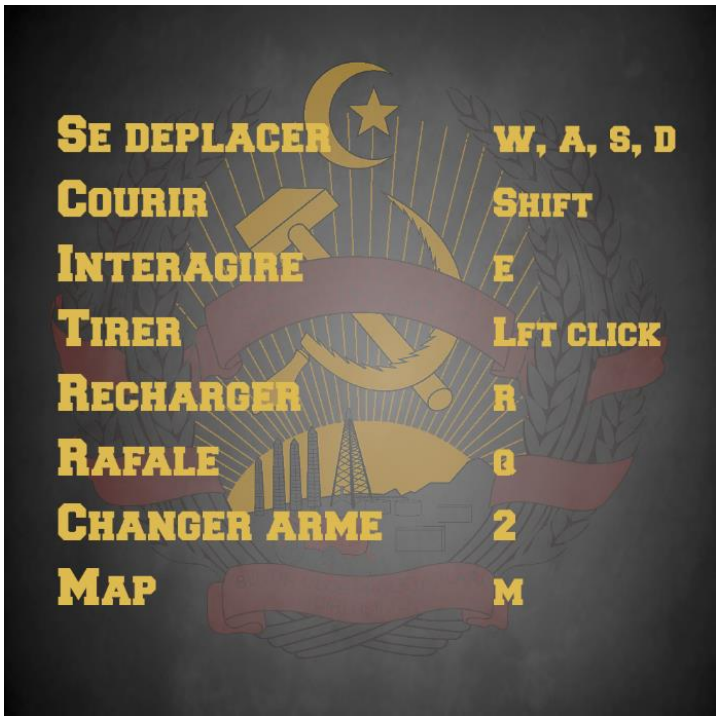
Si vous décidez de vous intéresser aux paramètres, cliquez sur le bouton prévu à cet effet et voici ce que vous pourrez paramétrer :

Les options sont assez évidentes à comprendre et il suffit de cliquer sur les boutons pour mettre à jour vos préférences. Une option intéressante est celle de la **Map.** Si vous laissez sur auto vous enchaînerez les map. Alors que si vous en sélectionnez une, votre partie se limitera à la map choisie.

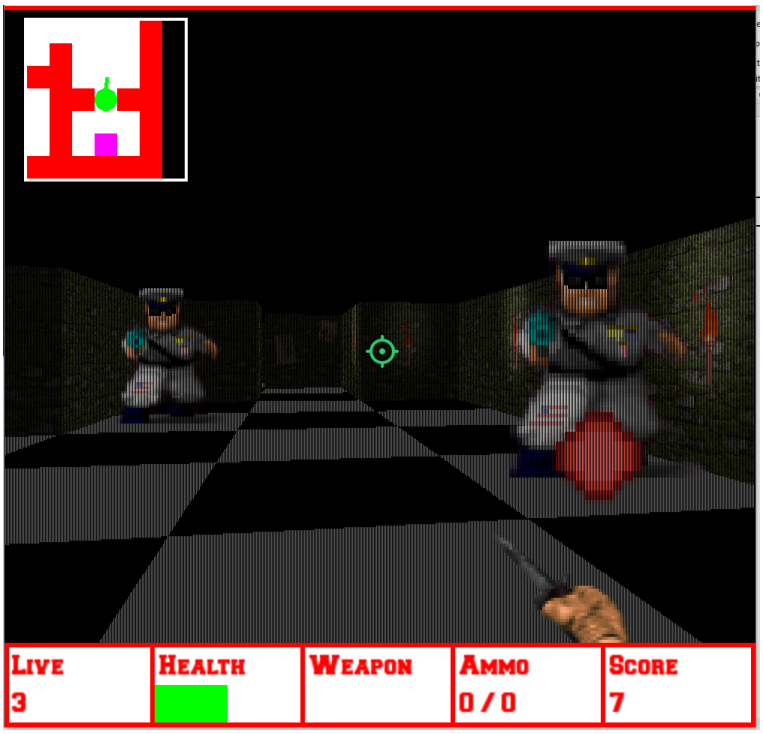
Une fois les paramètres enregistrés (clic sur enregistrer) vous pouvez lancer le jeu. Voici ce qui vous attend :



Vous possédez de nombreuses indications sur votre état telles que le nombre de vies restantes, votre score ou encore vos munitions. Pour interagir avec les entités que vous rencontrerez, appuyez la touche **E** de votre clavier. Pour plus de détails sur les commandes :



Si vous récupérez la clé se trouvant devant vous, vous la verrez apparaître dans votre inventaire de clé. Grâce à cette clé, il est vous est maintenant possible de déverrouiller la porte associée à cette clé. Dans ce cas, c'est la porte qui se trouve en face de vous. Pour déverrouiller une porte comme pour interagir avec toutes les entités, appuyez sur **E**. Vous voici maintenant sorti de la zone de départ et face à des ennemis.



Vous constaterez que votre barre de vie descend gentiment lorsque les ennemis vous tirent dessus. Vous pouvez tenter de tuer les ennemis avec votre couteau ou immédiatement prendre la fuite. Vous pourrez aussi trouver des coffres ou d'autres entités en explorant la map.

Le but du jeu est de se frayer un chemin dans les labyrinthes que vous rencontrerez pour trouver l'ascenseur qui vous permettra de gagner. Voici à quoi ressemble un ascenseur :

Si vous utilisez l'ascenseur, vous atteindrez soit le niveau suivant (choix map : auto) soit la fin du jeu, car votre partie se limite à une map définie. Voici à quoi ressemblerait une fin de partie en cas de victoire :

Cependant si vous veniez à perdre tous vos points de vie vous finiriez plutôt dans le menu de fin qui ressemble à ceci :

Sachez qu'à n'importe quel moment dans le jeu vous avez aussi la possibilité d'appuyer sur la touche **ESC** et un menu de pause comportant de nombreuses options vous permettra de faire le point sur votre vie…

10.2 - Cahier des charges

10.2.1 - Contexte

Dans le cadre de notre projet P2, nous avons pour but de réaliser un jeu. Nous avons décidé de réaliser un jeu dans le style du premier Wolfenstein.

Figure 46 : Illustration du jeu Wolfenstein , url : preterhuman.net

10.2.2 - Pitch

Vous Vladimir Poutine, devez aller sauver vos anciens collègues du KGB tombés dans un piège de l'horrible Biden. Pour ce faire, vous partirez armé de votre Makarov PV et irez affronter, seul, des hordes d'affreux soldats américains gardant une base tenue secrète dans le New Hampshire.

Heureusement, grâce à vos connaissances universelles vous savez exactement où se trouve l'entrée de cette base !

10.2.3 - Principe du jeu

Vous parcourez une série de labyrinthes et rencontrez divers ennemis que vous devez tuer.

10.2.4 - Moyen

- **IDE et langage** : Visual studio Community 2019, C++
- **Cible** : Windows (exe)
- **Technique 3D** : Raycasting
- **Librairie** : SFML

10.2.5 - Objectifs principaux

- Créer un jeu de tir à la première personne en 3D
- Créer des menus (accueil/GameOver)
- Ajouter des ennemis dans la map et la gestion des vies (gameplay rules)
- Ajouter des entités avec ou sans interactions
- Ajouter un HUD contenant toutes les infos du joueur

10.2.6 - Objectifs secondaires

- Ajouter les options dans le menu accueil (choix de la map, langue, difficulté)
- Ajouter l'univers soviétique par des textures personnalisées
- Ajouter un menu pause
- Ajouter un univers sonore (dont dialogue)
- Ajouter une histoire complète au jeu par des pnj et slides ou bulles de texte et audio
- Ajouter une mini-map pour se repérer
- Ajouter différentes armes et la possibilité de changer d'arme
- Lissage des textures
- Ajout d’animation

10.3 - Table des illustrations

Les figures 16 à 19 sont des images explicatives réalisées par nos soins.

Toutes les autres figures sont des captures d'écran de notre jeu.

10.4 - Bibliographies et références

10.4.1 - Sites Web

Tutoriel sur le raycasting : https://permadi.com/1996/05/ray-casting-tutorial-table-of-contents/

Concept du raycasting : https://lodev.org/cgtutor/raycasting.html

Wolfenstein : https://www.pinterest.it/pin/461407924312569128/

Configuration VS2019 pour SFML en C++ : https://www.sfml-dev.org/tutorials/2.5/start-vc.php

Structure du projet (gameState) : https://www.binpress.com/creating-city-building-game-with-sfml-state-manager/

Tutoriels et documentation SFML : https://www.sfml-dev.org/tutorials/2.5/

Raycasting avec SFML : https://lodev.org/cgtutor/raycasting.html#Textured_Raycaster

Animation de Sprite : https://www.youtube.com/watch?v=BLtnd54AMvw&ab_channel=ChiliTomatoNoodle

Animation de VertexArray : https://stackoverflow.com/questions/24616677/how-can-animate-textures-in-a-vertex-array et https://en.sfml-dev.org/forums/index.php?topic=9166.0

Exemple de classe bouton avec SFML : https://github.com/TheMaverickProgrammer/Swoosh/blob/master/ExampleDemo/Button.h

Raycasting avec textures : https://lodev.org/cgtutor/raycasting.html#Textured_Raycaster