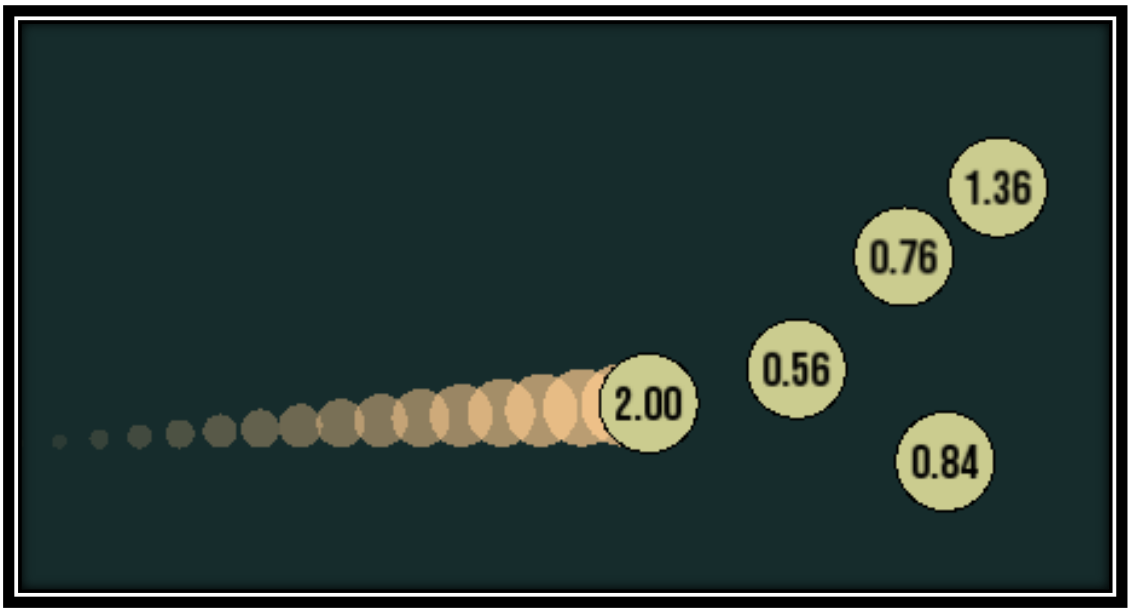
TPI - Physical Event Simulation

Simulation d'événements physiques en C++ avec la bibliothèque SFML



Auteur : Lucas Charbonnier

Classe : MID4 (2017-2021)

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

1 Glossaire

Étant donné la nature du projet – une simulation de physique – de nombreux termes techniques du domaine seront utilisés et, bien que ceux-ci soient tous définis et expliqués dans ce document, il est préférable de les regrouper ici.

La suite de ce document est rédigée en partant du principe que le lecteur a pris connaissance des termes définis dans le tableau ci-dessous.

Termes (triés de A-Z)	Définitions
Accélération	Grandeur définissant la rapidité avec laquelle la vitesse d'un corps augmente. Peut être un vecteur ou simplement un scalaire.
Balistique	La balistique est un domaine scientifique qui se concentre sur l'étude du mouvement des projectiles, leur **trajectoire**.
C++	Langage de programmation compilé multiparadigme.
Collision	Choc entre deux corps physiques. Une collision est qualifiée de « **parfaitement élastique** » si l'énergie cinétique totale est la même avant et après la collision. En d'autres termes, une collision parfaitement élastique implique une conservation totale de l'énergie cinétique. Dans les simulations de physique, une collision correspond à un état non-désirable entre deux corps et qui doit être **résolu**.
Fonction libre	Une fonction libre est, en C++, une fonction n'appartenant pas à une classe, contrairement à une **fonction membre** ou une **méthode**.
Frottement	Force subie par un corps dans la direction opposée à son déplacement.
Pas de temps / Δt / « timestep »	Intervalle de temps correspondant à la durée séparant le calcul de chaque frame dans une simulation discrète[1].
Scalaire	Réel qui multiplie un vecteur. Parfois, au lieu de calculer immédiatement la valeur exacte d'un vecteur, on calcule séparément la direction (un vecteur unitaire) et le scalaire, avant de les multiplier pour obtenir le vecteur final.
SFML	Sigle de **Simple and Fast Multimedia Library**. Bibliothèque informatique développée par Laurent Gomila permettant le développement d'applications graphiques en C++. C'est une bibliothèque bas-niveau basée sur **OpenGL**.

[1] Le concept de simulation « discrète » et son opposition aux simulations dites « continues » est expliqué plus loin dans ce document : Méthode de simulation

Simulation informatique	Programme informatique ayant pour but d'imiter un phénomène provenant du monde réel. Le but d'une simulation informatique est généralement de **prédire** l'évolution d'un système en fonction de ses conditions de départ.
Vecteur	Segment définit par une direction et une longueur. Les vecteurs sont couramment utilisés pour représenter : • La **position** d'un point sur un plan • La **vitesse** d'un corps • Une **accélération** subie par un corps • Une **force** appliquée à un corps
Vecteur unitaire	Un vecteur avec une magnitude/longueur de 1. Les vecteurs unitaires sont couramment utilisés pour représenter une direction.
Vélocité / vecteur vitesse	Vecteur représentant la **vitesse** ainsi que la **direction** dans laquelle se dirige un projectile. À ne pas confondre avec la vitesse, qui est un scalaire représentant la vitesse absolue d'un projectile.
Widget	Un widget est un élément d'une interface graphique (p.ex. un bouton ou un curseur).

2 Abstract

Dans le cadre du Travail Pratique Individuel de 4ème année à l'ETML, Lucas Charbonnier a disposé de 90 heures de travail (du 30 avril au 30 juin 2021) pour planifier, réaliser et documenter un projet de développement informatique. Le projet consistait en le développement d'une application comportant 2 simulations de physique en deux dimensions : Une simulation de collisions élastiques entre des billes de masses différentes mais de même rayon ainsi qu'une simulation de balistique prenant en compte la gravité et le frottement de l'air et du vent. Le but de ce projet est de valider les compétences de développement et de gestion de projet acquises durant la formation à l'ETML.

Pour mener à bien le projet, une planification initiale a été effectuée. Tout au long du projet, un journal de travail a été tenu afin de permettre au développeur, aux experts et au chef de projet de suivre l'avancement du TPI. Une fois par semaine, un rendu comportant le rapport, le journal de travail ainsi qu'une capture d'écran résumant l'avancement du développement a été envoyé aux experts et au chef de projet. Le développement a été effectué en C++ sur Visual Studio et avec la bibliothèque graphique bas-niveau SFML (Simple And Fast Multimedia Library).

Finalement, les deux simulations spécifiées dans le cahier des charges ont pu être implémentées dans leur entièreté. La simulation de collisions permet de prédire l'évolution d'un système physique composé de 5 corps rigides circulaires de masses différentes en simulant les différentes collisions élastiques pouvant se produire. La deuxième simulation est capable de prédire si la trajectoire d'un corps sphérique de masse variable au sein d'un fluide en mouvement (ou statique) atteindra une cible.

3 Analyse

3.1 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.

3.2 Technologies à utiliser

Ce chapitre contient l'analyse et le choix des différentes technologies qui seront utilisées pour le projet.

3.2.1 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.

3.2.2 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.

3.3 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.

3.3.1 Différence de langage pour le code et les commentaires

Tous les intervenants du projet parlent français et c'est la langue avec laquelle je suis le plus à l'aise pour expliquer des choses compliquées. Pour cette raison, la documentation (commentaires) du programme seront rédigés en français. De plus, si ce projet venait à être repris, il y a de grandes chances que la personne en charge soit également francophone. Il n'y a donc, à priori, aucune raison d'utiliser une autre langue que le français pour les commentaires.

Cependant, à part les commentaires, le reste du code sera rédigé en anglais. L'anglais est la langue universelle de l'informatique et c'est celle qui est utilisée par la majorité des bibliothèques (dont la SFML). Cette unicité permet d'éviter le mélange de langues dans le code, ce qui garantira une lecture plus facile de celui-ci.

De plus, l'utilisation de l'anglais permet d'éviter le problème des accents (é,è,à) qui ne sont pas gérés correctement par tous les IDEs.

3.4 Méthode de simulation

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 « Δt ») 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 Δt nécessitera 20 mises à jour alors que le deuxième Δt 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 :

$$x = v \times t$$

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 ***Δt*** et de sa vitesse :

$$x_{i+1} = x_i + v \times \Delta t$$

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.

3.4.1 Choix de la méthode

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.

3.4.2 Collisions élastiques

En physique, une collision est qualifiée d'élastique si l'énergie totale du système (les deux objets) est conservée après la collision. En d'autres termes, cela veut dire que l'énergie cinétique totale des deux billes est la même avant et après la collision.

Cette spécificité simplifie les formules de physique permettant de résoudre des collisions. S'il était question de collisions inélastiques, il faudrait introduire le concept de restitution, qui définit la quantité d'énergie conservée après la collision.

Cependant, la simulation de collisions parfaitement élastiques peut souvent paraître très irréaliste. Cela est dû au fait que de telles collisions ne peuvent avoir lieu dans le monde réel. En effet, quand 2 boule de billard

se collisionnent, une partie de l'énergie cinétique sera perdue[2]. Ces pertes sont la raison pour laquelle les boules de billard s'arrêtent et, si l'on ne les simule pas, nos billes ne vont tout simplement jamais s'arrêter de bouger.

Des collisions parfaitement élastiques ne seraient donc pas souhaitables si l'on développait, par exemple, un jeu vidéo où le rendu visuel doit correspondre le plus possible à l'intuition humaine (pour ne pas « frustrer » le joueur). Toutefois, Physical Event Simulation n'est pas un jeu vidéo, c'est une simulation d'évènements physique et, s'il fallait introduire le concept de restitution, il faudrait également défendre les valeurs de restitution choisies. Vu que les matériaux des « billes » ne sont pas spécifiés, ce choix serait donc arbitraire et c'est pour cette raison (et surtout parce que c'est ce qui est spécifié dans le cahier des charges) que des collisions purement et parfaitement élastiques seront simulées.

Le cahier des charges insiste sur la rigueur et l'exactitude des formules choisies. Il a donc été préféré de développer une simulation simple mais exacte, plutôt qu'une simulation complexe mais inexacte.

3.4.3 Analyse détaillée de certaines fonctionnalités

3.4.3.1 Collision entre le projectile et le récipient de l'expérience de balistique

Le cahier des charges mentionne que la simulation de balistique doit permettre la détection lorsque le projectile tombe dans le récipient.

Cependant, il est étonnamment complexe de définir à quel moment un projectile tombe dans le récipient. En effet, pour implémenter une telle fonctionnalité, il faudrait commencer par définir ce qu'on entend par « tomber dans le récipient ». En effet, cela soulève plusieurs questions :

- À partir de quel moment le projectile est-il dans le récipient ? Lorsqu'il entre dedans ou lorsqu'il est complètement à l'intérieur ?
- Est-ce que le projectile peut rebondir avec le récipient et, si oui, que fait on si le projectile tombe dans le récipient et rebondit en dehors (comme cela arrive parfois au basket) ?
- Le projectile doit-il être immobilisé à l'intérieur du récipient pour que le tir compte ? Et si oui, comment cela est-il possible si l'on ne simule que des collisions parfaitement élastiques ?
- Comment résoudre une collision avec une forme non-convexe ?

Pour ne pas perdre de temps à essayer à répondre à ces questions qui n'ont pas d'unique bonne réponse, il a été décidé de remplacer le récipient par une simple cible. Au lieu d'informer l'utilisateur lorsque le projectile tombe dans le récipient, on l'informera lorsque le projectile touchera la cible. Ce changement simplifiera grandement la simulation puisque on aura simplement à vérifier si le projectile collisionne avec la cible au lieu d'avoir à définir et détecter quand un récipient contient un autre objet.

3.4.3.2 Collisions entre les billes et le bord de la fenêtre

La gestion des collisions entre une bille et un bord de la fenêtre est sera extrêmement simple à gérer. Non seulement parce qu'il n'y a que deux sens de collision possible (horizontal et vertical) mais aussi parce que, étant donné que l'on ne résoudra que des collisions parfaitement élastiques, il suffira de repositionner la

[2] Entre guillemets car l'énergie cinétique n'est jamais vraiment perdue, elle est simplement convertie sous une autre forme : chaleur, déformation, son, etc…

bille au bord de la fenêtre et d'inverser la composante de sa vélocité en fonction de l'axe de collision concerné.

Prenons un exemple : Si une bille collisionne le bord de gauche, l'axe de collision est horizontal et cela veut dire qu'il faut inverser la vélocité horizontale de la bille. C'est-à-dire que si la composante x valait -3m/s, elle sera inversée et vaudra 3m/s. La composante verticale de la vélocité ne sera pas affectée car les collisions sont parfaitement élastiques et qu'il n'y a pas de frottement.

3.4.3.3 Trace des billes

L'affichage de la trace des projectiles ne devrait pas être trop compliqué à implémenter. Essentiellement, il suffit de garder un historique de toutes les positions occupées par le projectile et d'y afficher des formes différenciables des billes.

Pour donner un effet de « motion blur », on peut donner une couleur dont la transparence diminue progressivement avec l'ancienneté de la position.

3.5 Déploiement de l'application

Les exécutables modernes ne sont pas autosuffisants. Ils contiennent certes le code machine permettant leur exécution mais font également souvent référence à de nombreuses DLL (Dynamic Link Library). Une DLL est, comme un exécutable, un ensemble d'instructions destinées au processeur, à la seule différence qu'une DLL ne peut pas être exécutée directement, ce n'est pas son rôle. Les DLL contiennent du code qui a été externalisé pour pouvoir être utilisé par plusieurs programmes.

Les DLL de Windows par exemple, qui seront forcément utilisées par tous les programmes affichant des fenêtres sur l'écran, ne seront chargées qu'une seule fois dans la RAM pour que tous les autres programmes puissent y faire référence. Ce système permet d'économiser de la mémoire.

Pour prendre un autre exemple, la bibliothèque SFML a son ensemble de DLL qui doivent être présentes sur la machine (à côté de l'exécutable ou dans C:/Windows/System32) pour permettre l'exécution du programme.

C'est un système très ingénieux mais qui complique un peu le processus de déploiement. La machine du développeur contiendra forcément les DLL nécessaires à l'exécution du programme, mais comment être sûr que la machine du client a bien toutes les DLL ?

Une possibilité est de lister manuellement toutes les DLL dont le programme a besoin. Mais cette méthode, en plus d'être très rébarbative, n'est pas très sûre puisque on risque d'oublier une DLL. La solution à ce problème est d'utiliser un **installateur**. Un installateur est un programme qui se chargera de mettre en place l'environnement correctement en fonction de la machine sur laquelle il sera exécuté. Typiquement, les installateurs proposent de choisir des options telles que le répertoire d'installation ou l'architecture de la machine cible (64/32 bit). Les installateurs permettent également de définir automatiquement des variables d'environnement et ainsi de définir quel programme sera utilisé pour ouvrir un certain type de fichier ou d'ajouter l'exécutable au PATH, ce qui permettra de pouvoir le lancer directement depuis un terminal Windows.

Étant donné qu'il n'y a presque que des avantages à utiliser un installateur, c'est de cette manière que le déploiement de Physical Event Simulation sera assuré. Le seul point négatif étant qu'il faudra réserver du temps pour le développement et le test de celui-ci.

4 Planification initiale

4.1 Liste de tâches

Une fois l'analyse préliminaire du projet terminée, il a été possible de commencer la planification initiale. Elle consiste en la création d'une liste de tâches la plus exhaustive possible et en la répartition du temps du projet entre les différentes tâches.

Ci-dessous, vous trouverez la liste des tâches prévues pour le projet ainsi que le temps en heures prévu pour chacune. Certaines d'entre-elles sont déjà terminées puisque la planification initiale a été créée lors de l'analyse et que certaines de ces tâches concernent justement l'analyse.

Tâches - objectifs	Temps prévu
Absences - imprévus	7h10m
Réunion avec experts ou chef de projet	-
DOC - Journal de travail	2h
DOC - Planification initiale	7h
DOC - Analyse technologies à utiliser	30m
DOC - Analyse méthode de simulation	2h15
DOC - Analyse gestion des collisions	6h10
DOC - Analyse et choix méthodologie de projet	30m
DOC - Analyse et conception des maquettes	3h05m
DOC - Réalisation	9h15m
DOC - Analyse calcul collisions élastiques avec masse	3h55m
DOC - Analyse calcul trajectoire balistique avec vent	3h05m
DOC - Expliquer mise en place de l'environnement et prérequis	45m
DOC - Conception des tests	3h05m
DOC - Divers (rapport)	1h25m
DOC - Préparer le rendu du projet	1h
DEV - Mise en place de l'environnement	30m
DEV - Implémenter un widget "bouton"	2h10m
DEV - Implémenter le menu principal	3h5m
DEV - Implémentation du lanceur de billes	2h30m

DEV - Afficher la "trace" des billes	30m
DEV - Implémentation des collisions avec les bords	2h35m
DEV - Implémenter configuration des propriétés des billes 1-4	1h50m
DEV - Résoudre collisions élastiques avec masse entre deux billes	5h35m
DEV - Simulation expérience "chocs"	1h15m
DEV - Simulation expérience "balistique"	4h30m
DEV - Implémentation de la simulation balistique sans le vent	1h40m
DEV - Prendre en compte le vent dans la trajectoire balistique	3h05m
DEV - Détecter les collisions avec le "récipient"	3h05m
DEV - Création de l'installateur (setup)	3h05m
DEV - Divers	-
TEST - Validation des tests	3h55m

4.1.1 Commentaires

Tout d'abord, il est important de préciser que cette liste de tâches est commune au document de planification et au journal de travail. Ils font tout deux référence à cette même liste. Certaines tâches n'ont pas de temps planifié car elles n'auront lieu d'être que dans le journal de travail. Par exemple, la tâche « réunion avec experts ou chef de projet » ne peut pas être planifiée. On peut connaître l'heure et le date des visites mais il est difficile de planifier le temps qu'elles prendront. Pour la même raison, certaines tâches sont volontairement « vagues », telles que la tâche « DEV – Divers ». Ces tâches sont là pour pouvoir être référencées par le journal de travail lorsque le travail effectué n'entre dans aucune autre catégorie.

Vous aurez peut-être également remarqué que certaines tâches n'ont pas de temps attribué. Cela est dû au fait que certaines tâches étaient déjà terminées au moment de rendre la planification initiale. Je n'allais pas rétro-planifier des tâches déjà terminées (ce serait absurde) mais je les ai quand même ajoutées à ma planification pour pouvoir les référencer dans mon journal de travail.

Une absence est prévue le jeudi 6 mai 2021. C'est pour cela que la tâche « Absences – imprévus » a 7h10 de prévu.

4.1.1.1 Tâches principales

L'un des points les plus difficiles du projet sera la gestion des collisions. La création d'un moteur physique permettant la gestion des collisions ne sera pas simple. La conception des formules sera complexe car il ne s'agit pas simplement de créer un résultat visuellement agréable mais bien de créer un résultat le plus réaliste possible. Il faudra donc concevoir ces formules en partant de lois et concepts physiques élémentaires (tels que le principe de conservation d'énergie et de quantité de mouvement).

L'autre point qui prendra beaucoup de temps est la rédaction du rapport. Que ce soit pour mettre au propre l'analyse ou documenter le détail de la réalisation, la documentation du projet prendra beaucoup de temps. Ce choix est volontaire : il faut partir du principe que le projet va potentiellement être repris par un tiers un

jour. Même si la documentation prend du temps, elle permet en réalité d'en gagner beaucoup en cas de reprise du projet.

5 Conception

5.1 Maquettes de l'interface

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.

5.1.1 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.

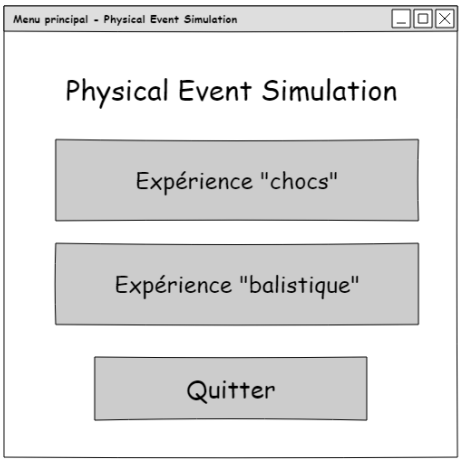


Figure 1 – Maquette du menu principal

5.1.2 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.

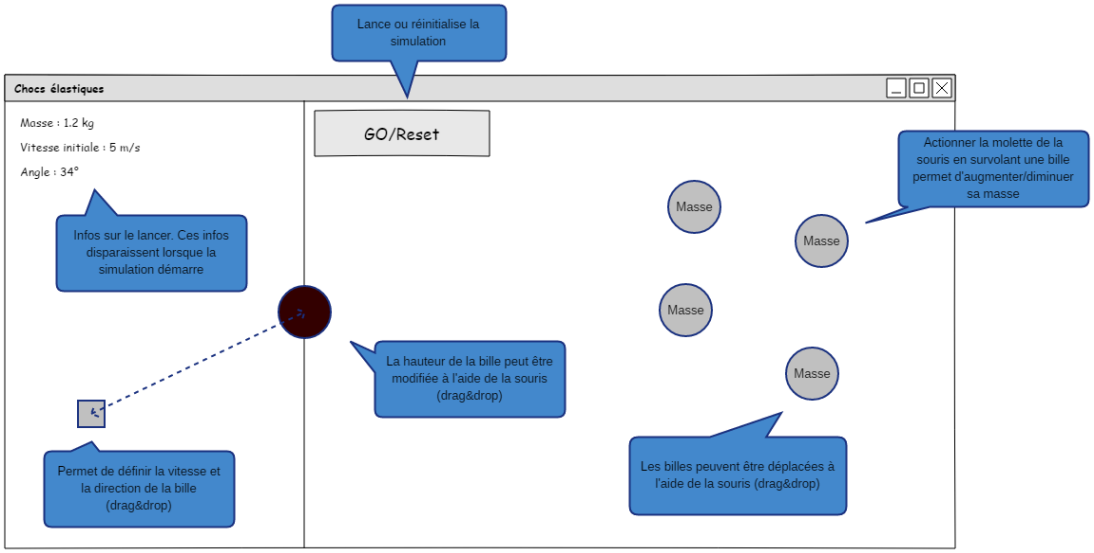


Figure 2 - Maquette de l'interface de la simulation de collisions

5.1.3 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).

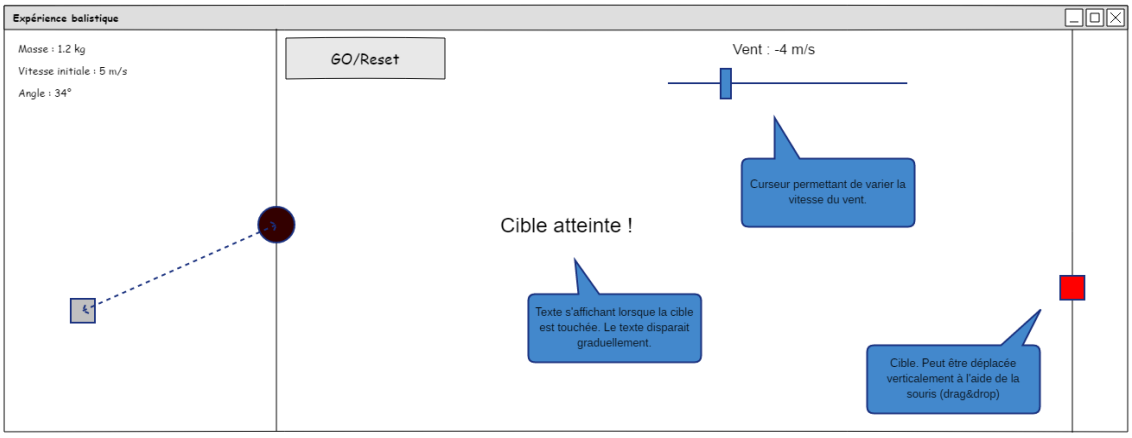


Figure 3 - Maquette de l'interface de la simulation de balistique

5.1.4 Police d'écriture utilisée

La SFML permet d'afficher du texte, à condition qu'un fichier de police (.ttf) soit spécifié dans le code. Il faut donc choisir une police pour le projet. La police choisie doit remplir deux conditions importantes :

- Elle doit permettre d'afficher des nombres de manière parfaitement lisible (on évitera une police type « calligraphie »)
- L'application est en français, il faudra donc qu'elle puisse afficher des accents au cas où
- Elle doit être libre de droits (domaine publique, si possible)

Après un peu de recherche sur le site web dafont.com qui répertorie de nombreuses polices d'écriture. La police « Bebas Neue » a été choisie (https://www.dafont.com/bebas-neue.font). Elle remplit tous les critères mentionnés ci-dessus.

5.1.5 Palette de couleur

Même si le projet consiste en une simple simulation, il est important que l'application soit agréable à utiliser et la palette de couleur utilisée pour son interface devra donc être choisie avec soin. Celle qui ont été utilisées pour les maquettes ci-dessus ne le sont qu'à titre d'exemple et il va de soi que des meilleures couleurs seront utilisées pour la réalisation de l'application.

5.2 Conception des formules de balistique

Nous allons voir les équations qui seront utilisées pour calculer la position des billes dans une simulation « discrète » (voir l'analyse pour la différence entre simulation continue et discrète) et, dans un deuxième temps, celles qui nous permettront de prendre en compte la résistance de l'air dans la simulation balistique.

Étant donné que les deux simulations sont en deux dimensions, nous travaillerons avec des vecteurs qui nous permettront de représenter la position et la vélocité des projectiles. La première équation nous permet de calculer la position d'un projectile en fonction de sa position actuelle (p_0), de sa vélocité (v) et du temps écoulé (Δ**t**) :

$$\vec{p} = \overrightarrow{p_0} + \vec{v} \cdot \Delta t$$

Cette équation suffira pour la première expérience puisqu'il s'agit de mouvement rectilignes uniformes. Cependant, la deuxième expérience introduit le concept de gravité, ce qui veut dire que la vélocité du projectile variera également en fonction du temps. De la même manière que pour la première équation, on peut calculer la nouvelle vitesse du projectile de la manière suivante :

$$\vec{v} = \overrightarrow{v_0} + \vec{a} \cdot \Delta t$$

Où **a** est un vecteur représentant l'accélération. Si a correspond à la gravité, le vecteur **a** pointera vers le bas et aura une magnitude égale à l'accélération gravitationnelle.

Ces deux équations très simples permettent de simuler une trajectoire balistique de manière discrète. Typiquement, une simulation discrète suivra le déroulement suivant :

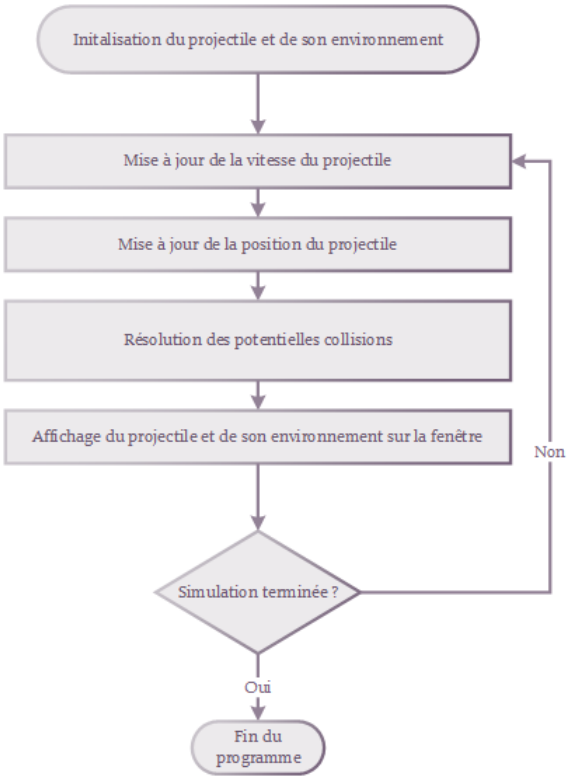


Figure 4 - Schéma de fonctionnement d'une simulation discrète

5.2.1 Frottement de l'air

Le calcul de frottement de l'air peut devenir extrêmement compliqué en fonction de la précision que l'on veut avoir. Pour éviter que le programme ne se transforme en une véritable simulation de fluide, nous allons utiliser des approximations.

La formule permettant de calculer le frottement de l'air F_x est la suivante[3] :

$$F_x = \frac{1}{2}\rho S C_x V_{rel}^2$$

Cette force dépend donc de 4 paramètres :

- Rhô (ρ) : la masse volumique du fluide ambiant (dans notre cas, de l'air)
- S : La surface de référence, c'est-à-dire, l'aire du projectile quand on le regarde de face (dans notre cas, on considérera le projectile comme une sphère et S sera donc équivalent à l'aire d'un cercle de même rayon).
- V_{rel} : La vitesse relative entre le projectile et le fluide.
- Cx : Le coefficient de trainée du projectile. Son calcul exact peut être assez compliqué et c'est là que nous utiliserons une approximation.

[3] https://fr.wikipedia.org/wiki/Tra%C3%AEn%C3%A9e

Nous allons maintenant adapter cette formule pour que l'on puisse déterminer l'intensité et la direction de l'accélération que subit le projectile.

Concernant la surface de référence, nous prendrons l'aire d'un cercle de même rayon que le projectile :

$$S = \pi r^2$$

La vitesse relative, elle, peut être déterminée en calculant le vecteur de vélocité relative entre le fluide et le projectile et en calculant sa longueur :

$$V_{rel} = ||\overrightarrow{V_{air}} - \overrightarrow{V_{proj}}||$$

Pour obtenir la direction de la force de frottement, on prend le vecteur unitaire de la vitesse relative entre le fluide et le projectile :

$$\vec{d} = \frac{\overrightarrow{V_{air}} - \overrightarrow{V_{proj}}}{V_{rel}}$$

La deuxième loi de Newton nous dit que l'accélération d'un corps est proportionnelle à la résultante des forces qu'il subit et inversement proportionnelle à sa masse. Concrètement, cela peut se traduire par l'équation suivante (si on a qu'une seule force) :

$$\vec{a} = \frac{\vec{F}}{m} = \frac{F \cdot \vec{d}}{m}$$

On peut combiner nos différentes équations pour exprimer directement l'accélération que subira le projectile en fonction de tous les paramètres :

- Sa vélocité $\mathbf{V_{proj}}$ (elle apparaît sous deux formes : comme un **vecteur** (avec la flèche) et comme un scalaire)
- Sa masse **m**
- La masse volumique de l'air ambiant **ρ**
- Sa vitesse relative par rapport à l'air ambiant : $\mathbf{V_{relative}}$
- Son rayon **r**
- Son coefficient de trainée $\mathbf{C_x}$

$$\overrightarrow{a_{proj}} = -\frac{\rho \pi r^2 C_x \cdot V_{rel}^2}{2m} \cdot \frac{\overrightarrow{V_{air}} - \overrightarrow{V_{proj}}}{V_{rel}}$$

Si on simplifie, cela donne l'équation suivante :

$$\overrightarrow{a_{proj}} = -\frac{\rho \pi r^2 C_x \cdot V_{rel}}{2m} \cdot (\overrightarrow{V_{air}} - \overrightarrow{V_{proj}})$$

5.2.1.1 Notes concernant la masse volumique de l'air et du coefficient de trainée

Des approximations du coefficient de trainée existent pour les formes simples fréquentes telles que les carrés, les cylindres ou les sphères. Pour les sphères, la valeur de **0.47** peut être utilisée tant que la vitesse

n'est pas trop élevée[4]. Au-delà d'un certain seuil, des turbulences apparaissent et celles-ci compliquent le calcul de la force de frottement puisqu'elles nécessitent le calcul de la valeur exacte du « nombre de Reynolds ». Pour éviter de transformer le programme en une véritable simulation de fluide, nous allons nous contenter de l'approximation.

La masse volumique de l'air est égale à **1.225 kg/m³** au niveau de la mer[5] et à 15°C. Étant donné que l'altitude du projectile ne variera pas de manière suffisamment grande pour changer significativement, on la considérera comme constante dans le programme.

5.3 Conception des formules pour la 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.

5.3.1 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.

5.3.1.1 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. En cas de collision, l'inéquation suivante se vérifie :

$$(c_{1x} - c_{2x})^2 + \left(c_{1y} - c_{2y}\right)^2 < (r_1 + r_2)^2$$

[4] https://fr.wikipedia.org/wiki/Coefficient_de_tra%C3%AEn%C3%A9e
[5] https://fr.wikipedia.org/wiki/Masse_volumique_de_l%27air

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 :

$$\vec{n} = \frac{\overrightarrow{c_b} - \overrightarrow{c_a}}{\left|\left|\overrightarrow{c_b} - \overrightarrow{c_a}\right|\right|}$$

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

La profondeur de la collision d peut être calculée en soustrayant la distance entre les centres des deux cercles par la somme de leurs rayons :

$$d = \left|\left|\overrightarrow{c_b} - \overrightarrow{c_a}\right|\right| - (r_a + r_b)$$

5.3.1.2 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 C_a, il faudra soustraire le vecteur de déplacement à sa position alors que pour C_b, il faudra l'additionner. Le calcul de la nouvelle position après un déplacement se fait de la manière suivante :

$$\overrightarrow{P'} = \vec{P} + \overrightarrow{\Delta P}$$

Pour calculer le déplacement (delta P), on amplifie la normale de collision par la moitié de la profondeur de la collision (d) :

$$\overrightarrow{\Delta P} = \vec{n} \cdot \frac{1}{2} d$$

On prend la moitié car ce vecteur sera appliqué pour les deux cercles, ce qui donnera un déplacement total égal à la profondeur. On peut maintenant calculer les nouvelles positions des cercles A et B :

$$\overrightarrow{c_a'} = \overrightarrow{c_a} - \overrightarrow{\Delta P}$$

$$\overrightarrow{c_b'} = \overrightarrow{c_b} - \overrightarrow{\Delta P}$$

5.3.1.3 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 quantité de mouvement instantané. La quantité de mouvement d'un corps correspond à sa vélocité (un vecteur) amplifié par sa masse :

$$\vec{p} = \vec{v} \cdot m$$

Dans la suite de cette réflexion, nous noterons l'impulsion avec la lettre J.

À partir de cette formule, on peut montrer que la vélocité finale d'un corps après l'application d'une impulsion J peut être calculée de la manière suivante (où v et v' sont respectivement les vélocités de départ et de fin et m correspond à la masse) :

$$\overrightarrow{V'} = \vec{V} + \frac{\vec{J}}{m}$$

Selon la troisième loi de Newton, si un objet exerce une force sur un deuxième objet, ce dernier exercera une force de **direction opposée** mais de **grandeur égale** sur le premier objet. Concrètement, cela veut dire que les deux billes recevront la même impulsion mais dans des sens différents.

La direction dans laquelle l'impulsion sera appliquée est connue, c'est la normale de collision. Comme pour le vecteur de déplacement, on l'inversera ou non en fonction de la bille concernée. Étant donné que l'on connait la direction du vecteur, il ne nous reste qu'à calculer le scalaire qui servira à amplifier ce vecteur. Nous noterons ce scalaire j.

Pour déterminer j, nous allons appliquer le principe de conservation d'énergie qui nous dit que l'énergie initiale totale du système est égale à l'énergie finale totale du système. En d'autres termes, il n'y a aucune perte d'énergie. Étant donné qu'il n'y a que de l'énergie cinétique dans notre système, on peut poser l'équation suivante :

$$\frac{1}{2}m_a v_{a1}^2 + \frac{1}{2}m_b v_{b1}^2 = \frac{1}{2}m_a v_{a2}^2 + \frac{1}{2}m_b v_{b2}^2$$

Elle montre simplement que l'énergie cinétique totale du système est conservée. Ici, v_{a1}, v_{a2}, v_{b1} et v_{b2} correspondent respectivement aux vitesses de départ et de fin des deux billes. On peut remplacer les vitesses de fin (v_{a2} et v_{b2}) en les exprimant en fonction des vitesses de départ :

$$v_{a2} = v_{a1} - \frac{j}{m_a}$$

$$v_{b2} = v_{b1} + \frac{j}{m_b}$$

On notera la différence de signe. Pour le premier objet, on soustrait l'impulsion à sa vitesse, alors que pour le deuxième, on l'additionne. Il est nécessaire d'avoir une soustraction et une addition puisqu'il y a un transfert d'énergie entre les deux objets.

Ici, j est en minuscule car ce n'est pas l'impulsion à proprement parler, mais simplement son intensité. Il est plus simple de d'abord mettre en place la formule et la simplifier en ne considérant que le scalaire j. Ce scalaire sera utilisé pour calculer l'impulsion J (majuscule) qui, elle, est belle est bien un vecteur.

Si on introduit les expressions des vitesses finales dans notre équation de conservation d'énergie, on obtient l'équation suivante :

$$\frac{1}{2}m_a v_{a1}^2 + \frac{1}{2}m_b v_{b1}^2 = \frac{1}{2}m_a\left(v_{a1} + \frac{j}{m_a}\right)^2 + \frac{1}{2}m_b\left(v_{b1} + \frac{j}{m_b}\right)^2$$

À première vue, cette équation peut paraître compliquée mais, si on la simplifie au maximum et qu'on isole notre inconnue j, on peut la présenter sous cette forme :

$$j = \frac{2(v_{a1} - v_{b1})}{\frac{1}{m_a} + \frac{1}{m_b}}$$

C'est presque fini, mais il reste quelque chose de très important à clarifier. Vous aurez sûrement remarqué que nous avons utilisé les vitesses (des scalaires) et non les vélocités (des vecteurs) des billes dans l'équation. En fait, cette équation permet de résoudre des collisions en 1 dimension, c'est-à-dire, avec des billes qui vont **exactement l'une contre l'autre**. Cependant, ce n'est pas toujours le cas en 2 dimensions et il va falloir tenir compte de cela.

La partie se trouvant en dessous de la barre de fraction ne va pas changer mais la partie du dessus, qui n'est rien d'autre que le double de la **vitesse relative** (vitesse avec laquelle les billes entrent en collision l'une contre l'autre), doit être adaptée pour une utilisation dans un plan à deux dimensions. Le calcul de la vitesse relative sera identique, à la différence près qu'on utilisera des vecteurs :

$$\overrightarrow{v_{relative}} = \overrightarrow{v_{a1}} - \overrightarrow{v_{b1}}$$

Cependant, la vitesse relative n'est pas suffisante. Ce qu'il nous faut est la vitesse relative **le long de la normale de collision**. En d'autres termes, à quelle vitesse les billes vont l'une vers l'autre. C'est cette valeur qui devra être utilisée pour calculer l'intensité de l'impulsion. Pour la calculer, c'est très simple, il suffit de projeter la vélocité relative sur la normale de collision, ce qui revient à effectuer le produit scalaire entre ces deux vecteurs :

$$v_{collision} = \vec{n} \cdot (\overrightarrow{v_{a1}} - \overrightarrow{v_{b1}})$$

Si l'on replace la vitesse relative de l'équation initiale par cette nouvelle expression, on obtient l'équation finale permettant d'exprimer l'intensité de l'impulsion en fonction :

$$j = \frac{2\vec{n} \cdot (\overrightarrow{v_{a1}} - \overrightarrow{v_{b1}})}{\frac{1}{m_a} + \frac{1}{m_b}}$$

Le vecteur de l'impulsion n'est rien d'autre que la normale de collision amplifiée par le scalaire j. L'impulsion sera la même pour les 2 billes mais l'accélération qui en résultera sera différente puisque les masses des billes peuvent être différentes.

On a maintenant tout ce qu'il faut pour exprimer les vélocités finales des deux billes en fonction de leur vélocité initiale, de leurs masses respectives, de la normale de collision et de l'intensité de l'impulsion :

$$\overrightarrow{v_{a2}} = \overrightarrow{v_{a1}} - \frac{j \cdot \vec{n}}{m_a}$$

$$\overrightarrow{v_{b2}} = \overrightarrow{v_{b1}} + \frac{j \cdot \vec{n}}{m_b}$$

Après le repositionnement des deux billes et l'application de l'impulsion, la collision peut être considérée comme résolue.

5.4 Conception des tests

Pour garantir le fait que toutes les fonctionnalités ont été implémentées, il sera nécessaire de tester chacune d'entre elles une par une. Un certain nombre de tests de validation devront donc être effectués.

Étant donné la courte durée du projet, il a été décidé de ne pas mettre en place de tests unitaires ou de tests d'intégration. En effet, ce genre de tests sont très utiles lorsqu'une application évolue continuellement car cela permet de vérifier que les fonctionnalités existantes fonctionnent toujours malgré l'ajout de nouvelles fonctionnalités mais, vu que les fonctionnalités à implémenter dans Physical Event Simulation sont toutes déjà connues, la création de tests unitaires n'est pas forcément avantageuse.

En clair, le temps que prendrait la création et la maintenance d'un projet de tests unitaires surpasse les bénéfices potentiels que l'on pourrait en tirer et c'est pour cette raison que seuls des tests de validation (manuels) seront effectués.

5.4.1 Modèle à suivre pour les tests de validation

Tous les tests de validation devront suivre le modèle suivant :

Fonctionnalité testée	Déroulement du test	Résultat attendu	Résultat obtenu	Validation
Ouverture du programme	Lancer le programme	Le menu doit s'ouvrir	Le menu s'ouvre	OK

Ce modèle permet de documenter avec beaucoup de détails le déroulement d'un test et ainsi de comprendre, en cas d'échec, exactement où le problème est survenu.

5.4.2 Liste des tests à effectuer

Le détail du déroulement de chaque test sera défini une fois le développement de l'application terminé car il est très possible que les différentes fonctionnalités soient implémentées d'une manière différente que celle prévue durant l'analyse. Voir section Tests

6 Réalisation

Cette section contient toutes les informations nécessaires pour pouvoir permettre à un potentiel tiers de reprendre le projet.

6.1 Emplacement du code source

Le code source, ainsi que tous les fichiers du projets (rapport, journal de travail, maquettes, etc…) sont disponibles sur le dépôt GitHub suivant : https://github.com/Raynobrak/etml-tpi. C'est un dépôt public, dont accessible par n'importe qui en possédant l'URL.

L'intérêt d'utiliser un dépôt est que l'on peut suivre l'évolution du projet et voir qui a apporté des modifications quel moment. L'autre avantage est que cela permet d'assurer un backup du projet. Si jamais le dépôt local est perdu, on peut toujours cloner le dépôt distant qui contiendra la dernière version du projet qu'on a push. Et, si jamais les serveurs GitHub venaient à être détruits (peu probable mais pas impossible), on disposera toujours de la copie locale.

L'utilisation d'un dépôt permet donc de faciliter le suivi du projet et de garantir un « backup », à condition bien sûr de push régulièrement ses modifications.

6.2 Versions des bibliothèques utilisées

Les dernières versions de la SFML (2.5.1) et de ma bibliothèque (Charbrary 2.0.0) ont été utilisées pour le développement de l'application.

6.3 Mise en place de l'environnement et compilation du projet

Cette section contient toutes les informations nécessaires à toute personne souhaitant compiler le code source sur sa machine.

6.3.1 Préparation de l'environnement

Les fichiers du projet sont regroupés dans une solution Visual Studio 2019 que l'on peut trouver dans le dossier source/physical-event-simulation. Pour pouvoir ouvrir cette solution, il est nécessaire que la charge de travail « Développement Desktop en C++ » soit installé sur Visual Studio. Si ce n'est pas le cas, elle peut être installée depuis Visual Studio installer :

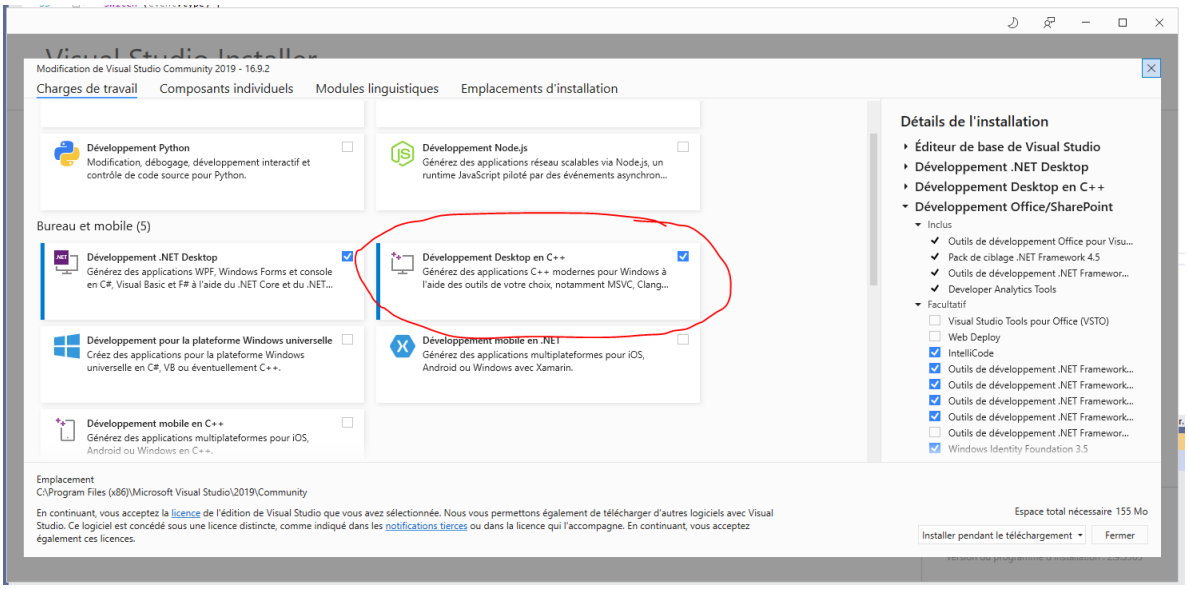


Figure 5 - Installation des modules C++ sur Visual Studio Installer

Pour pouvoir exécuter le projet, il faut encore faire 2 petites choses : Télécharger la SFML et ajouter les fichiers .lib ainsi que les DLL au dossier du projet.

Le repository Git a été mis en place de manière à éviter de Git des fichiers binaires (comme les .lib et .dll). Ceux-ci ont été omis du dépôt pour éviter de le surcharger inutilement mais ils peuvent être obtenus depuis le site officiel de la SFML en téléchargeant la dernière version :

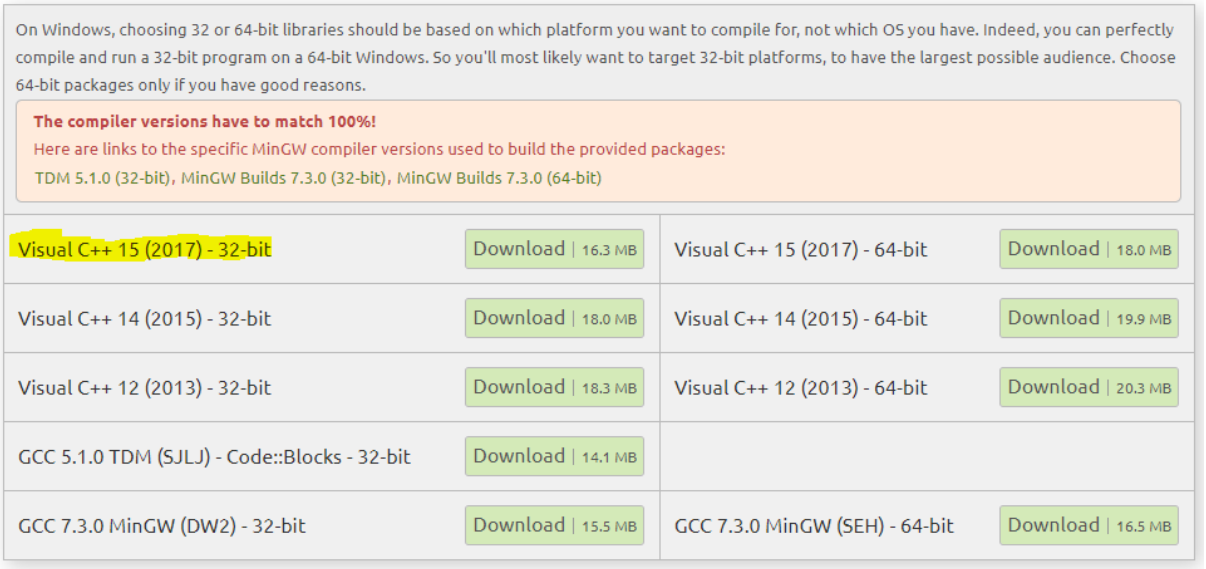


Figure 6 - Téléchargement de la bibliothèque SFML

La version de la SFML qui a été utilisée pour le projet est la version 2.5.1 précompilée pour Visual C++ 2017 en 32 bits.

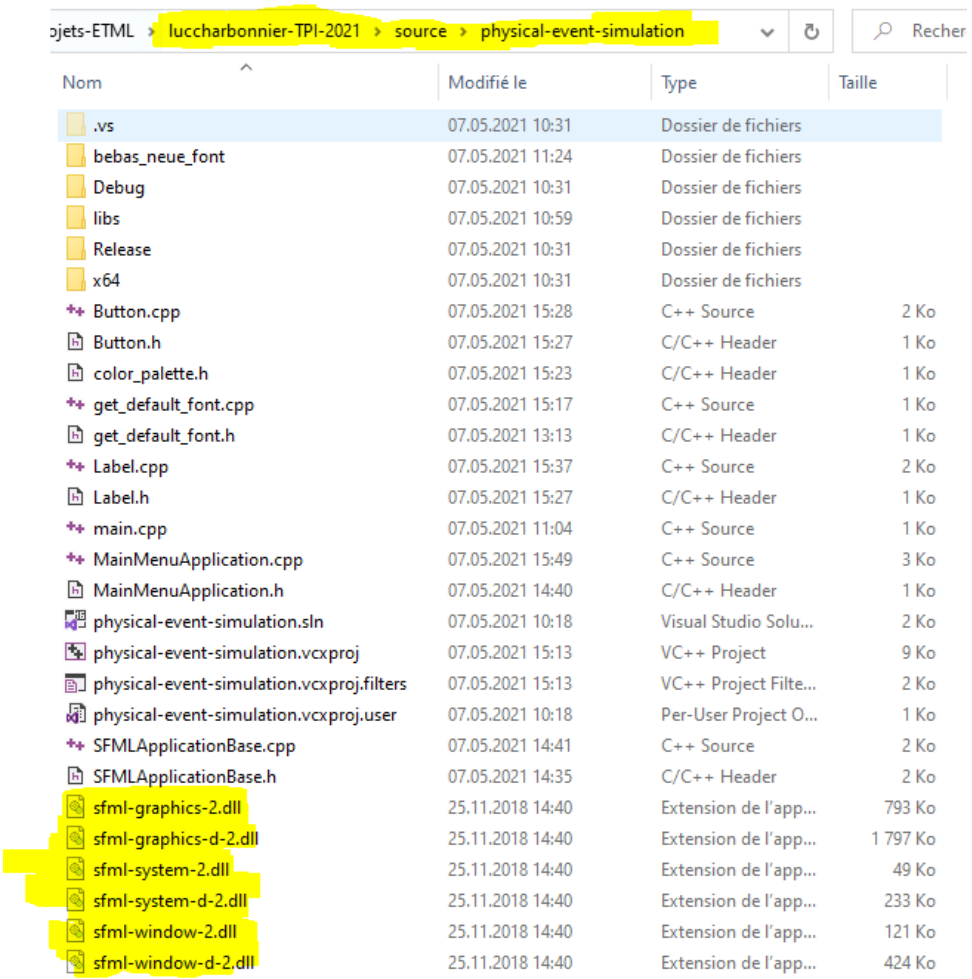


Figure 7 - Copie des DLL de la SFML dans le répertoire du projet

Dans l'archive téléchargée, il y aura un dossier bin. C'est ce dossier qui contient les DLL. Pour pouvoir exécuter le programme depuis Visual Studio, ces DLL doivent être mises à la racine du projet (dans le même dossier que la solution) ou à côté de l'exécutable produit par Visual Studio, les deux fonctionnent.

Cependant, il n'est pas nécessaire d'ajouter toutes les DLL contenues dans le dossier bin/ car nous n'utilisons que certains modules de la SFML (les modules **graphics**, **window** et **system**). Les DLL à ajouter sont celles visibles (surlignées en jaune) dans la capture d'écran ci-dessus. Il y en a 6 car il en faut deux pour chaque module : une pour la compilation normale (en release) et une pour la compilation permettant de débugger le programme (debug).

Dans l'archive téléchargée depuis le site de la SFML, il y aura également un dossier /lib. Ce dossier contient les fichiers .lib de la SFML. Vous pouvez copier-coller ce dossier dans le dossier libs/SFML-2.5.1-32/ (à côté du répertoire /include) :

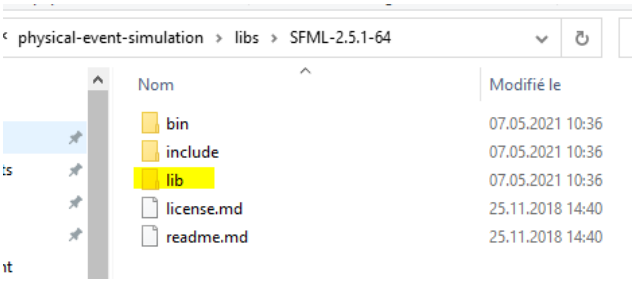


Figure 8 - Copier du dossier lib

On ne décrira pas ici[6] la manière dont la SFML est liée au projet Visual Studio. Toutefois, si ce processus vous intéresse, vous pouvez trouver la procédure complète sur le site de la SFML[7], mais sachez que ne devriez pas en avoir besoin pour compiler le projet puisque la SFML est déjà présente dans le dossier du projet.

6.3.2 Compilation

Étant donné que le projet est configuré pour la SFML en 32 bits, il est important que lors de la compilation, on sélectionne la configuration « x86 ». Si l'on souhaite déboguer le programme, il faut sélectionner la configuration debug et lancer le debugger de Visual Studio. Mais, sauf exception, il vaut mieux compiler en Release pour éviter que le programme soit ralenti inutilement.

Figure 9 - Configuration de build du projet

6.3.3 Troubleshooting

Cette section détaille les erreurs les plus fréquentes qui peuvent survenir lors de la compilation ou de l'exécution d'un projet lié à la SFML.

Le pop-up suivant apparaîtra lorsqu'on essaie d'exécuter le programme et qu'il y a une incohérence entre l'architecture (32/64 bits) cible des fichiers .lib et des .dll :

Figure 10 - Erreur de DLL

[6] Par soucis de temps et, surtout, pour éviter de réexpliquer inutilement un processus déjà détaillé par de nombreuses ressources sur Internet.

[7] https://www.sfml-dev.org/tutorials/2.5/start-vc-fr.php

Généralement, cela survient lorsque l'architecture des DLL de la SFML ne correspond pas à l'architecture pour laquelle le projet a été compilé. Pour résoudre ce problème, il faut s'assurer que les .dll et les .lib proviennent de la même version de la SFML.

L'erreur suivante est une erreur d'édition de liens (première étape de la compilation) :

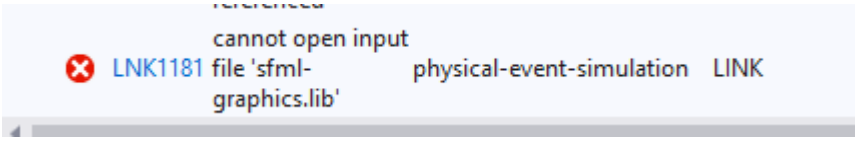


Figure 11 - Erreur d'édition de liens

Elle survient lorsque le compilateur n'arrive pas à trouver les fichiers .lib de la SFML. En principe, c'est parce que le dossier /libs n'a pas été ajouté au dossier de la SFML (comme décrit plus haut).

6.4 Boucle principale

Pour comprendre le fonctionnement du programme dans son ensemble, il faut comprendre le design pattern sur lequel il est basé. Ce « design pattern », souvent appelé « Game Loop », consiste à englober le fonctionnement du programme dans une boucle principale.

Cette manière de faire est presque indispensable lorsqu'on développe une application graphique qui a besoin de recevoir des informations provenant des périphériques de l'ordinateur (clavier, souris) car l'on ne sait pas quand ceux-ci seront activés et que bloquer le déroulement du programme en attendant un évènement (comme on le ferait lors de l'attente d'une entrée clavier dans un programme text-based en console) n'est pas une option.

Ce n'est cependant pas forcément la manière la plus intuitive de procéder. Les frameworks connus encapsulent généralement ce fonctionnement en introduisant plusieurs threads pour permettre au développeur de programmer de manière événementiel (on assigne une fonction à appeler lorsqu'un événement se produit).

Le paradigme événementiel est très pratique lorsque on développe une application avec une interface graphique comportant beaucoup de widgets (des boutons, des checkboxes, radiobuttons, etc…) car on peut assigner une fonction à chaque bouton et gérer très facilement les différents comportements à implémenter.

Cela dit, ce n'est pas du tout un paradigme approprié pour la création de simulation. À l'inverse, le paradigme « séquentiel » (avec un seul thread) est parfaitement adapté pour le développement de simulation discrètes utilisant un pas de temps.

Concrètement, voici comment le « Game Loop » pattern est mis en place pour une simulation : On commence par déclarer un chronomètre qui va mesurer le temps qui s'écoule entre chaque tour de boucle. Une fois que ce temps dépasse la valeur du pas de temps (un cinquantième de seconde, p.ex.), on met à jour la simulation. Cela veut dire que s'il y a un objet qui se déplace de 2 m/s vers la droite, il sera déplacé de 2 * (1/50) mètres (= 0.04 mètres) vers la droite lors d'une mise à jour de la simulation.

À chaque tour de boucle, en plus de mettre à jour la partie physique de la simulation, on va également prendre en compte les événements (clic de souris, touche de clavier, scroll molette, etc…) qui sont survenus depuis le dernier tour de boucle. L'affichage, lui, n'est pas forcément calqué sur le même pas de temps que la simulation, étant donné qu'il serait inutile de mettre à jour la simulation plus de 60 fois par secondes.

Concrètement, voici à quoi ressemble la mise en place de cette boucle principale dans l'application :

```
void SFMLApplicationBase::run() {
    // Chronomètres pour la mise à jour et le rendu de l'application.
    sf::Clock updateClock;
    sf::Clock renderClock;

    sf::Event event;

    float timeBetweenFrames = 1.f / fps_;

    // Boucle principale : tant que l'application ne doit pas fermer...
    while (!exitApplication_) {
        // Tant que la fenêtre est ouverte...
        while (window_.isOpen()) {
            // Si le temps écoulé depuis la dernière mise à jour dépasse le timestep...
            if (updateClock.getElapsedTime().asSeconds() >= fixedTimeStep_) {
                // Gestion des événements
                while (window_.pollEvent(event)) {
                    handleEvent(event);
                }

                updateClock.restart();

                // Mise à jour de l'application
                update(fixedTimeStep_);
            }

            // Si le temps écoulé depuis le dernier affichage dépasse le temps entre chaque frame...
            if (renderClock.getElapsedTime().asSeconds() >= timeBetweenFrames) {
                renderClock.restart();

                // Affichage
                render();
            }
        }
    }
}
```

Figure 12 - Boucle principale de l'application

6.4.1 Choix du pas de temps et du nombre de FPS

Le choix de la valeur du pas de temps est important. S'il est trop petit, la simulation paraîtra saccadée et risque d'être instable. À l'inverse, si le pas de temps est trop grand, le temps nécessaire au calcul d'un frame risque de dépasser la valeur du pas de temps, ce qui ralentira tout le programme.

Pour éviter que l'affichage soit saccadé, il faut évidemment que le temps entre chaque affichage soit inférieur ou égal à 1/30 seconde et que le pas de temps soit supérieur ou égal à cette valeur.

L'application tournera à **60 FPS**. C'est une fréquence d'image que la plupart des écrans modernes sont capables de suivre et qui donne un résultat plus agréable que 30 FPS.

Il faut maintenant choisir la valeur du pas de temps. Pour éviter le problème du tunneling (une instabilité qui survient lorsqu'une collision n'est pas détectée à cause de la très haute vitesse des objets concernés), il y a deux choses que l'on peut mettre en place. La première est de limiter la vitesse maximale des objets

de la simulation. C'est déjà plus ou moins le cas dans l'application car l'utilisateur est limité par la taille de la fenêtre pour l'application d'une vitesse à un projectile. La deuxième chose à faire est de diminuer la valeur du pas de temps. En effet, plus un pas de temps sera petit, plus la simulation sera précise.

Il a donc été décidé que le pas de temps sera égal à un **centième de seconde** (soit **100 mises à jour par seconde**). Cela veut dire que la simulation sera mise à jour plus souvent qu'elle ne sera affichée mais ce n'est pas un problème.

6.5 Documentation du code

Cette section va détailler les différents éléments du code. C'est-à-dire, les classes, les espaces de noms et certains fichiers. Le but ici est de présenter les différents éléments dans un ordre logique pour permettre une compréhension plus aisée, chose qui peut être compliquée si l'on ne dispose que du code et de ses commentaires.

6.5.1 Éléments externes

Comme expliqué précédemment, le programme utilise deux bibliothèques : la SFML et ma bibliothèque de calcul vectoriel. Ces deux bibliothèques sont utilisées intensivement dans tout le programme et, si le projet venait à être repris par une autre personne, il serait nécessaire d'avoir une compréhension basique des différentes classes ou structures que ces bibliothèques contiennent. Le tableau suivant contient une petite liste des classes les plus utilisées de ces deux bibliothèques :

Symbole[8]	Description
sf::Event[9]	Structure SFML représentant un événement détecté par une fenêtre. Par exemple, l'événement sf::MouseButtonPressed
Sf ::Drawable	Documentation
sf::Vector2f	Représente un vecteur à deux dimensions
ch::vec_t	Alias de sf ::Vector2f
ch::AABB	Représente une Axis-Aligned-Bounding-Box, un rectangle dont les côtés sont alignés sur les axes du référentiel. Plus clairement, c'est un rectangle que l'on ne peut pas faire pivoter. Les AABB sont positionnées à partir de leur coin haut-gauche.
ch::Circle	Représente un cercle (une position et un rayon). Les cercles sont positionnées à partir de leur centre.
ch::collision::	Espace de nom contenant des fonctions permettant la détection de collisions.

6.5.2 Affichage des fenêtres

Cette section énumère tous les éléments traitant de l'affichage des fenêtres.

6.5.2.1 Classe SFMLApplicationBase

Cette classe de base abstraite a pour but d'encapsuler le fonctionnement d'une fenêtre SFML. Cette classe encapsule également la mesure du pas de temps et du temps.

[8] L'espace de nom **sf::** appartient à la SFML alors que **ch::** appartient à la Charbrary
[9] Documentation de sf::Event : https://www.sfml-dev.org/documentation/2.5.1/classsf_1_1Event.php#details

Les classes dérivées n'auront que 3 méthodes à surcharger :

- La méthode **handleEvent(event)** qui devra contenir le code permettant de traiter l'événement SFML détecté.
- La méthode **update(dt)**, appelée automatiquement lorsque le temps écoulé depuis le dernier appel est supérieur au pas de temps, devra contenir le code permettant de mettre à jour l'application. Le paramètre **dt** correspond au pas de temps et peut, par exemple, être utilisé pour mettre à jour la position d'un projectile en fonction de sa vitesse.
- La méthode **customRender()**, appelée à la fréquence des FPS, devra contenir le code spécifique à l'affichage des éléments de l'application.

Cela dit, bien que cette classe s'appelle **SFMLApplicationBase**, rien n'empêche d'avoir plusieurs classes héritant de **SFMLApplicationBase** au sein d'un même programme. Ici, le terme « Application » doit être compris comme « Fenêtre ».

6.5.2.2 Classe MainMenuApplication

Hérite de **SFMLApplicationBase** et représente la fenêtre du menu principal de l'application.

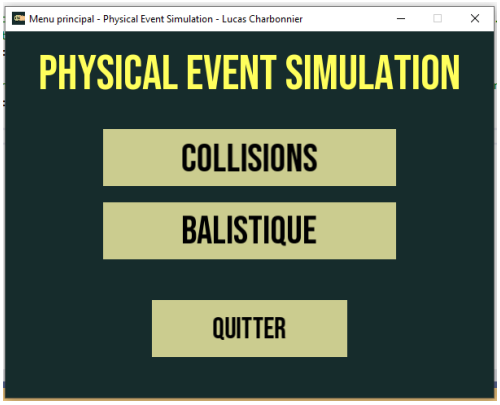


Figure 13 - Écran titre géré par la classe MainMenuApplication

6.5.2.3 Classe CollisionSimulationApp

Hérite de **SFMLApplicationBase** et représente la fenêtre de la simulation de collisions. Contient la logique de la simulation de collisions.

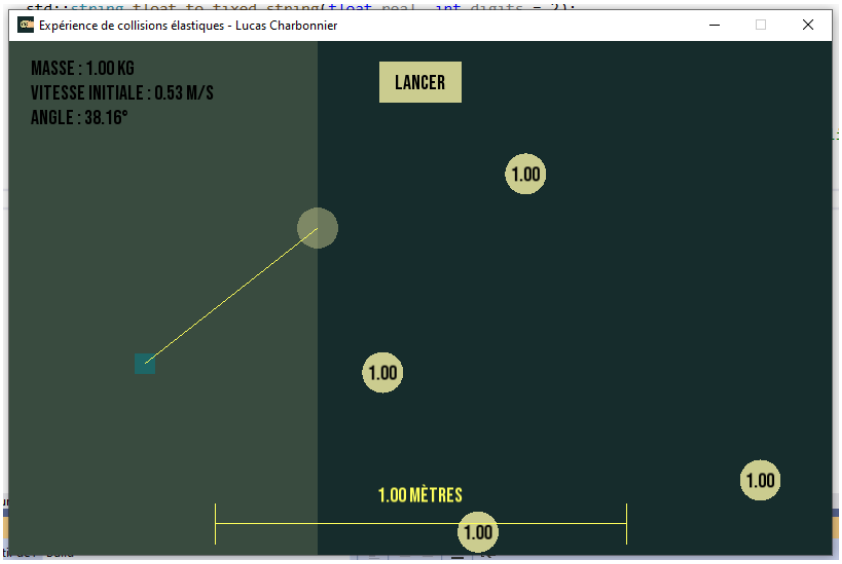


Figure 14 - Fenêtre de la simulation de collisions

6.5.2.4 Classe BallisticSimulationApp

Hérite de **SFMLApplicationBase** et représente la fenêtre de la simulation de balistique. Contient la logique de la simulation de balistique.

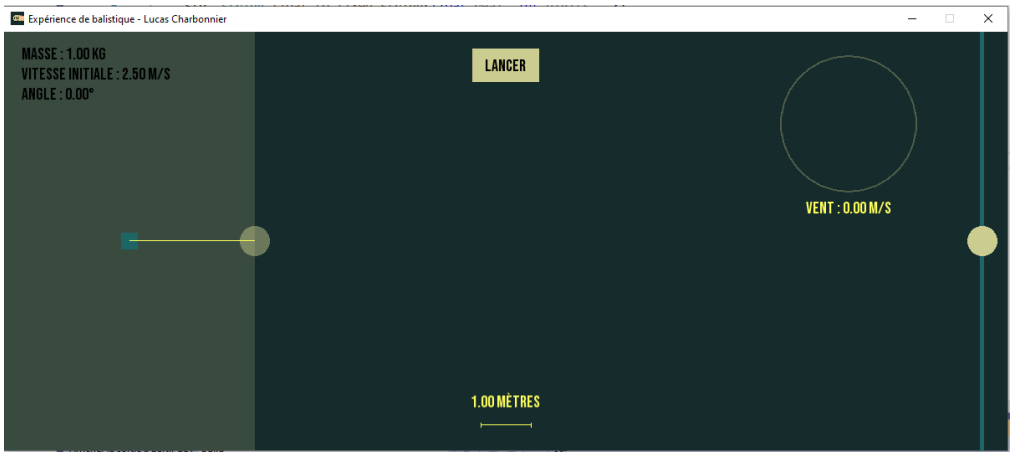


Figure 15 - Fenêtre de la simulation de balistique

6.5.3 Physique

Cette section détaille les classes ayant un rapport avec la physique.

6.5.3.1 Classe CircleRigidBody

La classe **CircleRigidBody** représente un corps rigide circulaire. En physique, un corps est dit « rigide » s'il ne peut pas se déformer ou se casser, donc que sa forme ne change pas.

C'est une classe utilisée pour représenter les projectiles dans les deux simulations. Elle contient des méthodes permettant de résoudre une collision entre deux corps rigides ainsi que pour résoudre les collisions avec les bords d'une fenêtre.

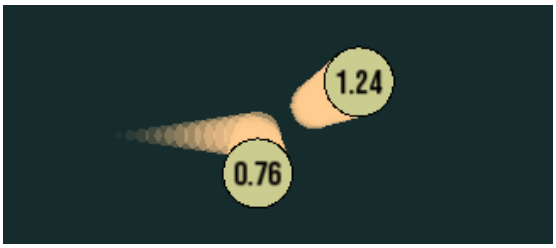


Figure 16 - Deux corps rigides après une collision

6.5.4 Widgets et éléments de l'interface

Les widgets sont des éléments visuels dont le but est de permettre à l'utilisateur d'interagir avec le programme ou simplement de lui donner des informations. Étant donné que la SFML est une bibliothèque relativement bas-niveau, elle ne propose aucun widgets (à part la classe **sf::Text**[10]) et ceux-ci doivent donc être implémentés à la main.

6.5.4.1 Classe Label

La classe Label encapsule le fonctionnement de la classe **sf::Text** de la SFML pour simplifier l'affichage de texte à l'écran. En particulier, elle gère de manière invisible l'obtention de la police d'écriture (**sf::Font**) qui doit être chargée depuis un fichier.

Cette classe permet aussi de positionner le texte de manière plus intuitive que la classe **sf::Text** de base ainsi que de le centrer au sein d'une zone (**AABB**).

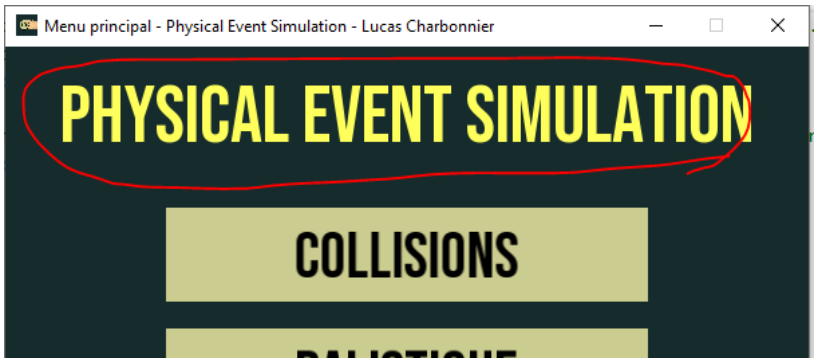


Figure 17 – Le titre du menu principal est un Label

6.5.4.2 Classe Button

La classe **Button** encapsule le fonctionnement d'un bouton et permet de savoir si il a été déclenché. Par défaut, les boutons sont déclenchés lorsque le clic gauche de la souris est **relâché** après avoir cliqué dans la zone du bouton.

[10] La classe **sf ::Text** permet d'afficher du texte à l'écran, à partir d'un fichier de police que l'on spécifie.

Figure 19 - Bouton

Figure 19 - Bouton pressé

Pour détecter le déclenchement du bouton, il faut, à chaque fois qu'un événement se produit, appeler la méthode **checkForMouseRelease(mousePos, event)** avec l'événement et la position actuelle de la souris. Cette méthode retournera un booléen si le bouton est déclenché. En plus de la détection du déclenchement, cette méthode mettra à jour l'état du bouton (pressé ou non ?).

Cette classe utilise un **Label** en interne pour afficher le texte du bouton.

6.5.4.3 Classe DraggableCircle

La classe DraggableCircle représente un cercle pouvant être glissé/déposé par l'utilisateur à l'aide de la souris. Elle peut être instanciée directement où être utilisée comme classe de base (comme c'est le cas pour **ConfigurableCircle**).

Pour mettre à jour le drag&drop, il faut appeler régulièrement la méthode **updateDragAndDrop()**.

6.5.4.4 Classe ConfigurableCircle

La classe **ConfigurableCircle** hérite de **DraggableCircle** et représente un futur projectile de la simulation. Les **ConfigurableCircle** peuvent être glissé/déposés n'importe où dans une zone donné et leur masse peut être changée.

Cette classe utilise un **Label** pour afficher la masse.

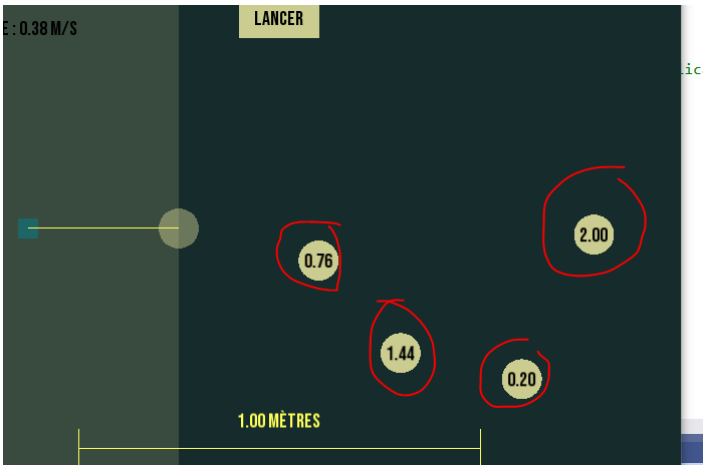


Figure 20 - Les ConfigurableCircle de la simulation de collisions

6.5.4.5 Classe ProjectileLauncher

Le lanceur de projectiles est un widgets complexe permettant à l'utilisateur de configurer les paramètres du lancer initial de la simulation. Il comporte 3 éléments importants :

- Un projectile pouvant être déplacé sur un axe vertical
- Un « bras de lancement » permettant de spécifier la vitesse initiale et la direction du projectile.
- Une zone de texte servant à afficher les informations du lancer.

Le projectile est représenté par un **DraggableCircle**, le bras de lancement par la classe **LauncherSling** et la zone de texte par un Label.

Comme pour les autres widgets, il faut régulièrement appeler la méthode **update(event, mousePos)** pour mettre à jour l'interface en fonction des événements déclenchés par l'utilisateur et de la position de la souris.

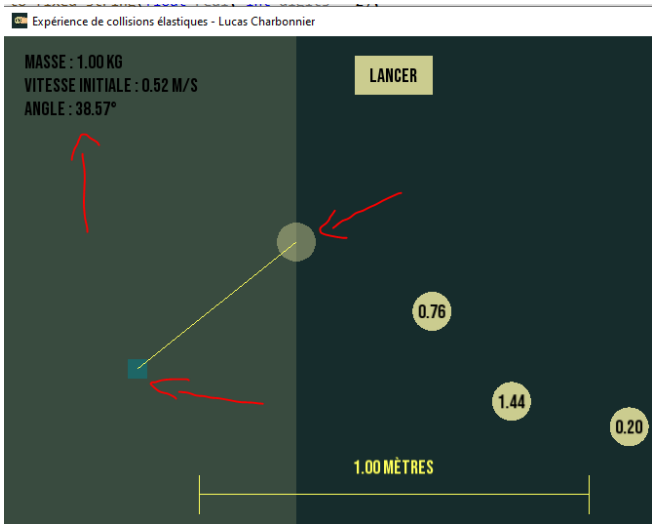


Figure 21 - Le lanceur de projectiles

6.5.4.6 Classe LauncherSling

La classe **LauncherSling** représente le bras de lancement du **ProjectileLauncher**. Le bras de lancement permet de glisser/déposer le petit carré servant à donner l'impulsion et la direction du projectile.

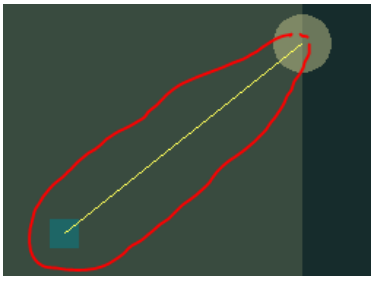


Figure 22 - Le bras de lancement du projectile

6.5.4.7 Classe TargetSlider

La classe **TargetSlider** est utilisée dans la simulation de balistique pour représenter la cible. Elle permet également à l'utilisateur de déplacer la cible sur son axe vertical.

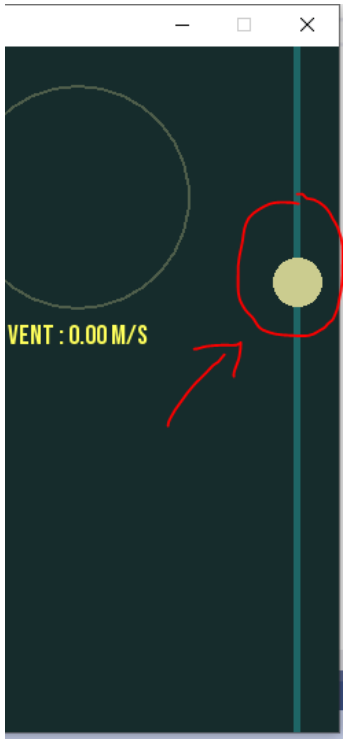


Figure 23 - La cible de la simulation de balistique

6.5.4.8 Classe WindPicker

La classe **WindPicker** permet à l'utilisateur de spécifier la direction et l'intensité du vent. Voir Sélection de la direction et de l'intensité du vent pour plus d'informations sur son fonctionnement.

6.5.4.9 Classe VectorArrow

La classe **VectorArrow** sert simplement à représenter graphiquement un vecteur sur la fenêtre. Elle est par exemple utilisée pour afficher la direction du vent.

Figure 24 - La flèche du WindPicker est une VectorArrow

6.5.4.10 Classe ScaleIcon

Le widget **ScaleIcon** représente une échelle (comme on pourrait en trouver sur une carte papier) permettant d'afficher la correspondance entre un mètre et un certain nombre de pixels. Ce n'est rien d'autre qu'une distance accompagnée d'une légende.

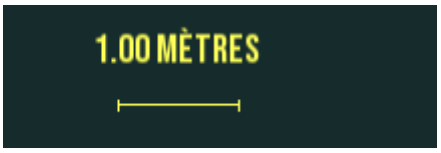


Figure 25 - Échelle de la simulation

6.5.4.11 Classe FadingText

La classe **FadingText** est utilisée pour l'affichage du message au centre de l'écran pour la simulation de balistique. Elle permet d'afficher du texte qui devient de plus en plus transparent jusqu'à sa disparition totale.

FadingText hérite de **Label**.

Figure 27 - FadingText venant d'apparaître

Figure 27 - FadingText proche de la disparition

6.5.5 Autres

À part les classes, il y a quelques éléments importants du code source qui méritent d'être mentionnés.

6.5.5.1 Fichier constants.h

Ce fichier est extrêmement important car il contient les constantes de l'application. Il ne contient pas toutes les constantes mais surtout celle qui sont intéressantes à modifier et qui utilisées à de nombreux endroits du code, telles que :

- La correspondance entre mètres et pixels
- Le rayon des projectiles
- La masse maximale des projectiles
- Le nombre d'objets dans la simulation de collisions
- La force de gravité
- La densité de l'air
- Etc…

Les autres constantes sont généralement présentes seulement là où elles sont utilisées pour éviter que l'entier du code source ne soit recompilé à chaque fois qu'on en change une.

6.5.5.2 Fichier color_palette.h

Le fichier **color_palette.h** contient le **namespace color_palette** qui lui-même contient des constantes relatives à la palette de couleurs de l'application.

6.5.5.3 Fichier main.cpp

Main.cpp contient la fonction **main()**, point de départ du programme. Cette fonction s'occupe uniquement de lancer la fenêtre du menu principal.

```
#include "MainMenuApplication.h"

int main() {
    MainMenuApplication menu;
    menu.run();
}
```

Figure 28 - Le main ne contient que deux lignes

6.5.5.4 Namespace « utils »

Le **namespace utils** contient des fonctions libres qui n'ont leur place nulle part ailleurs dans l'application. Il contient 3 fonctions :

- La fonction **float_to_fixed_string()** servant à convertir un nombre réel en un **std::string** avec un certaine précision (nombre de chiffres après la virgule).
- La fonction **get_default_font()** qui s'occupe de charger et de fournir une référence statique vers la police d'écriture de l'application (Bebas Neue, voir analyse : Police d'écriture utilisée).
- La fonction **get_application_icon_image()** qui fournit le même service que **get_default_font()**, mais pour l'image représentant l'icône de l'application.

6.6 Gestions des collisions

À chaque mise à jour de la simulation, on va vérifier si les objets entrent en collision. Dans cette section, les différents algorithmes utilisés pour la résolution des collisions seront décrits.

6.6.1 Collisions entre les billes et les bords de la fenêtre

Les collisions entre les billes et les bords de la fenêtre sont relativement simples à résoudre. Pour cela, on vérifie si la bille dépasse d'un des côté de la fenêtre, on la déplace de la distance dont elle dépasse et on inverse la composante de se vecteur de vitesse qui est perpendiculaire avec le côté de la fenêtre concerné.

Voici comment j'ai implémenté ces collisions :

```
6   void CircleRigidBody::collideWithWalls(ch::AABB enclosingWalls) {
7       auto circleAABB = ch::collision::enclosingAABB(circle_);
8       if (!ch::collision::aabb_contains(enclosingWalls, circleAABB)) {
9           auto collision = ch::collision::aabb_collision_info(enclosingWalls, circleAABB);
10
11          auto direction = -collision.normal;
12          float distance = circle_.diameter() - collision.absolutePenetrationDepthAlongNormal();
13
14          move(distance * direction);
15          accelerate(ch::vec_dot_product(vel_, collision.normal) * direction * 2.f);
16      }
17  }
```

Figure 29 - Gestion des collisions entre un corps rigide et les murs

C'est une fonction membre de la classe **CircleRigidBody** qui, comme on l'a vu, sert à représenter un objet physique. Le contenu de cette fonction peut paraître un peu compliqué à première vue, c'est pourquoi je vais le détailler un peu.

Tout d'abord, il faut comprendre à quoi correspond l'unique paramètre de la fonction. **enclosingWalls** est un rectangle définissant la zone dans laquelle le cercle est contenu. En pratique, cette zone sera positionnée

À la ligne 7, on convertit le cercle sous-jacent de l'objet physique en une AABB (c'est-à-dire, un rectangle). C'est une optimisation dont le but est de simplifier les futurs calculs. En effet, lorsque on essaie de garantir qu'un cercle est dans un rectangle, on peut réduire la forme du cercle à l'AABB la plus petite le contenant puisque, peu importe ce qui se passe, le cercle ne touchera jamais les bords avec un point autre que ses extrêmes verticaux et horizontaux.

Ensuite, on effectue la première vérification : Est-ce que l'AABB du cercle est contenue dans les bords de la simulation ? Si ce n'est pas le cas, il y a une collision et on procède aux vérifications plus complexes.

En cas de collision, on va chercher à obtenir plus de détails concernant cette collision. La fonction **aabb_collision_info()** permet d'obtenir l'état d'une collision entre deux rectangles (AABB). Cette fonction retourne une structure contenant la direction de la collision ainsi que la profondeur de pénétration de la deuxième AABB dans la première. C'est une fonction qui provient de ma bibliothèque et je ne vais pas détailler son fonctionnement car elle a été écrite en dehors du cadre de ce TPI.

Cependant, cette fonction est originellement destinée à résoudre des collisions entre deux rectangles de manière à ce qu'ils ne soient plus en collision. Ici, on veut exactement l'inverse : on cherche à faire en sorte que la deuxième AABB soit toujours dans la première. C'est pour cette raison qu'à la ligne 11, on déplacera le cercle dans une direction égale à l'inverse de la normale de collision.

À la ligne 12, on calcule la distance de déplacement pour qu'elle soit égale, non pas à la profondeur de collision (puisqu'elle définit la distance pour faire **sortir** la deuxième AABB de la première) mais au diamètre du cercle moins la profondeur de collision. Ce calcul nous donne la distance de laquelle la deuxième AABB dépasse de la première.

Une fois la bille repositionnée, il faut également lui appliquer une accélération. Le but ici est d'inverser la composante du vecteur de vitesse qui est parallèle à la normale de collision. Donc, si la bille touche le haut ou le bas de la fenêtre, on inversera la composante Y. Pour ce faire, on pourrait vérifier manuellement à l'aide de conditions (if…else) le côté concerné. Une autre manière d'atteindre ce résultat est d'appliquer une accélération égale au double de la vitesse dans l'axe concerné. Cela revient au même et permet d'appliquer cette accélération en une seule ligne de code (ligne 15).

6.6.2 Collisions entre deux projectiles

La logique de détection et de résolution de collisions entre deux projectiles a été placée dans la classe représentant les projectiles (**CircleRigidBody**). Pour résoudre une collision entre deux projectiles il suffit d'appeler la méthode **collideWith()** d'un des deux projectiles et de lui donner l'autre en paramètre. Voici la méthode :

```
void CircleRigidBody::collideWith(CircleRigidBody& other) {
    // Détection de la collision
    ch::CirclesCollision coll = ch::collision::circles_collision_info(circle_, other.circle_);

    // Si il n'y a pas de collision, on ne va pas plus loin.
    if (coll.normal == ch::NULL_VEC)
        return;

    // Calcul du vecteur de mouvement
    ch::vec_t movement = coll.normal * coll.absoluteDepth / 2.f;

    // Application du déplacement aux deux projectiles
    move(-movement);
    other.move(movement);

    // Calcul du vecteur d'impulsion
    float j = ch::vec_dot_product(2.f * coll.normal, vel_ - other.vel_) / (inverseMass() + other.inverseMass());
    ch::vec_t impulse = coll.normal * j;

    // Application de l'impulsion aux deux projectiles
    accelerate(-impulse / mass_);
    other.accelerate(impulse / other.mass_);
}
```

Figure 30 - Gestion des collisions entre deux corps rigides

Et un exemple de son utilisation dans la méthode **update()** de la classe **CollisionSimulationApp** :

```
for (size_t i = 0; i < circleRigidBodies_.size(); ++i) {
    auto& first = circleRigidBodies_[i];
    for (size_t j = i + 1; j < circleRigidBodies_.size(); ++j) {
        auto& second = circleRigidBodies_[j];

        first.collideWith(second);
    }
}
```

Figure 31 - Gestion des collisions dans l'application

La double boucle permettant la résolution de toutes les collisions entre les billes est particulière. On peut voir que la première commence à zéro, alors que la deuxième commence à i+1. Cette spécificité permet d'éviter que des collisions soit vérifié entre une bille et elle-même et également d'éviter que la collision entre deux billes soit vérifiée plusieurs fois. C'est une optimisation qui, dans notre cas, n'aura pratiquement pas d'impact sur les performances mais qui peut s'avérer très utile dans certains cas (avec plus d'un millier d'objets, par exemple).

6.7 Échelle de la simulation

Dans une simulation informatique, il n'y a pas d'unités. L'unité des nombres avec lesquels on travaille n'a pas de sens (mètres ? millimètres ? centimètres ?). Ou, du moins, jusqu'à ce qu'on décide d'afficher la simulation.

Quand on spécifie une position sur l'écran, on ne sait pas à quelle distance réelle cela correspond, on donne simplement le nombre de pixels depuis la gauche et depuis le haut de l'écran (ou de la fenêtre). Donc, par défaut, l'unité qui sera utilisée sera les pixels. Généralement cela ne pose pas problème, mais lorsque l'on développe une simulation qui donnera un résultat visuel, il est nécessaire que l'échelle de l'affichage aie du sens pour l'utilisateur.

Les « pixels » ne sont pas une bonne unité. Un pixel ne correspond à rien pour l'utilisateur et ce pour une bonne raison : la taille d'un pixel n'est pas fixe, elle peut varier d'un écran à l'autre. Ainsi, deux écrans de même taille peuvent comporter un nombre différent de pixels.

Pour résoudre ce problème, il faut mettre en place deux éléments importants. Tout d'abord, il faut définir précisément à quoi correspond un pixel. Pour cela, on définit une constante qui définira à combien de pixels un mètre correspond. Ensuite, il faut afficher cette échelle d'une manière ou d'une autre pour que l'utilisateur puisse observer la simulation et comprendre ses dimensions.

Voici la constante qui a été définie :

```
1   #pragma once
2
3   constexpr float PIXELS_PER_METER = 300.f;
4
```

L'avantage d'utiliser une constante est que l'on peut très simplement changer sa valeur pour adapter l'échelle de la simulation.

Une fois cette constante définie, il fallait un moyen de permettre à l'utilisateur de la voir. Le widget **ScaleIcon** a donc été créé (voir Classe ScaleIcon) pour permettre à l'utilisateur de mieux percevoir la taille de la simulation.

6.8 Application de la force de frottement du vent au projectile

Le calcul de la force de frottement s'est fait exactement comme prévu dans la conception. Pour rappel, la formule qui avait été développée était la suivante :

$$\overrightarrow{a_{proj}} = -\frac{\rho \pi r^2 C_x \cdot V_{rel}}{2m} \cdot \left(\overrightarrow{V_{air}} - \overrightarrow{V_{proj}}\right)$$

Et permettait de calculer l'accélération subie par un projectile sphérique (C_x = 0.47) de rayon r et de masse m, se déplaçant avec une vélocité V_{proj} dans un fluide avec un courant V_{air}, une densité rho et une vitesse relative V_{rel}. Cette formule est appelée à chaque mise à jour de la simulation de balistique. Voici comment elle a été implémentée dans le code :

```
void BallisticSimulationApp::applyDragForce(float dt) {
    // On récupère la vélocité de l'air (vitesse et direction du vent) à partir du WindPicker
    ch::vec_t fluidVel = windPicker_.computeWindIntensityAndDirection();

    // Vélocité du projectile
    ch::vec_t projVel = projectile_->getVelocity() / PX_PER_METER_BALLISTIC;

    // Masse et rayon
    float m = projectile_->getMass();
    float r = OBJECTS_RADIUS_BALLISTIC;

    // Vélocité relative entre le fluide et le projectile
    float relativeVelocity = ch::vec_magnitude(fluidVel - projVel);

    // Calcul de l'accélération
    ch::vec_t acceleration = (RHO * PI * r * r * SPHERE_DRAG_COEFFICIENT * relativeVelocity * (fluidVel - projVel)) / (2.f * m);

    // Application de l'accélération. On multiplie par le pas de temps pour que l'accélération soit proportionnelle au temps écoulé
    projectile_->accelerate(acceleration * dt);
}
```

Figure 32 - Application de la force de frottement au projectile dans la simulation de balistique

Les variables avec des noms en majuscules sont des constantes. Les constantes physiques utilisées pour le calcul du frottement sont définies dans le fichier **constants.h** :

```
constexpr float RHO = 1.225f; // Densité de l'air, en kg/m^3, au niveau de la mer, à 15°C
constexpr float PI = ch::FLT_PI; // Alias d'une autre constante de PI
constexpr float SPHERE_DRAG_COEFFICIENT = 0.47f; // Coefficient de frottement des objets sphériques
```

Figure 33 - Constantes de physique pour l'application du frottement

La constante **RHO** correspond à la densité du fluide (ici, de l'air) et **SPHERE_DRAG_COEFFICIENT** correspond au coefficient de frottement d'une sphère. Bien que la simulation de balistique ne soit qu'en deux dimensions, il reste plus logique d'utiliser le coefficient de frottement d'une sphère, et non d'un cercle (de toute façon, il n'est pas défini car un objet en deux dimensions ne peut pas subir de frottement).

6.9 Configuration du vent

Au moment de la conception des maquettes, il avait été décidé que le vent ne pourrait se déplacer qu'horizontalement. Cependant, lors de la conception des formules de physique, il est apparu que le vent serait de toute façon représenté par un vecteur en deux dimensions. Le fait de spécifier la force et la direction du vent à l'aide d'un vecteur (de vélocité) permet théoriquement de représenter un vent orienté dans n'importe quelle direction.

L'idée de développer un widget permettant de sélectionner l'intensité et la direction du vent à 360° a été analysée et il s'est avéré que ce n'était pas beaucoup plus compliqué que d'implémenter le simple curseur prévu dans la maquette.

Ainsi, le «**WindPicker**» a été implémenté :

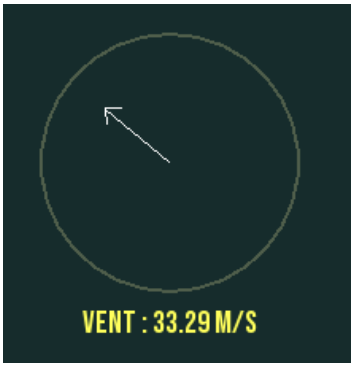


Figure 34 - Le WindPicker

Cette fonctionnalité permet notamment de réaliser une expérience assez intéressante. Si l'on définit la masse du projectile à 0.2 kg (= 200 grammes) et que l'on spécifie un vent orienté vers le haut à 30 m/s, on s'aperçoit que le projectile lévite dans l'air. Cela est dû au fait qu'un vent de 30 m/s provoque une force de frottement suffisamment forte pour compenser la force de gravité et donc l'annuler[11].

6.10 Affichage de la trace des projectiles

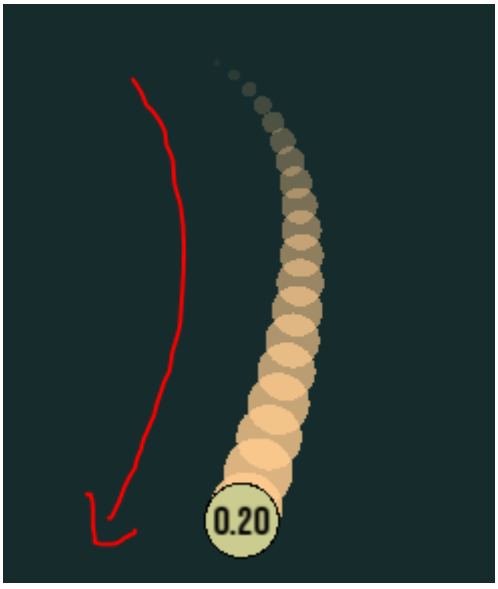


Figure 35 - Trace d'un projectile forcé de retourner en arrière à cause des frottements du vent

L'affichage de la trace des projectiles a été l'une des fonctionnalités les plus simples à implémenter.

Pour l'afficher, il suffit de sauvegarder la position du projectile toutes les X millisecondes et d'ajouter cette position à une file d'attente (**std::deque**[12]). Lorsque la file d'attente dépasse une certaine taille, on enlève

[11] C'est d'ailleurs le même phénomène qui est utilisé dans les simulateurs de chute libre.

[12] Documentation de **std::deque** : https://en.cppreference.com/w/cpp/container/deque

le dernier élément. Ensuite, il ne reste plus, au moment de l'affichage, qu'à affichant un cercle de taille décroissante centré à toutes les positions contenues dans la file d'attente.

Voici comment la trace, qui n'est autre qu'un historique des positions du projectile, est maintenue dans le code :

```
void CircleRigidBody::savePosition() {
    trace_.push_back(circle_.pos); // Ajout de la position actuelle à l'arrière de la file d'attente

    if (trace_.size() > OBJECTS_TRACE_HISTORY_LENGTH) { // Si la file d'attente dépasse sa taille max...
        trace_.pop_front(); // On enlève la position à l'avant de la file d'attente
    }
}
```

Figure 36 - Gestion de l'historique des positions du projectile.

6.11 Différences entre les maquettes et l'interface finale

Il y a plusieurs différences entre les maquettes conçues durant l'analyse et le résultat final. Cette section explique ces différences.

6.11.1 Cible remplacée par un cercle au lieu d'un carré

Dans la maquette de la simulation de balistique, la cible est représentée avec un carré. Cependant, elle a été remplacée par un cercle durant le développement. La raison de ce changement est dû au fait qu'une classe **DraggableCircle** a été implémentée et qui permettait de gérer la fonctionnalité de drag&drop des projectiles. Si la cible était carrée, j'aurais dû implémenter une autre class **DraggableAABB** (ou éventuellement **DraggableRectangle**) pour gérer le déplacement vertical de la cible mais je me suis dit qu'il était plus simple de juste remplacer le rectangle par un cercle et ainsi pouvoir utiliser la même class pour implémenter le déplacement de la cible.

De plus, les collisions entre deux cercles sont toujours plus simples (et plus rapides) à détecter que les collisions entre un cercle et un rectangle (dû aux multiples cas de collision possibles).

6.11.2 Remplacement du curseur de vitesse du vent par le WindPicker

Dans la maquette de la simulation de balistique, il n'y avait pas de WindPicker, seulement un curseur permettant de définir la vitesse du vent sur l'axe horizontal.

Cette fonctionnalité a été ajoutée car elle permettait de faire des expériences de balistiques assez intéressantes et qu'elle n'était pas plus complexe à concevoir qu'un curseur.

Plus d'informations dans la section Configuration du vent.

6.12 Création d'un installateur pour déployer l'application

Cette section explique comment l'installateur de l'application a été créé.

6.12.1 Compatibilité 32/64 bits

Le cahier des charges spécifie clairement que l'installateur doit pouvoir s'exécuter sur une machine Windows 32 out 64 bits. Étant donné qu'un projet ne peut pas être lié en 32 **et** 64 bits à la bibliothèque SFML, il a été décidé de ne produire qu'une version 32 bits.

Un programme compilé pour une architecture 64 bits ne fonctionnera que sur une machine en 64 bits alors qu'un exécutable compilé pour une architecture 32 bits pourra s'exécuter sur un système 32 ou 64 bits sans problème.

Pour simplifier l'installateur, il a été décidé que le programme déployé sera toujours prévu pour un système 32 bits.

6.12.2 Développement de l'installateur

Pour créer l'installateur, l'extension Visual Studio « Microsoft Visual Studio Installer Projects » doit être installée. Ensuite, il faut ajouter un nouveau projet à la solution Visual Studio de Physical Event Simulation et choisir le type de projet « Setup Wizard » :

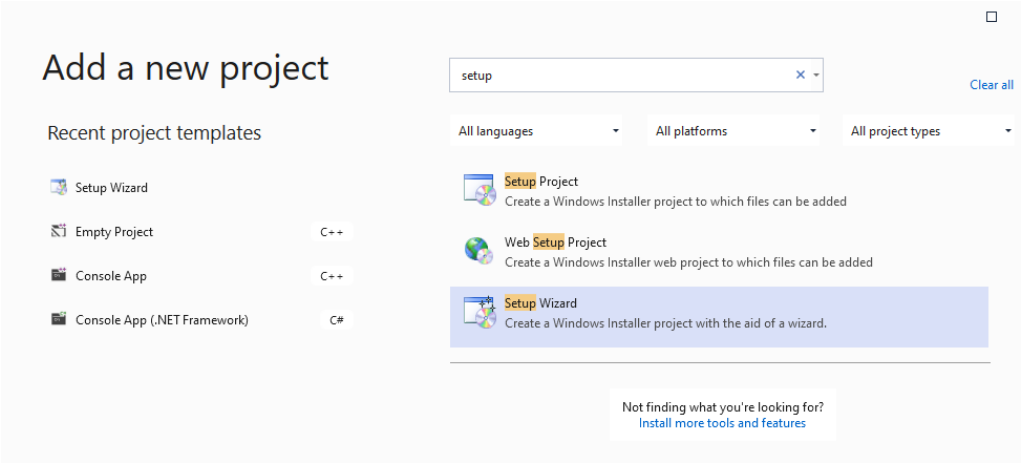


Figure 37 - Création d'un projet d'installateur avec MSVS Installer Projects

6.13 Bugs trouvé et leur résolution

Cette section répertorie les bugs détectés qui ont pu être résolus. Par « bug », ici, on entend tout comportement du programme étant indésirable du point de vue de l'utilisateur.

6.13.1 Pas de scroll sur laptop

Ce bug a été découvert lorsqu'un des experts, Monsieur Carrel, est venu voir l'avancement du projet lors de sa visite du 19 mai 2021. Après l'avoir aidé à compiler le projet, il a testé l'application et, étant donné qu'il était sur un laptop, n'a pas pu utiliser le scroll de la souris pour faire varier la masse des projectiles.

Étant donné qu'il n'était pas spécifié dans le cahier des charges que l'application devait être utilisable sur laptop, ce bug n'était pas une priorité absolue mais il a tout de même été résolu.

Pour résoudre ce problème, la possibilité de faire varier la masse à l'aide des touches fléchées du clavier a été implémentée (flèche du haut pour augmenter la masse et flèche du bas pour la diminuer).

7 Tests

7.1 Environnement de test

Pour pouvoir effectuer les tests détaillés ci-dessous, le testeur doit disposer d'une installation de Windows 10 32 ou 64 bits ainsi que de l'installateur (.msi ou .exe) de l'application Physical Event Simulation.

7.2 Liste des tests

Le tableau suivant décrit tous les tests effectués ainsi que les résultats obtenus :

Fonctionnalité testée	Déroulement du test	Résultat attendu	Résultat obtenu	Validation
Installation du programme via l'installateur	Sur une installation de Windows 32 bits, lancer le fichier de setup (le .msi ou le .exe) et laisser les paramètres par défaut de l'installation. Répéter le test pour une installation 64 bits.	L'application doit être installée dans un répertoire qui a du sens et elle peut être lancée sans erreur ou bug. L'icône de la fenêtre ainsi que la police d'écriture de l'application doivent être présentes.	Le programme est installé dans « C:/Program Files/ETML/Physical Event Simulation/ » et un raccourci vers le .exe est créé sur le bureau. Le programme se lance sans aucune erreur ou anomalie.	**OK**
Lancement du programme et navigation	Lancer le programme et naviguer entre les deux simulations	Le menu doit s'afficher, disparaître lorsque on ouvre une des deux simulations et réapparaître lorsqu'on la ferme.	Comme attendu	**OK**
Démarrage/arrêt d'une des deux simulations	Ouvrir la simulation de chocs, la lancer avec le bouton prévu à cet effet, puis l'arrêter. Faire pareil pour l'autre simulation.	À l'ouverture de la fenêtre, la simulation est en pause. Lorsque on clique sur le bouton démarrer, la simulation doit se lancer et elle doit s'arrêter et revenir aux paramètres de départ lorsque l'on reclique sur le même bouton. Ce test s'applique aux deux simulations	Comme attendu	**OK**
Orientation du lancer d'un projectile avec le lanceur	Ouvrir une des deux simulations. Lancer un projectile/une bille. Faire pareil pour l'autre simulation.	La bille doit être lancée dans la direction souhaitée. C'est-à-dire, à l'opposé du petit carré du lanceur.	La bille est lancée dans la direction donnée par le lanceur.	**OK**
Déplacement de la hauteur initiale du projectile	Ouvrir une des deux simulations et faire varier la hauteur du projectile et essayer de le sortir des limites	On peut changer la hauteur du projectile mais il ne peut pas dépasser en haut ni en bas.	On peut déplacer verticalement le projectile mais celui-ci est bloqué par les murs de la fenêtre	**OK**
Déplacement de la hauteur de la cible	Ouvrir la simulation de balistique et faire varier la hauteur de la cible et essayer de la sortir des limites.	On peut changer la hauteur de la cible mais elle ne peut pas dépasser en haut ni en bas.	On peut déplacer verticalement la cible mais celle-ci est bloquée par les murs de la fenêtre	**OK**

Variation de la vitesse du projectile	Ouvrir une des deux simulations. Lancer une bille à 45°. Arrêter la simulation et relancer une autre bille à 45° mais en choisissant une plus grande vitesse. Faire pareil pour l'autre simulation	Les deux lancers doivent suivre la même direction mais le deuxième lancer doit partir plus vite.	L'intensité dépend linéairement de la longueur du bras de lancement	**OK**
Variation de la masse des projectiles	Ouvrir une des deux simulations, modifier la masse du projectile. Faire pareil pour l'autre simulation.	La masse doit être changée et on doit pouvoir visualiser sa valeur exacte.	En scrollant la molette de la souris, on peut modifier la masse du projectile et celle-ci s'affiche en haut à droite	**OK**
Variation de la masse des billes dans la simulation de collisions	Ouvrir la simulation de collisions et essayer de changer la masse des billes avec la molette de.	La masse doit être changé et l'on doit pouvoir visualiser sa valeur exacte.	En scrollant la molette de la souris, on peut modifier la masse des billes et celle-ci s'affiche sur elles.	**OK**
Déplacement des billes dans la simulation de collisions	Ouvrir la simulation de collisions et essayer de drag&drop les billes dans la zone ainsi qu'à l'extérieur de la fenêtre.	Les billes doivent pouvoir être drag&drop au sein de la fenêtre mais il est impossible de les sortir de la fenêtre.	On peut les déplacer mais elles ne sortent pas des limites	**OK**
Application de la gravité sur le projectile de la simulation de balistique	Effectuer un même lancer dans la simulation de chocs puis de balistique	Dans la simulation de chocs, la bille doit partir tout droit. Cependant, le projectile doit suivre une trajectoire plus ou moins parabolique dans la simulation de balistique. Le test consiste en le constat de cette différence.	Gravité dans la simulation de balistique mais pas dans la simulation de collisions	**OK**
Frottement dans un fluide immobile	Effectuer deux lancers à 45° dans la simulation balistique. L'un avec un objet très léger, l'autre avec un objet très lourd. La vitesse du vent doit être de 0m/s.	La trajectoire du projectile léger doit montrer que sa masse faible fait que les frottements affectent beaucoup sa vitesse. En revanche, le projectile lourd doit être peu affecté par le frottement.	Le projectile va plus loin que celui qui est léger	**OK**
Frottement dans un fluide orienté en direction de la cible	Effectuer un lancer à 45° dans la simulation balistique avec un objet léger et avec un vent orienté vers la cible.	La trajectoire du projectile doit montrer que le vent l'accélère horizontalement en direction de la cible.	Le projectile va plus loin lorsqu'il est aidé par le vent	**OK**

Frottement dans un fluide orienté contre le projectile	Effectuer un lancer le plus vertical possible dans la simulation balistique avec un objet léger et avec un vent fort orienté contre le projectile.	La trajectoire du projectile effectuera un retour en arrière.	Avec un projectile très léger et un vent très fort orienté vers la gauche, le projectile retourne en arrière.	**OK**
Collision entre le projectile et la cible	Effectuer un lancer où le projectile atteint la cible	Un message doit s'afficher à l'écran pour prévenir l'utilisateur	Le message « cible atteinte ! » s'affiche.	**OK**

7.3 Bilan des tests

Comme on peut le constater, tous les tests ont pu être validés. Cependant, il est important de noter que tout ces tests sont de nature qualitative. Par exemple, les tests concernant l'application de la force de frottement au projectile sont basés uniquement sur ce que l'utilisateur peut voir mais pas sur les résultats exacts des calculs de physique (comme ce serait le cas si les tests étaient quantitatifs).

En d'autres termes, pour tester la simulation, on fait appel à l'intuition de l'utilisateur pour déterminer si le comportement perçu dans la simulation correspond à un comportement que l'on pourrait observer dans le monde réel. Si les comportements au sein de la simulation correspondent à l'intuition de l'utilisateur, on peut partir du principe que la simulation est « suffisamment réaliste » pour un être humain.

Cela dit, bien que ce niveau de précision pourrait être suffisant pour un jeu vidéo (où tout ce qui compte est ce que ressent l'utilisateur et pas forcément le réalisme de la simulation), l'application d'une telle stratégie de tests pour une simulation comme Physical Event Simulation est discutable.

Ainsi, bien que tous les tests aient été validés, il est possible que la simulation comporte des erreurs qui n'ont pas été détectées lors des tests.

8 Conclusion

Cette section sert à faire le point sur l'état du projet une fois la réalisation terminée.

8.1 Améliorations possibles et bugs connus

Dans cette section, les différentes améliorations possibles ainsi que les éventuels bugs connus sont détaillés.

8.1.1 Trace pour les projectiles

Actuellement, la trace des projectiles a un aspect très esthétique, très agréable à regarder. Cependant, ce n'est pas forcément optimal pour une simulation, en particulier si l'on recherche à avoir une prédiction détaillée de la trace.

Dans cet objectif, il serait donc peut-être plus judicieux de mettre en place une trace moins esthétique mais plus détaillée. On pourrait, par exemple, remplacer les cercles par des points et également ralentir (voir désactiver totalement) la disparition de la trace avec le temps.

8.1.2 Palette de couleur et aspect visuel de l'application

La palette de couleur utilisée par l'application a été créé rapidement pour combler un vide et en pensant que les couleurs choisies pourraient être améliorées plus tard. Finalement, aucun temps n'a été consacré à

l'amélioration de l'aspect visuel de l'application et ces couleurs de « test » ont été gardées. Les couleurs choisies ne sont pas forcément problématiques mais il ne fait aucun doute qu'elles pourraient être améliorées pour rendre l'application plus agréable à utiliser.

Il vaut le coup de préciser que je suis daltonien (vert/rouge). Je ne suis donc peut-être pas la personne de référence en ce qui concerne le choix et la conception d'une palette de couleur. C'est une des raisons qui m'a poussé à ne pas passer trop de temps là-dessus.

8.1.3 Impossible de tirer vers le haut si le projectile est positionné en bas

Le lanceur de projectiles comporte une limitation qui vaut le coup d'être soulignée. Pour rappel, il permet de définir la direction et la vitesse de départ du projectile initial dans les deux simulations. Pour ce faire, on doit positionner un carré dans la zone à gauche de l'écran. La hauteur du projectile de départ peut également être définie.

La limitation survient si l'on veut, par exemple, envoyer vers le haut un projectile situé tout en bas de la fenêtre. C'est tout simplement impossible puisqu'on ne pourra pas positionner l'embout du bras de lancement (le petit carré) en dessous du projectile, vu que celui-ci est déjà à l'extrême bas de la fenêtre.

Une solution possible serait de permettre de déplacer l'embout du bras de lancement en dehors de la fenêtre, mais ce ne serait pas très esthétique. Une autre possibilité serait de changer totalement la manière dont l'angle et la vitesse de départ du projectile sont spécifiés mais il faut avouer que le système actuel est quand même assez facile et intuitif à utiliser et il serait dommage de l'abandonner.

8.2 Comparaison entre la réalisation et la planification

Maintenant que la réalisation est terminée, on peut effectuer une comparaison des heures passées sur les différentes tâches. Pour cela, il suffit de consulter le bilan graphique de la planification et du journal de travail :

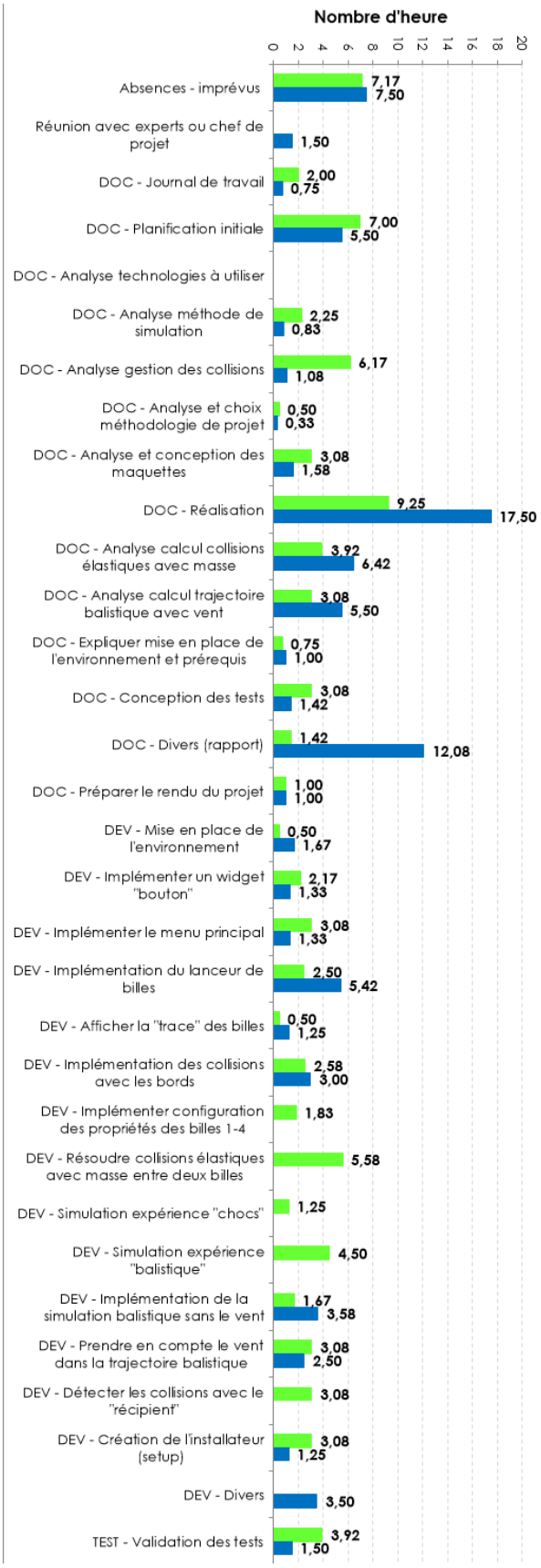


Figure 38 - Bilan graphique (le temps planifié en vert et la réalisation en bleu)

Malheureusement, vu que le projet a été séparé en de nombreuses petites tâches, le bilan graphique n'est pas très lisible. On peut toutefois noter quelques éléments intéressants.

8.2.1 Oubli de reporter certaines tâches

La première chose qu'on peut remarquer est que certaines tâches ont été planifiées mais pas réalisées (la tâche « DEV – Implémentation de la configuration des propriétés des billes 1-4 », par exemple). On pourrait croire que cela veut dire que la tâche n'a pas été effectuée mais en réalité c'est simplement dû au fait que le travail effectué a été inclus dans un autre tâche. Pour ce cas en particulier, il faut savoir que la principale complexité de la configuration des billes était l'implémentation du drag&drop et que celle-ci a été résolue durant l'implémentation du lanceur de projectiles. Ainsi, l'implémentation de la configuration des billes n'a peut-être pris qu'une dizaine ou une vingtaine de minutes et le temps a été attribué à la tâche « DEV - Implémentation du lanceur de projectiles ».

La raison de ce problème est tout bêtement un oubli de ma part. La tâche devait être si rapide à implémenter que j'ai dû simplement oublier que j'avais créé une tâche spécifique pour ce travail.

Un autre exemple est la tâche « DEV – Résoudre collisions élastiques avec masse entre deux billes », pour laquelle j'ai planifié ~5h30 de temps pour mais que j'ai apparemment totalement oublié de reporter dans le journal de travail. C'est forcément un oubli car les collisions ont bel et bien été implémentées (même si cela n'a pas pris beaucoup de temps, étant donné que les formules avaient été conçues durant l'analyse).

Personnellement, je pense que ces problèmes d'oublis sont dus en partie au format du fichier de planification et de journal de travail car, à mon avis, il n'est pas adapté à un projet de développement comportant de nombreuses petites tâches. Je pense qu'un tableau kanban (avec 3 colonnes : à faire, en cours et terminé) aurait été plus adapté.

8.2.2 Absences planifiées

On peut également remarquer que la « tâche » nommée « Absences-Imprévus » a du temps planifié. Ce temps planifié correspond simplement à la journée durant laquelle j'ai dû m'absenter pour participer à mon cours de conduite « 2-phases ».

8.2.3 Temps réalisé mais pas planifié

Certaines tâches, telle que « DEV – divers » ont été réalisées mais pas planifiées. C'est parfaitement normal. En fait, ces tâches ont été prévues comme tâches « tampons » qui seront utilisées pour comptabiliser le travail effectué qui ne rentre dans aucune autre tâche.

Il y a aussi la tâche « Réunion avec experts ou chef de projet » dans la même situation. En effet, je ne savais pas, au moment de la planification initiale au premier jour du projet, quand auraient lieu ni combien de temps prendraient les différentes réunions avec les intervenants du projet.

8.2.4 Travail à la maison

Le temps passé à la maison n'est pas explicitement montré sur le bilan graphique mais il a été reporté dans le journal de travail, à la séquence 22 :

Séquence	22		Date
Tâche	**Tranche [5min]**		**Explications: qu'est-ce qui se fait et comment ?**
TEST - Validation des tests	18	1h30min	Écrits les différents tests et effectué ceux-ci
DOC - Réalisation	20	1h40min	Continué de rédiger la partie "réalisation" du rapport
			Ci-dessous, le travail effectué durant le week-end du 22-24 mai 2021 :
DEV - Création de l'installateur (setup)	5	0h25min	Étant donné que l'installateur doit pouvoir être lancé sur une machine 32 bits comme sur une machine 64 bits, j'ai décidé de changer la version de la sfml liée au projet pour que le projet soit compilé en 32 bits et ainsi de n'avoir qu'une seule version de l'installateur.
DOC - Divers (rapport)	6	0h30min	Expliqué comment la compatibilité 32 et 64 bits sera assurée (voir section "5.12 Création d'un installateur pour déployer l'application" rapport).
DEV - Création de l'installateur (setup)	10	0h50min	Créé et testé l'installateur sur une machine virtuelle 32 bits
DOC - Divers (rapport)	12	1h	Mieux détaillé les tests et rédigé l'abstract en début de document + d'autres détails
Total tranche	71	5h55min	

Figure 39 - Travail effectué à la maison (2.75 heures)

8.2.5 Proportion du temps passé sur chaque partie du projet

Vous trouverez ci-dessous deux graphiques permettant de visualiser et de comparer le temps planifié pour chaque domaine du projet au temps réalisé.

Noter que pour ces deux graphiques, seul le temps total de travail a été compris dans le calcul (les absences et les imprévus ont été exclus), ce qui donne un total de 82 heures et 50 minutes de travail (994 tranches de 5 minutes). Les proportions qui suivent sont donc relative à ce temps total.

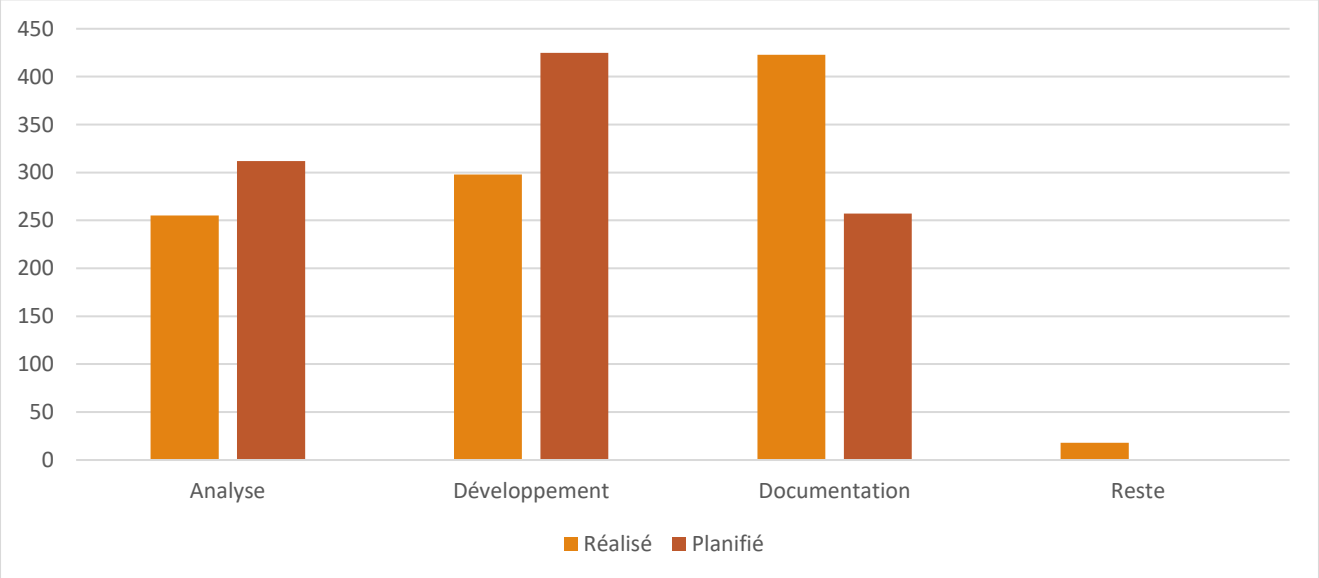


Figure 40 - Temps planifié et réalisé pour chaque partie du projet (en tranches de 5 minutes)

Le graphique ci-dessous est similaire mais présente la proportion sous forme de pourcentage :

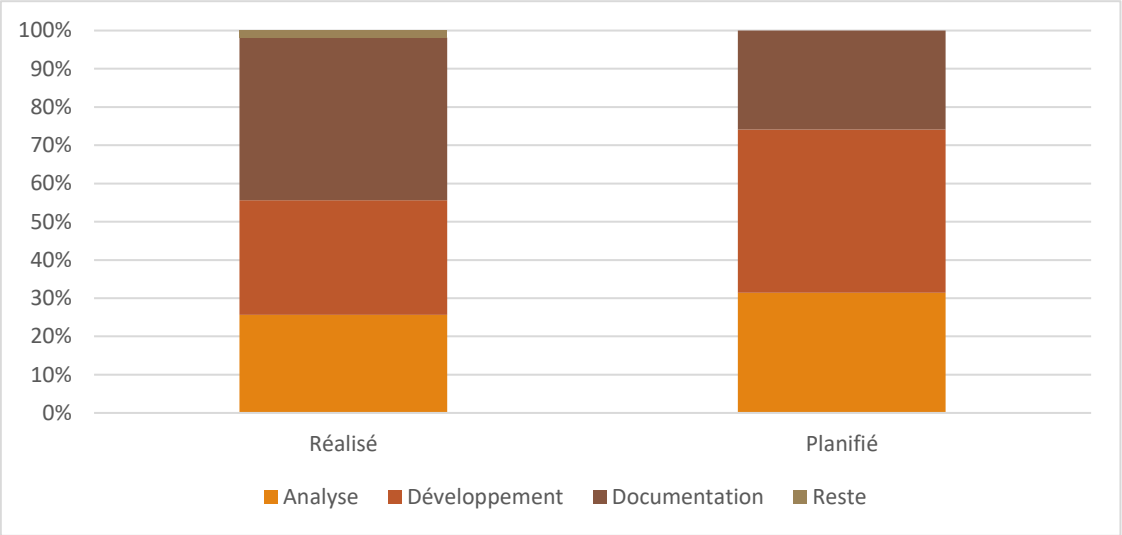


Figure 41 - Comparaison de la proportion de temps planifié et réalisé pour chaque partie du projet

Ces graphiques nous permettent de constater plusieurs détails intéressants concernant la planification du projet.

Tout d'abord, on peut voir que le temps prévu pour l'analyse et le développement ont été légèrement surestimés de respectivement 10% et 20% alors que le temps nécessaire à la documentation a, lui, été sous-estimé, avec une différence d'environ 20% aussi. Cette sous-estimation du temps consacré à la documentation est assez récurrente parmi tous les projets sur lesquels j'ai travaillé. Cela dit, je ne pense pas qu'elle est due à une réelle sous-estimation, mais plutôt à une compensation du temps gagné sur l'analyse et le développement.

Je m'explique. Même s'il n'y avait plus de travail à effectuer, ne « rien faire » ne serait pas une option. Peu importe la quantité de temps gagné sur le début du projet, la durée totale reste la même (on ne peut pas rendre en avance). Donc, si on gagne du temps quelque part, cette différence sera forcément compensée sur une autre partie du projet. Du coup, le fait de gagner du temps sur l'analyse et le développement ne va pas seulement diminuer leur proportion mais va également augmenter la proportion du temps passé sur la documentation.

Le fait que le projet ait une durée fixe est un détail important qui va influencer la manière dont ces graphiques devront être interprétés.

8.3 Autoévaluation

Pour l'autoévaluation, je vais reprendre un par un les « Points techniques évalués spécifiques au projet » et déterminer s'ils ont été remplis.

8.3.1 Analyse et description sur le choix des formules mathématiques

Le point 1 concernait les formules de mathématiques, leur **exactitude**, leur **justification** et leur **adaptation** dans un langage de programmation.

Il est difficile de juger l'exactitude des formules conçues. Je suis parti de formules exactes (conservation de l'énergie, quantité de mouvement, force de frottement, etc…) mais il est possible qu'une erreur ait été commise lors de la transformation de celles-ci (même si je doute que ce soit le cas). Comme expliqué dans

le Bilan des tests, les formules de physiques n'ont pas été testées quantitativement mais, à vue d'œil, la simulation semble cohérente.

Concernant la justification des formules, j'ai passé du temps dessus et je pense avoir bien réussi à expliquer mon raisonnement.

L'adaptation dans un langage de programmation n'a pas posé problème et s'est faite très rapidement puisque je disposais déjà de structures permettant de représenter les projectiles et leurs différentes propriétés (position, vélocité, masse, etc…).

8.3.2 Simulation du lancer

Ce point concernait la **visualisation** du tir et le **déplacement** du projectile.

La visualisation de la trajectoire a été mise en place pour qu'elle soit cohérente avec l'écoulement du temps réel. L'affichage de la trace permet aussi de faciliter la visualisation. Je pense donc avoir atteint cet objectif.

8.3.3 Représentation de l'effet de choc élastique

La représentation du choc élastique est faite via le changement de trajectoire des différents projectiles. Toutes les billes peuvent interagir entre-elles.

8.3.4 Paramétrage de la simulation

La simulation devait permettre la modification de la **vitesse**, de la **masse** (dans les limites), de la **direction**, de la **position** et de **l'angle** du projectile de départ ainsi que la masse et la position des projectiles 1-4.

Toutes ces propriétés peuvent être modifiées à l'aide de la souris et/ou du clavier. Le changement de masse se fait soit avec la molette de la souris ou avec les touches fléchées du clavier.

8.3.5 Passage en tout temps d'une simulation à l'autre

L'écran titre permet d'ouvrir les deux simulations et on peut y revenir en fermant la simulation actuellement ouverte.

8.3.6 Programme de déploiement de l'application (setup)

Un setup a été créé pour l'application. Par défaut, il installe l'application dans « C:/Programmes/ETML/Physical Event Simulation » et créé un raccourci sur le bureau. Le setup peut fonctionner sur un système 64 bits comme sur un système 32 bits.

8.3.7 Qualité du code fourni

Le code est divisé en de nombreux fichiers et classes. Toutes les classes sont séparées en deux fichiers :

- Un fichier d'en-tête (les fichiers .h) contenant la déclaration et l'interface publique et privée
- Un fichier d'implémentation (.cpp) contenant l'implémentation des membres de la classe.

Toutes les classes, fonctions (membres ou non) sont commentées à l'aide d'une courte description du service qu'elles rendent.

Comme expliqué dans l'analyse (Convention de code/nommage), le code ne respectent pas entièrement les normes de codages de l'ETML. Cependant, la convention proposée comme remplacement a été parfaitement respectée dans l'ensemble du code.

8.4 Bilan personnel

Dans l'ensemble, je suis satisfait de mon travail. J'ai réussi à implémenter toutes les fonctionnalités et je pense que la documentation que j'ai rédigée est d'assez bonne qualité. La planification aurait pu être meilleure mais je ne crois pas vraiment pouvoir faire mieux en une seule journée.

Que ce soit à l'ETML, lors des DemoMot de 1ère et de 2ème année, ou dans mon temps libre, j'avais déjà créé quelques simulations de physiques. C'est donc un domaine avec lequel je suis relativement à l'aise. Malgré cela, j'ai trouvé la partie physique/mathématiques du projet plus compliquée que prévu. En effet, bien que j'aie déjà implémenté des collisions et des simulations de balistique, je n'avais jamais pris le temps de me pencher sur les explications mathématiques des formules que j'utilisais.

Pour ce projet, je me suis forcé à partir depuis zéro et effectuer le développement mathématique du début à la fin en détaillant le plus possible les différentes étapes. Mon but était de produire une explication la plus claire possible. Je me suis rendu compte qu'il y avait une grande différence entre le fait de comprendre un concept et être capable de l'expliquer simplement.

Finalement, la conception des formules ressemble un peu à un tutoriel mais je pense que ce n'est pas une mauvaise chose et que cela facilitera la compréhension du lecteur.

Je tiens à remercier mon chef de projet, Monsieur Lymberis, qui m'a donné la possibilité de choisir le domaine du projet. Je suis passionné par les simulations informatiques et c'est un privilège incroyable d'avoir pu effectuer mon travail de diplôme là-dessus.

9 Bibliographie/Webographie

- Lien vers le dépôt GitHub du projet : https://github.com/Raynobrak/etml-tpi
- Site officiel de la SFML : https://www.sfml-dev.org/index-fr.php
- Dépôt de la Charbrary : https://github.com/Raynobrak/Charbrary
- Un tutoriel sur la détection et la résolution de collisions qui m'a beaucoup aidé : https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331
- Un livre sur les bonnes pratiques de programmation qui m'a beaucoup aidé : « Coder efficacement : Bonnes pratiques et erreurs à éviter (en C++) » de Philippe Dunski.

Les autres sources qui m'ont été utiles sont citées dans les références de mon journal de travail.

10 Table des illustrations

11 Annexes

Vous trouverez, joints à ce document, le **cahier des charges** signé suivi de la **planification** initiale et du **journal de travail**.