Simulation physique

L'objectif de ce TP est d'appliquer les notions de mécanique du point vues dans votre cours de physique pour réaliser une simulation numérique (à peu près) réaliste d'une balle. Dans un premier temps, vous devrez découvrir une nouvelle notion de C avant de l'appliquer au TP. La deuxième partie du TP vous permettra de découvrir les principes de base de la simulation temps réel à l'aide de la SDL.

Afin de vous aider à mener à bien cette tâche, des précisions théoriques sont formulées au début de chaque section. Le détails des instructions à réaliser sont systématiquement données dans une section précédée du mot-clé *Todo* écrit en orange.

1 Nouvelle notion de C : les structures

Une *structure* est un regroupement de plusieurs variables sous un seul nouveau type de données. Par exemple, supposons que l'on veuille définir un point de $\mathbb{R}^2$ en langage C. Un tel élément est composé de deux réels que l'on peut noter x et y et regrouper sous un terme générique que l'on appellerait `Point`. Ainsi un `Point` A peut être défini par un couple de réels, par exemple $A = (x, y) = (7, 3)$. Notre nouveau **type de donnée** s'appellerait alors `Point` et contiendrait deux éléments `x` et `y` chacun de type `float`. On pourrait alors définir un nouveau point nommé `A` à l'aide de l'instruction `Point A;`.[1]

1.1 Définition du nouveau type

La définition d'une structure est réalisée de façon générique, **dans un fichier header**, à l'aide des lignes de code suivantes.

```
struct _TypeStruct          // définition du type de la structure
{
 type1 nom1;                // type et nom du premier champ
 type2 nom2;                // type et nom du second champ
 ...
};          // on n'oublie pas le point-virgule après l'accolade
typedef struct _TypeStruct TypeStruct;    // renommage de type
```

Ces lignes ne contiennent que des définitions pour un nouveau type de données. Elles ne réalisent aucune modification de la mémoire. C'est la raison pour laquelle elles sont situées dans un fichier .h. Vous remplacerez bien évidemment le mot `TypeStruct` par le nom de votre nouveau type structuré, et les noms `type1`, `type2`, ... par les types de données à regrouper au sein de la structure.

La dernière ligne réalisant le renommage de type reste pour le moment non expliquée. Elle permet de simplifier la déclaration d'une nouvelle variable de type structurée.

1. Tout serait alors beau et merveilleux et nous pourrions enfin vivre dans un monde de Bisounours.

Exemple 1. Revenons maintenant sur notre exemple de point du plan. La définition du nouveau type de donnée se fait de la façon suivante (toujours dans un fichier .h) :

```
struct _Point   // Définition du nouveau type de donnée Point
{
  float x;
  float y;
};
typedef struct _Point Point;
```

1.2 Utilisation d'une variable de type structuré

La déclaration d'une nouvelle variable de type structuré est assez classique. On écrit en premier le nom du type puis le nom de la variable. On peut aussi initialiser ses champs de façon similaire à celle d'un tableau.

```
// Déclaration d'une nouvelle variable
TypeStruct var1;
// Déclaration et initialisation des deux premiers champs
TypeStruct var2 = {val1, val2};
```

Exemple 2. Revenons alors à notre exemple de `Point`. La déclaration d'une nouvelle variable de type `Point` est réalisée dans le .c telle que décrite précédemment :

```
Point A;           // Déclaration d'une nouvelle variable
Point B, C;        // Déclaration de plusieurs variables
Point P = {1,-2};  // Initialisation
```

Il ne reste plus qu'à modifier ce genre de variable. Étant donné qu'une variable de type structuré contient plusieurs champs, il est obligatoire de préciser le nom de la variable **ET** le nom du champ à modifier. On utilise pour cela le séparateur « `.` » de la façon suivante :

```
TypeStruct var1;    // Déclaration
var1.nom1 = val1;   // Modification de valeur
var1.nom2 = val2;
...
```

Exemple 3. En poursuivant l'exemple du `Point`, on modifie les points `A`, `B` et `C` déclarés précédemment :

```
A.x = -3;      // Modification du champ x du Point A
A.y = 13.37f; // Modification du champ y du Point A
B.y = A.x;     // Copie du champ x de A dans le champ y de B
C = P;         // Copie du point P dans le point C
```

1.3 Structures et pointeurs

Il existe principalement deux cas de figure pour la manipulation de pointeurs et de structures. Dans le premier cas, un champ de la structure est un pointeur. Par exemple, un des champs est une chaine de caractères de type `char *`. Cela ne change rien à l'utilisation de ce champ au travers de la variable : on utilise toujours le séparateur « . » pour accéder au champ concerné.

Le deuxième cas est plus délicat : la variable de type structuré est en fait un pointeur. En effet, il est tout à fait possible de définir un pointeur de type structuré, comme pour un type standard. Dans ce cas, on accède à un champ à l'aide du couple de symboles « `->` » (tiret et supérieur).

Exemple 4. On termine l'exemple du `Point` avec l'extrait de code suivant.

```
Point A = {0,0};   // Déclaration du Point A
Point B = {7,3};   // Déclaration du Point B
Point *pA = &A;    // Déclaration du pointeur pA
A.x = 2;           // Modification du champ x du Point A
pA->y = 4;         // Modification du champ y de A via le pointeur pA
B.x = pA->x;       // B est une variable et pA est un pointeur
pA->y = B.y;       // Modification du champ y de A via le pointeur pA
```

Todo. Vous possédez maintenant toutes les bases concernant la manipulation des structures. N'hésitez pas à jouer avec les différentes notions : définition de la structure, déclaration d'une variable, accès à un champ et pointeurs. Modifiez pour cela les codes qui sont fournis dans ce sujet afin de vous approprier complètement ces nouvelles notions.

2 Introduction à la modélisation physique

L'objectif de cette section est de développer un ensemble de primitives permettant de faire évoluer un solide soumis à la gravité terrestre et aux frottements de l'air. La trajectoire du solide sera calculée à l'aide des équations de physique que vous avez pu découvrir au cours du premier semestre. Le données du vol pourront être écrites dans le terminal puis visualisées à l'aide d'un tableur. Vous pourrez vérifier l'exactitude de votre code sous Moodle.

Pour réaliser cela, nous détaillons à nouveau les notions de mécanique du point appliquées à la chute d'un solide. Tout vecteur est noté avec une flèche au-dessus [2] ; par exemple l'accélération du solide est notée $\vec{a}$. Les scalaires réels ne comportent aucun signe distinctif : la masse du solide est par exemple notée m.

2.1 Notions théoriques

Notre objectif est de faire évoluer un solide dans le temps, en connaissant uniquement de façon théorique les forces qui lui sont appliquées. Le principe fondamental de la dynamique (ou 2ème loi de Newton) permet de relier l'accélération $\vec{a}$ d'un solide, sa masse m

2. N'en prenez pas trop l'habitude quand même ! Il va y a voir du changement au semestre 2 en maths !

et l'ensemble des forces $\sum \vec{F}$ qui lui sont appliquées :

$$\sum \vec{F} = m \times \vec{a}$$

Cette relation peut également s'écrire :

$$\vec{a} = \frac{1}{m} \sum \vec{F} \tag{1}$$

De l'accélération, on peut déduire la position $\vec{X}$ et la vitesse $\vec{v}$ car ces trois valeurs sont reliées par les équations suivantes :

$$\vec{a}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\vec{v}(t) \quad \text{et} \quad \vec{v}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\vec{X}(t)\,.$$

Ainsi, on trouve $\vec{X}(t) = \int \vec{v}(t)\mathrm{d}t$ et $\vec{v}(t) = \int \vec{a}(t)\mathrm{d}t$.

Le calcul de ces intégrales est possible, mais n'est intéressant qu'en physique théorique. Ici, on pratique ! Nous voulons tracer l'évolution du solide au cours du temps, et nous avons donc besoin de connaitre sa position à chaque instant de son évolution, pas seulement à la fin. Pour l'implémentation, nous utiliserons donc la méthode d'intégration d'Euler qui permet d'approcher le résultat d'un calcul intégral. Elle permet de déduire les vecteurs vitesse $\vec{v}(t)$ et position $\vec{X}(t)$ à partir de l'accélération :

$$\vec{v}(t + \mathrm{d}t) = \vec{v}(t) + \vec{a}(t)\mathrm{d}t\,, \tag{2}$$

$$\vec{X}(t + \mathrm{d}t) = \vec{X}(t) + \vec{v}(t)\mathrm{d}t\,. \tag{3}$$

C'est quoi dt ? Parce que la physique manque parfois de convivialité, on passe à côté de choses relativement simples à comprendre. Afin de mieux cerner d'où sont issues les formules précédentes, reprenons quelques bases. L'unité internationale permettant de décrire une distance est le mètre et l'unité de temps est la seconde. Ainsi, naturellement, la vitesse est déterminée en mètres par seconde, noté m/s.

Une accélération peut se définir comme un gain ou une perte de vitesse au cours du temps. Passer de $10m/s$ à $15m/s$ en une seconde correspond donc à une accélération de $+5m/s$ en une seconde, que l'on écrit $+5m/s^2$. Supposons maintenant que notre accélération soit constante à $+5m/s^2$ et que notre vitesse actuelle est de $15m/s$. On cherche à savoir quelle vitesse on aura atteint dans $0,1s$. Bien évidemment, le calcul est simple : $V = 15m/s + 15m/s \times 0,1s = 16,5m/s$. La valeur $0,1s$ correspond au fameux $\mathrm{d}t$.

Forces modélisées Pour cette première partie, nous ne prendrons en compte que deux forces.

- Le poids $\vec{P} = m \cdot \vec{g}$, avec $\vec{g} = \begin{pmatrix} 0 \\ -9.81 \end{pmatrix}$.
- Les forces de frottements dans l'air $\vec{f} = -\alpha \cdot \vec{v}$, le coefficient α étant le coefficient de friction.

2.2 Implémentation

Pour effectuer notre simulation, nous allons manipuler des vecteurs à deux dimensions car tous nos calculs seront effectués dans le plan à deux dimensions. Nous utiliserons donc le principe des structures pour les implémenter.

```
typedef struct Vec2_s
{
  float x;
  float y;
} Vec2;
```

Un vecteur sera désigné par une variable de type `Vec2` contenant deux champs : les composantes `x` et `y` du vecteur. Vous retrouverez la définition de ce vecteur dans le fichier Vector.h fourni sur Moodle. Votre première tâche est donc d'implémenter une librairie de manipulation de vecteurs. Vous pouvez utiliser le basecode fourni afin de lire la documentation liée aux fonctions que vous devez coder (fichiers Utils/Vector.c et .h) Vous disposez de plusieurs tests Moodle afin de valider vos fonctions avant de les intégrer au basecode de façon définitive.

Todo (Moodle). Complétez les fonctions données dans le test Moodle *Librairie Vector* afin de développer votre librairie de gestion des vecteurs. Vous pouvez créer en parallèle une paire de fichiers vector.c et vector.h ainsi qu'un fichier main.c afin de développer en local votre librairie.

Il est maintenant temps de faire bouger notre solide. Plutôt que d'utiliser le terme de *solide*, nous utiliserons le terme *balle*[3] qui peut être relativement bien assimilé à un solide. Notre balle va alors être également représentée grâce à une structure.

```
struct Ball_s
{
  Vec2 position;
  Vec2 velocity;
  float mass;
  float friction;
};
```

Todo. Dans votre `main`, déclarez une balle et initialisez-la aux valeurs que vous souhaitez. Pour avoir un comportement relativement réaliste par la suite, nous vous conseillons la valeur de 1 pour le coefficient de friction et 0.5 pour la masse. Utilisez les fonctions de manipulation de vecteurs précédemment codées pour initialiser votre balle.

Nous avons à présent tout ce qu'il nous faut pour calculer les positions successives de notre balle. Rappelons que le principe est de calculer les forces appliquées à la balle à un instant. La suite consiste à utiliser les équations (2) et (3) afin de calculer l'incrément de vitesse puis de position que la balle va gagner (ou perdre) en fixant un pas d'intégration. Plus ce pas sera petit, plus le nombre de positions intermédiaires données par la simulation sera grand, et plus notre simulation sera précise[4]. Dans notre cas, 10 ms (0.01 s) donneront

3. Pourquoi une balle et pas un ibijau ? Pour vous surprendre.
4. Il faut toutefois veiller à ne pas prendre un pas trop petit non plus sous peine de rencontrer des

un résultat satisfaisant pour ce premier programme.

Pour chaque itération vous devez réaliser les calculs suivants :

- calcul des forces appliquées : poids et forces de frottements ;
- calcul de l'accélération courante grâce à l'équation (1) ;
- mise à jour de la vitesse de la balle grâce à l'équation (2) ;
- mise à jour de la position de la balle grâce à l'équation (3).

La nouvelle position de la balle pourra alors être affichée dans la console après l'appel à la fonction. A nouveau, vous pouvez utiliser le basecode fourni afin de lire la documentation liée aux fonctions demandées. Un test Moodle à compléter en plusieurs fois est présent pour vous aider à les développer correctement.

> **Todo** (Moodle). Complétez les fonctions `Ball_UpdateVelocity_V1()` et `Ball_UpdatePosition()` du second test donné sur Moodle. La première fonction réalise les trois premières étapes de l'algorithme présenté précédemment : calcul des forces, de l'accélération et de la vitesse. La seconde fonction se charge de mettre à jour la position de la balle passée en paramètres. Elle doit également prendre en compte le rebond de la balle sur le sol.

Vous pouvez également vérifier vos calculs en affichant les positions successives de la balle dans le terminal. Vous pouvez alors copier les informations de votre terminal dans un nouveau fichier CSV puis tracer la trajectoire de votre balle dans tableur. Une balle envoyée à 5 m/s à l'horizontale à une hauteur de 1 mètre n'atteint pas le sol en abscisse 0. De même, elle ne peut pas non plus être à plus de 5 mètres de son point de départ après une seconde.

N'hésitez pas à modifier les valeurs initiales de votre balle. Masse, coefficient de friction, position et vitesse initiales sont autant de paramètres à faire varier pour vérifier graphiquement l'exactitude de votre code.

2.3 Liaison de deux balles par un ressort

Votre nouvel objectif est de simuler la force de rappel d'un ressort reliant deux balles. Commençons avec une approche simple dans laquelle une balle est reliée à un point fixe. La force de rappel s'exprime avec la formule suivante :

$$\vec{F} = k(l - l_0)\,\vec{I}$$

avec k la raideur du ressort (N/m), l l'allongement du ressort (m), l_0 la longueur du ressort au repos et $\vec{I}$ la direction vers le point d'attache du ressort (vecteur normalisé).

> **Todo** (Moodle). Complétez la fonction `Ball_UpdateVelocity_V2()` du test Moodle afin de simuler cette force de rappel. Votre ressort doit être fixé en position $(0.5, 0.5)$ et la fonction doit mettre à jour la vitesse de la balle passée en paramètres, cette fois-ci en prenant en compte la force de rappel du ressort en plus de la gravité et des frottements.

Il est temps de relier des balles. Commençons par observer plus en détail le basecode fourni. La définition de la structure `Ball` intègre maintenant le nombre de ressorts reliés à elle ainsi que leurs caractéristiques. Ces dernières sont donnée par une nouvelle structure

erreurs de précision dans le calcul flottant.

de type `Spring` comprenant un pointeur vers l'autre balle reliée au ressort ainsi que sa longueur au repos. Inutile de stocker la longueur actuelle du ressort puisqu'elle est recalculée à chaque mise à jour du moteur physique.

Afin de relier deux balles entre elles, la fonction `Ball_Connect()` doit être utilisée. L'appel à cette fonction met à jour automatiquement les différents champs liés au ressort dans les deux balles. Notez alors que la force de rappel s'exerce sur les deux balles. Si le ressort est détendu, les deux balles s'attirent, sinon, elles se repoussent.

> **Todo** (Moodle). Complétez la fonction `Ball_UpdateVelocity_V3()` du test Moodle afin de simuler cette force de rappel entre deux balles. L'objectif est cette fois de calculer l'ensemble des forces de rappel auxquelles la balle est soumise. Une nouvelle fois, la force de gravité et les frottements sont à prendre en compte.

3 Simple Physics Engine

Il est maintenant temps de visualiser tout cela en temps réel. Nous utiliserons pour cela la librairie SDL qui nous permettra d'ouvrir une nouvelle fenêtre dans laquelle nous pourrons afficher des éléments.

3.1 Introduction basecode

Un code de départ vous est fourni pour cette partie afin que vous n'ayez pas à vous préoccuper des aspects liés à la création d'une fenêtre et à l'affichage dans cette dernière. Le basecode fourni nécessite l'installation de la SDL2 pour son exécution correcte. Les commandes suivantes vous permettront d'installer les librairies nécessaires à la réalisation de ce TP. Elles sont données pour un système Ubuntu ou Mint, adaptez-les si besoin.

```
sudo apt update
sudo apt upgrade
sudo apt install libsdl2-dev libsdl2-image-dev
```

La compilation du projet est réalisée à l'aide d'un Makefile. Il s'agit d'un fichier permettant d'exécuter l'ensemble des compilations nécessaires à la création de l'exécutable final, nommé spe.bin (ou spe_d.bin pour la version debug). Pour compiler votre projet, il suffit donc de lancer l'une des deux commandes suivantes.

```
make
make debug
```

La première compile le projet de façon optimisée, la seconde affiche tous les avertissements possibles et utilise des options de debug pour faciliter l'usage d'un débogueur. Tant que vous n'avez pas fini de développer une fonction, utilisez `make debug`. Une fois que votre code est propre, utilisez `make`. Le lancement de l'exécutable produit provoquera l'ouverture d'une nouvelle fenêtre avec des graphismes figés. Il s'agit pour le moment d'un fonctionnement normal, la tâche suivante consiste à afficher les balles en mouvement dans cette fenêtre.

> **Todo.** Récupérez le basecode du projet, compilez-le et vérifiez que le programme se lance correctement. Si ce n'est pas le cas, demandez de l'aide rapidement à votre enseignant.

La fenêtre que vous avez vu apparaitre est au format 16/9 et représente un *monde* ayant pour largeur 19 mètres. Le point $(0, 0)$ est situé dans le coin inférieur gauche de la fenêtre. Ce monde est affiché dans une fenêtre qui est appelée *vue*. Dans le monde, les unités sont en mètres. Dans la vue, elles sont en pixels. Il existe donc dans le basecode des fonctions permettant de passer de l'un à l'autre. Vous prendrez garde dans la suite du projet aux unités que vous manipulerez dans les différentes fonctions.

3.2 Implémentation du PFD temps réel

Observons le basecode de plus près. On y retrouve de façon classique un fichier main.c accompagné d'un couple de fichiers Settings.c/.h qu'il ne faut pas modifier. Le dossier Utils/ contient toutes les fonctions permettant d'initialiser et utiliser la SDL pour afficher des éléments à l'écran. Le dossier Game contient quant à lui des fonctions inhérentes au jeu en lui même. Vous pouvez remarquer qu'il existe majoritairement des couples de fonction .c/.h. Sans rentrer dans les détails, un grand nombre de fonctions ne seront pas à modifier. Aussi, seuls les fichiers suivants seront à modifier :

- Utils/Vector.c contenant vos primitives de manipulation de vecteurs.
- Game/Ball.c contenant les fonctions de calcul physique relatives aux balles.
- Game/Scene.c, à modifier en fin de TP afin de gérer plusieurs balles.

L'utilisation de la SDL se fait en trois temps. Une fois que tous les éléments relatifs au chargement de la fenêtre sont réalisés, une *boucle de rendu* permet d'appeler trois types de fonction :

- la gestion du temps ;
- la mise à jour du moteur physique avec la récupération des évènements utilisateurs et la mise à jour des objets du jeu ;
- l'affichage des différents éléments.

Dans le basecode fourni, tout est déjà réalisé. Autrement dit, vous ne devez vous soucier que de la partie « implémentation de la physique ».

> **Todo.** Si vous ne l'avez pas déjà fait, complétez les fonctions présentes dans les fichiers Game/Ball.c et Utils/Vector.c à l'aide des codes que vous avez réalisé sur Moodle. Vérifiez que votre projet compile correctement. Si tout se passe bien, vous devriez voir trois balles à l'écran, reliées entre elles par des ressorts.

3.3 World of Goo

Nous vous proposons maintenant de recréer la physique cachée dans le jeu *World of Goo*. Jetez un œil rapide sur Internet pour comprendre l'objectif et avoir un aperçu du jeu. Vous pouvez également lancer la démo qui vous est fournie pour voir quelques possibilités qui s'offrent à vous.

Vous avez donc remarqué qu'en approchant la souris de certaines balles, un lien se traçait entre le pointeur de la souris et les quelques balles les plus proches. Votre objectif est de reproduire ce comportement à partir de la librairie qui vous est fournie.

En observant le `main()`, vous pouvez voir que la mise à jour des éléments du jeu se fait dans la fonction `Scene_Update()`. Un coup d'œil à cette fonction nous montre alors que toute la mise à jour du jeu est réalisée dans la fonction `Scene_UpdateGame()`. Un

joli `TODO` se présente alors à vous afin d'ajouter des balles lorsque l'utilisateur utilise le bouton gauche de sa souris.

La récupération de la position du pointeur de la souris vous permettra alors de rechercher les balles les plus proches de ce pointeur. Pour cela, vous complèterez les fonctions `Scene_getNearestBall()` et `Scene_getNearestBalls()`[5] disponibles dans le fichier Game/Scene.c. Voyez la documentation des fonctions pour le détail des prototypes.

Todo. Complétez la fonction `Scene_GetNearestBall()` du fichier Game/Scene.c. Vous pouvez utiliser la fonction `printf()` afin de vérifier si votre calcul est correct.

Maintenant que la balle la plus proche est clairement identifiée, il faut offrir la possibilité au joueur de placer une nouvelle balle à l'écran. L'ajout d'une balle est similaire aux premiers ajouts de balles qui ont été réalisés : vous les trouverez dans le bas de la fonction `Scene_New()`. N'oubliez pas de connecter la nouvelle balle à la balle la plus proche en utilisant la fonction `Ball_Connect()`.

Todo. Complétez la fonction `Scene_UpdateGame()` afin de relier la nouvelle balle à la balle la plus proche. Cette action doit se produire lorsqu'un clic souris est détecté. Vous pouvez maintenant compléter la fonction `Scene_GetNearestBalls()` du fichier Game/Scene.c afin de détecter les n balles les plus proches. Attention, cette fonction demande un peu de réflexion !

Vous venez de terminer la partie obligatoire qui vous assurera une note minimale de 10/20 si toutes les fonctions sont bien codées. Vérifiez que votre code est parfaitement stable en l'exécutant sur votre machine puis sur celle de votre binôme. La suite du TP consiste à ajouter des bonus à votre programme pour en faire un vrai jeu.

3.4 Expression libre

Malgré l'aspect simpliste de ce programme, il existe un grand nombre d'améliorations à apporter pour approcher au plus près un vrai jeu vidéo. Citons entre autres le fait de pouvoir maintenir une balle et la déplacer avec la souris, modifier la force des ressorts ou le nombre de plus proches voisins en fonction d'un certain score, briser des liens entre les balles si les forces appliquées sont trop grandes, ajouter d'autres balles avec des propriétés différentes etc. Les possibilités sont immenses, tout comme vos capacités !

4 Modalités de rendu du TP

Les modalités de rendu sont les suivantes, le non-respect entrainera la perte de points voire 0 :

- Lorsqu'un prototype de fonction, une définition de structure ou un format de fichier vous est précisé, vous NE devez PAS le modifier.
- La deadline est fixée au **vendredi 7 janvier 2022 à 18h**.
- Le TP se fera par 2 étudiants.
- Le mode de rendu se fera sur Moodle.

5. Non, il n'y a pas de faute de frappe, il y a bien un 's' à la fin du nom de la seconde fonction.

- Le TP sera remis sous forme d'archive .zip correctement formée. Le nom de l'archive sera formé par le nom en majuscule de l'étudiant et de son binôme éventuel séparé par le caractère underscore ('_'). Exemple : NORRIS.zip ou NORRIS_VANDAMME.zip. Elle devra créer lors de l'extraction un dossier du même nom (sans l'extension .zip bien entendu).
- Le code source devra être commenté et prêt à compiler sous LINUX. Éventuellement un fichier *readme.txt* pourra accompagner de dossier pour décrire l'usage du programme.