TPI - Physical Event Simulation

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

Auteur : Lucas Charbonnier

Classe : MID4

Lieu : ETML

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

Chef de projet : Dimitri Lymberis

Experts : Xavier Carrel, Charles-Henri Hayoz

Table des matières

[1 Analyse 2](#_Toc70930266)

[1.1 Méthodologie de projet 2](#_Toc70930267)

[1.2 Technologies à utiliser 3](#_Toc70930268)

[1.2.1 Langage de programmation 3](#_Toc70930269)

[1.2.2 Bibliothèques 3](#_Toc70930270)

[1.3 Convention de code/nommage 3](#_Toc70930271)

[1.4 Méthode de simulation : discrète ou continue 3](#_Toc70930272)

[1.4.1 Choix de la méthode et design pattern 4](#_Toc70930273)

[2 Planification initiale 4](#_Toc70930274)

[3 Conception 5](#_Toc70930275)

[3.1 Maquettes de l’interface 5](#_Toc70930276)

[3.1.1 Menu principal 5](#_Toc70930277)

[3.1.2 Interface de la première expérience : « chocs » 5](#_Toc70930278)

[3.1.3 Interface de la deuxième expérience « balistique » 6](#_Toc70930279)

[3.2 Formules mathématiques pour la gestion des collisions 6](#_Toc70930280)

[3.2.1 Détection et résolution d’une collision entre 2 billes 7](#_Toc70930281)

[4 Réalisation 10](#_Toc70930282)

[4.1 Mise en place de l’environnement pour compiler le code 10](#_Toc70930283)

[5 Tests 10](#_Toc70930284)

[6 Conclusion 10](#_Toc70930285)

[7 Bibliographie 10](#_Toc70930286)

# Spécifications

Merci de vous référer au cahier des charges pour toutes les spécifications concernant le projet.

# Analyse

## Méthodologie de projet

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

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

## Technologies à utiliser

### Langage de programmation

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

### Bibliothèques

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

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

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

## 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.

### 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.

## Méthode de simulation : discrète ou continue

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

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

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

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

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

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

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

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

### Choix de la méthode

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.

### 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[[1]](#footnote-1). 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 inéxacte.

### Analyse détaillée de certaines fonctionnalités

#### 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.

#### 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 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.

#### Trace des billes

<todo> dessin de la trace

## 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.

# Planification initiale

## 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 |
|

### 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.

#### 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.

## 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.

### 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.

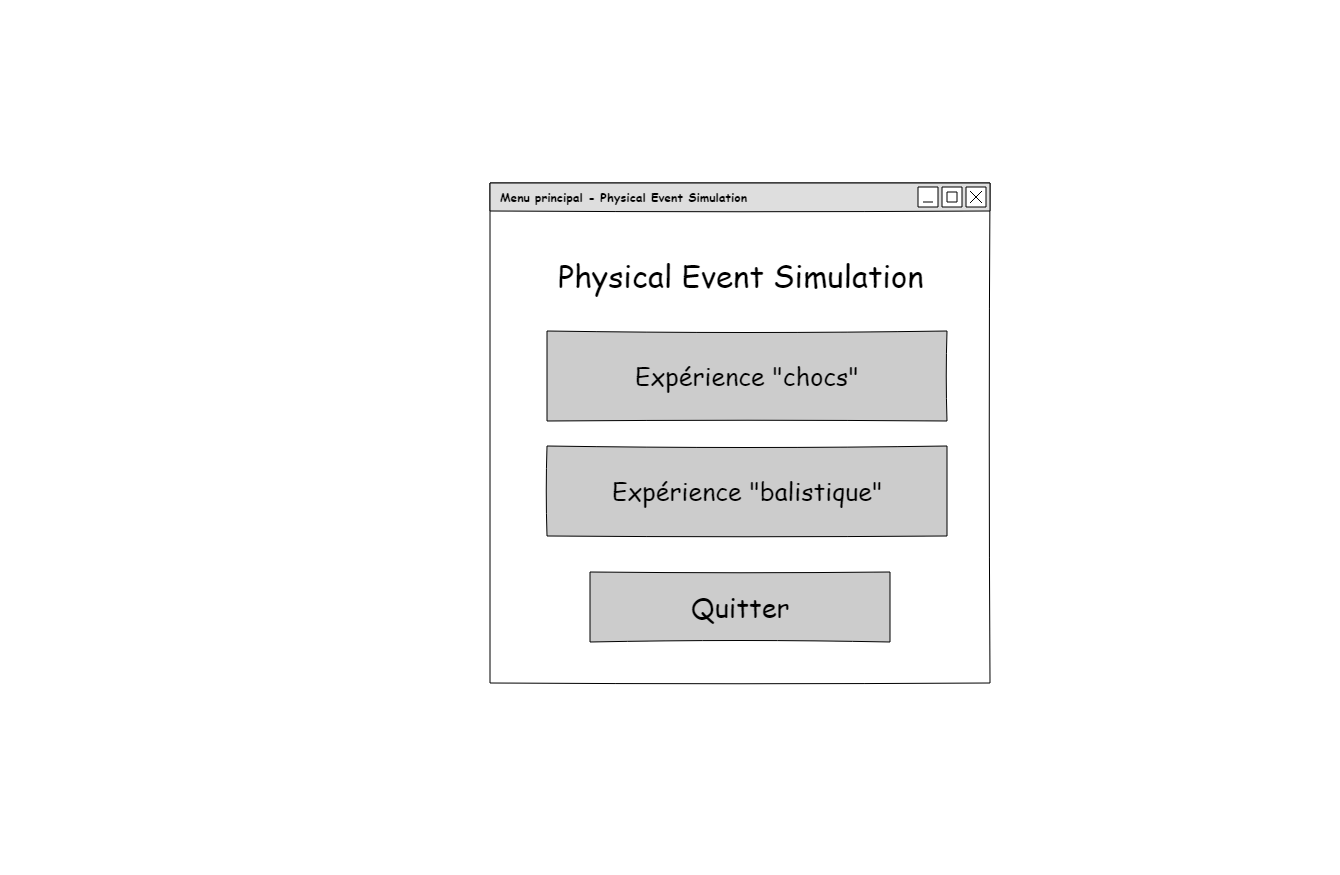


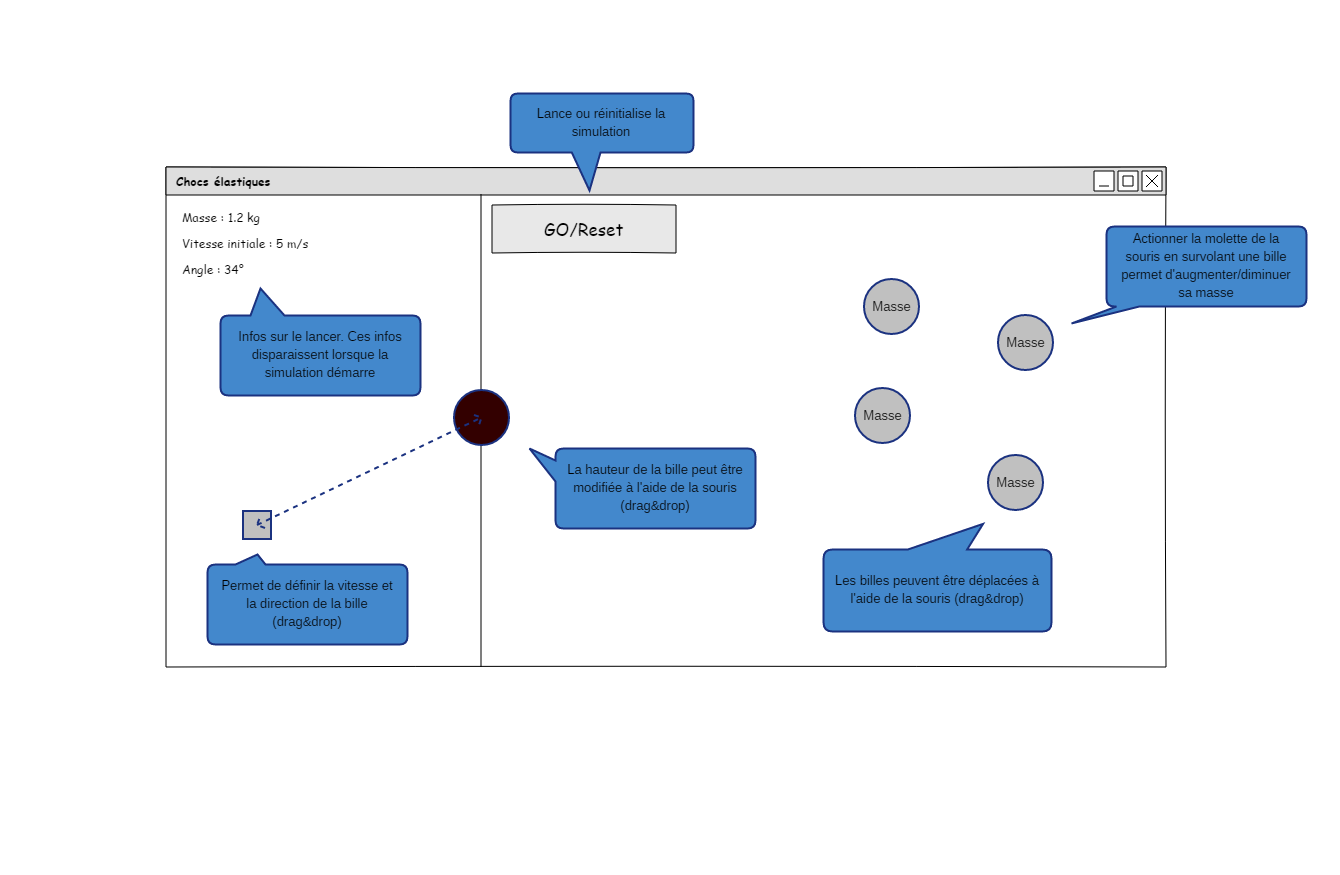
Figure maquette menu

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

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

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

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

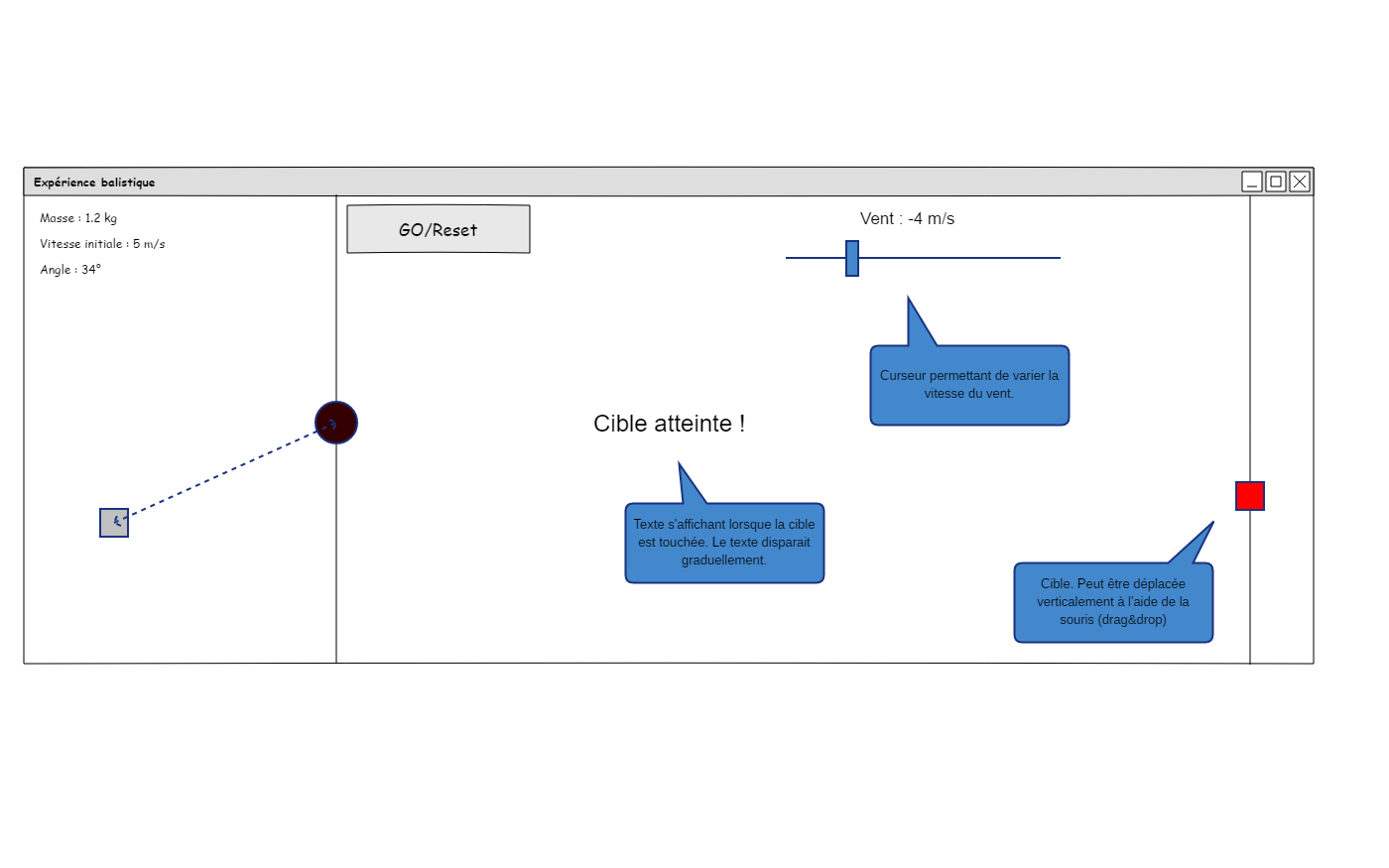


### Interface de la deuxième expérience « balistique »

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

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

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



### 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.

### 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.

## 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.

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

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

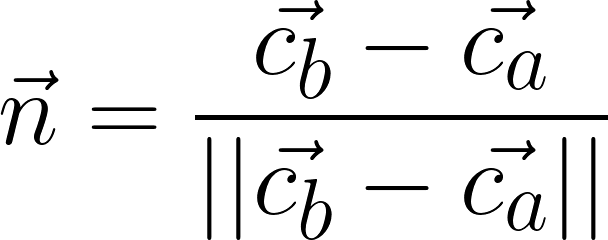
#### Détection de la collision

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

collisions billes

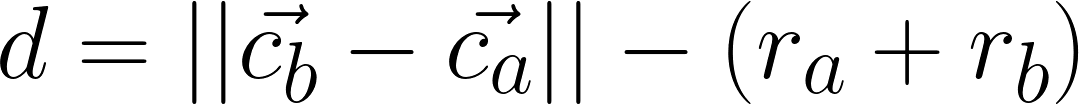
S’il n’y pas de collision, il va de soi que le programme n’ira pas plus loin. En revanche, si une collision est détectée, il va être nécessaire de déterminer deux informations essentielles : La normale de collision et la profondeur de collision.

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 :



On a maintenant une normale de collision pointée en direction du cercle Ca. Elle indique la direction vers laquelle le cercle Ca doit être « poussé » pour résoudre la collision. Pour le cercle Ca, 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 :



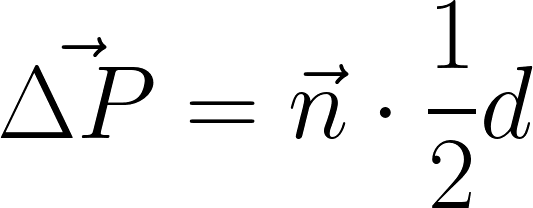
#### Résolution de la collision : déplacement

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.

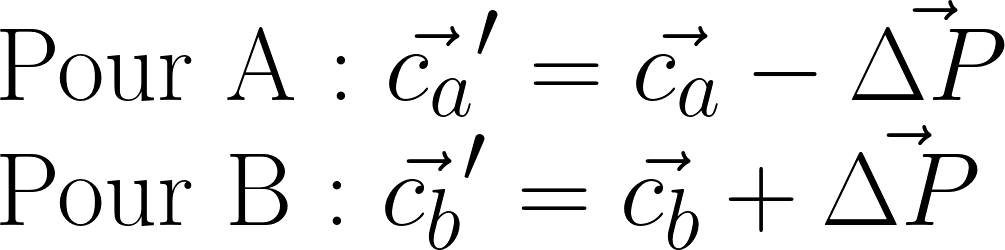
Pour cela, il faut calculer le vecteur de translation qui sera additionné ou soustrait en fonction du cercle concerné. Les propriétés de ce vecteur seront identiques pour les deux cercles, à une différence près : son sens. Pour repositionner le cercle Ca, il faudra soustraire le vecteur de déplacement à sa position alors que pour Cb, il faudra l’additionner. Le calcul de la nouvelle position après un déplacement se fait de la manière suivante :

C:\Users\luccharbonnier\AppData\Local\Microsoft\Windows\INetCache\Content.Word\déplacement.png

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

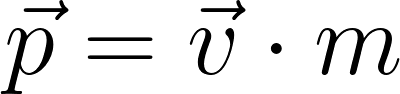


#### Résolution de la collision : impulsion

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.

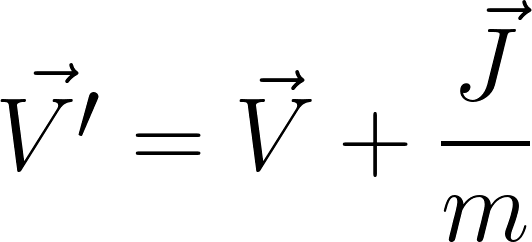
C’est ici que les choses se compliquent un peu. En plus de prendre en compte la normale de collision ainsi que la position des cercles, il va également falloir tenir compte de leurs vélocités et de leurs masses pour pouvoir leur appliquer une impulsion. Pour rappel, la vélocité est un vecteur représentant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



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) :



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 :

Elle montre simplement que l’énergie cinétique totale du système est conservée. Ici, va1, va2, vb1 et vb2 correspondent respectivement aux vitesses de départ et de fin des deux billes. On peut remplacer les vitesses de fin (va2 et vb2) en les exprimant en fonction des vitesses de départ :

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 :

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 :

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 :

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 :

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 :

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 :

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

## Conception des formules de balistique

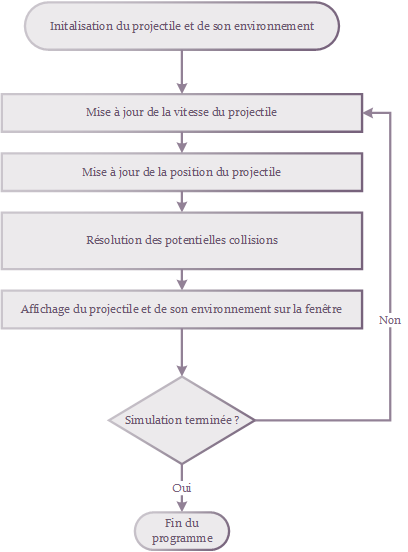
Nous allons voir les équations qui seront utilisées pour calculer la position des billes dans une simulation « discrète » (voir l’analyse pour la différence entre simulation continue et discrète) et, dans un deuxième temps, celles qui nous permettront de prendre en compte la résistance de l’air dans 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 (p0), de sa vélocité (v) et du temps écoulé (**∆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 :

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 :



### 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 Fx est la suivante[[2]](#footnote-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).
* Vrel : 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.

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 :

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 :

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

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) :

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

* Sa vélocité **Vproj** (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 : **Vrelative**
* Son rayon **r**
* Son coefficient de trainée **Cx**

Si on simplifie, cela donne l’équation suivante :

#### 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[[3]](#footnote-3). 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[[4]](#footnote-4) 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.

## Conception du format 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. Pour ce faire, et pour s’assurer de ne rater aucun cas, les tests suivront tous le même format. Le cartouche suivant contient un exemple et sera utilisé comme modèle pour tous les tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

Évidemment, les tests ne seront effectués qu’après la fin du développement.

# Réalisation

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

## 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>.

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.

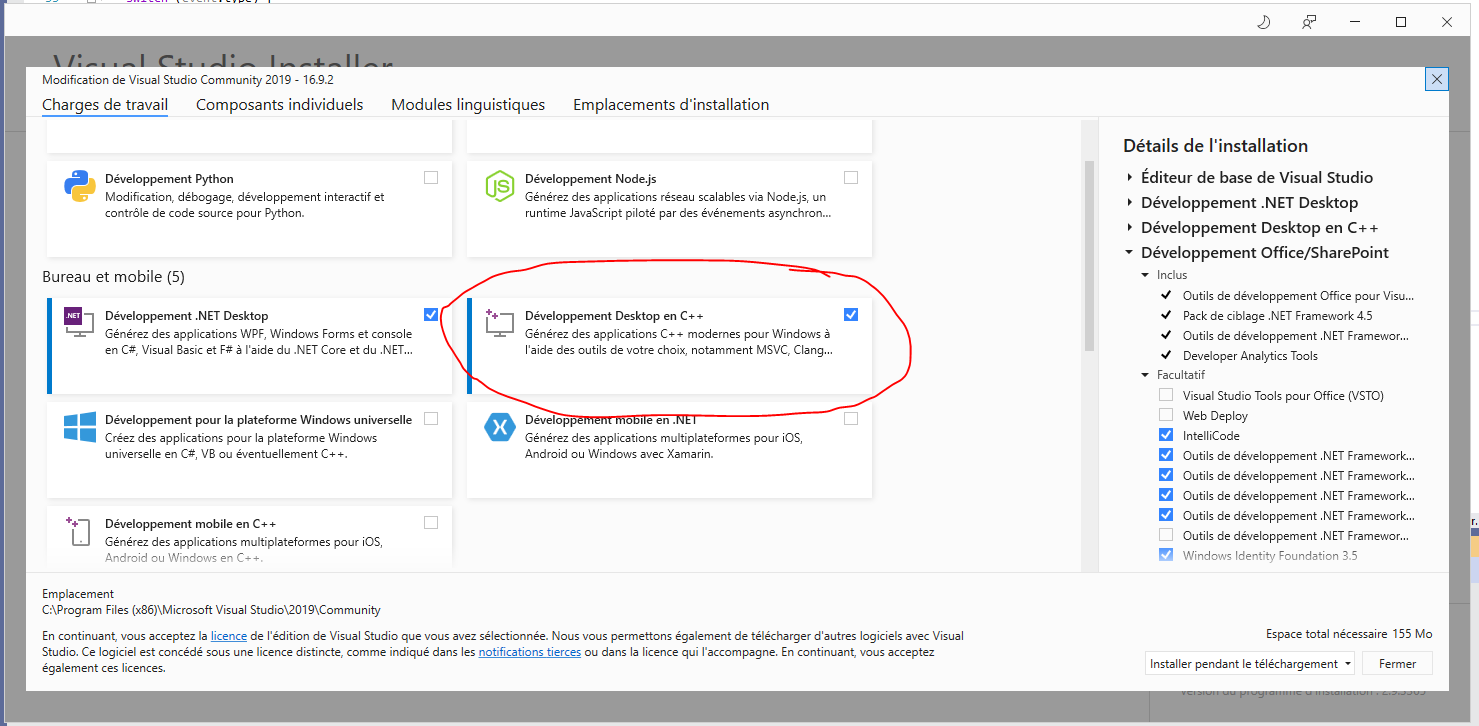
L’utilisation d’un dépôt permet donc de faciliter le suivi du projet et de garantir un « backup », à condition bien sûr de push régulièrement ses modifications.

## Versions des bibliothèques utilisées

Les dernières versions de la SFML (2.5.1) et de ma bibliothèque (Charbrary 2.0.0) seront utilisées.

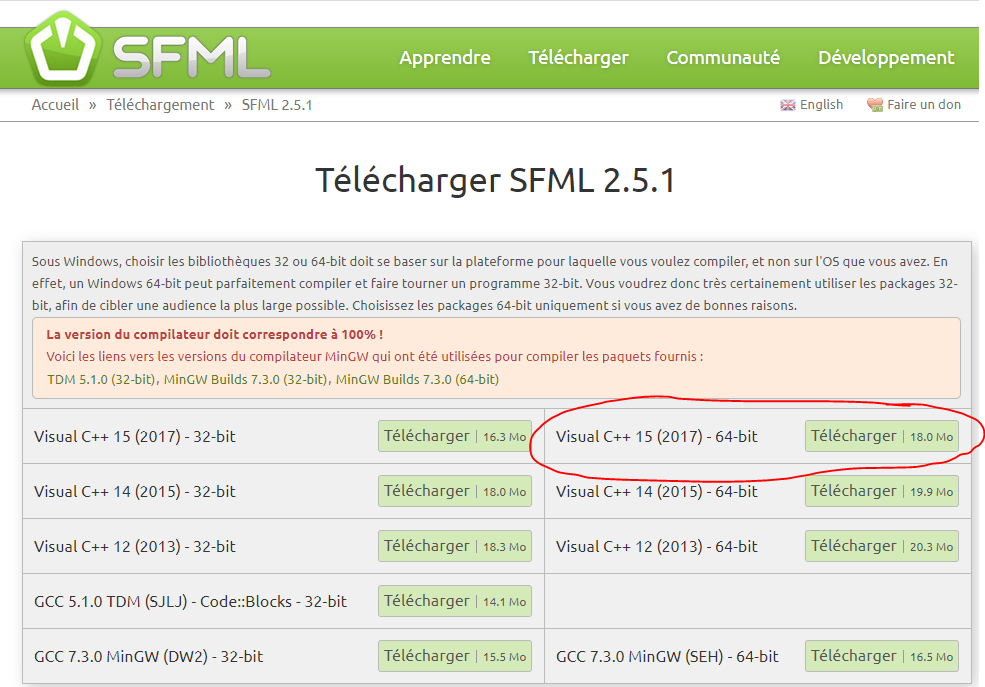
## Mise en place de l’environnement et compilation du projet

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 :



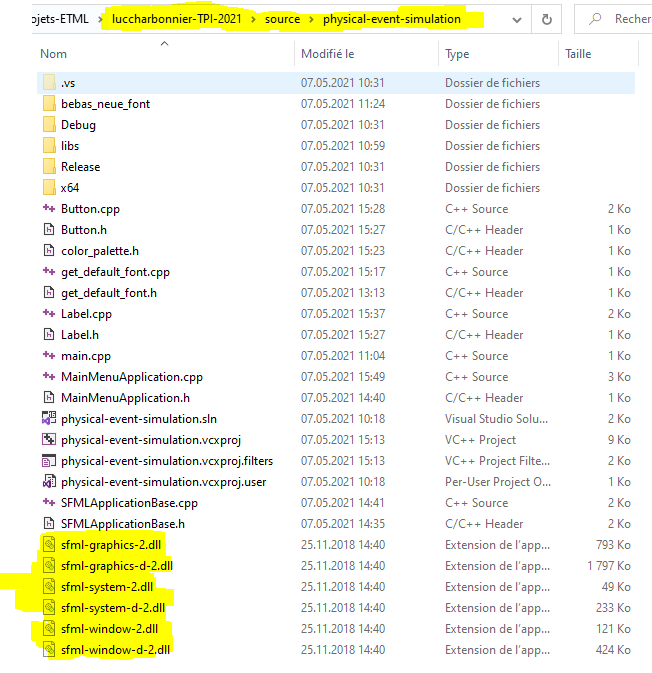
<todo> dossier /libs manquant

Une fois cela fait, la solution pourra être ouverte et le projet pourra, en principe, être compilé. Cependant, l’exécutable résultant ne pourra pas être exécuté car il manque les DLL de la SFML. Ceux-ci ont été omis du dépôt pour éviter de le surcharger mais ils peuvent être obtenus depuis le site officiel de la SFML en téléchargeant la dernière version :



La version de la SFML qui a été utilisée pour le projet est la version 2.5.1 précompilée pour Visual C++ 2017 en 64 bits.

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).

On ne décrira pas ici[[5]](#footnote-5) 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[[6]](#footnote-6), mais sachez que ne devriez pas en avoir besoin pour compiler le projet puisque la SFML est déjà présente dans le dossier du projet.

## 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).

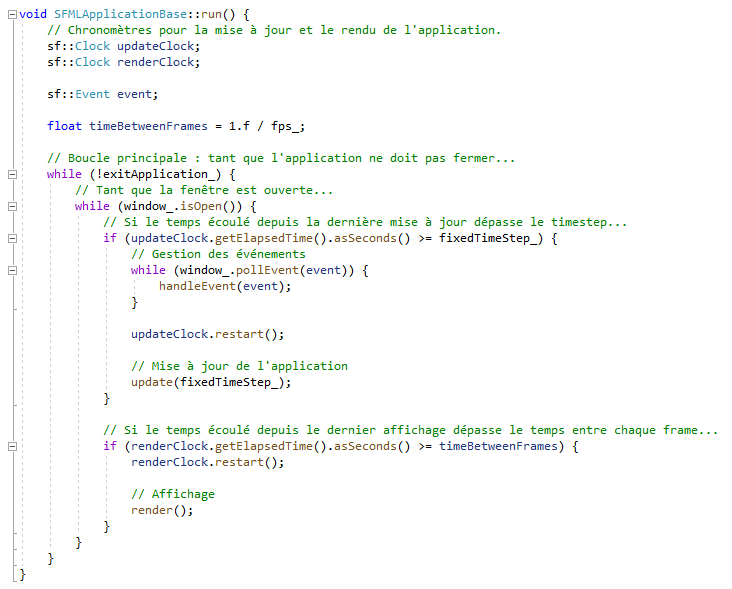
Le paradigme événementiel est très pratique lorsque on développe une application avec une interface graphique comportant beaucoup de widgets (des boutons, des checkboxes, radiobuttons, etc…) car on peut assigner une fonction à chaque bouton et gérer très facilement les différents comportements à implémenter.

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 :



### 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 la simulation soit saccadée, il faut évidemment que temps entre 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.

## Liste des classes et de leur utilité

<todo>

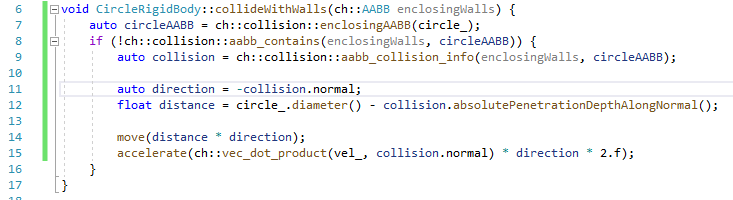
## Gestions des collisions

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.

### 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 :



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.

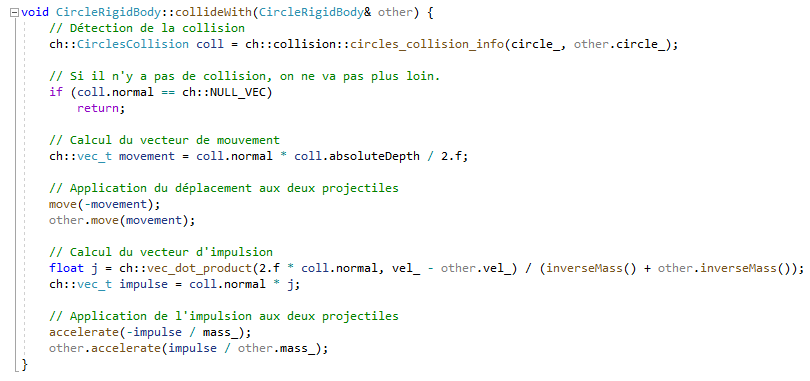
Cependant, cette fonction est originellement destinée à résoudre des collisions entre deux rectangles de manière à ce qu’ils ne soient plus en collision. Ici, on veut exactement l’inverse : on cherche à faire en sorte que la deuxième AABB soit toujours dans la première. C’est pour cette raison qu’à la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

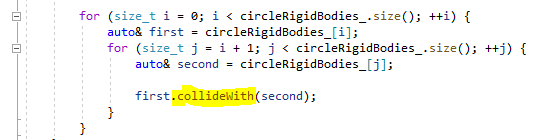
Une fois la bille repositionnée, il faut également lui appliquer une accélération. Le but ici est d’inverser la composante du vecteur de vitesse qui est parallèle à la normale de collision. Donc, si la bille touche le haut ou le bas de la fenêtre, on inversera la composante Y. Pour ce faire, on pourrait 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).

### 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 :



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



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).

### Résolution quand vitesses opposées

<todo>

## Affichage de la trace des objets

<todo>

## Différences entre les maquettes et l’interface finale

### Cible remplacée par un cercle au lieu d’un carré

<todo>

### Choix de la vitesse et de la direction du vent

<todo> windPicker

## Détail de certaines fonctionnalités

<todo>

### 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 :



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

Dans un deuxième temps, une échelle a été mise en place

<todo>

## Bugs ou améliorations possibles

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

<todo>

# Tests

<todo : à continuer>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fonctionnalité testée | Déroulement du test | Résultat attendu | Résultat obtenu | Validation |
| 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 lorsque on la ferme. |  |  |
| Démarrage/arrêt d’une des deux simulations | Ouvrir la simulation de chocs, la démarrer, 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 |  |  |
| 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. |  |  |
| 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. |  |  |
| 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. |  |  |
| 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. |  |  |
| 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. |  |  |
| 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 orienté contre le projectile. | La trajectoire du projectile effectuera un retour en arrière. |  |  |
| Collision entre le projectile et la cible |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# Conclusion

## Améliorations possibles

* Bloquer redimensionnement fenêtre
* Trace fixe pour balistique
* Visualisation trace après le tir

<todo>

## Autoévaluation

<todo>

# Bibliographie

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

1. Entre guillemets car l’énergie cinétique n’est jamais vraiment perdue, elle est simplement convertie sous une autre forme : chaleur, déformation, son, etc… [↑](#footnote-ref-1)
2. <https://fr.wikipedia.org/wiki/Tra%C3%AEn%C3%A9e> [↑](#footnote-ref-2)
3. <https://fr.wikipedia.org/wiki/Coefficient_de_tra%C3%AEn%C3%A9e> [↑](#footnote-ref-3)
4. <https://fr.wikipedia.org/wiki/Masse_volumique_de_l%27air> [↑](#footnote-ref-4)
5. Par soucis de temps et, surtout, pour éviter de réexpliquer inutilement un processus déjà détaillé par de nombreuses ressources sur Internet. [↑](#footnote-ref-5)
6. <https://www.sfml-dev.org/tutorials/2.5/start-vc-fr.php> [↑](#footnote-ref-6)