TPI - Physical Event Simulation

Simulation d’expérience physique en C++ avec la bibliothèque SFML

Auteur : Lucas Charbonnier

Classe : MID4

Lieu : ETML

Temps à disposition : 90 heures (28 avril - 28 mai 2021)

Chef de projet : Dimitri Lymberis

Experts : Xavier Carrel, Charles-Henri Hayoz

Table des matières

[Analyse 2](#_Toc70689510)

[Méthodologie de projet 2](#_Toc70689511)

[Technologies à utiliser 3](#_Toc70689512)

[Langage de programmation 3](#_Toc70689513)

[Bibliothèques 3](#_Toc70689514)

[Méthode de simulation : discrète ou continue 3](#_Toc70689515)

[Choix de la méthode et design pattern 4](#_Toc70689516)

[Gestion des collisions 4](#_Toc70689517)

[Détection et résolution d’une collision entre 2 billes 4](#_Toc70689518)

[Convention de code/nommage 6](#_Toc70689519)

[Interface graphique 6](#_Toc70689520)

[Menu principal 6](#_Toc70689521)

[Interface de la première expérience : « chocs » 7](#_Toc70689522)

[Interface de la deuxième expérience « balistique » 7](#_Toc70689523)

[Planification initiale 8](#_Toc70689524)

[Réalisation 8](#_Toc70689525)

[Mise en place de l’environnement pour compiler le code 8](#_Toc70689526)

[Tests 8](#_Toc70689527)

[Conclusion 8](#_Toc70689528)

[Bibliographie 8](#_Toc70689529)

# Analyse

## Méthodologie de projet

Pour mener à bien le projet, il a été décidé que la **méthode des 6 pas** serait utilisée pour structurer son déroulement. Cela veut dire que le projet sera séparé en plus ou moins 6 étapes :

* S’informer 🡪 prise de connaissances du cahier des charges
* Planifier 🡪 planification initiale détaillée
* Décider 🡪 analyse et conception des fonctionnalités
* Réaliser 🡪développement de l’application
* Contrôler 🡪 Effectuer les différents tests
* Évaluer 🡪 En général, l’évaluation d’un projet sert à savoir ce qui a bien fonctionné (ou non) dans le but d’améliorer le ou les processus de travail.

## Technologies à utiliser

### Langage de programmation

Le langage de programmation utilisé sera le C++. C’est un langage bas-niveau et adapté à la création de simulation de physique. De plus, c’est le langage avec lequel je suis le plus familier.

### Bibliothèques

Le langage C++ seul ne permet de créer des applications graphiques, c’est pour cela que la SFML sera également utilisée. La SFML (Simple and Fast Multimedia Library) est une bibliothèque multimédia permettant, entre autres, de créer des applications graphiques.

Il est important de noter que la SFML n’est pas un moteur de jeu à proprement parler. C’est une bibliothèque graphique relativement bas niveau qui permet d’afficher des formes (rectangles, cercles, etc…) sur une fenêtre Windows mais elle ne propose pas de fonctionnalités avancées telles que l’affichage de widgets ou le calcul vectoriel. Le site officiel de la SFML : <https://www.sfml-dev.org/index-fr.php>

La détection/résolution de collisions élastiques nécessitera l’utilisation de calcul vectoriel, c’est pour cela que, en plus de la SFML, une bibliothèque de calcul vectoriel sera utilisée. J’utiliserai ma propre bibliothèque que j’ai développée durant mon temps libre : <https://github.com/Raynobrak/Charbrary>.

## Méthode de simulation : discrète ou continue

Il existe deux grandes manières de simuler des systèmes physiques : de manière discrète ou de manière continue. Il est nécessaire d’examiner ces deux méthodes pour pouvoir choisir laquelle est la plus adaptée au développement du projet.

Dans une simulation discrète, on calcule la simulation bout par bout jusqu’à ce qu’elle se termine. Pour cela, on définit un « pas de temps » (également appelé « time step » ou « ∆t ») qui correspond à la fréquence à laquelle la simulation sera mise à jour.

Un pas de temps petit (p.ex. 50 millisecondes) donnera un résultat plus précis qu’un pas de temps long (p.ex. 500 millisecondes) mais cela sera également plus couteux en puissance de calcul puisque, pour calculer 1 seconde de simulation, le premier ∆t nécessitera 20 mises à jour alors que le deuxième ∆t n’en nécessitera que 2. De plus, un pas de temps trop petit risque de donner un visuel saccadé.

Dans une simulation continue, le système physique est représenté comme une équation ou un système d’équations. On peut connaître l’état de la simulation à un certain moment en faisant varier le paramètre t dans l’équation.

C’est la méthode qui donnera le résultat le plus précis mais elle nécessite de disposer d’un modèle mathématique permettant de représenter le système sous forme d’équation, ce qui n’est pas toujours possible.

Prenons l’exemple d’un objet qui se déplace à une vitesse constante de gauche à droite. Dans une simulation continue, on pourrait calculer la position ***x*** de l’objet en fonction de la vitesse ***v*** et du temps écoulé ***t*** depuis le début de la simulation :

Alors que dans une simulation discrète, la position de l’objet à une certaine frame ***i+1*** sera calculée en fonction de sa position à la frame précédente ***i*** et en fonction du ***∆t*** et de sa vitesse :

En d’autres termes, dans une simulation continue on calcule directement la position absolue alors que dans une simulation discrète celle-ci se calcule en faisant la somme de tous les déplacements.

### Choix de la méthode et design pattern

Les simulations discrètes sont presque toujours plus simples à implémenter que les simulations continues parce que les formules à concevoir sont forcément plus simples (pour la même raison qu’il est plus facile d’approximer une intégrale plutôt que de calculer sa valeur exacte).

Une simulation continue aurait pu être considérée pour la deuxième expérience mais le fait qu’il faille gérer les collisions avec le « panier » compliquerait la formule. La première expérience, elle, est constituée de plusieurs objets et serait pratiquement impossible à simuler de manière continue, étant donné que c’est un système chaotique (une petite variation des paramètres de départ mènera à une situation finale totalement différente).

Il a donc été décidé que les 2 expériences seraient simulées de manière discrète.

## Gestion des collisions

La gestion des collisions sera une partie essentielle au fonctionnement de l’application. Dans une simulation de physique, on s’attend à ce que le résultat ressemble le plus possible à ce qu’on pourrait observer dans le monde réel, il faudra donc apporter un soin particulier à cette partie pour que le résultat paraisse le plus naturel possible.

Avant d’aller plus loin, il est important de bien expliquer en quoi consiste la « gestion des collisions ». Elle se divise en deux parties : la détection des collisions et la résolution des collisions.

La détection consiste à analyser la position, la forme et la taille de deux objets et à déterminer s’ils se superposent ou non. De plus, la détection des collisions englobe également le calcul d’informations telles que la normale de collision ainsi que la profondeur de la collision (nous reviendrons sur ces deux éléments).

Un état de collision est un état non désirable et c’est surtout un état abstrait qui ne peut exister que dans une simulation informatique. Dans la vraie vie, les objets ne peuvent pas occuper le même espace, une superposition est donc impossible. Pour que la simulation apparaisse réaliste, il est donc nécessaire de réarranger les objets pour qu’ils ne soient plus en collision. Ce processus s’appelle la résolution de collisions.

La résolution de collision consiste à déplacer et/ou appliquer une impulsion aux objets en état de collision dans le but qu’ils ne se collisionnent plus lors de la prochaine vérification.

### Détection et résolution d’une collision entre 2 billes

Nous allons voir comment détecter et résoudre une collision élastique entre 2 billes de masses différentes. Une collision élastique est une collision ou l’énergie cinétique totale des 2 billes est conservée et qui implique donc l’application d’une impulsion aux deux billes lors de la collision.

#### Détection de la collision

Nous sommes en 2D donc les billes sont analogues à des cercles. Les collisions entre deux cercles sont très simples à vérifier : il suffit de vérifier si la distance entre les centres des cercles est inférieure à la somme de leurs rayons. Si c’est le cas, les deux cercles se superposent. Pour simplifier et retirer la racine carrée (très couteuse au niveau du CPU), on peut élever au carré :



S’il n’y pas de collision, il va de soi que le programme n’ira pas plus loin. En revanche, si une collision est détectée, il va être nécessaire de déterminer deux informations essentielles : La normale de collision et la profondeur de collision.

La normale de collision correspond à l’axe le long duquel la collision se produit. D’une certaine manière, elle représente la « direction » de la collision. C’est une droite mais on la représente généralement sous forme d’un vecteur unitaire. Pour calculer ce vecteur unitaire, il faut diviser la position relative des deux cercles par la distance les séparant :

<formule>

On a maintenant une normale de collision pointée en direction du cercle C\_2. Elle indique la direction vers laquelle le cercle C\_2 doit être « poussé » pour résoudre la collision. Pour le cercle C\_1, on prendra l’opposé de la normale.

La profondeur de la collision peut être calculée en faisant le produit scalaire entre la normale de collision et la distance relative entre les deux cercles sous forme de vecteur :

<formule>

#### Résolution de la collision : déplacement

On dispose maintenant de la normale de collision et de la profondeur de la superposition des deux cercles. Il est temps de passer à la résolution. La première étape de la résolution consiste à repositionner les cercles pour qu’ils ne se superposent plus.

Pour cela, il faut calculer le vecteur de translation qui sera additionné ou soustrait en fonction du cercle concerné. Les propriétés de ce vecteur seront identiques pour les deux cercles, à une différence près : son sens. Pour repositionner le cercle C1, il faudra soustraire le vecteur de déplacement à sa position alors que pour C2, il faudra l’additionner.

Pour calculer ce vecteur, on amplifie la normale de collision par la moitié de la profondeur de la collision. On prend la moitié car ce vecteur sera appliqué pour les deux cercles, ce qui donnera un déplacement total égal à la profondeur :

<formule de la nouvelle position : position’ = position + déplacement>

<formule du calcul du vecteur de déplacement pour C1 et C2>

#### Résolution de la collision : impulsion

Lorsque deux billes se collisionnent, elles ne sont pas juste déplacées, elles rebondissent l’une contre l’autre. Il y a donc un changement au niveau de leur vitesse et direction de déplacement.

C’est ici que les choses se compliquent un peu. En plus de prendre en compte la normale de collision ainsi que la position des cercles, il va également falloir tenir compte de leurs vélocités et de leurs masses pour pouvoir leur appliquer une impulsion. Pour rappel, la vélocité est un vecteur représentant la vitesse et la direction de déplacement d’un objet.

Avant d’aller plus loin, il est nécessaire d’introduire le concept d’impulsion. Une impulsion est définie comme un changement de vitesse instantané. La direction dans laquelle l’impulsion est appliquée est connue, c’est la normale de collision. Comme pour le déplacement, l’impulsion sera additionnée ou soustraite à la vélocité en fonction de la bille concernée.

<à terminer>

La deuxième étape de la résolution consiste en l’application d’une impulsion aux deux billes.

## Convention de code/nommage

Les normes de l’ETML en matière de programmation définissent la manière dont les classes, méthodes et variables doivent être nommées.

Cependant, ces normes sont définies pour le langage C#, PHP et javascript, alors que le projet sera développé en C++. Pour cette raison, les modifications suivantes ont été apportées :

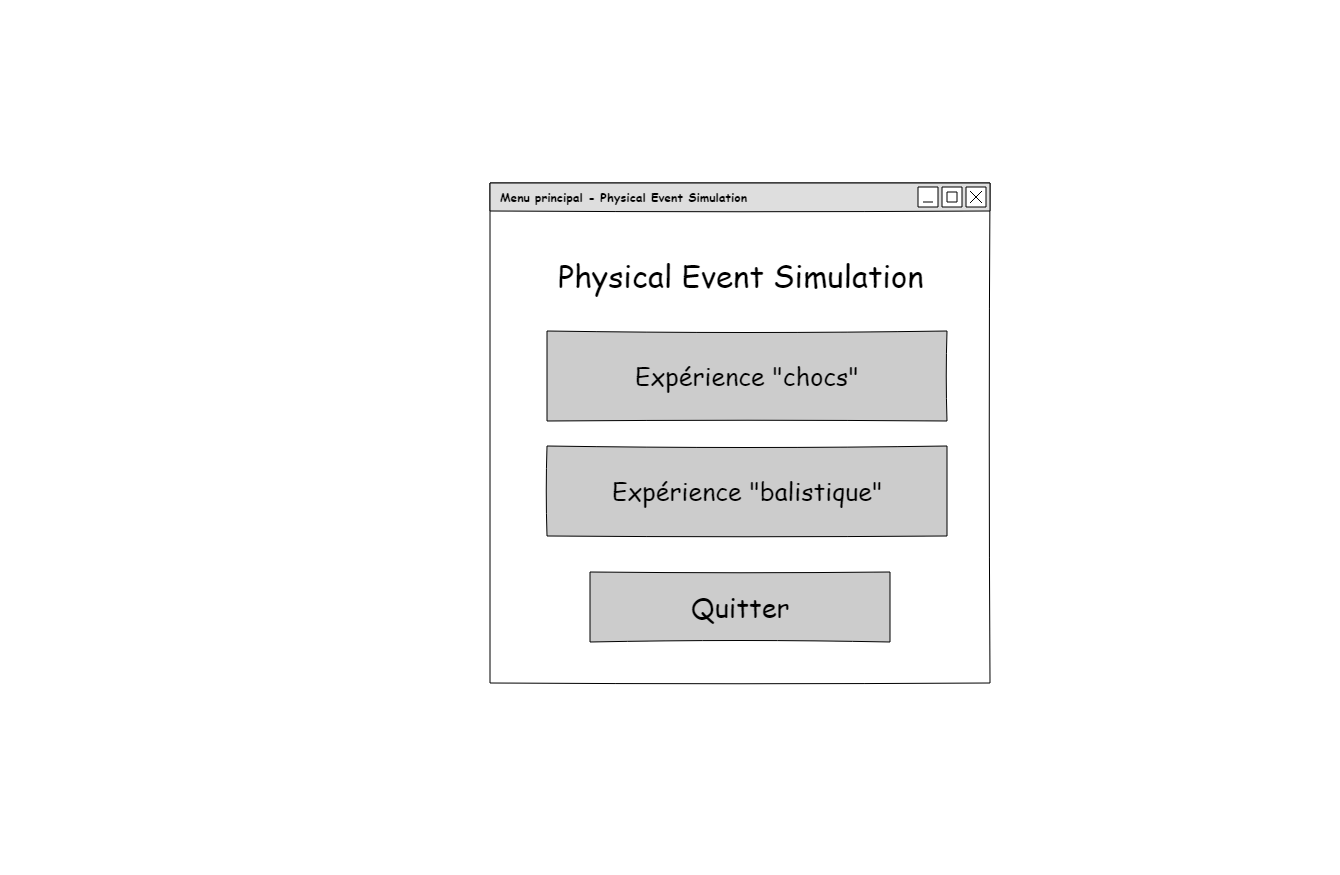
* Les fonctions membres d’une classe suivront le lowerCamelCase au lieu du UpperCamelCase. C’est une convention la majorité des développeurs C++ suivent.
* Les fonctions libres (non-membre) suivront le snake\_case. Cela permet de les différencier clairement des fonctions membres.
* Les variables membres privées d’une classe seront suivies (au lieu d’être précédées) d’un underscore : « foo\_ ». Les variables précédées d’un underscore sont réservées aux implémentations de la STL du langage C++. Le respect de cette convention facilitera la maintenance du code puisqu’elle permettra d’éviter d’éventuels conflits de noms.

## Interface graphique

Étant donné la nature de l’application à développer (une simulation), un soin particulier devra être apporté au visuel. Le but est de développer une interface simple et agréable à utiliser. Cependant, la bibliothèque graphique utilisée (SFML) est d’assez bas-niveau et c’est pour cela que j’ai choisi de partir sur un design « minimaliste ». Ci-dessous, vous trouverez les maquettes que j’ai conçues pour l’application.

### Menu principal

Le menu principal est très simple et ne contient qu’un titre avec 2 boutons permettant d’accéder aux différentes expériences et un dernier bouton pour quitter l’application.

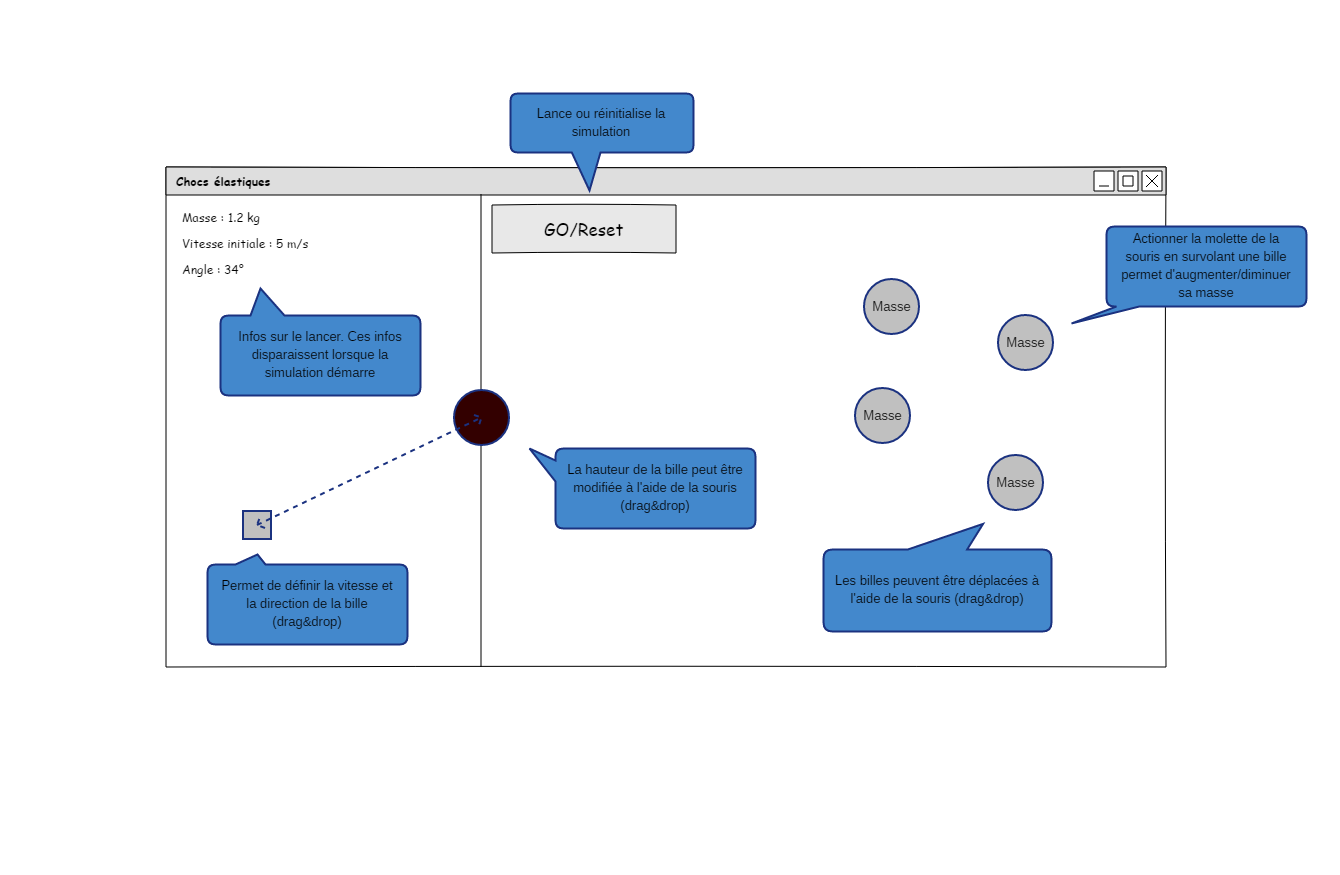


### Interface de la première expérience : « chocs »

Pour permettre de définir la vitesse et la direction de la bille principale, j’ai créé un « lanceur » (le petit carré connecté à la bille sombre). La vitesse de départ ainsi que la direction seront défini en calculant la distance relative séparant la bille et le carré.

Les billes pourront être déplacées en les glissant avec la souris (drag&drop). Il va de soi que la bille principale ne bourra bouger que verticalement et que les billes ne pourront pas se trouver en dehors de l’écran ou derrière le lanceur.

La masse des billes pourra être définie en actionnant la molette de la souris en les survolant (glisser vers l’avant pour augmenter la masse et vers l’arrière pour la diminuer). La masse exacte de chaque bille sera affichée dessus.

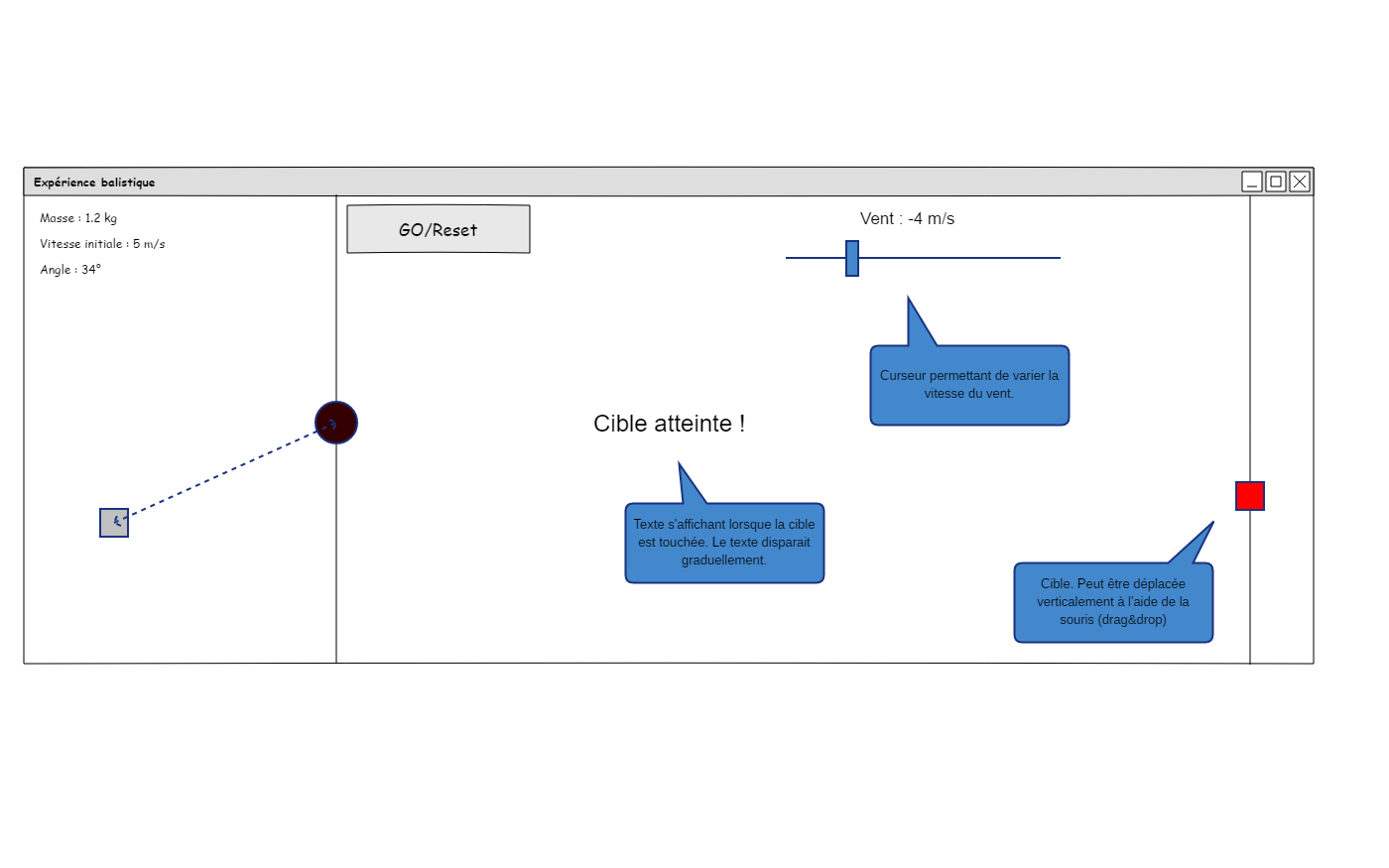


### Interface de la deuxième expérience « balistique »

La configuration de la bille de lancer est identique à l’autre expérience. La cible est positionnée de la même façon que la bille (en la glissant à l’aide de la souris).

Il est possible de modifier l’intensité et la direction du vent à l’aide du curseur. Une valeur négative indiquera un vent allant « contre » la bille et une valeur positive correspondra à un vent « poussant » la balle vers la droite.

Si la bille atteint la cible (collision), un message s’affichera au centre de l’écran. Le message disparaîtra graduellement (fade out).



# Planification initiale

La planification initiale du projet est disponible dans le même fichier que le journal de travail, en annexe.

# Réalisation

À faire

## Mise en place de l’environnement pour compiler le code

À faire : link sfml

# Tests

À faire

# Conclusion

# Bibliographie

|  |  |
| --- | --- |
| [1] | «Simulation à événements discrets,» [En ligne]. Available: https://fr.wikipedia.org/wiki/Simulation\_%C3%A0\_%C3%A9v%C3%A9nements\_discrets. |
| [2] | «Simulation informatique,» [En ligne]. Available: https://fr.wikipedia.org/wiki/Simulation\_informatique. |