TP C# 9: EPITA Space Program

Assignment

Archive

You must submit a zip file with the following architecture :

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

- You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
- The code **MUST** compile.
- In this practical, you are allowed to implement methods other than those specified. They will be considered bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where `$` represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal:

```
echo "* firstname.lastname" > AUTHORS
```

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

Contents

1 Introduction **4**
1.1 Foreword . . . 4
1.2 Objectives . . . 4

2 Lesson **4**
2.1 Interfaces . . . 4
2.1.1 Using interfaces . . . 6
2.1.2 Differences between abstract classes and interfaces . . . 7
2.2 Operator overloading . . . 7

3 Exercise: Vector2 **10**
3.1 Operator overloading . . . 11
3.2 Norm of a vector . . . 11
3.3 Normalized vector . . . 12
3.4 Distance between two vectors . . . 13
3.5 Vector2.Zero . . . 13

4 Exercise: Simulating kinematics **13**
4.1 Understanding the simulation environment . . . 15
4.2 A refresher in kinematics . . . 16
4.3 Approximating the instantaneous rate of change . . . 16
4.4 The simulation loop . . . 18
4.5 Newton's second law . . . 19
4.6 The gravity of the situation . . . 19
4.7 Bonus 1: A simple pendulum . . . 21
4.8 The harmonic oscillator . . . 23
4.9 The damped harmonic oscillator . . . 24
4.10 Describing a circular motion . . . 25
4.10.1 The dephased harmonic oscillator . . . 25
4.10.2 A composite body . . . 26
4.10.3 Circular motion . . . 27
4.10.4 Bonus 2: To infinity and beyond . . . 29

5 Exercise: EPITA Space Program **30**
5.1 The N-body problem . . . 30
5.2 Bonus 3: Creating new scenes . . . 34

List of Figures

1 Scene 0. Test . . . 14
2 Error Screen . . . 15
3 Scene 1. Gravity . . . 21
4 Scene 1.1. Pendulum . . . 22
5 Scene 2. Spring . . . 24
6 Scene 3. Damper . . . 25
7 Scene 4.1. Circling Demo . . . 26
8 Scene 4. Circling . . . 28
9 Scene 4.2. Circling Complex . . . 28
10 Scene 5. Infinity . . . 29
11 Scene 5.2. Infinity Complex . . . 30
12 Scene 6. Two Body . . . 32
13 Scene 7. Three Body . . . 32
14 Scene 8. Bouncy Earth . . . 33
15 Scene 9. Magnetism . . . 33

1 Introduction

1.1 Foreword

Oh glorious day! The moment you have been waiting for your entire life has finally come. At long last, the mad physics skills you have gathered for the past decade will finally come to fruition. Fret no more, for this was no vain effort!

Your journey will take you to the magical lands of kinematics and dynamics, employing such wonders as Mass-Spring-Damper systems and Newton's Law of Universal Gravitation.

More importantly, you will be provided with a way to express your creativity as you have never been able to before. The laws of physics are yours to overhaul. Whole galactic systems will bend to your will!

1.2 Objectives

During this practical, you will make heavy use of the Object-Oriented Programming (OOP) notions you have learned in the previous practicals. You will have the opportunity to write deep inheritance chains that make sense, and hopefully develop an intuitive sense of what inheritance is good for, and in which areas it falls short.

This practical will also teach you advanced C# OOP notions such as *interfaces* and *operator overloading*.

Moreover, for the first time, you will be able to visualize the execution of your code in *real time* thanks to the visualisation tools that will be provided to you.

2 Lesson

2.1 Interfaces

You have already learned a bit about inheritance, classes and abstract classes in a previous practical. For a good reminder on the subject, you can check out **TP C# 6: WestWorld Tycoon** as well as the C# doc on the subject.

Interfaces are a way to define the signature of methods that a class should *implement*, without actually implementing them. The idea of interfaces is to share behaviors *horizontally*, between classes that might have no common ancestors, at any level of the inheritance tree. On the other hand, class inheritance is a way to share behaviors *vertically*, from top to bottom.

Another way to think about it is that interfaces represent *contracts* that classes must respect.

```
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 }
```

For instance, in the example above, the `Star` class *extends* the `Shape` class, but *implements* the `ITranslatable` interface. The `Point` class does not extend any super-class, but *implements* the `ITranslatable` interface.

Tip

You'll note that, just like with abstract methods, methods declared in an interface do not have a body (`{ ... }`). Instead, declarations are directly followed by a semicolon.

This means that we're just providing the *prototype* or *signature* of the method and not its actual *implementation*, which we leave out for the actual classes that implement our interface.

Going further

The `I` prefix in the name of an interface is a convention. For instance, we could have called the `ITranslatable` interface `Translatable` instead, nothing in the language itself prevents us from doing so. However, in order to make it clearer that we are handling an interface and not a class, developers prefer to explicitly differentiate them through naming conventions.

Most interface names also end with "-able", as a way to indicate that an interface merely represents what a class can do.

If you're interested to learn more about C# naming conventions, you can read the official recommendations on the subject.

2.1.1 Using 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 }
```

In the above example, we aim to apply a translation to a set of objects. Regardless of their class of origin, all we know about them is that they implement the `ITranslatable` interface. As such, they provide a `Translate` public method that accepts a `Point` object as the sole argument.

As you can observe, in the `foreach` loop, we consider elements of the `translatables` list to be of type `ITranslatable`. Indeed, once we add an element of type `B` to a list of elements of type `A`, with the prerequisite of the type `B` inheriting from (class) or implementing (interface) the type `A`, its class of origin is lost to us. All we now know is that it can be considered as an

element of type `A`.

That is why, when we iterate over the elements of `translatables`, we consider them all to be of type `ITranslatable`, instead of their types of origin (`Star` and `Point`). The only method we can call on them (except for methods inherited from `System.Object`, such as `ToString`) is thus `Translate`, since `ITranslatable` declares no other method.

Going further

You can still manually cast an object of an interface or super-class type to the type of a class that implements or extends it using the explicit cast operation:

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

Be aware that this is **extremely unsafe**, as there usually is no guarantee that the object you are *downcasting* is, in fact, of the correct class. There are still ways to retrieve the original type of an object with absolute certainty, but we will not go over them in this practical.

2.1.2 Differences between abstract classes and interfaces

If you have followed the course on abstract classes closely, you might be asking yourself about the difference between abstract classes declaring abstract methods and interfaces. One such difference is that abstract classes can actually *implement* methods instead of just declaring them with the `abstract` prefix. Abstract classes can also declare and assign attributes, while interfaces only declare methods.

An advantage of interfaces is that a class can implement any number of interfaces. However, since C# does not support multiple inheritance, a class can only inherit from a single super-class.

Finally, to reiterate, there is a semantic difference between classes and interfaces. Class inheritance represents vertical relationships while interface implementation represents horizontal relationships.

2.2 Operator overloading

If you remember your OCaml days (sweet, sweet days), you might recall being able to declare *infix* functions, that is, functions that you can call using the infix form.

```
(*
Let us define the `<~>` infix operator that returns the difference between
two integers.
*)
> let (<~>) a b = abs (a - b);;
val ( <~> ) : int -> int -> int = <fun>
> 1319 <~> 1277;;
- : int = 42
> (<~>) 1319 1277;; (* Force the prefix mode *)
- : int = 42;;
```

Wouldn't it be cool if we were able to do the same in C# ?

Well, we can't. Nada. Not possible. End of the argument. OCaml was too pure for this world.

However, what we *can* do is *overload* existing operators to assign them to custom operations.

Say we have a `Vector2` type, for example. This type represents a vector in two-dimensional space. A point, if you will, with coordinates X and Y. In physics and mathematics, we have grown accustomed to manipulating this exact kind of object using the +, -, *, /, *etc.* operators, just as we would for numbers. However, in C# , we have only been using those operators with actual number types so far (think `int`, `float`, *etc.*).

C# allows for *operator overloading*, which is the practice of assigning a custom operation to a particular infix operator between two types. For instance, if we wanted to be able to add two `Vector2` objects together, we would need to overload the `+` operator on the `Vector2` class.

The syntax is as follows:

```
public class Vector2
{
    public double X { get; }
    public double Y { get; }

    public Vector2(double x, double y)
    {
        X = x;
        Y = y;
    }

    // Unary minus: `-v`.
    public static Vector2 operator -(Vector2 v)
    {
        return new Vector2(-v.X, -v.Y);
    }

    // Addition: `v1 + v2`.
    public static Vector2 operator +(Vector2 v1, Vector2 v2)
    {
        return new Vector2(v1.X + v2.X, v1.Y + v2.Y);
    }

    // Scalar multiplication: `42 * v`.
    public static Vector2 operator *(double factor, Vector2 v)
    {
        return new Vector2(factor * v.X, factor * v.Y);
    }
}
```

Our little *Vector2* class is starting to become really useful, allowing us to represent operations such as vector negation, addition, and scalar multiplication in C# just like we would in the mathematical language.

Going further

As you can see, the different operands of an overloaded operator can be of different types (*cf.* the vector scalar multiplication). The order of the parameters is important, as it represents the order of the operands. As such, in the scalar multiplication we implemented, the scalar factor should always be first. If we wanted to be able to put the scalar factor last (*eg* `v * 42`), we'd need to implement `operator *(Vector2 v, double factor)`.

When overloading operators, at least one of the arguments should be of the class where the operation is defined. In our case, at least one argument should be of type `Vector2`.

3 Exercise: Vector2

Important

The boilerplate of the practical's exercise is available on the assistants' website.

The project already contains all the classes you'll need for the exercises, excluding the bonuses. It contains the following folders and files:

- `ACDC`: This folder contains the code of the simulation program. It **must not** be tampered with in any way. It will be automatically replaced during the correction. As such, any change you make there is void and will most likely result in your project failing to compile.
- `Entities` and `Scenes`: Those folders contain the skeleton of classes you'll have to implement.
- `Vector2.cs`: This file should be completed as part of the first exercise.
- `Program.cs`: This file is the main entry point of the program. Its purpose will be explained later.

Remember that `Vector2` class we were talking about? Now has come the time to implement it properly.

Open the `Vector2.cs` file in your editor. It is situated at the root of the project. As you can see, it already contains the definition of most of the operations and functions you'll need to implement.

Warning

Functions defined but left for you to implement usually throw a `NotImplementedException` in their body, like the following:

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

You should **always** remove this exception once you actually implement the function. The purpose of this line of code is to communicate the fact that something has been left undone and should be worked on later.

Similarly, `// TODO` and `// FIXME` comment lines are only temporary and should be removed once the function they are documenting is properly implemented.

Warning

In this document, when a function declaration is followed by a semicolon (`;`) and not a body (`{ ... }`), it means that we are only specifying the *prototype* of the function.

Unless you are writing an interface or an abstract method in an abstract class — which you won't have to do in this practical — you should never put a semicolon after the declaration of a method. This will probably cause a syntax error and prevent you from compiling your code.

3.1 Operator overloading

In this exercise, you will have to implement useful operators to manipulate your `Vector2` objects.

Warning

Infix operators should **never** mutate any of their operands. In this exercise, this means you shouldn't ever modify the coordinates of the `Vector2` objects you take as input. Instead, you should return a new `Vector2` object that is the result of the operation.

Another way to think about it is that when you add two numbers, `a + b`, neither `a` nor `b` are modified. Instead, the operation returns a new number that is the sum of both.

Subtraction

```
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);
```

Examples

```
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 Norm of a vector

The euclidean distance, or L^2 norm of a vector $\mathbf{u}$ of coordinates (x, y), is defined as the square root of the sum of the square of its coordinates.

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

or

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

Tip

A **bold variable** in an equation represents a vector.

Going further

This definition generalizes to higher-dimensional vectors, such as vectors in the three-dimensional space, and beyond.

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

Now implement the `Norm` method that returns the L^2 norm of a `Vector2`.

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

Examples

```
new Vector2(0, 0).Norm() // 0
new Vector2(1, 1).Norm() // 1.4142135623730951
new Vector2(1, 2).Norm() // 2.236067977499979
```

3.3 Normalized vector

The *normalized vector*, or *versor* of a vector $\mathbf{u}$ is defined as:

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

$\hat{\mathbf{u}}$ is a unit vector with the same direction as $\mathbf{u}$.

Now implement the `Normalized` method that returns the normalized vector of a `Vector2`.

Warning

The `Normalized` method **must not** modify the original vector, but rather return a new vector equal to its normalized vector.

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

Examples

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

3.4 Distance between two vectors

The euclidean distance between two vectors **u** and **v** is defined as the L^2 norm of their difference.

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

Expanded, this gives the following form, with which you're probably much more familiar:

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

Now implement the `Dist` static method that takes two `Vector2` objects and returns the value of the euclidean distance between them.

```
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

We'll make great use of the zero-vector **0** of coordinates $(0, 0)$ in the rest of the practical.

For convenience, define a static attribute `Zero` on the `Vector2` class that is equal to the zero-vector.

Going further

Since none of the methods of a `Vector2` object can mutate its coordinates, our `Vector2` data structure is completely *immutable*. That means, once you have a `Vector2`, you can not modify it directly.

As such, if you want to obtain a new *Vector2*, you need to either construct a new one from scratch or combine other vectors using the operators you have overloaded.

This property ensures that `Vector2.Zero` will always retain the same coordinates.

4 Exercise: Simulating kinematics

The second (and most interesting) part of this practical is dedicated to simulating physical bodies.

For this purpose, we have developed a state-of-the-art simulation environment that aims to recreate real-life physics with extreme precision. Well, tried to anyway.

When testing your `Vector2` implementation, you might have noticed that `RunSimulation` method in the `Program` class. Now is the time to call it!

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

Now run the program. If all goes well, you will notice a link is printed to the console. Click on it!

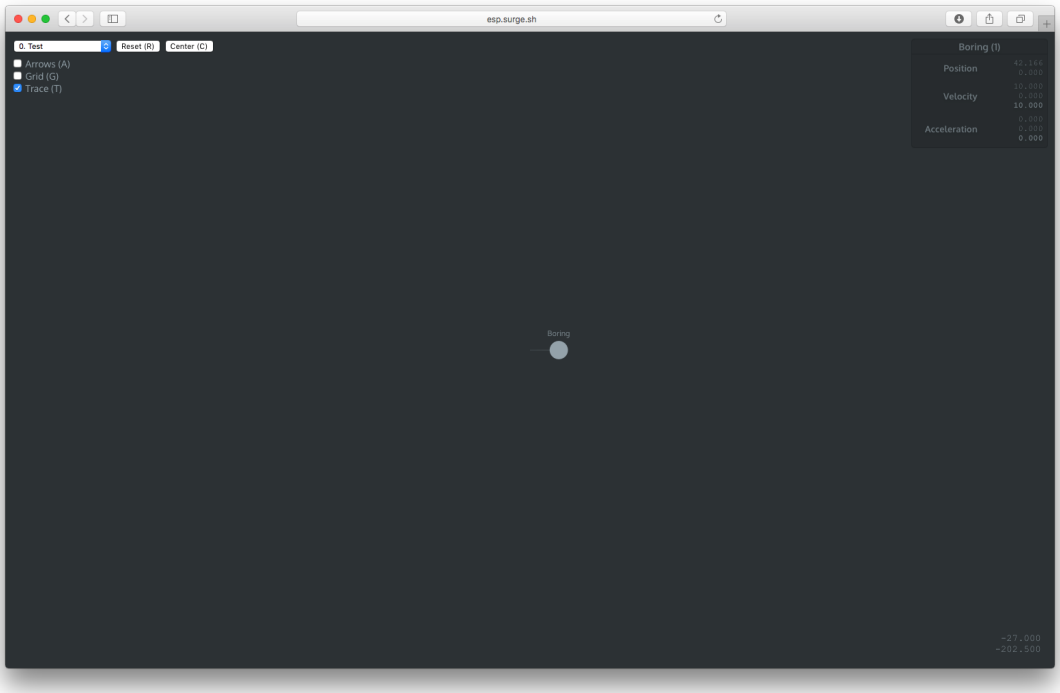

Figure 1: Scene 0. Test

You should see something akin to Scene 0. Test. This is the simulation environment. You will notice a few interface elements:

- The **white circle** annotated "Boring" represents a physical body. You'll eventually get to create your own!
- The **fading white line** represents the trajectory of the physical body.
- The scene **dropdown** lets you select a scene among a selection of pre-crafted scenes. Note that only the scene titled "0. Test" will show a body moving at all. The others will come to life as you implement the different entities in the `Entities` folder.
- The **Reset** button lets you reset the current scene. All bodies will go back to their initial position, velocity and acceleration.
- The **Center** button moves the camera back to the center of the scene, at coordinates $(0, 0)$.
- The **Arrows** checkbox lets you toggle the display of the velocity (in blue) and acceleration (in red) vectors for the different bodies of the scene. The only body in the "Test" scene

has no acceleration and a non-zero initial velocity. As such, only its velocity vector will show. The magnitude of the velocity and acceleration vectors represent, respectively, what distance the body will travel during one unit of time (second), and what speed the body will gain during the next unit of time.

- The **Grid** checkbox lets you toggle the display of the grid. This grid is here to help you situate bodies relative to the environment. Thin lines are drawn every 10 units of distance (think meters), while bold lines are drawn every 100 units of distance.
- The **Trace** checkbox lets you toggle the display of bodies' trajectories.
- On the right, you will find information about all the bodies in the current scene. Clicking on the information of a body will **follow** it by keeping it at the center of the screen at all times.
- In the bottom right corner of the screen are displayed the coordinates of the position of the mouse **relative to the scene**.

You can **zoom** using your mousewheel or by scrolling on your touchpad, and you can **pan** by clicking anywhere on the screen and moving your mouse pointer.

Warning

If the console doesn't print a link when you run the project, or if you see Error Screen when clicking on the link, please contact an assistant.

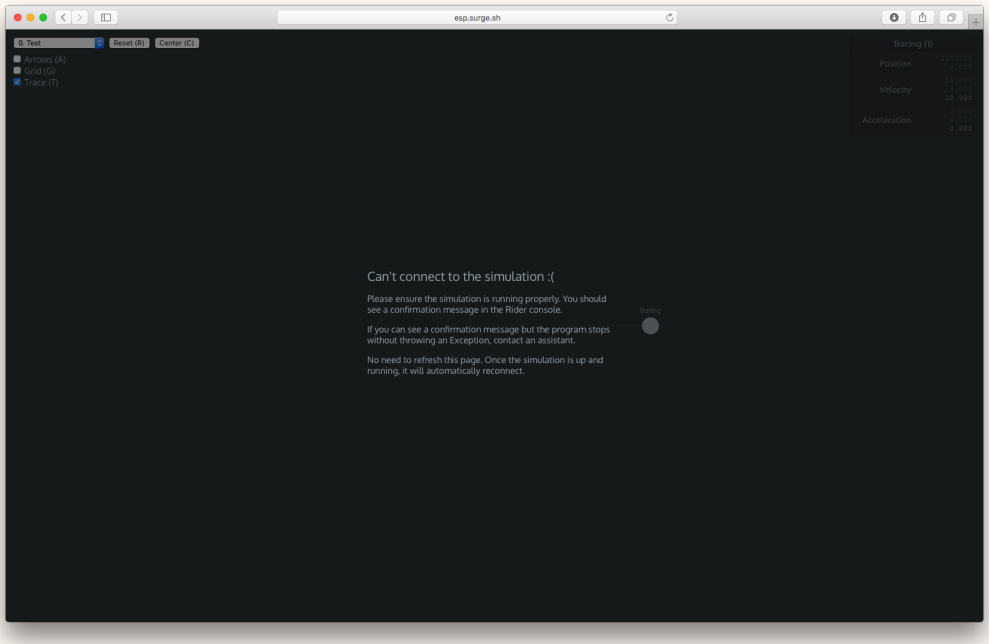

Figure 2: Error Screen

4.1 Understanding the simulation environment

In order to get a hang on the simulation environment, we recommend you fiddle with the `TestBody` and `TestScene` classes. You will find more explanations of how they work in their

source files.

Make sure to take a few minutes to familiarize yourself with the simulation environment. Understanding how it works and what everything represents is important for the rest of the practical.

4.2 A refresher in kinematics

Tip

To refresh your memory:

- A **physical body** is just an object in space that has mass and a position. You are a body, your computer is a body, your assistants are (nice) bodies.
- **Velocity** is a measure of the change of a body's position over time. In this practical, velocity is measured in m/s, that is, by how much the position of a body (in meters) changes over one unit of time (a second). Since we're dealing with two-dimensional space, our velocity is actually a two-dimensional vector. The **magnitude**, **length**, or **norm** of this vector represents what we traditionally refer to as **speed**, while the **direction** of this vector represents the direction in which the body is moving.
- **Acceleration** is a measure of how much speed a body gains over time. In this practical, acceleration is measured in m/s^2, that is, by how much the velocity of a body (in m/s) changes over one unit of time (a second). This is a harder concept to grasp than velocity, but a simple way to think about it is that acceleration is to velocity what velocity is to position. And just like our velocity, our acceleration is a vector.
- A **unit of time** in this practical will refer to a second. Technically, it does not matter whether it is a second, an hour, or any arbitrary time delta (if any). However, to make things more natural and in order to provide a meaningful visualisation, we have chosen the second as the unit of time of choice in this practical. Similarly, a **unit of distance** in this practical will refer to a meter (but will be represented as one or more pixels). The same comment can be made as to the arbitrary nature of that choice.

If any of these concepts are still unfamiliar to you, please do not hesitate to ask your assistant for a better explanation. This practical is no substitute for an introductory course on the basics of kinematics and dynamics.

4.3 Approximating the instantaneous rate of change

Warning

This section is not necessary to understand in order to complete this practical. However, having a good mental grasp of how all the pieces of the simulation fit together will help greatly in the exercises to come.

Feel free to move on to the next part and come back to this one later on.

Let r be the position of a body, v its instantaneous velocity, and a its instantaneous acceleration. We know from high school kinematics that:

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

We also know from the definition of the derivative that the instantaneous rate of change of a function f at point a is equal to:

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

Now let's say we know $f(a)$ and $\dot{f}(a)$ for some point a, and want to *approximate* $f(a+h)$ for some sufficiently small delta h. Getting rid of the limit, we get:

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

Now, swapping f in the above formula for r, we get:

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

Which tells us that from an initial position r_0 and an initial velocity v_0, we can approximate the position at the next instant $t_1 = t_0 + \Delta t$ to be $(t_1 - t_0) \cdot v_0 + r_0$. We can proceed in a similar manner for the velocity, with $v_1 = (t_1 - t_0) \cdot a_0 + v_0$.

You might ask yourself, what was the point of all that? Since we're trying to simulate real-time physics on a computer with a frequency much, much lower than that of the real world — think $\frac{1}{t_P}$ where t_P is the Planck time, so a frequency of approximately $1.86 \cdot 10^{34}$ GHz distributed over countless particles — we need to make trade-offs in the precision of our simulation. As such, the above formula lets us approximate changes in position and velocity using the actual tick rate of our computer as our time delta. Obviously, since this is an approximation, our error increases greatly over time. But for the purpose of this simulation, our accuracy is more than enough.

The actual logic for updating the simulation can be found in `Scene.Update`, where the delta time is computed using the elapsed ticks between the current update and the previous update:

```
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  }
```

As well as in `Body.Update`, where we update the position and velocity of the body according to the approximations we found above:

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

Going further

While this all might seem excessively tedious to some of you, it actually has applications in software you use daily. Indeed, your favorite game's physics engine implements this very approximation. For those of you who chose Unity as their game engine of choice (or lack thereof), the above explanation might help alleviate some of the confusion about the game loop, about the difference between `Update` and `FixedUpdate`, and about the meaning of `Time.deltaTime`. It might even provide the beginning of an answer as to why scaling time up introduces serious precision errors.

4.4 The simulation loop

The way our simulation works is that each and every simulated object gets updated at a frequency defined in `Constants.cs`. Indeed, all the objects of our simulation *implement* the `IEntity` interface, which declares an `Update` method.

During each update, the velocity of each object is updated using its acceleration, then its position is updated using the new value for its velocity. The acceleration is then reset to 0.

What we ask of you during the following exercises is to implement the `Update` methods of a few different kinds of entities. In those `Update` methods, you will come to affect the instantaneous acceleration of your objects in order to apply a change of velocity, and thus of position, over time.

You might ask, why not directly change the position following the parametric equation of a kind of movement?

Tip

A parametric equation of movement is an equation of the form:

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

where the position r of an object depends directly upon the current instant t. As such, at each timestep t_i, all we have to do is recompute $r(t_i)$ and update the position of the object instantaneously and precisely.

Using parametric equations to describe a movement certainly is the simplest (and most precise) option. However, as soon as our movement gets more complex, or involves a variety of bodies, our equations can get extremely complicated.

You can find an example of using a parametric equation to update the position of a body as a comment in the `TestBody` class, instantiated in the `TestScene` class.

4.5 Newton's second law

Newton's second law states:

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

where $\sum \mathbf{F}$ represents the sum of all forces applied to a body in Newton ($\mathrm{kg} \cdot \mathrm{m/s^2}$), m is the mass of the body in kilograms, and $\mathbf{a}$ is the acceleration vector of the body in $\mathrm{m/s^2}$.

Moving terms around, we find:

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

This basically says that applying a force to a body will change its acceleration by a factor proportional to the force itself, and inversely proportional to the mass of the body. As such, heavier bodies will require more force to move, while increasing the magnitude of the force will also move the body faster. Finally, applying a force to a body will accelerate the body in the direction of the force.

Write the `ApplyForce` method inside of `Body.cs`. This function applies a discrete force $\mathbf{F}$ to the object using the following formula.

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

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

Warning

Keep in mind that an object might call `ApplyForce` multiple times during a single update step. The desired behavior is then to apply a composite force, and not just the force of the last call to `ApplyForce`. This is the reason why you should *sum* the new acceleration with the previous one, as not to cancel the effects of previously applied forces. This sum is already part of the above formula.

Tip

Remember that the acceleration of an object is reset at each game loop (at the beginning of an update step). As such, the first time an object calls `ApplyForce` during an `Update` step, the acceleration will initially be null.

Don't overthink this method. Remember the operators you wrote for the `Vector2` class. The actual body of the function should be no longer than a single line of code.

4.6 The gravity of the situation

As we have studied it in class, the force of gravity is a constant force applied to an object. At the earth's surface, the force of gravity has a magnitude of 9.81 N and a direction pointing down to the earth's core. In this exercise, we aim to implement a body with a force of gravity applied to it.

Write the constructor and `Update` method of the `Gravity` class.

Since the `gravity` is a constant vector, and you will need it in your `Update` method, you should store it in an attribute of the `Gravity` class. You will be able to test and visualize your object in the environment, see Scene 1. Gravity.

Tip

Remember to always call `base.Update(delta)` at the very beginning of your `Update` methods.

Warning

You may add any method or attribute you want to the class, as well as to any of the other classes not defined in the `ACDC` folder. However, you should **never** change the signature of an existing method or constructor.

Important

You may **not** modify the position or velocity of an object anywhere else than in its constructor. The whole purpose of this exercise is to get you to simulate natural movement by manipulating acceleration.

Remember the `ApplyForce` method we asked you to write earlier. You'll need it quite extensively!

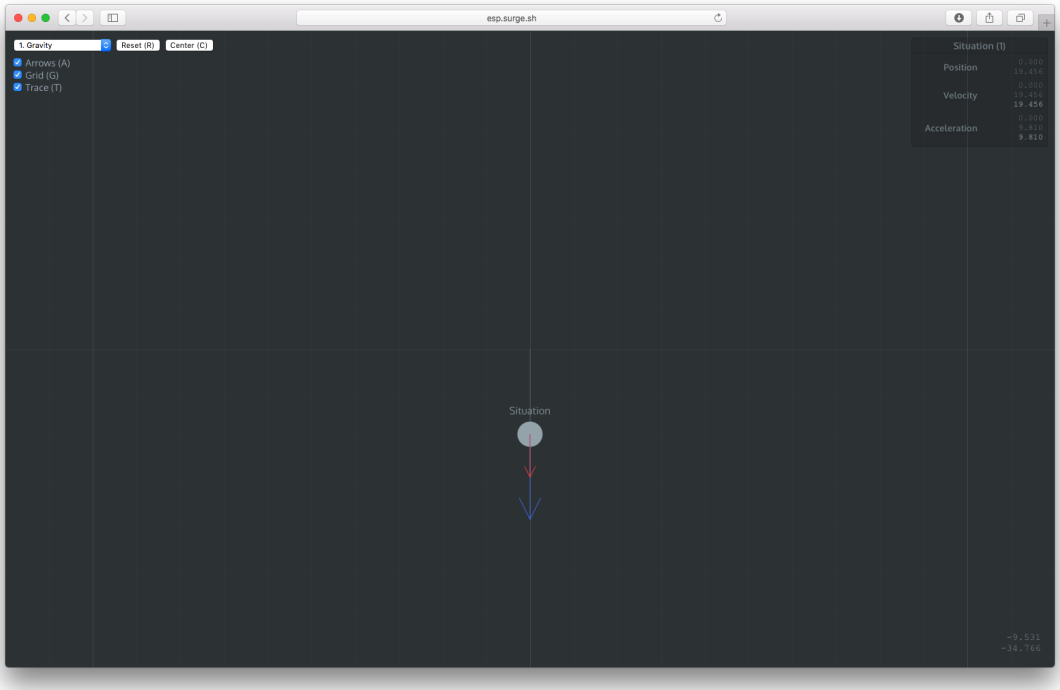

Figure 3: Scene 1. Gravity

4.7 Bonus 1: A simple pendulum

Tip

This practical contains three bonus exercises, scattered through the sections. If you do not wish to implement the bonuses (although we strongly recommend you do), you may move on to the next exercise.

A pendulum is an object attached to a string of constant length.

In order to simulate a pendulum, we need to ensure that our object keeps a constant distance to a particular point, which we'll call the **pivot point**. In short, after applying the different forces to our object, we need to apply an additional force to rectify its position so the constraint is respected. We call this force the **constraint force** $\mathbf{F}_c$.

The formula for the constraint force $\mathbf{F}_c$ is as follows:

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

where $\sum \mathbf{F}$ is the sum of all other forces applied to the object, $\mathbf{r}$ is the position of the object **relative** to the pivot point, m is the mass of the object, and $\mathbf{v}$ is the velocity of the object. The $\cdot$ operator is the **dot product** of two vectors.

You can retrieve $\sum \mathbf{F}$ from the acceleration with Newton's second law:

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

In order to implement the above formula, you'll first need to implement the `Vector2.Dot` static method of the `Vector2` class.

Then, write the constructor and `Update` method of the `DistanceJoint` class. You will not need to call `Update` yourself on the `Body` that is provided in the constructor, that will be handled by the `Scene` automatically. You will be able to test and visualize your object in the environment, see Scene 1.1. Pendulum.

Going further

You might have noticed that our pendulum eventually strays away from its pivot point. Since the constraint force is always calculated relative to the current position of the object, the constraint is progressively loosened as we introduce error.

If you're interested to learn more about constraints in physics engines, or if you wish to implement a better constraint system, you will find a more detailed explanation of constraints in physics simulations in this excellent tutorial, which we've used as reference in the making of this exercise.

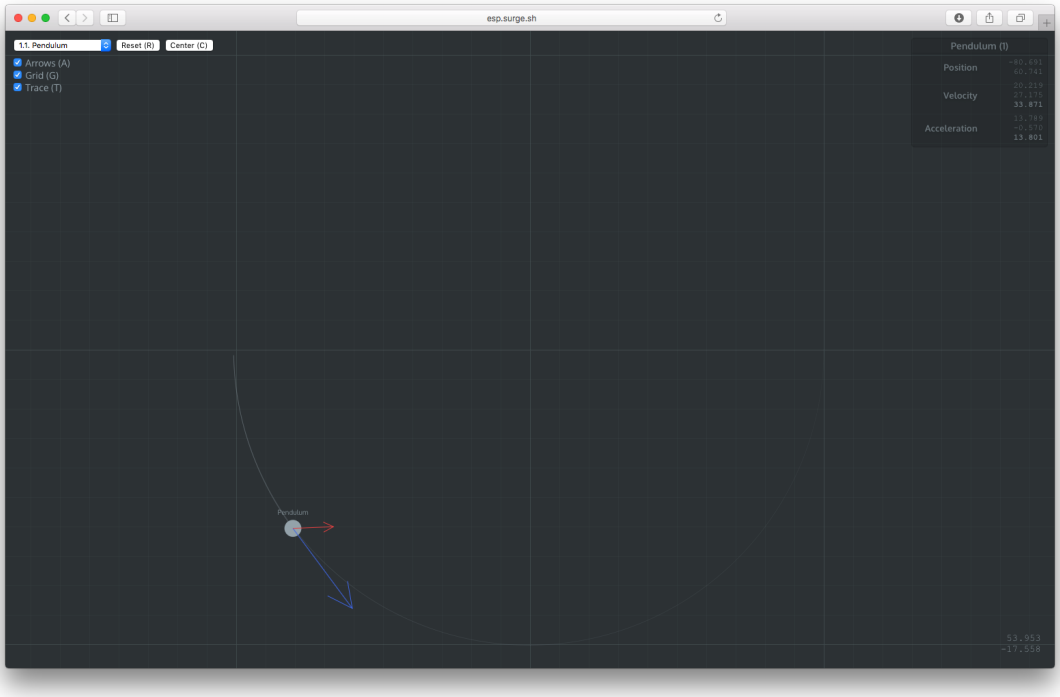

Figure 4: Scene 1.1. Pendulum

4.8 The harmonic oscillator

Wikipedia's definition of a harmonic oscillator

In classical mechanics, a harmonic oscillator is a system that, when displaced from its equilibrium position, experiences a restoring force, $\mathbf{F}$, proportional to the displacement, $\mathbf{x}$:

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

where k is a positive constant.

Now look at the constructor of `Spring.cs`. You'll see that it receives five arguments. You can forget about `name` and `density`, as they are only useful for the visualization. `initialPosition` will be set by `Body`'s constructor as the initial `Position` vector, while `mass` will be set as the constant `Mass`. Both `Position` and `Mass` are getters declared in `Body.cs` that you can access in the methods of `Spring`.

The `Spring` constructor also receives an `origin` vector and a `spring` constant. In the above formula, k represents this `spring` constant, while $\mathbf{x}$ represents the displacement of the oscillator at instant t, which is simply equal to `Position - origin`.

Since `origin` is a constant, but `Position` changes over time, you should store the `origin` vector in an attribute of the `Spring` class, so you may access it inside of your `Update` method. You should do the same with the `spring` constant.

Write the constructor and `Update` method of the `Spring` class. You will be able to test and visualize your object in the environment, see Scene 2. Spring.

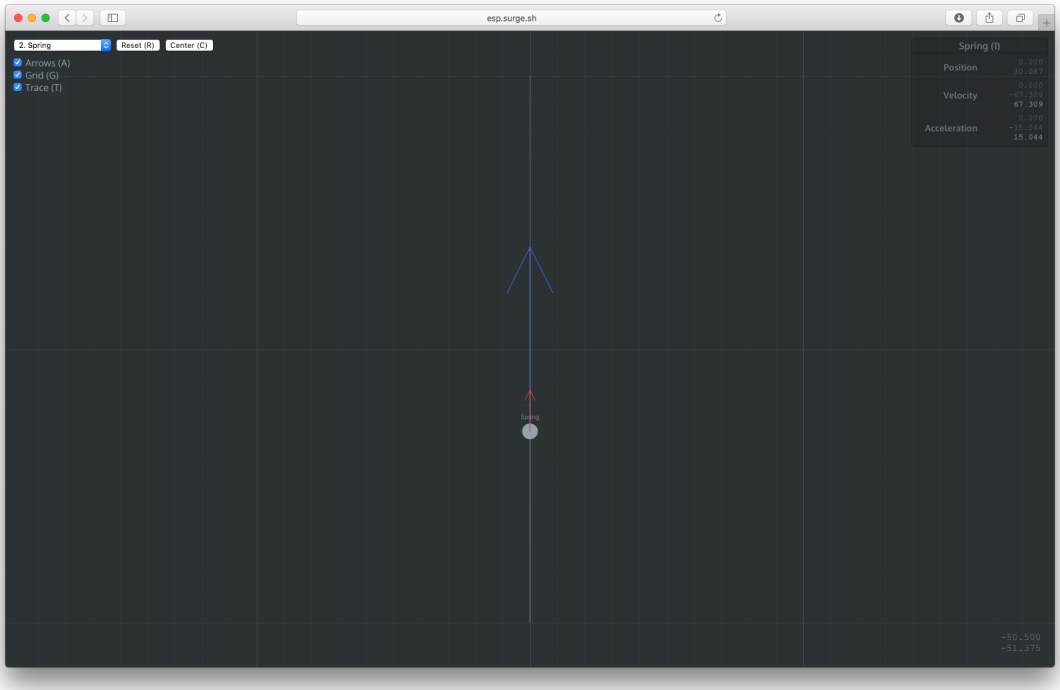

Figure 5: Scene 2. Spring

4.9 The damped harmonic oscillator

A damped harmonic oscillator is a harmonic oscillator with an additional damping force $\mathbf{F}_d$ applied to it :

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

where c is the *damping* constant and $\dot{\mathbf{x}}$ is the derivative of the displacement.

So the composite force applied to our damped harmonic oscillator is:

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

Since we're in a simulation, $\dot{\mathbf{x}}$ can be approximated as:

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

where $\mathbf{x}_1$ is the current displacement and $\mathbf{x}_0$ is the displacement during the last update.

In code, this simply gives:

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

Write the constructor and `Update` method of the `DamperSpring` class. During each update step, you will need to store the current displacement you computed in order to use it in the next update step. You will be able to test and visualize your object in the environment, see Scene 3. Damper.

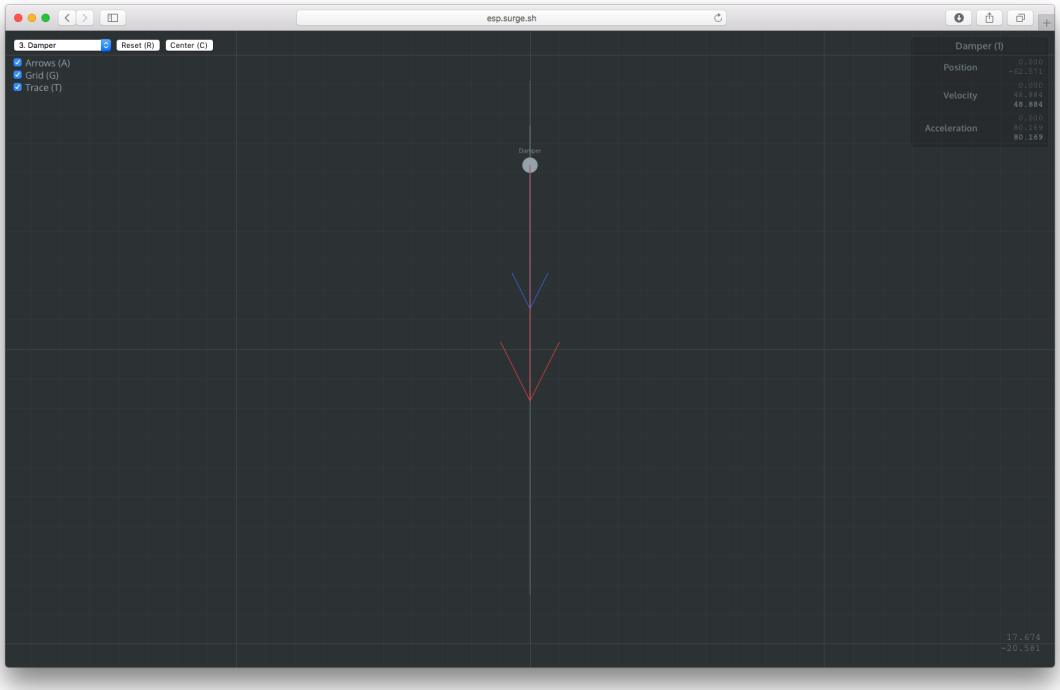

Figure 6: Scene 3. Damper

4.10 Describing a circular motion

4.10.1 The dephased harmonic oscillator

A circular motion can be described as the composition of two orthogonal harmonic oscillators.

To get an intuitive sense of why that is, have a look at this cool animation. The movement of the red and blue dots are those of two harmonic oscillators with a different phase.

The exact phase difference of the two harmonic oscillators is equal to $\frac{\pi}{2}$, which is the phase difference between *cos* and *sin*:

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

An issue we face is that our current harmonic oscillator implementation starts at $t = 0$. As such, all the harmonic oscillators we add to our simulation will all initially be in sync. In order to have a *dephased* harmonic oscillator, its initial velocity and position must be different.

The parametric equation of a harmonic oscillator states:

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

We know that $\varphi = \frac{\pi}{2}$, so substituting it and computing $r(0)$ and $v(0)$ we get:

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

So all we have to do to write our new dephased oscillator is to extend the current oscillator and set a new initial position and velocity according to the formulas above.

Tip

Remember the initial position we found above is relative to the oscillator's origin.

Write the constructor of the `SpringMax` class (which is the oscillator we are talking about). No need to implement its `Update` method : it already inherits it from its `Spring` super-class. You will be able to test and visualize your object in the environment, see Scene 4.1. Circling Demo.

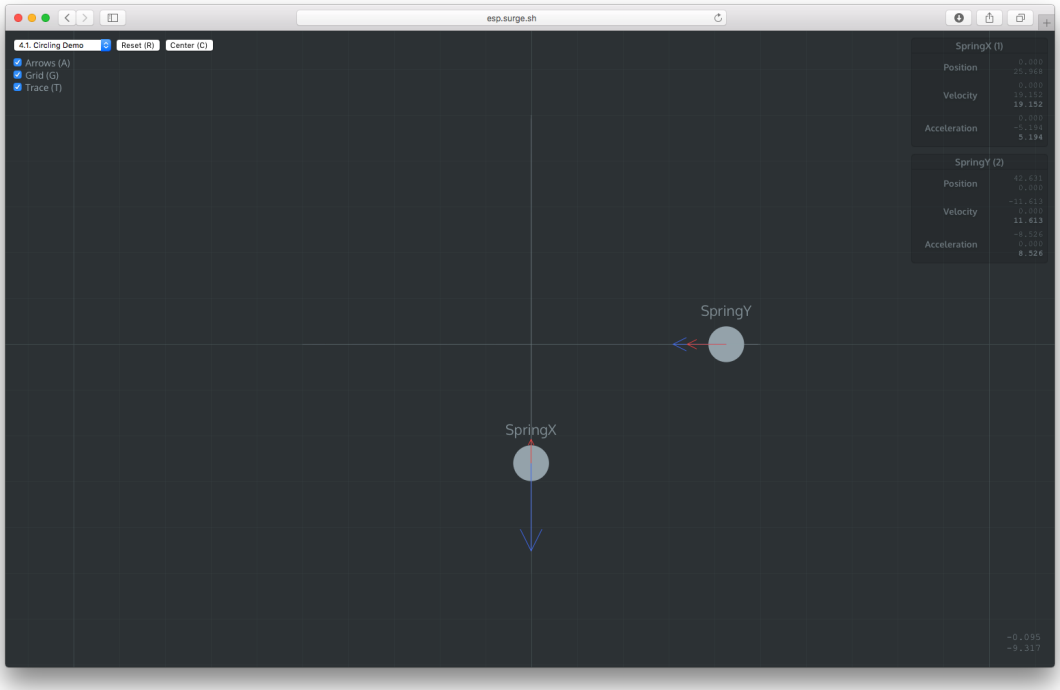

Figure 7: Scene 4.1. Circling Demo

4.10.2 A composite body

Now that we have our dephased harmonic oscillator, we need a way to compose two or more physical bodies.

A `CompositeBody` contains and updates other bodies. At all times, its own acceleration is equal to the sum of the accelerations of its children.

Write the constructor, `Update` method and `Add` method of the `CompositeBody` class.

Tip

When writing the `Add` method, remember to correctly add the initial velocity and acceleration of the children bodies to that of the `CompositeBody`.

The `Add` method will only ever be called in the constructor of sub-classes of `CompositeBody`, so you're perfectly allowed (and encouraged) to do so.

4.10.3 Circular motion

You now possess all the elements needed to describe a circular motion as the composition of a `Spring` and `SpringMax`.

You will need to change the initial position of your `SpringMax` so its direction and initial velocity are correct in all cases. You can take a look at the "4.1. Circle Demo" scene to get an example as to how the `SpringMax` should be offset relative to the `Spring`.

Write the constructor of the `CirclingSpring` class.

Tip

Since `CirclingSpring` is the composition of two bodies, you should instantiate these bodies with the correct parameters in the constructor of `CirclingSpring` and pass them to the `Add` method inherited from `CompositeBody`.

You do not need to implement an `Update` method for the `CirclingSpring` class or for the `InfinitySpring` class below, since the `Update` method of `CompositeBody` already does the work.

You can test and visualize your `CirclingSpring` object in the environment, see Scene 4. Circling and Scene 4.2. Circling Complex.

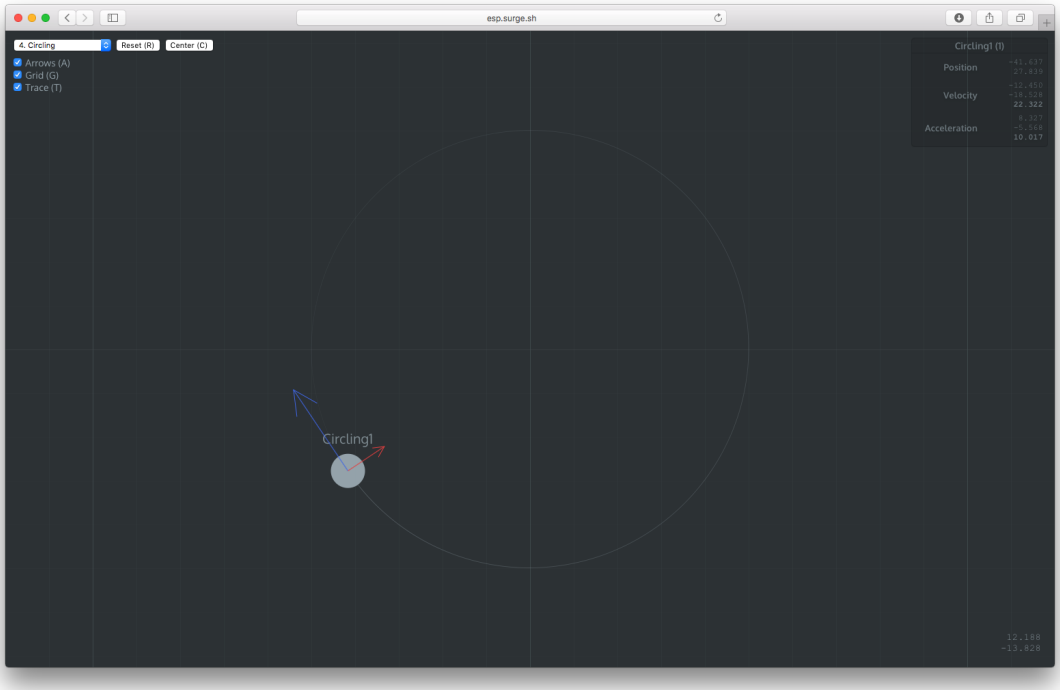

Figure 8: Scene 4. Circling

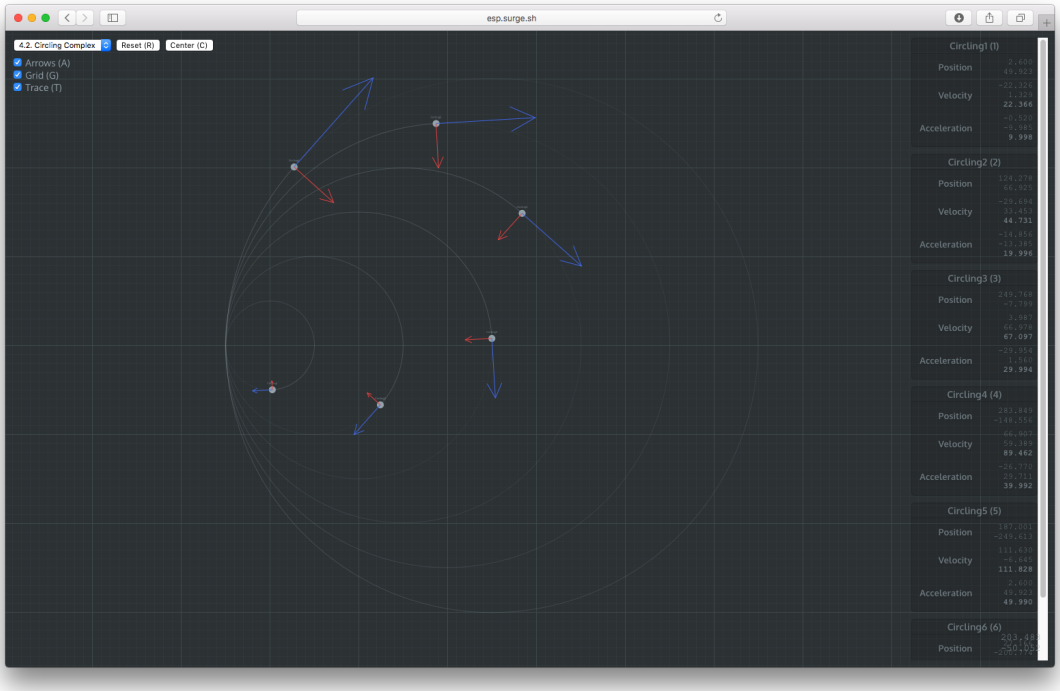

Figure 9: Scene 4.2. Circling Complex

4.10.4 Bonus 2: To infinity and beyond

The infinity motion (∞) is almost exactly the same as the circular motion. However, the `SpringMax`'s amplitude (maximum displacement) should be half that of the `Spring`, while its frequency twice that of the `Spring` (for the same mass). As such, its springness constant should be four times that of the `Spring`. You can take a look at the "5.1. Infinity Demo" scene to get an example as to how the `SpringMax` should be offset relative to the `Spring`.

Write the constructor and `Update` method of the `InfinitySpring` class.

You can test and visualize your `InfinitySpring` object in the environment, see Scene 5. Infinity and Scene 5.2. Infinity Complex.

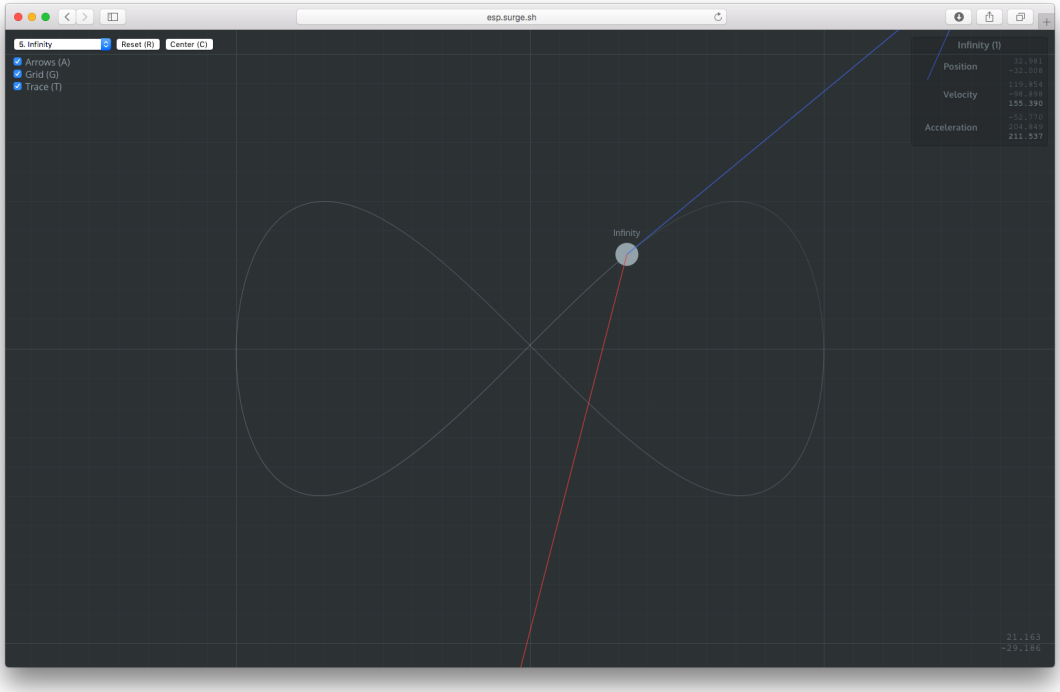

Figure 10: Scene 5. Infinity

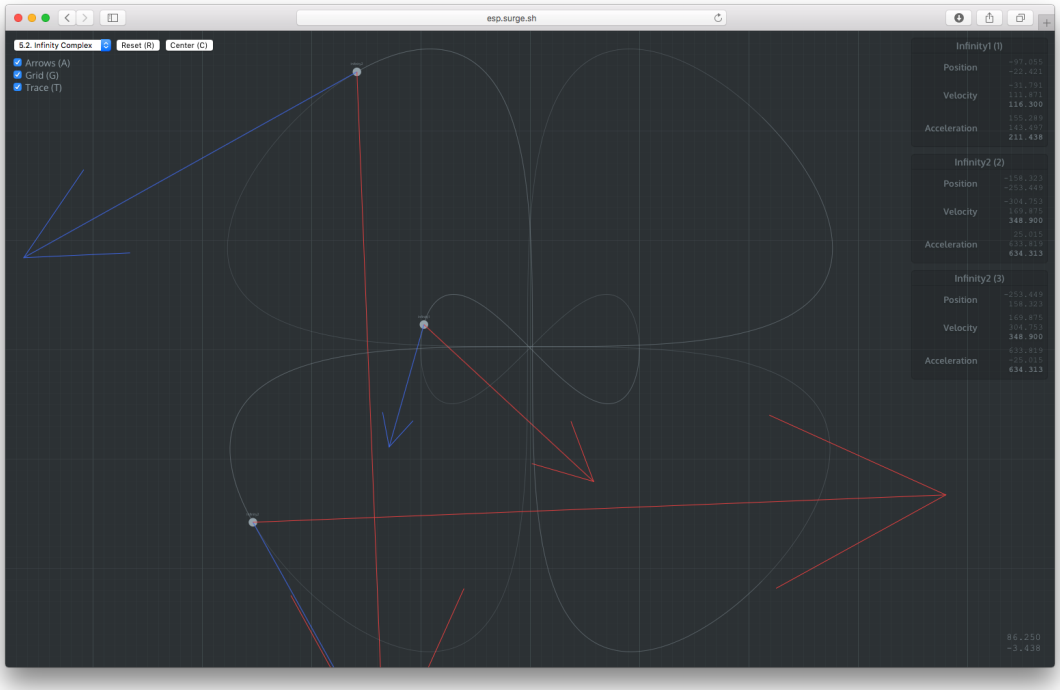

Figure 11: Scene 5.2. Infinity Complex

5 Exercise: EPITA Space Program

Delos Incorporated has mandated you to build and launch a new space-themed park into orbit. As a recent graduate from EPITA, you haven't had much experience launching objects much further than a mouse across a room.

Thankfully, you remember your physics class at EPITA and are confident in being able to complete the task.

5.1 The N-body problem

Wikipedia's entry on the N-body problem

In physics, the n-body problem is the problem of predicting the individual motions of a group of celestial objects interacting with each other gravitationally. Solving this problem has been motivated by the desire to understand the motions of the Sun, Moon, planets and the visible stars. In the 20th century, understanding the dynamics of globular cluster star systems became an important n-body problem. The n-body problem in general relativity is considerably more difficult to solve.

Newton's third law states that all forces between two objects exist in equal magnitude and opposite direction.

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

Newton's law of universal gravitation states that the amplitude of the attractive force between two objects of mass m_1 and m_2, separated by a distance d, is equal to:

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

where $G = 6.674 \cdot 10^{-11}$ N is the gravitational constant.

Finally, this force points along the line going through both of those object's center of mass. As such, applied to one object, the force points to the other object, and vice versa as per Newton's third law.

Write the constructor, `Add` method and `Update` method of the `System` class. A `System` is an entity that contains other entities. The `g` argument in the constructor represents the gravitational constant, as we might want to use other constants than Newton's. The `Add` method is useful in order to add entities to the system, while the `Update` method iterates over all entities pairs and applies opposite forces to each member.

You can test and visualize your code in the environment, see Scene 6. Two Body and Scene 7. Three Body. If you implemented the `InfinitySpring` class, you will also be able to visualize how multiple rules interact, see Scene 8. Bouncy Earth. And if you implemented the `DistanceJoint` class, you will witness the dance of two magnetic bodies as they struggle to meet each other in Scene 9. Magnetism.

Going further

You will notice that the "Two body", "Three body" and "Bouncy Earth" scenes are considerably zoomed out and sped up. Otherwise, we're afraid observing the movements in these scenes in real time wouldn't be much fun. You will find the exact factors of scale and speed in their respective files as the arguments to the base `Scene` constructor.

However, the coordinates and measures displayed on the visualization are accurate. As such, even if the scene is sped up, the velocity and acceleration of the different objects are still in m/s and m/s^2, respectively.

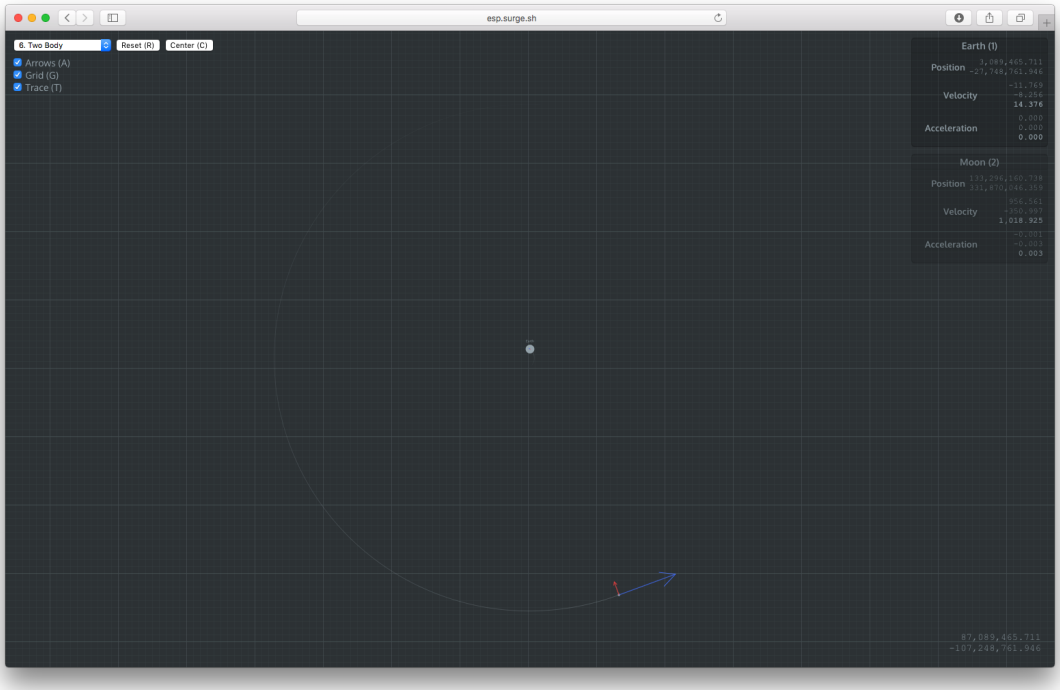

Figure 12: Scene 6. Two Body

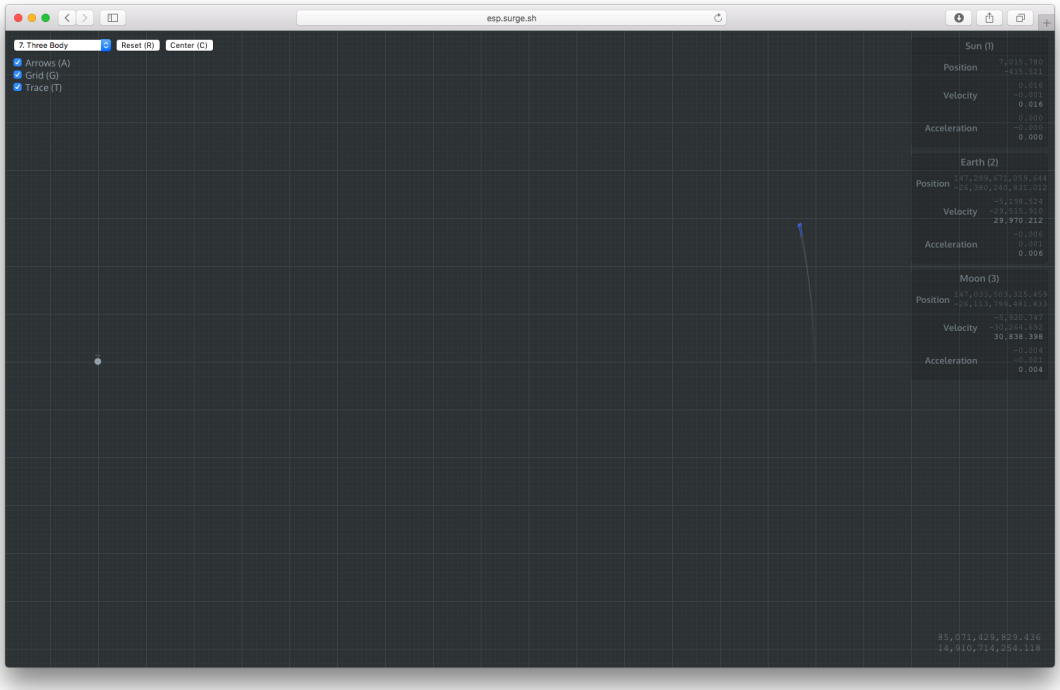

Figure 13: Scene 7. Three Body

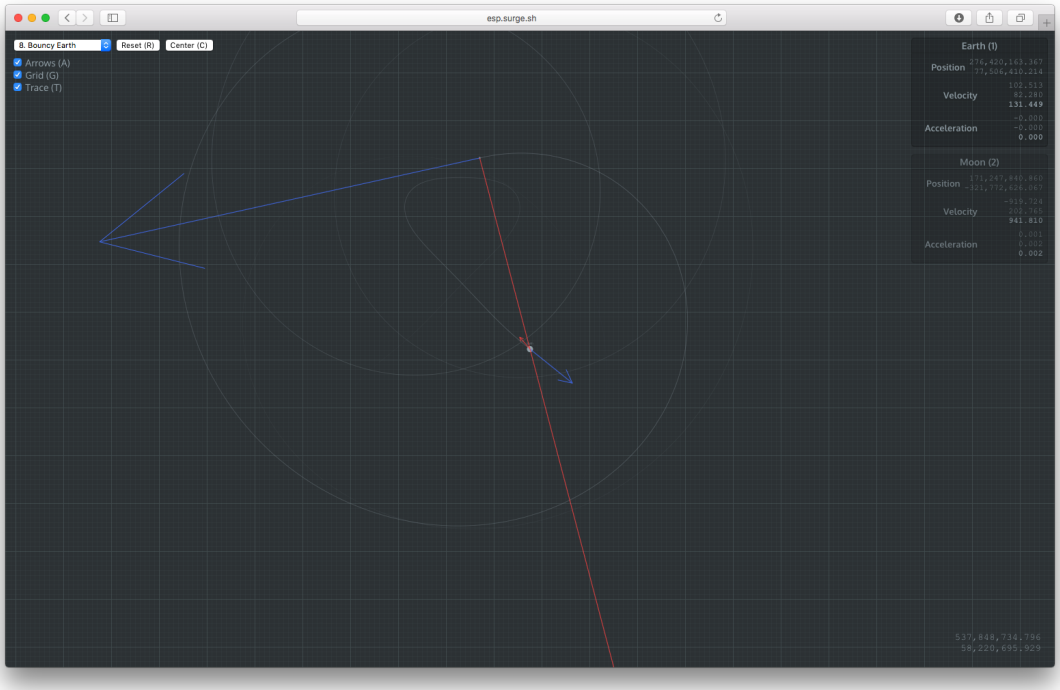

Figure 14: Scene 8. Bouncy Earth

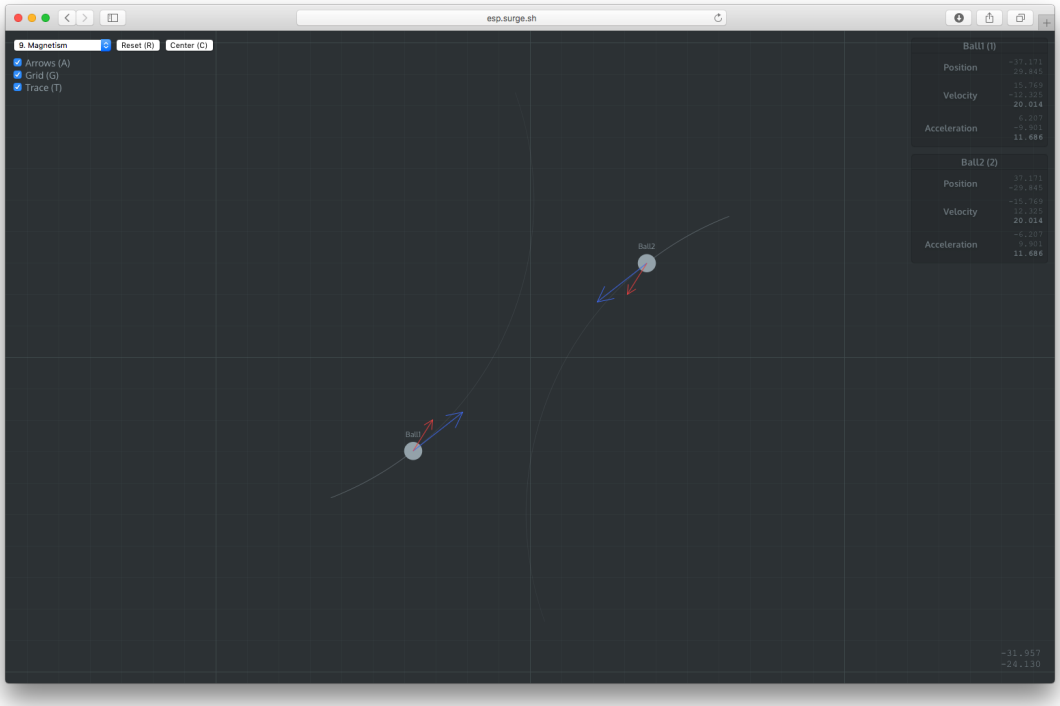

Figure 15: Scene 9. Magnetism

5.2 Bonus 3: Creating new scenes

A `Scene` is an object containing multiple `IEntity` objects and updating them. If you take a look inside of the `Scenes` folder, you will see the definition of all the scenes you have been using to visualize your code.

For this bonus, we ask that you create two new scenes. Creating a new scene is as simple as creating a new class inside of the `Scenes` folder that extends the `Scene` abstract class, then instantiating and registering it inside of the `RunSimulation` method in `Program.cs`. Don't forget to implement the constructor of your scene, where you will add all the objects that you want to appear in the scene.

You are free to chose which scenes to create, but be original! You will be graded both on the creativity and the complexity of your scenes. You are free to create new types of entities, in the `Update` of which you can manipulate the position and velocity freely — although we'd rather you kept using `ApplyForce` for more natural motions.

Examples of scenes you can implement are:

- The entire solar system, with the satellites of all the planets and a new space park orbiting around the earth.
- Better and more diverse constraint types (*cf.* Bonus 1: A simple pendulum). For instance, you could implement joints and try to reproduce the butterfly effect: two systems with very slightly different initial states eventually drifting away from each other. You could also implement collisions by making sure objects are pushed away from each other when they intersect.
- A binary star system.
- *etc.*

These violent deadlines have violent ends.