TP C# 9 : EPITA Space Program

Consignes de rendu

A la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
rendu-tp-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- EpitaSpaceProgram/
        |-- EpitaSpaceProgram.sln
        |-- EpitaSpaceProgram/
            |-- Everything except bin/ and obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

- Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
- Les fichiers `AUTHORS` et `README` sont obligatoires.
- Pas de dossiers `bin` ou `obj` dans le projet.
- Respectez scrupuleusement les prototypes demandés.
- Retirez tous les tests de votre code.
- **Le code doit compiler !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
*␣prenom.nom$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* prenom.nom" > AUTHORS
```

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

Table des matières

Table des figures

1 Introduction

1.1 Avant-propos

Oh jours glorieux ! Le moment que vous avez attendu toute votre vie est finalement arrivé. Enfin, les compétences que vous avez pu acquérir en physique ces 10 dernières années vont porter leurs fruits. Ne vous en faites plus, car vos efforts n'auront pas été en vain !

Votre voyage vous mènera jusqu'au pays magique de la cinématique et de la dynamique, en employant des merveilles telles que les oscillateurs amortis et la loi de la gravitation universelle de Newton.

Mais le plus important est que nous vous offrons enfin un moyen d'exprimer votre créativité ! Les lois de la physique sont entre vos mains. Des systèmes galactiques entiers plieront sous votre volonté !

1.2 Objectifs

Durant ce TP un peu particulier, vous ferez un usage intensif de la programmation orientée objet (POO), une notion que vous avez découverte dans les TPs précédents. Vous aurez l'opportunité d'implémenter des chaînes d'héritage en profondeur, ce qui, nous l'espérons, développera chez vous une intuition de l'utilité de l'héritage, et de ses limites.

Ce TP vous enseignera également des notions avancées de POO en C# comme les *interfaces* et la *surcharge d'opérateurs*.

Enfin, pour la première fois, vous allez pouvoir visualiser l'exécution de votre code en *temps réel*, grâce à l'outil de visualisation que nous vous fournissons.

2 Cours

2.1 Interfaces

Vous avez déjà entendu parler un peu d'héritage, de classes et de classes abstraites lors des TPs précédents. Pour rappel, vous pouvez consulter le **TP C# 6 : WestWorld Tycoon** et la documentation officielle C# .

Une interface est une manière de définir le prototype des méthodes qu'une classe doit *implémenter*, sans les implémenter directement. L'idée de l'interface est de partager un comportement de manière *horizontale* entre des classes n'ayant potentiellement aucun ancêtre commun. À l'inverse, l'héritage permet de partager un comportement de manière *verticale*, du haut vers le bas.

Une autre manière de se le représenter mentalement est de considérer les interfaces comme un contrat que les classes doivent respecter.

```
1  interface ITranslatable
2  {
3      void Translate(Point delta);
4  }
5
6  class Point : ITranslatable
7  {
8      public double X { get; }
9
10     public double Y { get; }
11
12     public Point(double x, double y)
13     {
14         X = x;
15         Y = y;
16     }
17
18     public void Translate(Point delta)
19     {
20         // FIXME
21     }
22 }
23
24 abstract class Shape
25 {
26     public Point Origin { get; }
27
28     protected Shape(Point origin)
29     {
30         Origin = origin;
31     }
32 }
33
34 class Star : Shape, ITranslatable
35 {
36     public double Width { get; private set; }
37
38     public Star(Point origin, double width) : base(origin)
39     {
40         Width = width;
41     }
42
43     public void Translate(Point delta)
44     {
45         // FIXME
46     }
47 }
```

Dans l'exemple ci-dessus, la classe `Star` *étend* la classe `Shape`, mais *implémente* l'interface `ITranslatable`. La classe `Point` n'étend aucune super-classe, mais *implémente* l'interface `ITranslatable`.

Conseil

Notez bien que, tout comme pour les classes abstraites, les méthodes déclarées dans une interface n'ont pas de corps (`{ ... }`). À la place, les déclarations sont directement suivies d'un point virgule.

Cela signifie que nous ne faisons que fournir le *prototype* ou la *signature* d'une méthode, et non pas son *implémentation*, que nous laissons aux classes qui implémenteront notre interface.

Pour aller plus loin

Le préfixe `I` dans le nom d'une interface est une convention. Plus haut, nous aurions plus appeler l'interface `ITranslatable` tout simplement `Translatable`, ce n'est pas le langage en lui-même qui nous en empêche. Cependant, afin d'expliciter que nous manipulons une interface et non une classe, les développeurs préfèrent les différencier clairement au travers d'une convention de nommage.

La plupart des interfaces ont aussi un nom finissant en "-able", comme manière d'indiquer qu'une interface ne fait que représenter ce qu'une classe peut faire.

Si vous vous intéressez à la question des conventions de nommages en C# , vous pouvez lire la documentation officielle à ce sujet.

2.1.1 Utiliser des interfaces

```
1  // `translatables` can only contain objects whose class implements the
2  // `ITranslatable' interface.
3  List<ITranslatable> translatables = new List<ITranslatable>();
4
5  Star myLittleStar = new Star(new Point(42, 42), 42);
6  Point origin = new Point(0, 0);
7  translatables.Add(myLittleStar);
8  translatables.Add(origin);
9
10 foreach (ITranslatable translatable in translatables)
11 {
12     // Move all objects 10 units to the right and 10 units down.
13     translatable.Translate(new Point(10, 10));
14 }
```

Dans l'exemple ci-dessus, nous voulons effectuer une translation sur un groupe d'objets. En dehors de leur classe d'origine, tout ce que nous savons de ces objets est qu'ils implémentent l'interface `ITranslatable`. Ainsi, ils fournissent la méthode publique `Translate` qui accepte un objet `Point` comme unique paramètre.

Comme vous pouvez le voir, dans la boucle `foreach`, nous considérons les éléments de la liste `translatables` comme de type `ITranslatable`. En effet, quand on ajoute un élément de type

`B` à une liste d'éléments de type `A`, avec pour prérequis que le type `B` hérite de ou implémente (interface) le type `A`, la classe d'origine de l'élément ne nous est plus connue. Tout ce que nous savons, c'est qu'il peut être considéré comme un élement de type `A`.

C'est pourquoi, lorsqu'on itère sur des élements de `translatables`, nous considérons qu'ils sont tous de type `ITranslatable`, plutôt que de types (`Star` ou `Point`), qui sont leur type d'origine. La seule méthode que nous pouvons utiliser avec ces élements (à part les méthodes héritées de `System.Object`, comme la méthode `ToString`) est `Translate`, puisque `ITranslatable` ne déclare aucune autre méthode.

Pour aller plus loin

Vous pouvez *caster* ou coercer manuellement un objet dont le type connu est celui d'une interface ou d'une super-classe, vers le type d'une classe l'implémentant ou l'étendant en utilisant l'opération de *cast* ou de coercition explicite suivante :

```
1 foreach (ITranslatable translatable in translatables)
2 {
3     // Cast `translatable` to the `Star` type.
4     // You can imagine the `Sparkle` method to be defined in the
5     // `Star` class.
6     ((Star)translatable).Sparkle();
7 }
```

Restez conscients que cette manière de faire est **extrêmement dangereuse**, puisqu'il n'y a aucune garantie que l'objet que vous soyez en train de *downcast* ou *sous-typer* appartienne à la bonne classe. Il existe tout de même des manières de retrouver le type original d'un objet avec certitude, mais nous n'en parlerons pas dans ce TP.

2.1.2 Les différences entre classes abstraites et interfaces

Si vous avez suivi de près le cours sur les classes abstraites, vous devez vous demander : quelle est la différrence entre une classe abstraite déclarant des méthodes abstraites et une interface ?

Une première différence est qu'une classe abstraite peut *implémenter* ses méthodes au lieu de juste les déclarer avec le préfixe `abstract`. Les classes abstraites peuvent aussi déclarer et assigner des valeurs à leurs attributs, tandis que les interfaces ne peuvent déclarer que des méthodes.

Un avantage d'une interface est qu'une classe peut implémenter autant d'interfaces qu'elle le souhaite. Cependant, puisque le C# ne permet pas l'héritage multiple, une classe peut seulement hériter d'une seule et unique super-classe.

Enfin, au risque de nous répéter, il y a une différence sémantique entre les classes et les interfaces. L'héritage d'une classe permet de représenter une relation verticale tandis que l'implémentation d'une interface permet de représenter une relation horizontale.

2.2 La surcharge d'opérateurs

Si vous vous rappelez de l'époque OCaml (que de bons moments), vous vous souvenez peut-être que vous pouviez déclarer des fonctions *infixes*. Soient des fonctions que vous pouviez appeler sous la forme infixe.

```
1  (*
2  Let us define the `<~>` infix operator that returns the difference between
3  two integers.
4  *)
5  > let (<~>) a b = abs (a - b);;
6  val ( <~> ) : int -> int -> int = <fun>
7  > 1319 <~> 1277;;
8  - : int = 42
9  > (<~>) 1319 1277;; (* Force the prefix mode *)
10 - : int = 42;;
```

Ne serait-ce pas tip top si nous pouvions faire de même en C# ?

Bon ben, on ne peut pas. Nada. Pas possible. Fin de la question. Le OCaml est trop bon pour ce monde.

Cependant, ce que nous *pouvons* faire, c'est redéclarer des opérateurs existants afin de leur attribuer un comportement particulier.

Imaginons que nous ayons un type `Vector2`. Ce type représente un vecteur en deux dimensions. Un point, si vous voulez, avec des coordonnées X et Y. En physique et en mathématiques, nous sommes habitués à manipuler ces objets avec les opérateurs +, -, *, /, *etc.*, tout comme nous le ferions pour des nombres. Cependant, en C# , nous n'avons jusqu'ici utilisé ces opérateurs qu'avec des nombres (pensez aux `int`, `float`, *etc.*).

Le C# permet la *surcharge d'opérateurs*, qui est l'action d'assigner une opération quelconque à un opérateur infixe particulier entre deux types. Par exemple, si nous voulions additionner deux objets `Vector2` ensemble, nous devrions surcharger l'opérateur `+` de la classe `Vector2`.

La syntaxe est la suivante :

```
1  public class Vector2
2  {
3      public double X { get; }
4      public double Y { get; }
5
6      public Vector2(double x, double y)
7      {
8          X = x;
9          Y = y;
10     }
11
12     // Unary minus: `-v`.
13     public static Vector2 operator -(Vector2 v)
14     {
15         return new Vector2(-v.X, -v.Y);
16     }
17
18     // Addition: `v1 + v2`.
19     public static Vector2 operator +(Vector2 v1, Vector2 v2)
20     {
21         return new Vector2(v1.X + v2.X, v1.Y + v2.Y);
22     }
23
24     // Scalar multiplication: `42 * v`.
25     public static Vector2 operator *(double factor, Vector2 v)
26     {
27         return new Vector2(factor * v.X, factor * v.Y);
28     }
29 }
```

Notre petite classe *Vector2* commence à devenir très utile, et nous permet de représenter des opérations comme l'addition, la soustraction ou encore la multiplication scalaire de vecteurs en C# comme nous l'aurions fait dans un langage mathématique.

Pour aller plus loin

Comme vous pouvez le constater, les différents opérandes d'un opérateur redéclaré peuvent être de types différents (*cf.* la multiplication scalaire de vecteurs). L'ordre des paramètres est important, puisqu'il représente l'ordre des opérandes. Ainsi, dans la multiplication scalaire que nous avons implémentée, le facteur scalaire devrait toujours être en premier. Si nous voulions pouvoir placer le facteur scalaire en dernier (*cad* `v * 42`), nous aurions besoin d'implémenter `operator *(Vector2 v, double factor)`.

Quand on surcharge un opérateur, au moins un des paramètres doit être de la classe dans laquelle l'opérateur est déclaré. Dans notre cas, au moins un paramètre doit être de type `Vector2`.

3 Exercices : Vector2

Important

Le squelette de ce TP est téléchargeable sur le site web des assistants.

Le projet contient déjà l'ensemble des classes dont vous aurez besoin pour les exercices, à l'exception des bonus. Il contient les dossiers et fichiers suivants :

- `ACDC` : Ce dossier contient le code du programme de simulation. Ils ne **doit pas** être altéré de quelque manière que ce soit. Il sera automatiquement remplacé pendant la correction. Ainsi, tout changement que vous y apporterez sera en vain et empêchera sûrement votre projet de compiler.
- `Entities` et `Scenes` : Ces dossiers contiennent les squelettes de classes que vous allez devoir implémenter.
- `Vector2.cs` : Ce fichier devra être complété lors du premier exercice.
- `Program.cs` : Ce fichier est le point d'entrée de votre programme. Son utilité sera expliquée plus tard.

Vous souvenez-vous de la classe `Vector2` dont nous parlions plus haut ? Il est grand temps de l'implémenter correctement !

Ouvrez le fichier `Vector2.cs` dans votre éditeur de texte. Il se trouve à la racine du projet. Comme vous pouvez le voir, il contient déjà la définition de la plupart des opérateurs et fonctions que vous allez devoir implémenter.

Attention

Les fonctions définies, mais qu'il vous reste à implémenter lancent souvent une `NotImplementedException`, comme ici :

```
1  throw new NotImplementedException("some error message");
```

Vous devez **toujours** retirer cette exception une fois la fonction concernée implémentée. Le but de cette ligne est d'indiquer que quelque chose n'a pas été terminé et devra être réglé ultérieurement.

De même, `// TODO` et `// FIXME` sont des lignes de commentaire temporaires et doivent être supprimées une fois la fonction implémentée.

Attention

Dans ce document, lorsque la déclaration d'une fonction est suivie d'un point-virgule (`;`) et non d'un corps (`{ ... }`), cela signifie que nous ne faisons que spécifier le *prototype* de la fonction.

À moins que vous ne soyiez en train d'écrire une interface ou une méthode abstraite dans une classe abstraite *ce que vous ne ferez pas dans ce TP* , vous ne devez jamais mettre un point-virgule après la déclaration d'une méthode. Cela provoquera probablement une erreur de syntaxe et vous empêchera de compiler.

3.1 La surcharge d'opérateurs

Dans cet exercice, vous devrez implémenter des opérateurs utiles pour manipuler vos objets `Vector2`.

Attention

Les opérateurs infixes ne doivent **en aucun cas** modifier un de leurs opérandes. Dans cet exercice, cela signifie que vous ne devrez jamais modifier les coordonnées des objets `Vector2` que vous prenez en paramètres. À la place, vous devrez retourner un nouvel objet `Vector2`, résultat de l'opération.

Considérez l'addition de deux nombres. Dans `a + b`, ni `a` ni `b` ne sont modifiés. À la place, l'opération renvoie un nouveau nombre, somme des deux autres.

Soustraction

```
1  public static Vector2 operator -(Vector2 v1, Vector2 v2);
```

Division

```
1  public static Vector2 operator /(double factor, Vector2 v);
```

```
1  public static Vector2 operator /(Vector2 v, double factor);
```

Exemples

```
1  new Vector2(1, 1) / 2 // [0.5, 0.5]
2  1 / new Vector2(2, 2) // [0.5, 0.5]
3  new Vector2(1, 1) - new Vector(2, 2) // [-1, -1]
```

3.2 Norme d'un vecteur

La distance euclidienne, ou norme L^2 d'un vecteur **u** de coordonnées (x, y), est définie comme la racine carrée de la somme des carrés de ses coordonnées.

$$\|\mathbf{u}\|_2 = \sqrt{x^2 + y^2}$$

ou

$$\|\mathbf{u}\|_2 = (x^2 + y^2)^{\frac{1}{2}}$$

Conseil

Une **variable en gras** dans une équation représente un vecteur.

Pour aller plus loin

Cette définition se généralise aux vecteurs de dimensions supérieures, comme les vecteurs à trois dimensions et au delà.

$$\|\mathbf{u}\|_2 = (x_1^2 + x_2^2 + ... + x_n^2)^{\frac{1}{2}}$$

Implémentez maintenant la méthode`Norm` qui retourne la norme L^2 d'un `Vector2`.

```
1  public double Norm();
```

Exemples

```
1  new Vector2(0, 0).Norm() // 0
2  new Vector2(1, 1).Norm() // 1.4142135623730951
3  new Vector2(1, 2).Norm() // 2.23606797749979
```

3.3 Vecteur normalisé

Le *vecteur normalisé*, ou *vecteur normé* d'un vecteur **u** est défini comme :

$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{\|\mathbf{u}\|_2}$$

$\hat{\mathbf{u}}$ est un vecteur unitaire de même direction que **u**.

Maintenant, implémentez la méthode `Normalized` qui retourne le vecteur normalisé d'un `Vector2`.

Attention

La méthode `Normalized` ne doit **en aucun cas** modifier le vecteur original, mais plutôt retourner un nouveau vecteur égal à son vecteur normé.

```
1  public Vector2 Normalized();
```

Examples

```
1  new Vector2(1, 1).Normalized() // [0.707106781186475, 0.707106781186475]
2  new Vector2(1, 0).Normalized() // [1, 0]
3  new Vector2(0, 0).Normalized() // [NaN, NaN] (and the rocket goes boom)
```

3.4 Distance entre deux vecteurs

La distance euclidienne entre deux vecteurs **u** et **v** est définie comme la norme L^2 de leur différence.

$$d(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|_2$$

Ce qui donne la forme suivante, qui vous est sûrement plus familière :

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}$$

Implémentez maintenant la méthode statique `Dist` qui prend deux objets `Vector2` et retourne la valeur de la distance euclidienne entre eux.

```
1  public static double Dist(Vector2 v1, Vector2 v2);
```

Examples

```
1  Vector2.Dist(new Vector2(123, 321), new Vector2(123, 321)) // 0
2  Vector2.Dist(new Vector2(0, 0), new Vector2(1, 1)) // 1.4142135623730951
3  Vector2.Dist(new Vector2(42, 0), new Vector2(0, 42)) // 59.39696961966999
```

3.5 Vector2.Zero

Nous allons beaucoup utiliser le vecteur nul **0** de coordonnées $(0, 0)$ durant le reste de ce TP.

Pour faciliter les choses, définissez un attribut statique `Zero` égal au vecteur zéro dans la classe `Vector2`.

Pour aller plus loin

Puisqu'aucune méthode d'un objet `Vector2` ne peut modifier ses coordonnées, notre structure de données `Vector2` est *immuable*. Cela signifie que, lorsque vous avez un `Vector2`, vous ne pouvez pas le modifier directement.

Ainsi, si vous voulez obtenir un nouveau *Vector2*, vous devez soit en créer un nouveau, soit combiner deux vecteurs existants grâce aux opérateurs que vous avez surchargés.

Cette priorité assure que le `Vector2.Zero` aura toujours les mêmes coordonnées.

4 Exercices : Simulation cinématique

Cette seconde partie du TP (et la plus intéressante) est dédiée à la simulation de corps physiques.

Dans ce but, nous avons développé un environnement de simulation à la pointe de la technologie, qui a pour finalité de recréer la physique de notre monde avec une précision extrême. Enfin, nous avons au moins tenté.

Lorsque vous avez testé votre classe `Vector2`, vous avez peut-être remarqué la méthode `RunSimulation` de la classe `Program`. Eh bien, c'est le moment de l'utiliser !

```
1  public static void Main()
2  {
3      RunSimulation(); // And may the odds be ever in our favor...
4  }
```

Lancez maintenant le programme. Si tout se passe bien, vous remarquerez qu'un lien s'est affiché dans la console. Cliquez dessus !

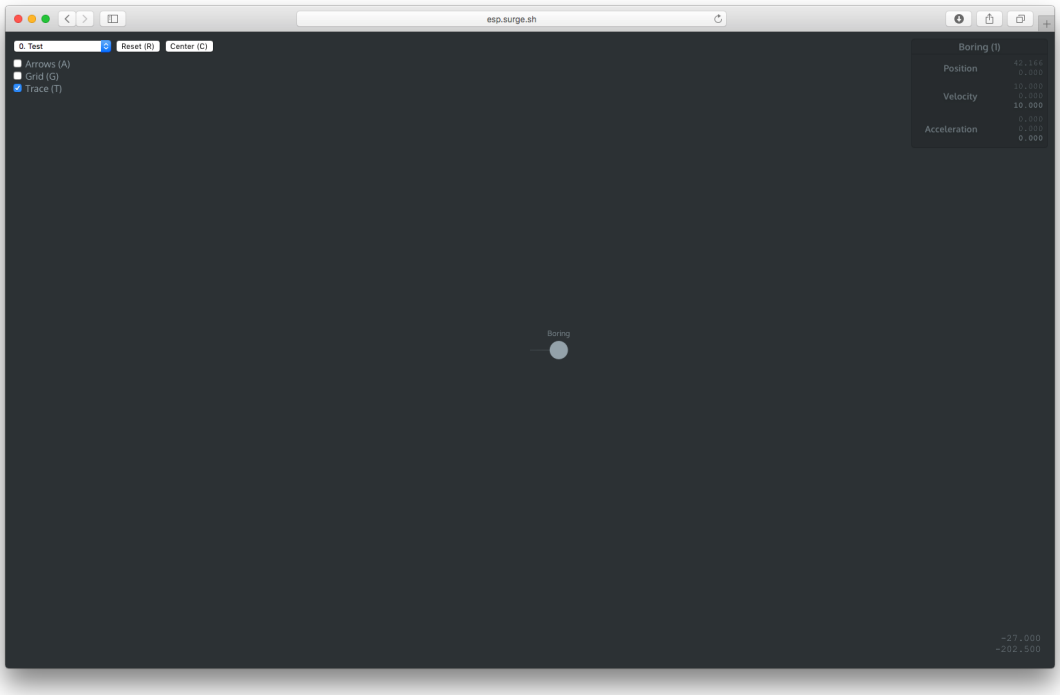

FIGURE 1 – Scène 0. Test

Vous devriez voir quelque chose ressemblant à Scène 0. Test. Il s'agit de l'environnement de simulation. Vous remarquerez quelques éléments de cette interface :

- Le **cercle blanc** annoté "Boring" représente un corps physique. Dans un moment, vous pourrez en créer d'autres !
- La **ligne blanche** qui disparaît progressivement représente la trajectoire du corps physique.
- Le **menu déroulant** vous permet de choisir une scène parmi une sélection. À noter que seule la scène intitulée "0. Test" affichera un corps en mouvement. Les autres s'animeront au fur et à mesure que vous implémenterez les entités du dossier `Entities`.
- Le bouton **Reset** vous permet de réinitialiser la scène en cours. Tous les corps reviendront à leur position, leur vitesse et leur accélération initiales.
- Le bouton **Center** remet la caméra au centre de la scène, aux coordonnées $(0, 0)$.
- L'option **Arrows** vous laisse afficher les vecteurs de vitesse (en bleu) et d'accélération (en rouge) pour les différents corps de la scène. Le seul corps de la scène "Test" n'ayant par défaut pas d'accélération et une vitesse initiale non-nulle, seul son vecteur de vitesse apparaîtra. La magnitude de la vitesse et de l'accélération représentent, respectivement, la distance qu'un corps traversera en une unité de temps (seconde), et la vitesse que ce corps gagnera lors de l'écoulement de la prochaine unité de temps.

— L'option **Grid** vous permet d'afficher une grille. Cette grille est là pour vous aider à situer les corps par rapport à l'environnement. Les lignes les plus fines sont dessinées toutes les 10 unités de distance (par exemple, le mètre), tandis que les lignes plus marquées apparaissent toutes les 100 unités de distance.
— L'option **Trace** vous permet d'afficher les trajectoires des corps en mouvement.
— Sur la droite, vous trouverez des informations sur tous les corps de la scène actuelle. Cliquer sur les informations d'un corps permet de le **suivre** : quoi qu'il advienne, il restera au centre de l'écran.
— Dans le coin en bas à droite de l'écran sont affichées les coordonnées de la souris **par rapport à la scène**.

Vous pouvez **zoomer** en utilisant la molette de votre souris ou votre pavé tactile, et vous pouvez vous déplacer en cliquant n'importe où sur l'écran et en déplaçant le curseur de votre souris.

Attention

Si la console n'affiche pas de lien lors de l'exécution du projet, ou si vous observez un Écran d'erreur en cliquant sur le lien, veuillez contacter un assistant.

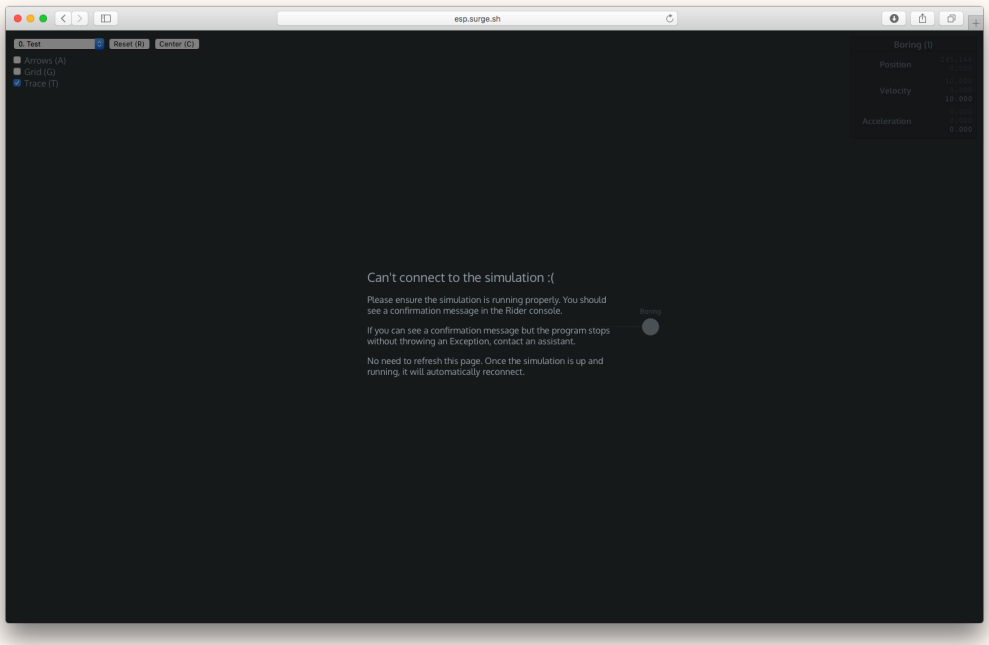

FIGURE 2 – Écran d'erreur

4.1 Understanding the simulation environment

Afin de vous familiariser avec l'environnement de simulation, nous vous recommandons de jouer avec les classes **TestBody** et **TestScene**. Vous trouverez plus d'explications sur leur fonctionnement dans leurs fichiers sources respectifs.

Assurez-vous de prendre quelques minutes pour faire le tour de l'environnement de simulation. Comprendre son fonctionnement et la manière dont les corps sont représentés est important pour le reste de ce TP.

4.2 Remise à niveau en cinématique

Tip

Pour vous rafraîchir la mémoire :

- Un **corps physique** est un objet dans l'espace qui a une masse et une position. Vous êtes un corps, votre ordinateur est un corps, vos assistants sont des (super) corps.
- La **vitesse** est une mesure de l'évolution de la position d'un corps en fonction du temps. Dans ce TP, la vitesse est mesurée en m/s, c'est-à-dire en nombre de mètres parcourus en une unité de temps (une seconde). Puisque nous gérons des objets en deux dimensions, notre vitesse est un vecteur à deux dimensions. La **magnitude**, **longueur**, ou **norme** de ce vecteur représente la vitesse à laquelle il se déplace par rapport à l'origine, tandis que la **direction** de ce vecteur représente la direction dans laquelle il se déplace.
- L'**accélération** est une mesure de l'évolution de la vitesse d'un objet au cours du temps. Dans ce TP, l'accélération est mesurée en m/s^2, c'est-à-dire la variation de la vitesse en une unité de temps (une seconde). C'est un concept plus compliqué à appréhender que la vitesse, mais un moyen simple de le voir est que l'accélération est à la vitesse ce que la vitesse est à la position. Et tout comme la vitesse, notre accélération est un vecteur.
- Une **unité de temps** dans ce TP représente une seconde. Techniquement, il n'est pas important qu'il s'agisse d'une seconde, d'une heure ou d'une durée arbitraire. Cependant, pour rendre les choses plus naturelles et pouvoir fournir une visualisation qui ait du sens, nous avons choisi la seconde comme unité de temps. De même, l'**unité de distance** de ce TP sera le mètre (mais sera représentée par des pixels).

Si certaines de ces notions vous semblent encore floues, n'hésitez-pas à demander une meilleure explication à vos assistants. Ce TP n'est pas destiné à remplacer une introduction aux bases de la cinématique et de la dynamique.

4.3 Approximer le taux de variation instantané

Attention

Il n'est pas nécessaire de comprendre cette section pour terminer ce TP. Cependant, comprendre la manière dont les composants de la simulation fonctionnent les uns avec les autres vous sera d'une grande aide dans les prochains exercices.

Soit r la position d'un corps, v sa vitesse instantanée et a son accélération instantanée. Nous savons grâce à nos leçons de cinématique du lycée que :

$$a(t) = \dot{v}(t) = \frac{dv(t)}{dt}$$
$$v(t) = \dot{r}(t) = \frac{dr(t)}{dt}$$

Nous savons également grâce à la définition de la dérivée que le taux de variation instantané d'une fonction f à un point a est égal à :

$$\dot{f}(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

Supposons maintenant que nous connaissons $f(a)$ et $\dot{f}(a)$ pour un certain point a, et que nous voulons *approximer* $f(a+h)$ pour un delta h suffisamment petit. En supprimant la limite, on obtient :

$$\dot{f}(a) \approx \frac{f(a+h) - f(a)}{h}$$
$$f(a+h) \approx h \cdot \dot{f}(a) + f(a)$$

Maintenant, en remplaçant f par r dans la formule ci-dessus, on obtient :

$$\begin{aligned} r(t_0 + \Delta t) &\approx \Delta t \cdot \dot{r}(t_0) + r(t_0) \\ r(t_0 + \Delta t) &\approx \Delta t \cdot v(t_0) + r(t_0) \\ r(t_0 + \Delta t) &\approx \Delta t \cdot v_0 + r_0 \\ r(t_1) &\approx (t_1 - t_0) \cdot v_0 + r_0 \end{aligned}$$

Ce qui nous dit que depuis une position initiale r_0 et avec une vitesse initiale v_0, nous pouvons approximer la position à l'instant suivant $t_1 = t_0 + \Delta t$ comme $(t_1 - t_0) \cdot v_0 + r_0$. Nous pouvons procéder de la même manière pour la vitesse, avec $v_1 = (t_1 - t_0) \cdot a_0 + v_0$.

Vous vous demandez peut-être : pourquoi faisons-nous tout ça ? Eh bien, puisque nous essayons de représenter des modèles physiques sur notre ordinateur à une fréquence bien moins haute que celle du monde réel — pensez $\frac{1}{t_P}$ où t_P est la constante de Planck, soit une fréquence d'approximativement $1.86 \cdot 10^{34}$ GHz distribuée sur un nombre incalculable de particules — nous devons faire des concessions sur la précision de notre simulation. Ainsi, la formule précédente nous permet d'approximer les changements de position et de vitesse en utilisant la fréquence même de notre processeur comme unité de temps. Évidemment, puisqu'il s'agit d'une approximation, notre taux d'erreur augmente énormément au fil du temps. Mais dans le cadre de cette simulation, notre précision est largement suffisante.

La logique derrière la mise à jour de la simulation se trouve dans `Scene.Update`, où le *delta time* est calculé à partir du temps écoulé entre la dernière mise à jour et l'instant présent.

```
1  private void Update(object stateInfo)
2  {
3      var delta = _span.Ticks / (double)TimeSpan.TicksPerSecond * Speed;
4      foreach (var entity in _entities)
5          entity.Update(delta);
6  }
```

Ainsi que dans `Body.Update`, où nous mettons à jour la position et la vitesse d'un corps en fonction des approximations trouvées plus haut :

```
1  public virtual void Update(double delta)
2  {
3      Velocity += Acceleration * delta;
4      Position += Velocity * delta;
5      Acceleration = Vector2.Zero;
6  }
```

Pour aller plus loin

Bien que tout cela puisse vous paraître fort ennuyeux, ces méthodes ont des applications dans des logiciels que vous utilisez au quotidien. En effet, le moteur physique de votre jeu préféré utilise cette même approximation. Pour ceux qui ont choisi Unity comme moteur de jeu, l'explication ci-dessus vous aidera à différencier `Update` et `FixedUpdate` dans les boucles de jeu, et vous démystifiera la variable `Time.deltaTime`. Cela pourra même vous aider à comprendre pourquoi accélérer le passage du temps peut introduire de sérieuses erreurs d'imprécision.

4.4 La boucle de simulation

La manière dont notre simulation fonctionne est que chaque object simulé est mis à jour à une fréquence définie dans `Constants.cs`. En effet, tous les objets de notre simulation *implémentent* l'interface `IEntity`, qui déclare une méthode `Update`.

Durant chaque mise à jour, la vitesse de chaque objet est mise à jour en fonction de son accélération, puis sa position est mise à jour en fonction de la nouvelle valeur de la vitesse. L'accélération est ensuite remise à zéro.

Ce que nous vous demandons de faire dans les exercices suivants est d'implémenter la méthode `Update` de plusieurs types d'entités. Dans ces méthodes `Update`, vous en viendrez à altérer l'accélération de vos objets dans l'optique de provoquer un changement de vitesse, et ainsi de position, au cours du temps.

Vous pourriez vous demander : pourquoi ne pas directement changer la position de nos objets selon l'équation paramétrique d'un certain mouvement ?

Conseil

Une équation paramétrique de mouvement est une équation de la forme :

$$r(t) = ...$$

où la position r d'un objet dépend directement de l'instant t actuel. Ainsi, à chaque instant t_i, nous n'avons qu'à recalculer $r(t_i)$ pour mettre à jour la position de notre objet de manière instantanée et précise.

Utiliser des équations paramétriques pour décrire un mouvement est certainement l'option la plus simple (et la plus précise). Cependant, dès que notre mouvement devient plus complexe, ou implique plusieurs corps, nos équations peuvent devenir extrêmement compliquées.

Vous trouverez un exemple d'utilisation d'une équation paramétrique pour mettre à jour la position d'un corps en commentaire de la classe `TestBody`, instanciée dans la classe `TestScene`.

4.5 La seconde loi de Newton

La seconde loi de Newton stipule :

$$\sum \mathbf{F} = m\mathbf{a}$$

Où $\sum \mathbf{F}$ est la somme des forces appliquées à un corps en Newtons ($\text{kg} \cdot \text{m/s}^2$), m est la masse du corps en kilogrammes, et $\mathbf{a}$ est le vecteur accélération du corps en m/s^2.

En déplacant des variables, nous trouvons :

$$\mathbf{a} = \frac{\sum \mathbf{F}}{m}$$

Cela signifie qu'appliquer une force à un corps modifiera son accélération par un facteur proportionnel à la force elle-même, et inversement proportionnelle à la masse de ce corps. Ainsi, des corps plus lourds nécessiteront une force plus importante pour être déplacés, tandis qu'augmenter la magnitude de la force déplacera aussi le corps plus rapidement. Enfin, appliquer une force à un corps accélérera ce corps dans la direction de cette force.

Implémentez la méthode `ApplyForce` dans le fichier `Body.cs`. Cette fonction applique une force discrète sur un objet à l'aide de la formule suivante :

$$\mathbf{a} = \mathbf{a} + \frac{\mathbf{F}}{m}$$

```
1  public void ApplyForce(Vector2 force);
```

Attention

Gardez à l'esprit qu'un objet peut appeler plusieurs fois `ApplyForce` dans une seule étape de mise à jour. Le comportement attendu est alors d'appliquer une force composée et non pas simplement la force calculée lors du dernier appel à `ApplyForce`. C'est pourquoi vous devrez faire la *somme* de l'ancienne accélération et de la nouvelle, afin de ne pas annuler les forces appliquées auparavant. Cette somme est déjà présente dans la formule donnée plus haut.

Conseil

Souvenez-vous que l'accélération d'un objet est réinitialisée à chaque boucle de simulation (au début de la mise à jour). C'est pour cela que lors du premier appel à `ApplyForce` par la fonction `Update`, l'accélération sera initialement nulle.

Ne vous prenez pas la tête sur cette méthode. Souvenez-vous des opérateurs que vous avez implémentés dans la classe `Vector2`. Le corps de la fonction ne devrait pas contenir plus d'une ligne de code.

4.6 La gravité de la situation

Telle que nous l'avons étudiée en cours, la gravité est une force constante appliquée à un objet. À la surface de la Terre, la gravité a une magnitude de 9.81 N et une direction pointant vers le centre de la Terre. Dans cet exercice, nous souhaitons implémenter un objet soumis à une force gravitationnelle.

Implémentez le constructeur et la méthode `Update` de la classe `Gravity`.

La valeur de `gravity` étant un vecteur constant, et puisque vous allez en avoir besoin dans votre méthode `Update`, vous devrez la conserver dans un attribut de la classe `Gravity`. Vous pourrez tester et visualiser l'exécution de votre code dans l'environnement, voir Scène 1. Gravity.

Conseil

Faites attention à toujours appeler `base.Update(delta)` au début des méthodes `Update`.

Attention

Vous pouvez ajouter n'importe quel méthode ou attribut dans la classe, comme dans toutes les classes n'étant pas situées dans le dossier `ACDC`. Cependant, vous ne devez **en aucun cas** modifier le prototype d'une méthode ou d'un constructeur existant.

Important

Vous ne pouvez **en aucun cas** modifier la position ou la vitesse d'un objet à un autre endroit que dans son constructeur. Le but de cet exercice est de vous faire simuler des mouvements naturels en manipulant des accélérations.

Souvenez-vous de la méthode `ApplyForce` que nous vous avons demandé d'écrire tout à l'heure. Vous l'utiliserez beaucoup !

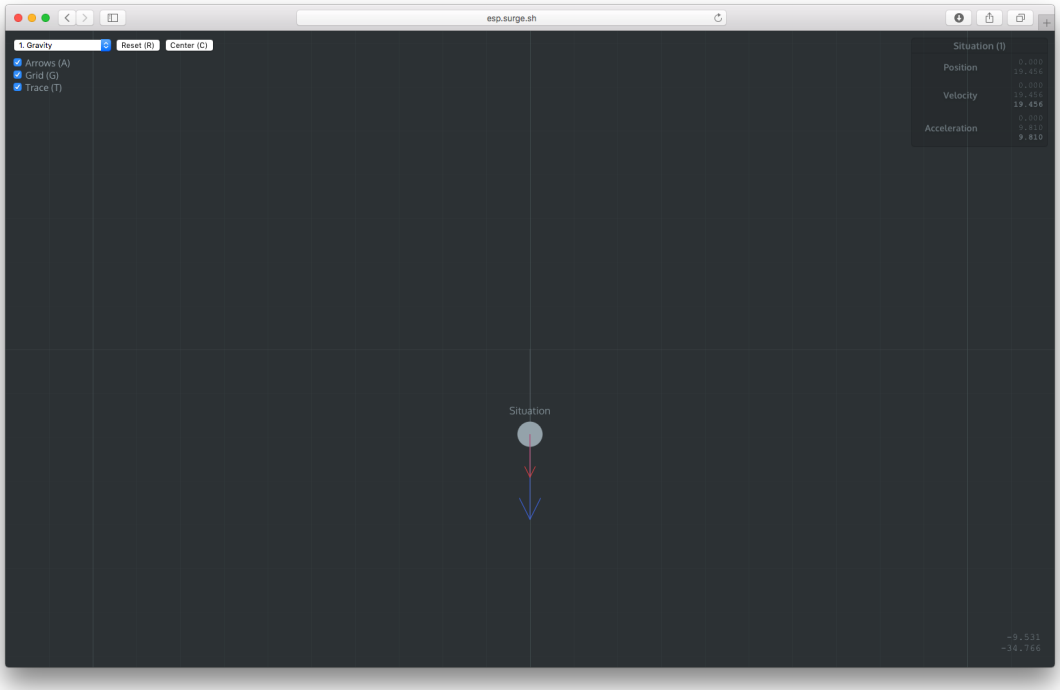

FIGURE 3 – Scène 1. Gravity

4.7 Bonus 1 : Un pendule simple

Tip

Ce sujet contient trois exercices bonus, éparpillés parmi les sections. Si vous ne désirez pas implémenter les bonus (même si nous vous le recommandons fortement), vous pouvez passer à l'exercice suivant.

Un pendule est un objet attaché à une corde de longueur fixe.

Afin de simuler le mouvement d'un pendule, nous devons nous assurer que notre objet garde une distance constante à un certain point, que nous appellerons **point pivot**. En clair, après avoir appliqué les différentes forces à notre objet, nous devons appliquer une force supplémentaire afin de rectifier la position de notre objet, dans l'optique de respecter la contrainte. Nous appelons cette force la **force de contrainte** $\mathbf{F}_c$.

La formule de la force de contrainte $\mathbf{F}_c$ est la suivante :

$$\mathbf{F}_c = \mathbf{r}\frac{-\sum \mathbf{F} \cdot \mathbf{r} - m\mathbf{v} \cdot \mathbf{v}}{\mathbf{r} \cdot \mathbf{r}}$$

où $\sum \mathbf{F}$ est la somme de toutes les forces appliquées à l'objet, $\mathbf{r}$ est la position de l'objet **relative** au point de pivot, m est la masse de l'objet, et $\mathbf{v}$ est sa vitesse. L'opérateur $\cdot$ est le **produit scalaire** de deux vecteurs.

Grâce à la seconde loi de Newton, on retrouve $\sum \mathbf{F}$ à partir de l'accélération de l'objet :

$$\sum \mathbf{F} = m\mathbf{a}$$

Afin d'implémenter la formule ci-dessus, vous devrez d'abord implémenter la méthode statique `Vector2.Dot` de la classe `Vector2`.

Puis, implémentez le constructeur et la méthode `Update` de la classe `DistanceJoint`. Vous n'aurez pas à appeler la méthode `Update` du `Body` fourni dans le constructeur vous-même, ce sera géré par la `Scene` de manière automatique. Vous pourrez tester et visualiser l'exécution de votre code dans l'environnement, voir Scène 1.1. Pendulum.

Pour aller plus loin

Vous aurez peut-être remarqué que notre pendule finit par s'éloigner de son point de pivot. Puisque nous calculons la force de contrainte de manière relative à la position actuelle de l'objet, la contrainte est progressivement affaiblie et l'erreur s'accroit.

Si vous vous intéressez au problème des contraintes dans les moteurs physiques, ou si vous désirez implémenter un système de contraintes plus robuste, vous trouverez des explications plus détaillées sur les contraintes dans les simulations physiques dans cet excellent tutoriel, que nous avons utilisé comme référence dans cet exercice.

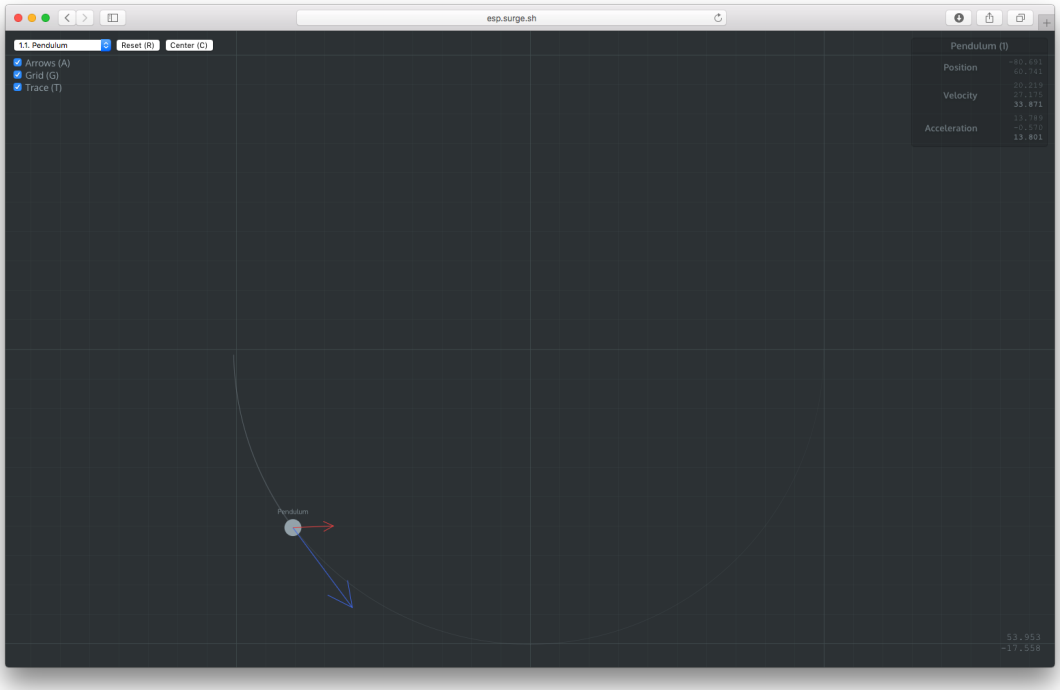

FIGURE 4 – Scène 1.1. Pendulum

4.8 L'oscillateur harmonique

La définition Wikipédia d'un oscillateur harmonique

Dans la mécanique classique, un oscillateur harmonique est un système qui, quand il est déplacé de sa position d'équilibre, subi une force de rappel $\mathbf{F}$ qui est proportionnelle à son déplacement $\mathbf{x}$:

$$\mathbf{F}_h = -k\mathbf{x}$$

où k est une constante positive.

Jetez maintenant un oeil au constructeur de `Spring.cs`. Vous verrez qu'il prend cinq arguments. Vous pouvez oublier `name` et `density`, qui ne servent qu'à la visualisation. `initialPosition` sera définie par le constructeur de `Body` comme le vecteur initial `Position`, tandis que `mass` sera définie comme la constante `Mass`. `Position` et `Mass` possèdent des getters déclarés dans `Body.cs`, auquels vous pouvez accéder depuis les méthodes de `Spring`.

Le constructeur de `Spring` reçoit également un vecteur `origin` et une constante `spring`. Dans la formule ci-dessus, k représente la constante `spring`, tandis que `x` représente le déplacement de l'oscillateur à un instant t, qui est tout simplement égal à `Position - origin`.

Puisque `origin` est une constante, mais que `Position` varie au cours du temps, vous devrez stocker le vecteur `origin` dans un attribut de la classe `Spring`, afin de pouvoir y accéder depuis votre méthode `Update`. Vous devrez faire de même avec la constante `spring`.

Implémentez le contructeur et la méthode `Update` de la classe `Spring`. Vous pourrez tester

et visualiser l'exécution de votre code dans l'environnement, voir Scène 2. Spring.

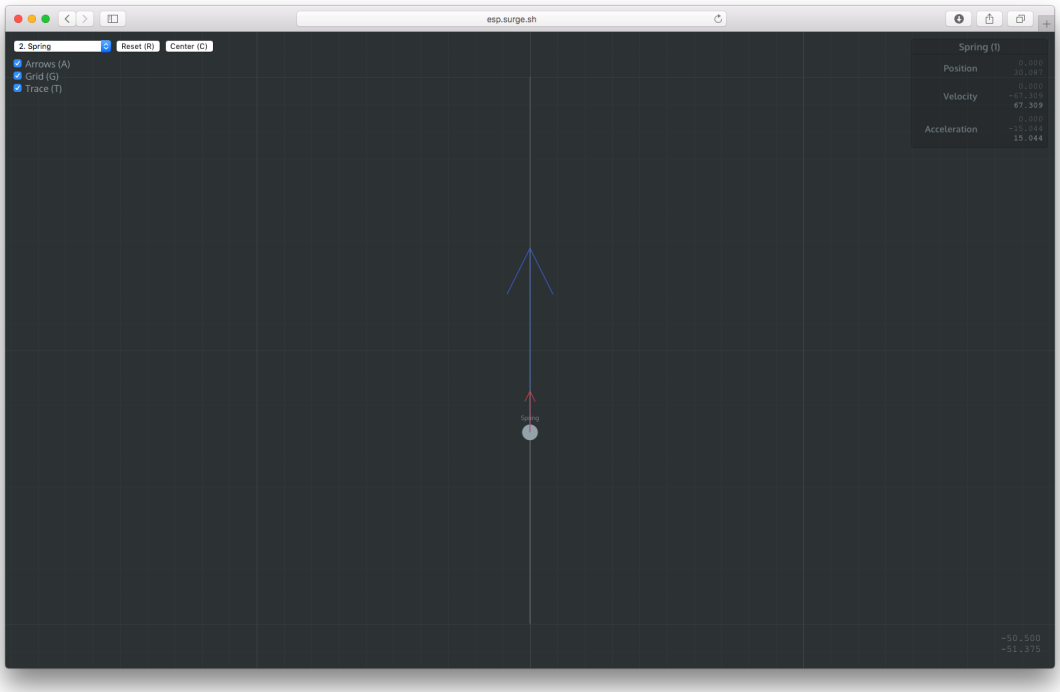

FIGURE 5 – Scène 2. Spring

4.9 L'oscillateur harmonique amorti

Un oscillateur harmonique amorti est un oscillateur harmonique qui subi une force supplémentaire d'amortissement $\mathbf{F}_d$:

$$\mathbf{F}_d = -c\dot{\mathbf{x}}$$

où c est la constante d'*amortissement* et $\dot{\mathbf{x}}$ est la dérivée du déplacement.

La force composée appliquée à notre oscillateur harmonique amorti est donc :

$$\mathbf{F} = \mathbf{F}_h + \mathbf{F}_d$$

Puisque nous sommes dans une simulation, $\dot{\mathbf{x}}$ peut être approximé comme :

$$\dot{\mathbf{x}} \approx \frac{\mathbf{x}_1 - \mathbf{x}_0}{\Delta t}$$

où $\mathbf{x}_1$ est le déplacement actuel, et $\mathbf{x}_0$ est le déplacement lors de la mise à jour précédente.

En code, cela donne simplement :

```
1  var displacementVelocity = (currDisplacement - prevDisplacement) / delta;
```

Implémentez le constructeur et la fonction `Update` de la classe `DamperSpring`. À chaque mise à jour, vous devrez stocker le déplacement actuel calculé afin de pouvoir l'utiliser lors de la mise

à jour suivante. Vous pourrez tester et visualiser l'exécution de votre code dans l'environnement, voir Scène 3. Damper.

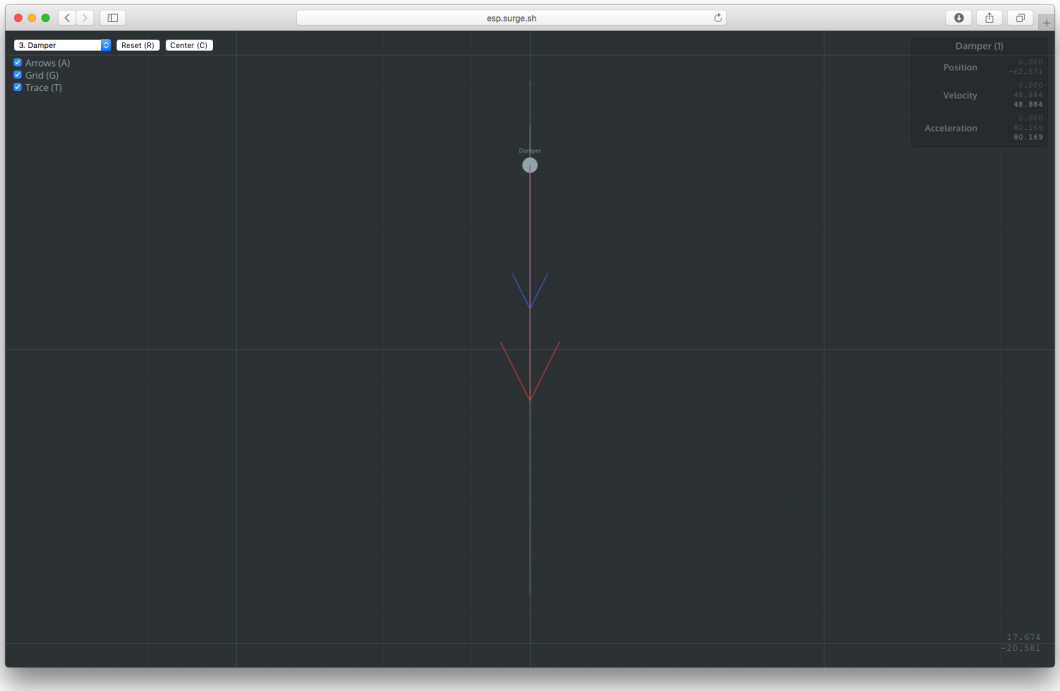

FIGURE 6 – Scène 3. Damper

4.10 Description d'un mouvement circulaire

4.10.1 L'oscillateur harmonique déphasé

Un mouvement circulaire peut être décrit comme la composition de deux oscillateurs harmoniques orthogonaux.

Afin de développer votre intuition sur le sujet, jetez un oeil à cette animation super cool. Le mouvement des points rouges et bleus sont ceux de deux oscillateurs harmoniques de phase différente.

La différence exacte de phase entre ces deux oscillateurs harmoniques est égale à $\frac{\pi}{2}$, ce qui est la même différence de phase qu'entre *cos* et *sin* :

$$cos(\theta) = sin(\theta + \frac{\pi}{2})$$

Un problème que nous allons rencontrer est que notre implémentation actuelle d'un oscillateur harmonique commence à l'instant $t = 0$. Ainsi, tous les oscillateurs harmoniques que nous ajouterons à notre simulation seront initialement synchronisés. Afin d'avoir des oscillateurs harmoniques *déphasés*, leur vitesse et position initiales doivent être différentes.

L'équation paramétrique d'un oscillateur harmonique est :

$$\omega = \sqrt{\frac{k}{m}}$$
$$r(t) = x_{max} * \cos(\omega * t + \varphi)$$
$$v(t) = -x_{max} * \omega * \sin(\omega * t + \varphi)$$

Nous savons que $\varphi = \frac{\pi}{2}$, donc par substitution et en calculant $r(0)$ et $v(0)$ nous obtenons :

$$\begin{aligned} r(t) &= x_{max} * \cos(\omega * t + pi/2) \\ v(t) &= -x_{max} * \omega * \sin(\omega * t + \varphi) \\ r(0) &= x_{max} * \cos(pi/2) \\ &= 0 \\ v(0) &= -x_{max} * \omega * \sin(pi/2) \\ &= -x_{max} * \omega \end{aligned}$$

Donc tout ce que nous avons à faire pour écrire notre nouvel oscillateur déphasé est d'étendre notre oscillateur actuel et de définir ses nouvelles position et vitesse initiales à partir de la formule d'au-dessus.

Conseil

Souvenez-vous que la position initiale trouvée plus haut est par rapport l'origine de l'oscillateur.

Implémentez le constructeur de la classe `SpringMax` (qui est celle de l'oscillateur dont nous parlons). Pas besoin d'implémenter sa méthode `Update` : il l'hérite de sa classe-mère `Spring`. Vous pourrez tester et visualiser l'exécution de votre code dans l'environnement, voir Scène 4. Circling.

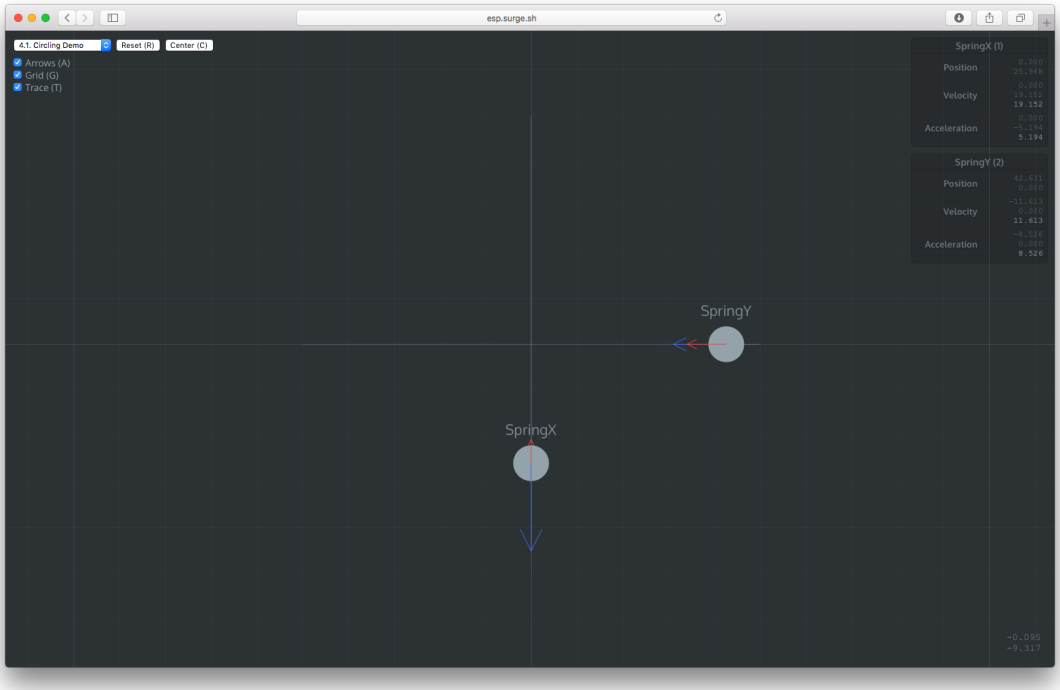

FIGURE 7 – Scène 4.1. Circling Demo

4.10.2 Un corps composé

Maintenant que nous avons nos oscillateurs harmoniques déphasés, nous avons besoin d'un moyen de composer deux corps physiques ou plus.

Un `CompositeBody` contient et met à jour d'autres corps. Sa propre accélération est toujours égale à la somme de l'accélération de ses enfants.

Implémentez le constructeur et les méthodes `Update` et `Add` de la classe `CompositeBody`.

Conseil

En écrivant la méthode `Add`, n'oubliez pas d'ajouter correctement la vitesse et l'accélération initiales des corps enfants à celles de `CompositeBody`.

La méthode `Add` sera uniquement appelée dans les constructeurs des sous-classes de `CompositeBody`, vous avez donc le droit de le faire (et nous vous y encourageons).

4.10.3 Mouvement circulaire

Vous possédez maintenant tous les éléments nécessaires pour décrire un mouvement circulaire comme la composition d'un `Spring` et d'un `SpringMax`.

Vous devrez changer la position initiale de votre `SpringMax` pour que sa direction et sa vitesse initiale soient correctes dans tous les cas. Vous pouvez jeter un oeil à la scène "4.1. Circle Demo" pour un exemple de comment le `SpringMax` doit être positionné par rapport au `Spring`.

Implémentez le constructeur de la classe `CirclingSpring`.

Conseil

Puisque `CirclingSpring` est la composition de deux corps, vous devriez instancier ces corps avec les bons paramètres dans le constructeur de `CirclingSpring` et les passer à la méthode `Add` héritée de `CompositeBody`.

Vous n'avez pas besoin d'implémenter la méthode `Update` pour les classes `CirclingSpring` ou `InfinitySpring` (plus bas), puisque la méthode `Update` de `CompositeBody` s'occupe déja de tout.

Vous pourrez tester et visualiser l'exécution de votre code dans l'environnement, voir **??** et Scène 4.2. Circling Complex.

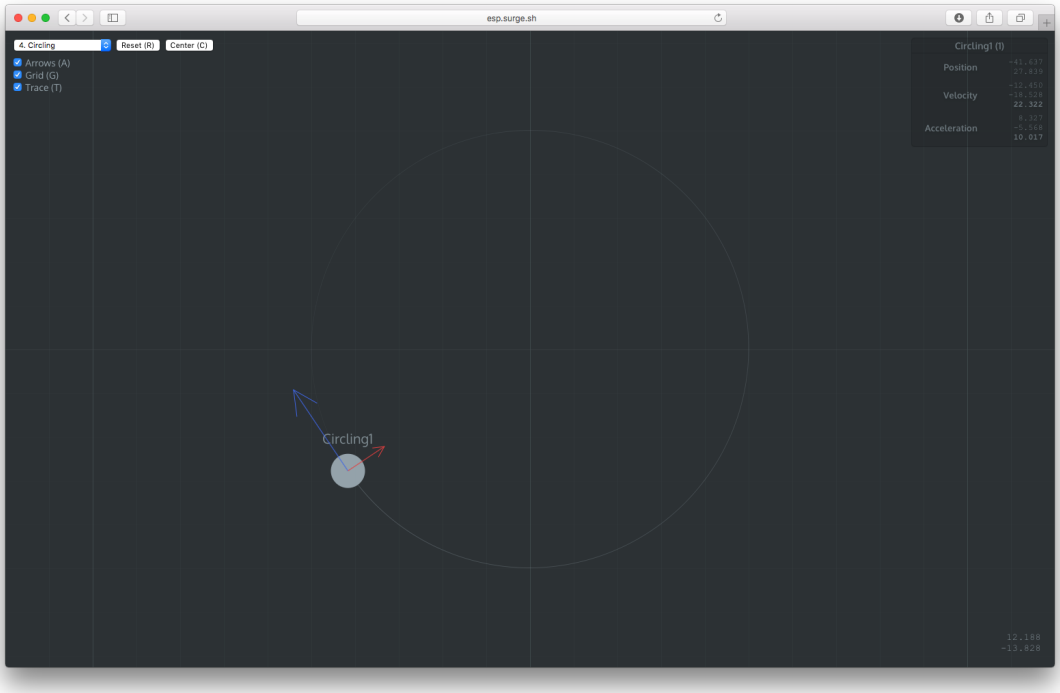

FIGURE 8 – Scène 4. Circling

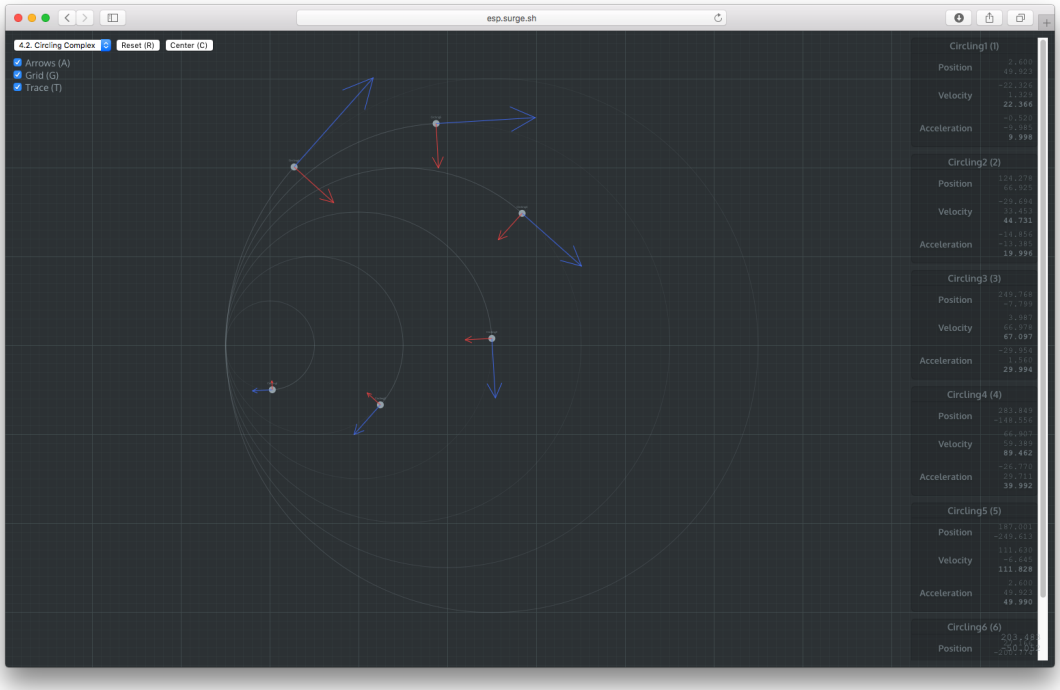

FIGURE 9 – Scène 4.2. Circling Complex

4.10.4 Bonus 2 : Vers l'infini et au delà

Le mouvement suivant le symbole infini (∞) est presque identique au mouvement circulaire.

Cependant, l'amplitude de `SpringMax` (le déplacement maximal) devrait être la moitié de celle de `Spring`, tandis que sa fréquence est deux fois celle de `Spring` (pour la même masse). Ainsi, sa constante d'élasticité doit être quatre fois celle de `Spring`.

Implémentez le constructeur et la méthode `Update` de la classe `InfinitySpring`.

Vous pourrez tester et visualiser l'exécution de votre code dans l'environnement, voir **??** et Scène 5.2. Infinity Complex.

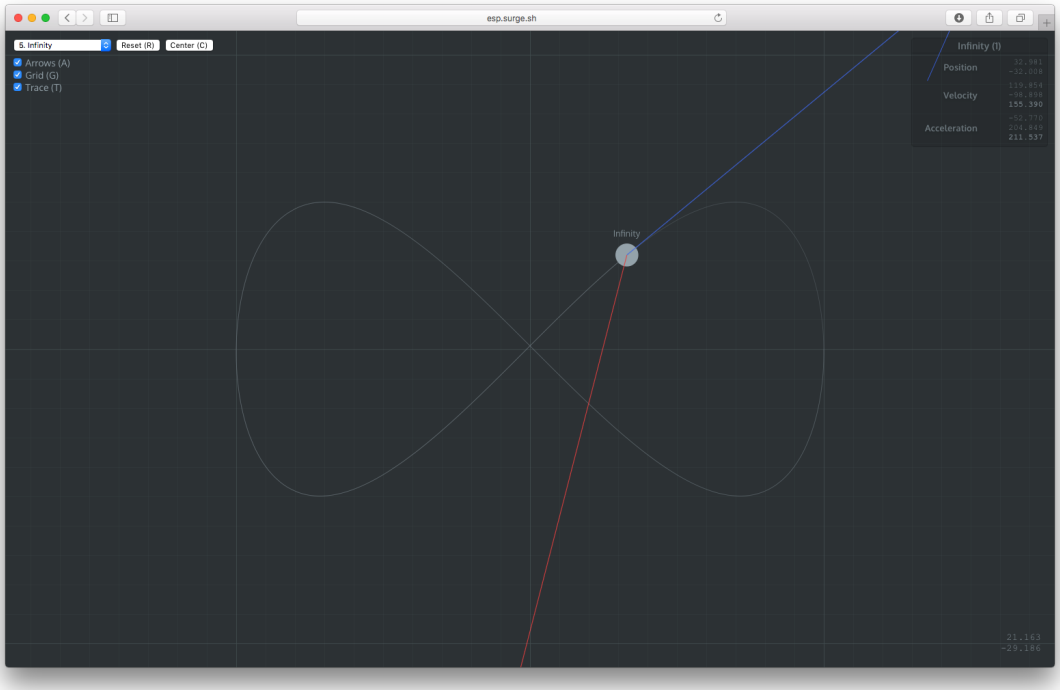

FIGURE 10 – Scène 5. Infinity

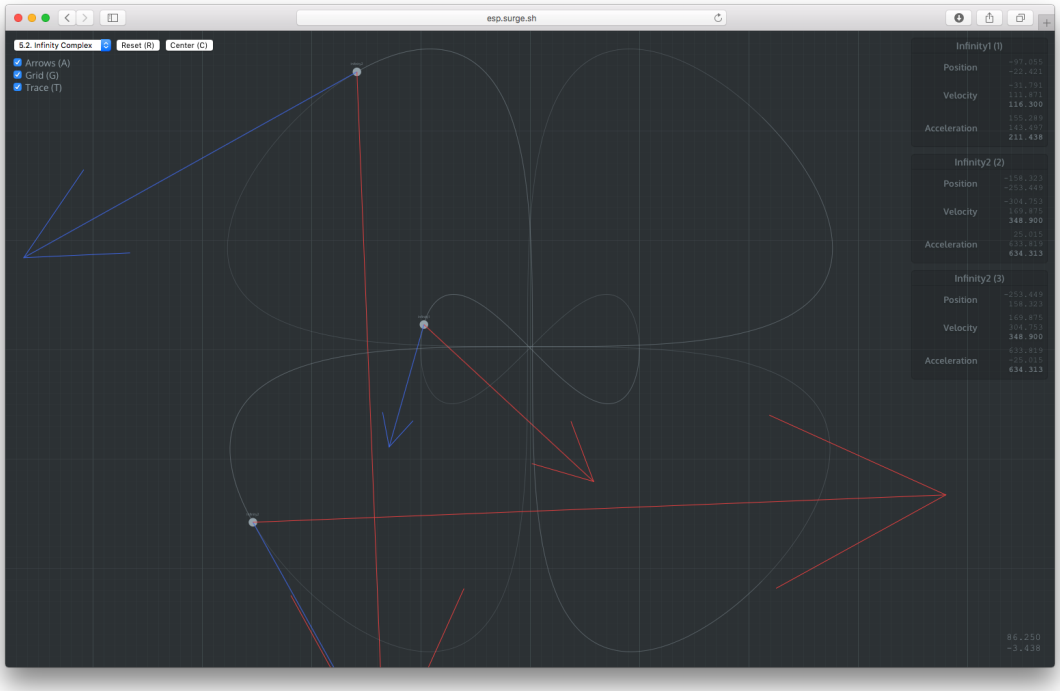

FIGURE 11 – Scène 5.2. Infinity Complex

5 Exercice : EPITA Space Program

Delos Incorporated vous a demandé de construire et de lancer en orbite un parc d'attraction sur le thème de l'espace. En temps que jeune élève diplômé de l'EPITA, vous n'avez pas beaucoup d'expérience dans le lancement d'objets, à part peut-être les souris en salle machine.

Heureusement, vous vous souvenez de vos cours de physique à l'EPITA, et vous êtes sûr(e) et certain(e) que vous serez en mesure de réaliser cette tâche.

5.1 Le problème à N corps

La page Wikipédia sur le problème à N corps

En physique, le problème à N corps est le problème de la prédiction des mouvements individuels d'un groupe d'objets célestes qui intéragissent gravitationnellement les uns avec les autres. La résolution de ce problème est motivée par le désir de comprendre les mouvements du Soleil, de la Lune, des planètes et des étoiles visibles. Au 20ème siècle, comprendre les dynamiques de regroupements global des systèmes solaires est devenu un problème important à N corps. Le problème à N corps en relativité générale est considérablement plus difficile à résoudre.

La troisième loi de Newton déclare que toutes les forces entre deux objets existent en magnitude égale et de direction opposées.

$$\mathbf{F}_{A/B} = -\mathbf{F}_{B/A}$$

La loi de la gravitation universelle de Newton déclare que l'amplitude de la force attractive entre deux objets de masse m_1 et m_2, séparés par une distance d, est égale à :

$$F = G\frac{m_1 m_2}{d^2}$$

où $G = 6.674 \cdot 10^{-11}$ N est la constante de gravitation.

Enfin, cette force existe sur la ligne séparant les deux centres de masse de ces objets. Ainsi, appliquée à un objet, la force pointe sur l'autre, et vice-versa par la troisième loi de Newton.

Implémentez le constructeur et les méthodes `Add` et `Update` de la classe `System`. Un `System` est une entité qui contient d'autres entités. L'argument `g` du constructeur représente la constante gravitationnelle, car nous pourrons en venir à utiliser d'autres constantes que celle de Newton. La méthode `Add` est utilisée pour ajouter des entités dans le système, tandis que la méthode `Update` itère sur toutes les paires d'entités et applique des forces opposées à chaque membre.

Vous pourrez tester et visualiser l'exécution de votre code dans l'environnement, voir Scène 6. Two Body et `??`. Si vous avez codé la classe `InfinitySpring`, vous verrez également comment vos règles intéragissent, voir Scène 8. Bouncy Earth. Enfin, si vous avez implémenté la classe `DistanceJoint`, vous serez témoin de la danse de deux corps dans leur lutte perpétuelle pour se rejoindre, voir Scène 9. Magnetism.

Pour aller plus loin

Vous remarquerez que les scènes "Two body", "Three body" et "Bouncy Earth" sont considérablement rétrécies et accélérées. Autrement, nous avons bien peur qu'observer les mouvements des entités de ces scènes en temps réel n'aurait pas grand intérêt. Vous trouverez les facteurs exacts de rétrécissement et d'accélération dans les fichiers respectifs de ces scènes, comme arguments au constructeur de base de `Scene`.

Cependant, les coordonnées et mesures affichées sur la visualisation sont correctes. Ainsi, même si une scène est accélérée, la vitesse et l'accélération des différents objets reste respectivement en m/s et en m/s^2.

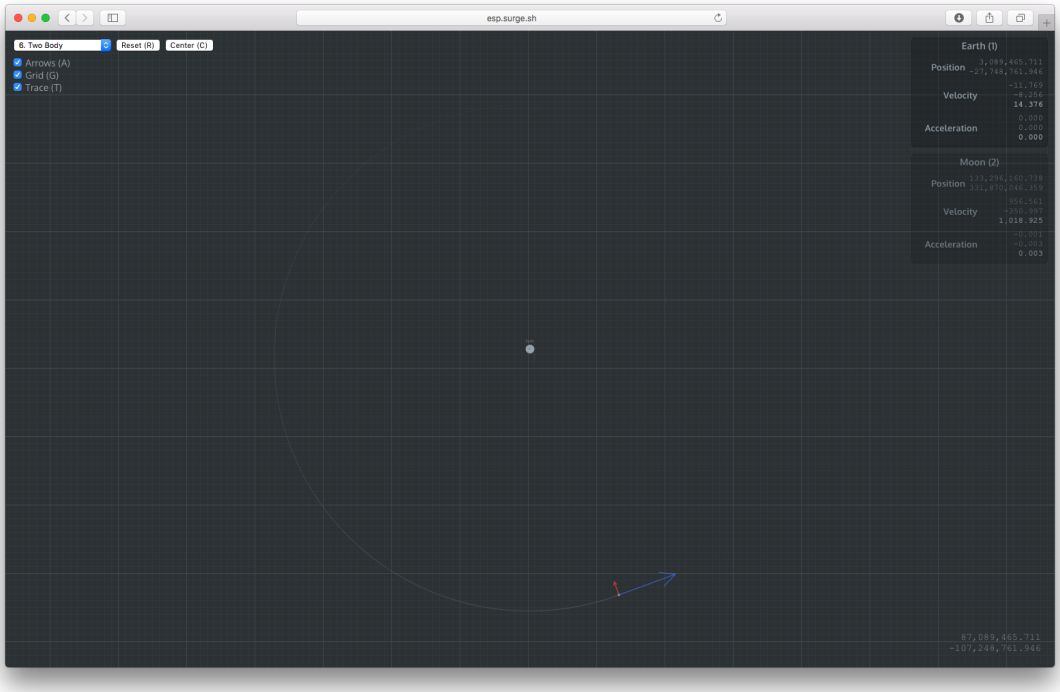

FIGURE 12 – Scène 6. Two Body

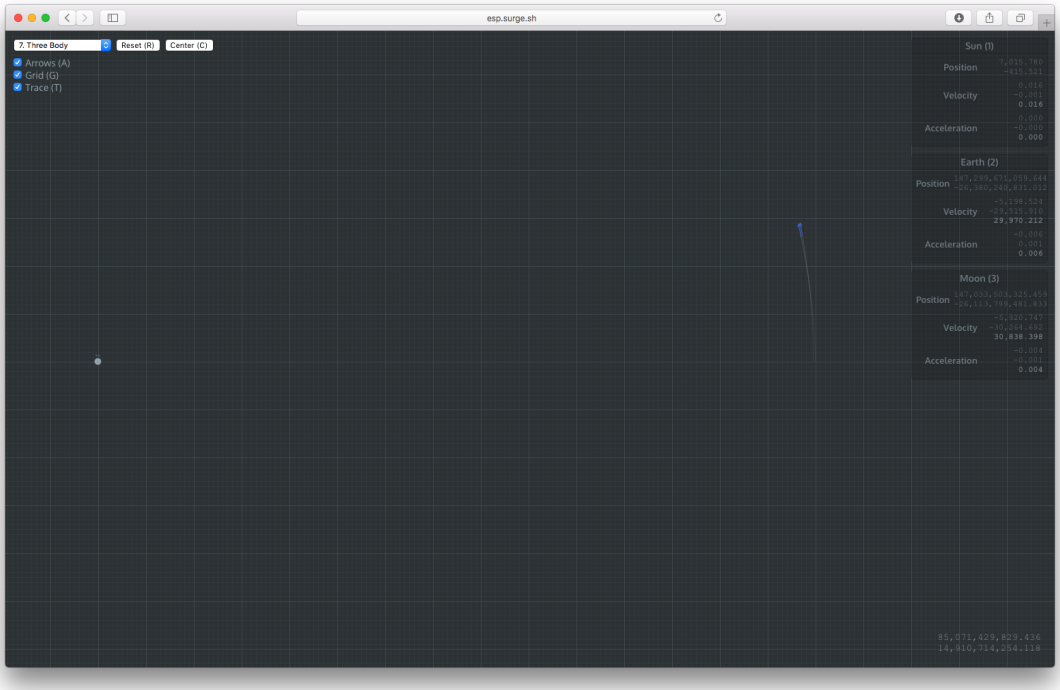

FIGURE 13 – Scène 7. Three Body

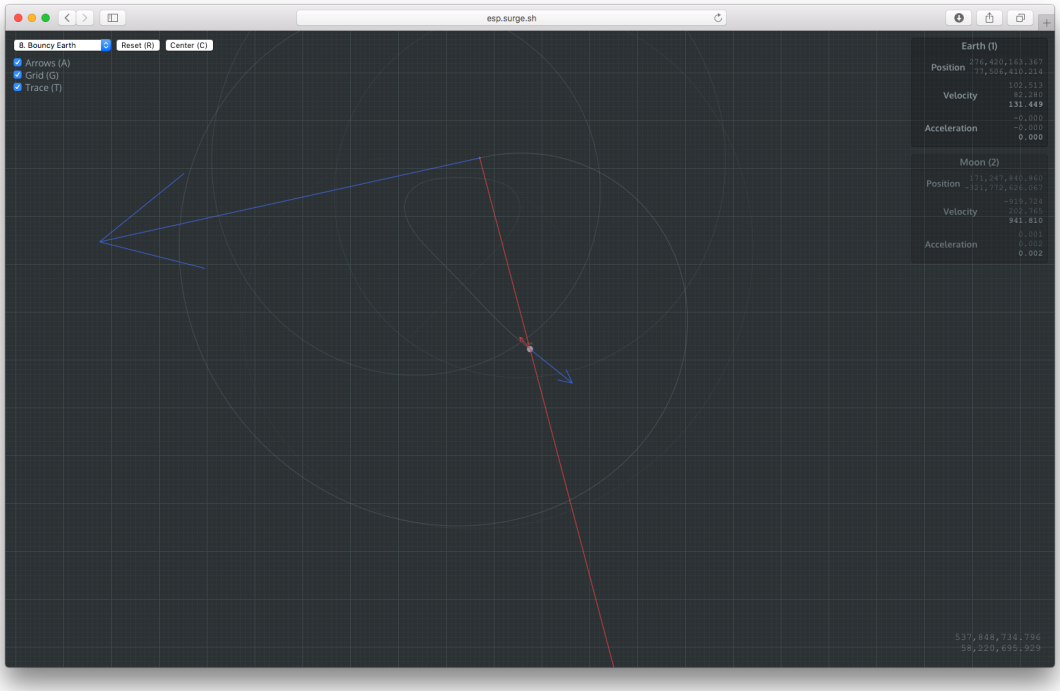

FIGURE 14 – Scène 8. Bouncy Earth

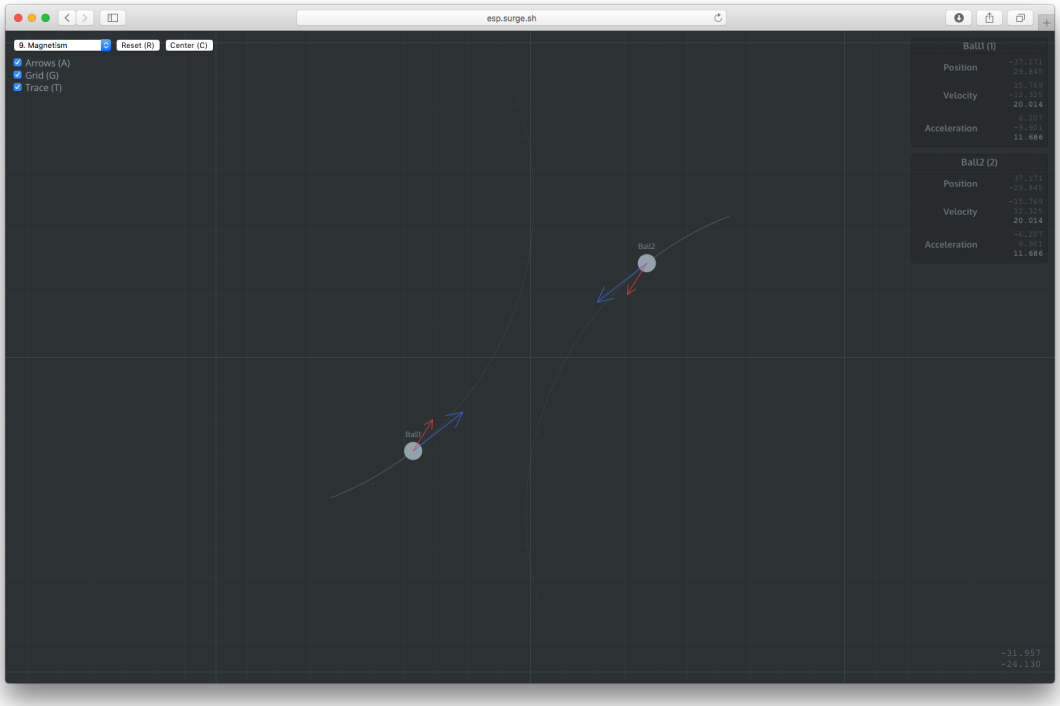

FIGURE 15 – Scène 9. Magnetism

5.2 Bonus 3 : Créer des nouvelles scènes

Une `Scene` est un objet containant plusieurs objets `IEntity` et qui les met à jour. Si vous jetez un oeil à l'intérieur du dossier `Scenes`, vous verrez les définitions de toutes les scènes que vous avez utilisées pour visualiser votre code.

Pour ce bonus, nous vous demandons de créer deux nouvelles scènes. Créer une nouvelle scène consiste à créer une nouvelle classe à l'intérieur du dossier `Scenes`, qui étend la classe abstraite `Scene`, l'instancie et l'enregistre dans la méthode `RunSimulation` de `Program.cs`. N'oubliez pas d'implémenter le constructeur de votre scène, où vous devrez ajouter tous les objets que vous voulez voir appaître sur votre scène.

Vous êtes libre de choisir quelles scènes créer, mais soyez original(e) ! Vous serez noté(e) à la fois sur l'originalité et la complexité de vos scènes. Vous êtes libres de créer un nouveau type d'entités, dont vous pourrez manipuler la vitesse et la position directement comme vous le souhaitez dans la méthode `Update` — mais nous préférerions que vous utilisiez la méthode `ApplyForce` pour des mouvements plus naturels.

Des exemples de scènes que vous pourriez implémenter :
- L'ensemble du système solaire, avec les satellites de toutes les planètes et un nouveau parc d'attraction orbitant autour de la Terre
- De meilleurs types de contraintes (*cf.* Bonus 1 : Un pendule simple). Par exemple, vous pourriez implémenter les joints et essayer de reproduire l'effet papillon : deux systèmes aux conditions initiales très proches finissant par s'éloigner. Vous pourriez aussi implémenter les collisions en s'assurant que deux objets sont écartés lorsqu'ils se croisent.
- Un système binaire d'étoiles.
- *etc.*

These violent deadlines have violent ends.