

SnapGames

EDITION

Développer mon jeu avec Java

Frédéric Delorme

Table of Contents

Preface	1
Les Fondations	2
Le langage Java	3
Bases de la POO	4
Plongeons dans Java !	12
UML : Langage de Modélisation Unifié	14
Conclusion	16
Configuration	17
Contexte	17
Les sections	17
Implementation de la configuration	18
Fenêtre et boucle de jeu	23
Entité et ennemis	41
Ajoutons un ennemi	41
La classe Entity	43
Conclusion	51
Définir une Scène	52
La Scène	52
La scène de jeu	54
Modifions MonProgramme	56
Node ?	59
Conclusion	61
Comportement dans les entités	62
Qu'est-ce qu'un comportement	62
Passons au code	63
Délégation aux Comportements dans les phases du cycle de jeu	64
Conclusion	69
Camera, Monde et Aire de jeu	70
L'objet Camera	70
Allons un peu plus loin	75
World, un nouveau monde	77
Conclusion	81
Délégation de la boucle de jeu	82
Un peu d'histoire	82
Passons au code	84
Implementation de StandardGameLoop	86
Conclusion	88
Collision et réponse	89

La Détection	89
Reponse aux collisions	90
Nouvelle Scene	90
Ajoutons un peu de Physique	93
Contexte	93
Un peu de théorie	93
Un peu de code Java	95
Une Entité	95
Le service PhysicEngine	96
Les limites liées au jeu	98
La classe World	99
l'effet Material	102
Les WorldArea	105
Ajout de Systems et de leur manager	109
Thank	110
Index	111

Preface

Les jeux vidéo m'ont toujours fasciné. Pas seulement pour les aventures qu'ils offrent ou les souvenirs qu'ils laissent, mais pour cette magie invisible qui transforme des lignes de code en mondes vivants. Je me suis souvent demandé : Comment ça marche ? Comment un simple écran noir peut devenir un terrain de jeu où tout semble possible ?

Développeur depuis 25 ans et architecte logiciel depuis 15 ans, j'ai passé une bonne partie de ma carrière à résoudre des casse-têtes techniques dans l'industrie. Mais ma vraie passion, celle qui m'a poussé à explorer, à apprendre et à bidouiller, c'est le jeu vidéo. Ce livre est une sorte de mélange : un projet personnel où mes compétences professionnelles rencontrent mon amour pour le rétro-gaming.

Mon but ? Vous emmener dans les coulisses d'un jeu vidéo, pour construire ensemble un moteur 2D avec Java. Pas besoin d'outils modernes ou de bibliothèques complexes, juste AWT et Swing, les bons vieux outils du JDK. C'est simple, accessible, et parfait pour comprendre les fondamentaux : une boucle de jeu (game-loop), des entités, des scènes, des comportements, des systèmes et même une gestion des ressources. Bref, tout ce qu'il faut pour donner vie à vos idées.

Pourquoi Java ? Parce que c'est une boîte à outils polyvalente que je connais par cœur, mais aussi parce que ça prouve qu'on n'a pas besoin des dernières technologies pour s'amuser et apprendre. Ce projet est une invitation à retourner à l'essentiel, à jouer avec le code comme on jouait avec des LEGO quand on était gamins.

Alors, que vous soyez curieux, passionné de rétro-gaming, ou simplement en quête d'un défi créatif, ce livre est fait pour vous. Prenez votre clavier, mettez-vous à l'aise, et embarquons ensemble dans cette aventure. Vous verrez, construire un jeu vidéo, c'est un peu comme jouer à un jeu... sauf que vous êtes le créateur.

Les Fondations

Avant de nous lancer dans la création de notre moteur de jeu, nous devons poser des bases solides. Comme construire une maison, un bon projet logiciel repose sur des fondations claires et stables. Dans ce chapitre, nous allons explorer les principes de la programmation orientée objet (OOP) et leur mise en œuvre avec le langage Java. Ces concepts seront les briques essentielles pour construire tout ce qui va suivre.

Pourquoi Java ?

Java est un langage robuste, portable et facile à apprendre. Avec ses 25 ans d'existence, il a prouvé sa valeur dans des domaines allant de l'industrie aux jeux vidéo. Ce n'est peut-être pas le choix le plus courant pour développer un moteur de jeu, mais c'est justement ce qui en fait un terrain d'apprentissage idéal : en travaillant avec Java, nous apprenons à penser différemment et à maximiser les capacités d'outils simples.

La Programmation Orientée Objet : un modèle puissant À la base, la programmation orientée objet repose sur quatre piliers principaux :

- L'encapsulation : c'est l'art de cacher les détails internes d'un objet tout en offrant une interface claire pour interagir avec lui. Imaginez une voiture : vous appuyez sur l'accélérateur sans avoir besoin de connaître le fonctionnement du moteur.
- L'héritage : cette notion permet de réutiliser du code en établissant des relations entre les classes. Par exemple, dans un jeu, une classe Entity pourrait définir des propriétés générales, et des sous-classes comme Player ou Enemy pourraient hériter de ces propriétés tout en ajoutant leur propre comportement.
- Le polymorphisme : ce concept rend votre code plus flexible en permettant à un même type de se comporter différemment en fonction du contexte. Par exemple, une méthode render() pourrait s'appliquer à différentes entités d'un jeu, chacune ayant sa propre manière de se dessiner à l'écran.
- L'abstraction : c'est la capacité à se concentrer sur les aspects importants en cachant les détails complexes. Dans notre projet, nous utiliserons des interfaces et des classes abstraites pour définir des comportements sans nous attarder sur leur implémentation immédiate.

Java et l'OOP en action

Java incarne ces principes dans sa conception même. Avec ses classes, ses interfaces, et son riche écosystème, il offre tous les outils nécessaires pour structurer proprement un projet, qu'il s'agisse d'une application d'entreprise ou d'un moteur de jeu.

Dans ce chapitre, nous allons revoir les notions clés de l'OOP à travers Java. Si vous débutez avec ce langage ou si vous avez besoin d'une petite mise à jour, pas d'inquiétude ! Nous avancerons étape par étape, en construisant des exemples concrets qui serviront de point de départ à notre moteur de jeu. Chaque concept sera expliqué avec un œil sur son utilité dans notre projet.

Préparez-vous : cette introduction va poser les bases de tout ce que nous allons créer ensemble. Vous serez bientôt prêt à transformer des idées en code, un objet à la fois.

Le langage Java

Qu'est-ce que Java ?

Java est un langage de programmation orienté objet de haut niveau développé par Sun Microsystems (maintenant partie d'Oracle Corporation) au milieu des années 1990. Il est conçu pour être indépendant de la plateforme, permettant aux développeurs d'écrire du code une fois et de l'exécuter partout (grâce à la machine virtuelle Java). Java est connu pour sa simplicité, sa robustesse, ses fonctionnalités de sécurité et ses bibliothèques étendues, ce qui le rend largement utilisé pour la création d'applications web, d'applications mobiles, de logiciels d'entreprise, et plus encore. Sa syntaxe est similaire à celle de C++, ce qui aide les programmeurs à passer facilement de ces langages.

Java a été développé au début des années 1990 par une équipe dirigée par James Gosling chez Sun Microsystems. Initialement destiné à la télévision interactive, le projet a évolué en un langage de programmation à usage général. Voici quelques jalons clés de l'histoire de Java :

- **1991** : Le projet Java, initialement appelé Oak, a été lancé pour créer un langage pour les systèmes embarqués.
- **1995** : Le langage a été officiellement renommé Java et publié au public. Il a gagné en popularité grâce à sa capacité "écrire une fois, exécuter partout", grâce à la machine virtuelle Java (JVM).
- **1996** : La première version officielle, Java 1.0, a été publiée, établissant Java comme un acteur majeur dans le développement web avec des applets.
- **1998** : Java 2 a été introduit, incluant des améliorations significatives et l'introduction de la bibliothèque GUI Swing.
- **2004** : Java 5 (également connu sous le nom de J2SE 5.0) a introduit des fonctionnalités majeures comme les génériques, les annotations et la boucle 'for' améliorée.
- **2006** : Sun Microsystems a rendu Java gratuit et open-source sous la licence publique générale GNU.
- **2010** : Oracle Corporation a acquis Sun Microsystems, et Java a continué d'évoluer avec des mises à jour régulières et de nouvelles fonctionnalités.
- **2020 et au-delà** : Java a maintenu sa pertinence avec des améliorations continues, se concentrant sur la performance, la sécurité et la productivité des développeurs.
- **Aujourd'hui**, Java reste l'un des langages de programmation les plus utilisés au monde, en particulier dans les environnements d'entreprise et le développement d'applications Android.

Ouvrons la boîte

Le Java Development Kit (JDK) est un environnement de développement logiciel utilisé pour développer des applications Java. Il comprend plusieurs composants essentiels pour le développement Java :

- **Compilateur Java (javac)** : Convertit le code source Java (.java) en bytecode (.class) qui peut être exécuté par la machine virtuelle Java (JVM).
- **Environnement d'exécution Java (JRE)** : Fournit les bibliothèques, la machine virtuelle Java

(JVM) et d'autres composants nécessaires pour exécuter des applications Java.

- **Machine virtuelle Java (JVM)** : Exécute le bytecode Java et fournit un environnement d'exécution pour les applications Java.
- **Outils de développement** : Comprend divers outils en ligne de commande pour des tâches telles que :
 - **jar** : Crée et gère des fichiers Java Archive (JAR).
 - **javadoc** : Génère de la documentation à partir des commentaires du code source Java.
 - **jdb** : Débogueur Java pour résoudre les problèmes des applications Java.
 - **javap** : Désassemble les fichiers de classe pour fournir des informations sur leur structure.
- **Bibliothèques de classes Java** : Un ensemble de bibliothèques préconstruites qui fournissent des fonctionnalités pour des tâches comme l'entrée/sortie, le réseau, les structures de données et les interfaces graphiques.
- **Programmes d'exemple et documentation** : Code exemple et documentation extensive pour aider les développeurs à comprendre comment utiliser le JDK efficacement.
- **JavaFX** : Un ensemble de packages graphiques et multimédias pour construire des applications client riches (inclus dans certaines distributions JDK).

Le JDK est essentiel pour les développeurs Java car il fournit tous les outils et bibliothèques nécessaires pour créer, compiler et exécuter des applications Java.

Bases de la POO

Cette série de tutoriels ne sera pas un cours sur la POO (Programmation Orientée Objet), mais nous introduisons Java à travers la POO. La Programmation Orientée Objet est basée sur un ensemble de modèles et de processus décrits à travers des classes et des objets.

La Programmation Orientée Objet (POO) est un paradigme de programmation basé sur le concept d'"objets", qui peuvent contenir des données et du code. Voici les principes fondamentaux de la POO :

- **Encapsulation** : Ce principe consiste à regrouper les données (attributs) et les méthodes (fonctions) qui opèrent sur les données en une seule unité appelée classe. Il restreint l'accès direct à certains composants de l'objet, ce qui aide à prévenir les interférences et les abus non intentionnels. L'accès aux données est généralement contrôlé par des méthodes publiques (getters et setters).
- **Abstraction** : L'abstraction se concentre sur la dissimulation des détails d'implémentation complexes et montre uniquement les caractéristiques essentielles d'un objet. Elle permet aux programmeurs de travailler à un niveau de complexité plus élevé sans avoir besoin de comprendre tous les détails sous-jacents. Cela est souvent réalisé par le biais de classes abstraites et d'interfaces.
- **Héritage** : L'héritage permet à une classe (l'enfant ou sous-classe) d'hériter des propriétés et des méthodes d'une autre classe (le parent ou superclasse). Cela favorise la réutilisation du code et établit une relation hiérarchique entre les classes. Une sous-classe peut également remplacer les méthodes de sa superclasse pour fournir une fonctionnalité spécifique.

- **Polymorphisme** : Le polymorphisme permet aux méthodes de faire des choses différentes selon l'objet sur lequel elles agissent. Cela peut être réalisé par la surcharge de méthode (même nom de méthode avec des paramètres différents) et le remplacement de méthode (la sous-classe fournit une implémentation spécifique d'une méthode déjà définie dans sa superclasse). Il permet à une seule interface de représenter différentes formes sous-jacentes (types de données).

Ces principes travaillent ensemble pour faciliter un code modulaire, réutilisable et maintenable, ce qui fait de la POO un choix populaire pour le développement de logiciels.

Une Classe

La **classe** dans la POO est l'outil pour décrire un **objet**, comme un plan peut l'être pour une maison ou une voiture.

Elle définit des attributs (certaines valeurs spécifiques pour ce type d'objet) et des méthodes, une liste d'interactions possibles avec l'objet.

Par exemple, une classe nommée **Car** pourrait avoir des attributs comme **color**, **make**, et **model**, et des méthodes comme **start()**, **stop()**, et **accelerate()**.

Un Objet

Un **objet** est une instance d'une classe.

Lorsqu'une classe est définie, aucune mémoire n'est allouée jusqu'à ce qu'un objet de cette classe soit créé.

Chaque objet peut avoir son propre état (valeurs des attributs) et peut exécuter les méthodes définies dans sa classe.

Par exemple, si **Car** est une classe, alors **myCar** pourrait être un objet de cette classe avec des valeurs spécifiques comme **color = "red"**, **make = "Toyota"**, et **model = "Corolla"**.

```
// Définition de la classe
public class Car {
    String color;
    String make;
    String model;

    void start() {
        System.out.println("Voiture démarrée");
    }
}

public class Main {
    public static void main(String[] args) {
// Création d'un objet
        Car myCar = new Car();
        myCar.color = "red";
    }
}
```

```

        myCar.make = "Toyota";
        myCar.model = "Corolla";
        myCar.start(); // Sortie : Voiture démarrée
    }
}

```

Une interface

Qu'est-ce qu'une Interface ?

Une interface en programmation orientée objet est un contrat qui définit un ensemble de méthodes qu'une classe doit implémenter, sans fournir d'implémentation concrète pour ces méthodes. Voici quelques points clés illustrés par l'exemple de la classe Car. Caractéristiques d'une Interface

Définition de Méthodes :

Une interface peut contenir des déclarations de méthodes que les classes qui l'implémentent doivent définir. Par exemple, nous pourrions créer une interface Vehicle :

```

interface Vehicle {
    void start();

    void stop();

    void accelerate();
}

```

- Pas d'État :** Les interfaces ne contiennent pas de variables d'instance. Elles peuvent seulement définir des constantes. Cela signifie que Vehicle ne peut pas avoir d'attributs comme color ou make.
- Multiples Implementations :** Une classe peut implémenter plusieurs interfaces. Par exemple, si nous avions une autre interface Electric, une classe ElectricCar pourrait implémenter à la fois Vehicle et Electric.
- Polymorphisme :** Les interfaces permettent le polymorphisme. Par exemple, nous pouvons traiter différents types de véhicules de manière uniforme :

```

public class VehiculeTest {
    public VehiculeTest(){}
}

public static void main(String[]arg){
    Vehicle myCar = new Car();
    // Appelle la méthode start() de la classe Car
    myCar.start();
}

```

Encapsulation de Comportements

Les interfaces définissent des comportements attendus sans se soucier de leur réalisation. Cela favorise la séparation des préoccupations. Par exemple, Car doit fournir des implémentations pour start(), stop(), et accelerate() :

```
public class Car implements Vehicle {  
    @Override  
    public void start() {  
        System.out.println("Voiture démarrée");  
    }  
  
    @Override  
    public void stop() {  
        System.out.println("Voiture arrêtée");  
    }  
  
    @Override  
    public void accelerate() {  
        System.out.println("Voiture accélérée");  
    }  
}
```

Avantages des Interfaces

- **Flexibilité** : Les interfaces permettent de changer les implémentations sans affecter le code qui les utilise. Par exemple, si nous décidons de créer une nouvelle classe **Truck** qui implémente **Vehicle**, nous pouvons le faire sans modifier le code qui utilise **Vehicle**.
- **Testabilité** : Elles facilitent le test unitaire en permettant de créer des classes simulées (mocks) qui implémentent les interfaces. Cela est utile pour tester des comportements sans dépendre des implémentations concrètes.
- **Clarté** : Elles fournissent une documentation claire des comportements attendus des classes. Dans notre exemple, toute classe qui implémente **Vehicle** doit savoir comment démarrer, arrêter et accélérer.

En résumé, une interface comme **Vehicle** est un outil puissant pour structurer et organiser le code, en favorisant la réutilisation et la maintenabilité, tout en garantissant que les classes comme **Car** fournissent les comportements nécessaires.

 Car
□ color: String
□ make: String
□ model: String
● start(): void
● stop(): void
● accelerate(): void

Qu'est-ce que l'Héritage de Classe ?

L'héritage de classe est un concept fondamental en programmation orientée objet qui permet de créer une nouvelle classe (appelée sous-classe ou classe dérivée) à partir d'une classe existante (appelée superclasse ou classe parente). La sous-classe hérite des attributs et des méthodes de la superclasse, ce qui favorise la réutilisation du code et la création de relations hiérarchiques entre les classes.

Exemple avec Vehicle et Car

Définition de la Superclasse

Nous commençons par définir une classe `Vehicle` qui contient des attributs et des méthodes communs à tous les types de véhicules.

```
public class Vehicle {  
    String color;  
    String make;  
  
    public void start() {  
        System.out.println("Véhicule démarré");  
    }  
  
    public void stop() {  
        System.out.println("Véhicule arrêté");  
    }  
}
```

Création de la Sous-classe

Ensuite, nous créons la classe `Car` qui hérite de la classe `Vehicle`. Cela signifie que `Car` a accès aux attributs `color` et `make`, ainsi qu'aux méthodes `start()` et `stop()`.

```
public class Car extends Vehicle {  
    String model;  
  
    public void accelerate() {  
        System.out.println("Voiture accélérée");  
    }  
}
```

Nous pouvons également envisager une classe `Truck` pour modéliser un camion, qui est, lui aussi un véhicule:

```
public class Truck extends Vehicle {  
    String model;  
    int wheels;
```

```

public void accelerate() {
    System.out.println("Camion accélérée");
}

public void load() {
    System.out.println("Camion chargé");
}

public void unload() {
    System.out.println("Camion déchargé");
}
}

```

Voici une illustration de l'héritage :

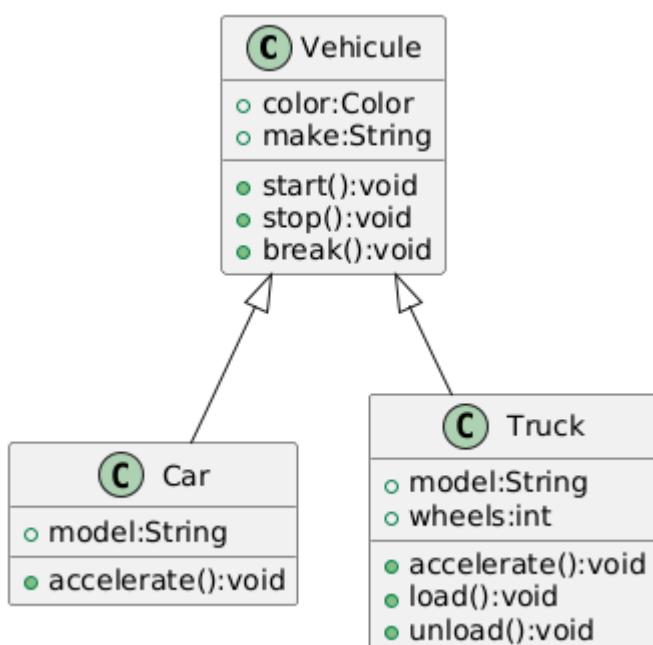


Figure 1. Diagramme UML illustrant l'héritage entre les classes **Vehicule**, **Car** et **Truck**

Utilisation de l'Héritage

Dans la classe Main, nous pouvons créer un objet Car et utiliser à la fois les méthodes de Car et celles héritées de Vehicle.

```

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.color = "rouge";
        myCar.make = "Toyota";
        myCar.model = "Corolla";

        myCar.start();      // Sortie : Véhicule démarré
        myCar.accelerate(); // Sortie : Voiture accélérée
        myCar.stop();       // Sortie : Véhicule arrêté
    }
}

```

}

Avantages de l'Héritage

1. **Réutilisation du Code :** L'héritage permet de réutiliser le code de la superclasse dans la sous-classe. Dans notre exemple, Car n'a pas besoin de redéfinir les méthodes start() et stop(), car elles sont déjà définies dans Vehicle.
2. **Organisation Hiérarchique :** L'héritage crée une structure hiérarchique qui aide à organiser les classes. Cela facilite la compréhension des relations entre différents types d'objets.
3. **Extension de Fonctionnalités :** La sous-classe peut ajouter des méthodes et des attributs spécifiques tout en conservant les comportements de la superclasse. Par exemple, Car a une méthode accelerate() qui n'est pas présente dans Vehicle.
4. **Polymorphisme :** L'héritage permet également le polymorphisme. Par exemple, nous pouvons traiter un objet Car comme un objet Vehicle, ce qui permet d'utiliser des méthodes communes sans connaître le type exact de l'objet.

Conclusion

L'héritage de classe, comme illustré avec **Vehicle** et **Car**, est un puissant mécanisme qui favorise la réutilisation du code, l'organisation hiérarchique et l'extension des fonctionnalités. Cela permet de créer des systèmes plus modulaires et maintenables tout en simplifiant la gestion des comportements communs entre différentes classes.

La portée des variables

Une notion supplémentaire est à comprendre, celle de la portée de variable dans le langage Java.

En effet, lorsque vous créez une variable, l'endroit où la créée a une grande importance, ainsi que la caractérisation de sa portée : cela définit la visibilité de la variable par les autres classes ou méthodes.

La portée des variables et attributs en Java, peut être définie par code en intégrant les modificateurs d'accès **private**, **protected** et **public**.

1. Modificateurs d'Accès :

- **private** : L'attribut ou la méthode est accessible uniquement au sein de la classe où il est défini. Il n'est pas accessible depuis d'autres classes, même celles qui héritent de cette classe.
- **protected** : L'attribut ou la méthode est accessible dans la classe où il est défini, dans les classes du même package, et dans les sous-classes (même si elles sont dans des packages différents).
- **public** : L'attribut ou la méthode est accessible depuis n'importe quelle autre classe, sans restriction.

2. Portée des Variables Locales :

Les variables déclarées à l'intérieur d'une méthode, d'un bloc ou d'une boucle sont des variables locales. Leur portée est limitée à ce bloc spécifique. Elles ne peuvent pas être utilisées en dehors de celui-ci.

```

public void maMethode() {
    int x = 10; // variable locale
    System.out.println(x); // accessible ici
}
// System.out.println(x); // erreur : x n'est pas accessible ici

```

- Portée des Attributs de Classe (ou Variables d'Instance) :** Les attributs déclarés au niveau de la classe (en dehors des méthodes) peuvent avoir différents modificateurs d'accès :

```

public class MaClasse {
    private int attributPrive; // accessible uniquement dans MaClasse
    protected int attributProtege; // accessible dans le même package et dans les
    sous-classes
    public int attributPublic; // accessible de partout

    public void maMethode() {
        attributPrive = 5; // accessible ici
        attributProtege = 10; // accessible ici
        attributPublic = 15; // accessible ici
    }
}

```

- Portée des Attributs Statiques :** Les attributs déclarés avec le mot-clé static sont partagés entre toutes les instances de la classe. Ils peuvent également avoir des modificateurs d'accès :

```

public class MaClasse {
    private static int attributStatiquePrive; // accessible uniquement dans MaClasse
    protected static int attributStatiqueProtege; // accessible dans le même package
    et dans les sous-classes
    public static int attributStatiquePublic; // accessible de partout

    public static void maMethodeStatique() {
        attributStatiquePrive = 10; // accessible ici
        attributStatiqueProtege = 20; // accessible ici
        attributStatiquePublic = 30; // accessible ici
    }
}

```

- Portée des Paramètres de Méthode :** Les paramètres passés à une méthode sont des variables locales, avec une portée limitée à cette méthode.

```

public void maMethode(int param) { // param est un paramètre
    System.out.println(param); // accessible ici
}

```

Résumé

La portée détermine où une variable ou un attribut peut être utilisé dans le code. Les modificateurs d'accès (private, protected, public) contrôlent la visibilité des attributs et des méthodes. Les variables locales ont une portée limitée à leur bloc, tandis que les attributs de classe et les attributs statiques peuvent avoir une portée plus large, selon leur modificateur d'accès.

Plongeons dans Java !

Maintenant que nous avons revu quelques bases de la POO, les classes, les interfaces, l'héritage, il est temps de s'intéresser de plus près à JAVA !

La méthode main

Lorsque vous créez un programme en Java et que vous souhaitez l'exécuter, il est nécessaire de suivre certaines règles. L'une des plus importantes est de définir un point d'entrée pour démarrer votre programme : c'est la méthode **main** !

Cette méthode reçoit un seul paramètre, qui est un tableau de chaînes de caractères (String) contenant tous les arguments passés depuis la ligne de commande.

Par exemple, si vous exécutez votre programme avec la commande suivante (nous aborderons les détails de la commande **java** plus tard) :

```
java MonProgramme.java Parametre1=1 Parametre2=2
```

Les arguments passés à main seront alors la liste suivante ["Parametre1=1", "Parametre2=2"].

Voici à quoi ressemble la classe Java **MonProgramme** :

```
public class MonProgramme {  
  
    public static void main(String[] args) {  
        for (String arg : args) {  
            System.out.println("argument : " + arg);  
        }  
    }  
}
```

Si nous exécutons ce code avec la commande proposée, nous obtiendrons la sortie suivante dans la console :

```
$> java MonProgramme.java Parametre1=1 Parametre2=2  
nb args:2  
argument : Parametre1=1  
argument : Parametre2=2
```

Nous savons maintenant comment lancer un programme java.

Améliorons notre programme

Ajoutons quelques éléments à notre programme. Nous allons réorganiser le code pour déplacer le traitement des arguments de la méthode `main` vers une méthode distincte de la classe `MonProgramme`.

```
public class MonProgramme2 {  
  
    ①   public MonProgramme2() {  
        System.out.println("Démarrage de mon Programme2");  
    }  
  
    ②   public void run(String[] args) {  
        System.out.println("nb args:" + args.length);  
        for (String arg : args) {  
            System.out.println("argument : " + arg);  
        }  
    }  
  
    ③   public static void main(String[] args) {  
        ④       MonProgramme2 prog = new MonProgramme2();  
        prog.run(args);  
    }  
}
```

1. Notre nouvelle classe `Programme2` a maintenant un constructeur, qui initialise des informations dès la création de l'instance de la classe, ici, nous affichons un message indiquant le démarrage du programme.
2. La méthode `run` contient le code précédemment présent dans `main`, c'est-à-dire l'affichage du nombre d'arguments et de leur contenu.
3. Dans la nouvelle méthode `main`, nous commençons par créer une instance de notre classe `MonProgramme2`,
4. Ensuite, nous appelons la méthode `run` de cette instance.

Nous venons d'écrire notre premier programme Java, Bravo !

Note
 Si vous êtes familier avec le langage C++, vous remarquerez l'absence d'une méthode de destruction (destructeur) pour l'instance.

Une fenêtre ?

Nous avons maintenant un pied dans le code java, nous allons passer à ce qui nous intéresse ici, la

création d'une jeu. En premier lieu, il est nécessaire d'ouvrir une fenêtre dans laquelle nous pourrons afficher ce jeu.

Pour ce faire, nous passerons par la bibliothèque de composants graphique fournie nativement par java. Non pas JavaFX, mais AWT et Swing. Ces 2 bibliothèques historiques nous donneront tous les services et composants dont nous aurons besoin.

Comment créer une fenêtre ? cette création passe par l'utilisation d'une classe : [JFrame](#).

UML : Langage de Modélisation Unifié

Nous pouvons également utiliser les méthodes et outils UML pour décrire de telles classes et objets.

UML, ou Langage de Modélisation Unifié, est un langage de modélisation standardisé utilisé en ingénierie logicielle pour visualiser, spécifier, construire et documenter les artefacts d'un système logiciel. Il fournit un ensemble de techniques de notation graphique pour créer des modèles visuels de systèmes logiciels.

Caractéristiques clés de l'UML :

1. **Représentation Visuelle** : UML utilise des diagrammes pour représenter différents aspects d'un système, ce qui facilite la compréhension des systèmes complexes.
2. **Notation Standardisée** : UML fournit un moyen cohérent de représenter les composants du système, ce qui aide à la communication entre les parties prenantes (développeurs, designers, analystes commerciaux, etc.).
3. **Multiples Diagrammes** : UML comprend divers types de diagrammes qui peuvent être classés en deux groupes principaux :
 - **Diagrammes Structurels** : Ces diagrammes représentent les aspects statiques d'un système, tels que :
 - Diagramme de Classe
 - Diagramme de Composant
 - Diagramme d'Objet
 - Diagramme de Paquet
 - **Diagrammes Comportementaux** : Ces diagrammes représentent les aspects dynamiques d'un système, tels que :
 - Diagramme de Cas d'Utilisation
 - Diagramme de Séquence
 - Diagramme d'Activité
 - Diagramme d'État
4. **Modélisation des Systèmes Logiciels** : UML peut être utilisé pour modéliser divers aspects des systèmes logiciels, y compris les exigences, l'architecture, la conception et l'implémentation.
5. **Utilisation Interdisciplinaire** : Bien qu'il soit principalement utilisé en ingénierie logicielle, UML peut également être appliqué dans d'autres domaines pour modéliser des systèmes

complexes.

UML aide les équipes à communiquer efficacement et sert de plan pour construire des applications logicielles, facilitant de meilleures pratiques de conception et de documentation.

Quelques exemples

1. Un Diagramme de Classe

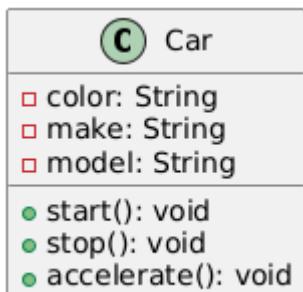


Figure 2. Diagramme UML de classe

1. Un Diagramme d'Activité

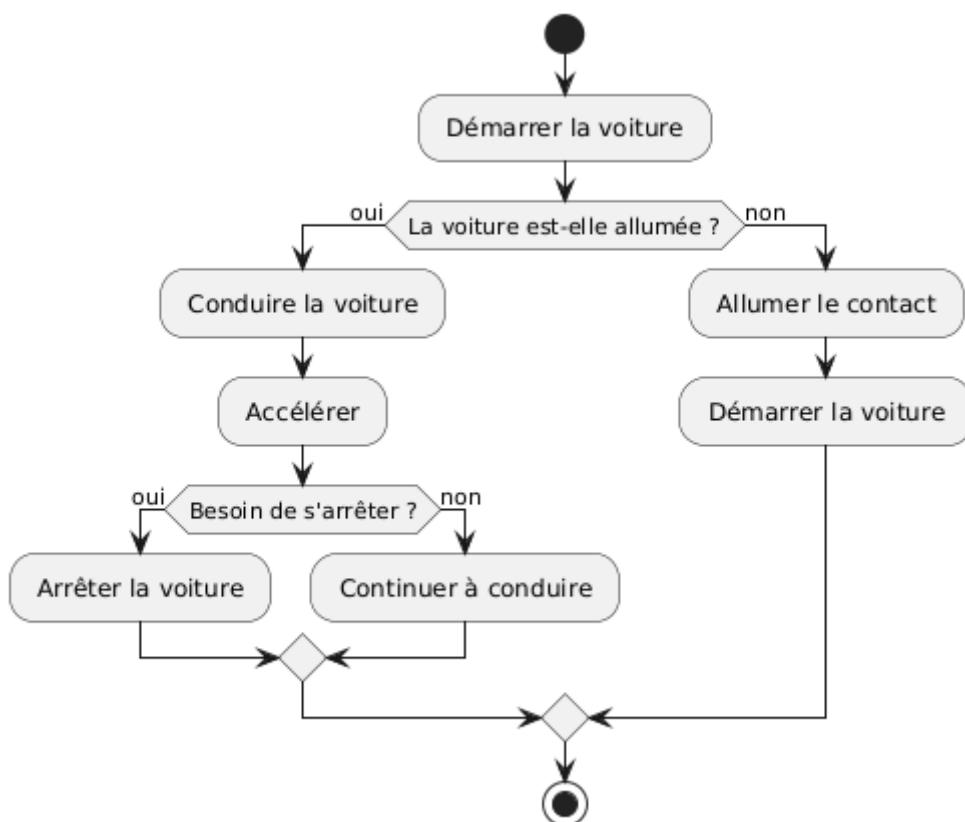


Figure 3. Diagramme UML d'activités

Pour en découvrir, je vous invite à visiter la page de wikipedia relative à l' [Unified Modeling Language \(UML\)](https://fr.wikipedia.org/wiki/UML_(informatique)) : [https://fr.wikipedia.org/wiki/UML_\(informatique\)](https://fr.wikipedia.org/wiki/UML_(informatique)).

Conclusion

Nous avons fait un tour, certes rapidement, de ce que propose le langage Java et la modélisation UML. Nous utiliserons donc le langage Java, et l'UML pour expliquer un peu les approches de conception du code proposé.

Il est grand temps de se lancer dans l'aventure avec nos premiers programmes d'exploration.

Configuration

Avant de plonger dans le cœur de notre moteur de jeu, nous devons établir un système flexible pour gérer la configuration. En nous appuyant sur un fichier `*.properties`, bien connu des développeurs Java, nous allons lire des paramètres clés, les stocker dans une map et les rendre accessibles sous une forme typée. Ce système servira de socle pour adapter facilement les comportements de notre moteur tout au long du projet.

Contexte

La configuration sera servie via un fichier java ".properties". Les différents services et traitement internes auront leur propre section dans ce fichier.

Quelques règles :

1. Chaque clé de configuration dans le fichier aura son nom préfixé apr `app..`
2. Le nom de section doit suivre le préfixe `app..`, exemple `app.physic` définira toutes les clés pour le moteur physique.
3. Pas plus de QUATRE suffixes dans le nommage, après le nom de la section correspondant au service.

Les sections

Rendering

La section rendering fournira la taille de la fenêtre d'affichage ainsi que la résolution du buffer de rendu interne. Ces 2 entrées seront au format `[width]x[height]`

Voici les valeurs par défaut :

- `app.render.window.title = Java com.snapgames.framework.GameInterface In A Week`
- `app.render.window.size = 640x400`
- `app.render.buffer.size = 320x200`

Physic

La section physic permettra de définir la zone de jeu ainsi que la gravité

- `app.physic.world.play.area.size = 1080x800`
- `app.physic.world.gravity = (0,-0.981)`

Scene

La section Scene définit la liste des implementations de `Scene` pour le jeu ainsi que la scene à activer par défaut au démarrage du jeu.

- `app.scene.list` = play:com.snapgames.demo.scenes.PlayScene,
- `app.scene.default` = play

Mode test et debug

Certaines fonctions ne doivent être activées que dans un contexte d'exécution lié aux tests ou au débogage. C'est ce que permettront de configurer les clés suivantes :

- `app.debug.level` = le niveau de finesse du débogage, de 0 à 5,
- `app.debug.filter` = list de package/entité, au format text, pour lesquels vous souhaitez activer la trace de débogage, exemple: `com.snapgames.scene.PlayScene`
- `app.test.mode` = false/true, la valeur `true` activant le mode test. La game loop ne se jouant qu'un nombre limité de fois, au minimum 1 fois si la clé `app.test.loop.maxcount` n'est pas définie.
- `app.test.loop.max.count` indique, si nécessaire, le nombre maximum de fois que l boucle principale doit être exécutée.

Maintenant que nous avons fait le tour des clés de configuration, nous allons passer à l'implémentation de ce service.

Implementation de la configuration

Tout d'abord, nous allons installer cette configuration dans un nouveau package que nous allons appeler `com.snapgames.framework.utils`. Ensuite, la classe implémentant le service de configuration se nommera `Config`.

```
public class Config {  
}
```

Ce nouvel object recevra à sa création l'application parente, ici notre classe `com.snapgames.framework.GameInterface` (dont nous verrons plus tard les détails)

```
public class Config {  
    com.snapgames.framework.GameInterface app;  
  
    public Config(com.snapgames.framework.GameInterface app) {  
        this.app = app;  
        // ...  
    }  
}
```

Ensuite, nous allons définir les valeurs par défaut. Nous stockerons les valeurs de configuration dans une Map. Pour nous faciliter la gestion des valeurs de configuration, nous allons étendre la class `HashMap` du JDK et ainsi bénéficier de toutes ses fonctionnalités.

```

public class Config extends HashMap<String, Object> {
    GameInterface app;

    public Config(GameInterface app) {
        this.app = app;
        put("app.test", false);
        put("app.test.loop.max.count", 1);
        put("app.debug.level", 0);
        put("app.render.window.title", "Java com.snapgames.framework.GameInterface In
A Week");
        put("app.render.window.size", new Dimension(640, 400));
        put("app.render.buffer.size", new Dimension(320, 200));
        put("app.physic.world.play.area.size", new Rectangle2D.Double(0, 0, 640, 400)
);
        put("app.physic.world.gravity", new Point2D.Double(0, -0.981));
        put("app.scene.default", "");
        put("app.scene.list", "");
    }
}

```

Nous avons maintenant un ensemble de valeurs prêtes à être servie via le getter de la Map:

```

import com.snapgames.framework.GameInterface;

public class examples.MonProgrammeConfig1 extends TestGame {
    private String configurationFilePath = "/config.properties";
    private Config config;

    public examples.MonProgrammeConfig1() {
        System.out.println("Démarrage de mon Programme3");
        config = new Config(this);
    }

    public void run(String[] args) {
        System.out.printf("configuration for title:%s%n", (String) config.get(
"app.render.window.title"));
    }

    public static void main(String[] args) {
        examples.MonProgrammeConfig1 prog = new examples.MonProgrammeConfig1();
        prog.run(args);
    }
}

```

En exécutant cette classe `MonProgrammeConfig1`

```

javac -d target/demo-classes src/main/java/com/snapgames/framework/GameInterface.java
src/test/java/*.java src/test/java/**/*.java

```

```
java -cp target/demo-classes examples.MonProgrammeConfig1
```

Vous obtenez l'affichage suivante sur la console :

```
java -cp target/demo-classes examples.MonProgrammeConfig1
# Démarrage de examples.MonProgrammeConfig1
=> Configuration for title:Default Title
```

Initialization depuis un fichier

Passons au plus intéressant : chargeons un fichier `*.properties` et parcourons ses valeurs afin de les typer et les stocker dans la map.

L'opération de lecture est grandement facilité par l'utilisation de l'objet `Properties` du JDK, il faut ensuite parcourir chaque valeur et interpréter chaque valeur pour stocker une valeur typée, c'est-à-dire convertie en `Integer`, `Long`, `Boolean`, `Double` ou en toute autre classe nécessaire, correspondante dans notre map.

1. Chargement du fichier

Nous avons un fichier de propriétés qui contient les valeurs suivantes :

```
## Debug & Test
app.exit=false
app.debug.level=3
app.render.window.title="Test Game App"
## Render
app.render.window.size=640x400
app.render.buffer.size=320x200
## Physic Engine
app.physic.world.play.area.size=1080x800
app.physic.world.gravity=(0,-0.981)
## Scene
app.scene.default=play
app.scene.list=play:com.snappgames.demo.scenes.PlayScene,
# error
app.unknown.key=not known
```

Modifions maintenant notre classe `Config` pour lire le fichier de propriétés avec `Properties.load(String)` :

```
public class Config extends HashMap<String, Object> {
    GameInterface app;

    public Config(GameInterface app) {
        //..
    }
}
```

```

public void load(String filePath) {
    try {
        props.load(this.getClass().getResourceAsStream(configFilePath));
        props.forEach((k, v) -> {
            System.out.printf("%s=%s%n", k, v);
        });
        parseAttributes(props.entrySet().parallelStream().collect(Collectors
.toList()));
    } catch (IOException e) {
        System.err.printf("Unable to read configuration file: %s", e.getMessage()
);
    }
}
}
}

```

Il faut maintenant parcourir toutes les entrées du fichier créer les vraies valeurs typées:

1. Parcours des valeurs

```

public class Config extends HashMap<String, Object> {
//...

private void parseAttributes(List<Entry<Object, Object>> collect) {
    collect.forEach(e -> {
        switch (e.getKey().toString()) {
            case "app.render.window.title" -> {
                put("app.render.window.title", (String) e.getValue());
            }
            case "app.exit" -> {
                app.setExit(Boolean.parseBoolean(props.getProperty("app.exit")));
            }
            case "app.debug.level" -> {

                app.setDebug(Integer.parseInt(props.getProperty("app.debug.level")));
            }
            case "app.render.window.size" -> {
                String[] values = ((String) e.getValue()).split("x");
                put("app.render.window.size", new
Dimension(Integer.parseInt(values[0]), Integer.parseInt(values[1])));
            }
            case "app.render.buffer.size" -> {
                String[] values = ((String) e.getValue()).split("x");
                put("app.render.buffer.size", new
Dimension(Integer.parseInt(values[0]), Integer.parseInt(values[1])));
            }
            case "app.physic.world.play.area.size" -> {
                String[] values = ((String) e.getValue()).split("x");
                put("app.physic.world.play.area.size", new Rectangle2D.Double(0,
0, Double.parseDouble(values[0]), Double.parseDouble(values[1])));
            }
        }
    });
}
}
}

```

```
        }
        case "app.physic.world.gravity" -> {
            String[] values = ((String) e.getValue()).substring(((String)
e.getValue()).indexOf("(") + 1, ((String) e.getValue()).lastIndexOf("))).split(",");
            put("app.physic.world.gravity", new
Point2D.Double(Double.parseDouble(values[0]), Double.parseDouble(values[1])));
        }
        case "app.scene.default" -> {
            put("app.scene.default", (String) e.getValue());
        }
        case "app.scene.list" -> {
            put("app.scene.list", ((String) e.getValue()).split(","));
        }
        default -> {
            System.err.printf("Unknown value for %s=%s%n", e.getKey(),
e.getValue());
        }
    }
});
```

La méthode `parseAttribute(List<Entry<Object, Object>> collect)` permet de parcourir la collection clé/valeur et entrée par entrée, exécuter la conversion correspondante à chaque clé connue.

par exemple, pour la clé `app.exit` dont la valeur typée correspondante doit être un booléen:

```
//...
case "app.exit"->{
    app.setExit(Boolean.parseBoolean(props.getProperty("app.exit")));
}
//...
```

Nous pouvons voir ici que la valeur obtenue est positionnée directement, dans l'instance de `app`.

Dans le second exemple, la valeur est stockée dans la map pour un usage futur :

```
//...
case "app.render.window.title"->{
    put("app.render.window.title", (String) e.getValue());
}
//...
```

Dans ce troisième et dernier exemple, la valeur du fichier de propriété est convertie en une instance de [Dimension](#), et est stockée dans la map:

```
//...  
case "app.render.window.size":>{
```

```

String[] values = ((String) e.getValue()).split("x");

put("app.render.window.size",
    new Dimension(
        Integer.parseInt(values[0]),
        Integer.parseInt(values[1])));
}

// ...

```

En exécutant cette classe `MonProgrammeConfig2`

```

javac -d target/demo-classes src/main/java/com/snapgames/framework/GameInterface.java
src/test/java/*.java src/test/java/**/*.java
java -cp target/demo-classes examples.MonProgrammeConfig2

```

Vous obtenez l'affichage suivante sur la console :

```

# Démarrage de examples.MonProgrammeConfig2
# Load configuration Properties file /config2.properties
- app.scene.list=play:com.snapgames.demo.scenes.PlayScene,
- app.render.window.size=640x400
- app.exit=false
- app.physic.world.play.area.size=1080x800
- app.physic.world.gravity=(0,-0.981)
- app.scene.default=play
- app.debug.level=3
- app.unknown.key=not known
- app.render.window.title="Test Game App (config2)"
- app.render.buffer.size=320x200
~ Unknown value for app.unknown.key=not known
=> Configuration for title:"Test Game App (config2)"

```

Nous voilà fin prêt à passer à un autre sujet, l'affichage dans une fenêtre.

Fenêtre et boucle de jeu

Contexte

Nous allons maintenant nous intéresser à l'affichage. Mais pour cela, nous devons mettre en place une structure de code bien connu des développeurs de jeu : la "boucle de jeu" ou "Game Loop".

Introduction

Historiquement, cette structure est héritée des premiers jeux vidéos et des contraintes techniques des années 70-80. Et nous devons bien avouer que depuis, même si les frameworks ont terriblement évolués, la notion de GameLoop a perduré, et aujourd'hui encore, elle est présente dans les gros moteurs.

Cette structure est assez simple et repose sur un triplet d'actions :

1. initialiser les ressources du jeu
2. gérer les entrées du joueur,
3. mettre à jour en respectant la mécanique du jeu les objects du jeu,
4. faire le rendu graphique des objets concernés.
5. libérer toutes les ressources

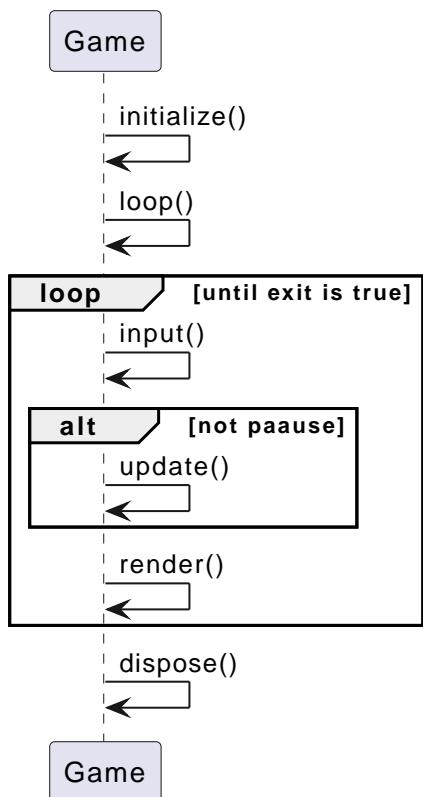


Figure 4. Diagramme de séquence de notre boucle de jeu

Ce qui se résume assez bien par :

- input
- update
- render

Si nous mettons ajour notre précédent exemple de code issu du chapitre de la configuration, nous avons :

```
public class MonProgrammeWinAndLoop1 extends TestGame {  
    //...  
    public MonProgrammeWinAndLoop1() {  
        //...  
    }  
  
    public void loop() {  
        while (!isExitRequested()) {
```

```

        input();
        update();
        render();
    }
}

private void initialize() {}

private void input() {}

private void update() {}

private void render() {}

private void dispose() {}

public void run(String[] args) {
    System.out.printf("=> Configuration for title:%s%n", (String) config.get(
"app.render.window.title"));
    initialize();
    loop();
    dispose();
}

//...

public static void main(String[] args) {
    MonProgrammeWinAndLoop1 prog = new MonProgrammeWinAndLoop1();
    prog.run(args);
}
}

```

Ansi, tant que rien ne demande de quitter le jeu, on boucle sur les trois actions.

Comme nous l'avons introduit dans le chapitre précédent, nous devons prendre en compte un mode test limitant le nombre de 'tours' dans la boucle, aussi, adaptons notre code en conséquence :

```

public class MonProgrammeWinAndLoop1 extends TestGame {
    //...
    private boolean testMode = false;
    private int maxLoopCount = 1;

    public initialize() {
        testMode = config.get("app.test");
        maxLoopCount = (int) config.get("app.test.loop.max.count");
        System.out.printf("# %s est initialisé%n", this.getClass().getSimpleName());
    }

    public void loop() {
        int loopCount = 0;

```

```

        while (!isExitRequested() && (!testMode || loopCount < maxLoopCount)) {
            input();
            update();
            render();
            loopCount++;
        }
        System.out.printf("> Game loops %d times\n", loopCount);
    }

//...
private void dispose() {
    System.out.printf("# %s est terminé.\n", this.getClass().getSimpleName());
}
//...
}

```

Si vous positionnez les valeurs suivantes dans le fichier de configuration :

```

## Debug & Test
app.test=true
app.test.loop.max.count=3

```

Et que vous lancez l'exécution de notre nouvel exemple :

```

javac -d target/demo-classes src/main/java/com/snapgames/framework/GameInterface.java
src/test/java/*.java src/test/java/**/*.java
java -cp target/demo-classes examples.MonProgrammeWinAndLoop1

```

Vous obtenez sur la console la sortie suivante :

```

# Démarrage de MonProgrammeWinAndLoop1
# Load configuration Properties file /config2.properties
- app.exit=false
- app.debug.level=3
- app.unknown.key=not known
- app.test=true
- app.test.loop.max.count=3
- app.render.window.title="Test Game App (config2)"
- app.render.buffer.size=320x200
~ Unknown value for app.exit=false
~ Unknown value for app.unknown.key=not known
=> Configuration for title:"Test Game App (config2)"
# MonProgrammeWinAndLoop1 est initialisé
=> Game loops 3 times
# MonProgrammeWinAndLoop1 est terminé.

```

Une fenêtre

Passons maintenant à l'affichage ! je vous propose pour cela d'ajouter une fenêtre à notre programme. Cette fenêtre sera défini en taille par une configuration. Le titre de ladite fenêtre sera également issu d'une entrée dans le fichier la configuration.

Afin d'implémenter cette fenêtre, nous utiliserons l'API SWING et AWT du JDK, et ce même si JavaFX existe, ce dernier ne permettant pas facilement de réaliser un rendu "old school" à base de gros pixels.

```
public class MonProgrammeWinAndLoop2 extends TestGame {  
    //...  
    private JFrame window;  
    //...  
  
    public void initialize() {  
        testMode = config.get("app.test");  
        maxLoopCount = (int) config.get("app.test.loop.max.count");  
        System.out.printf("# %s est initialisé%n", this.getClass().getSimpleName());  
  
        createWindow();  
    }  
  
    private void createWindow() {  
        // Create the Window  
        window = new JFrame((String) config.get("app.render.window.title"));  
        window.setPreferredSize(config.get("app.render.window.size"));  
        window.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);  
        window.pack();  
        window.setVisible(true);  
        window.addKeyListener(this);  
    }  
}
```

Et nous ajoutons 2 nouvelles lignes dans le fichier de configuration :

```
## Render  
app.render.window.title=Test Game App (win-and-loop)  
app.render.window.size=640x400
```

Ainsi, sont défini le titre et la taille en pixels de la fenêtre affichée.

Ajoutons maintenant un peu d'interaction, nous allons ajouter le traitement des évènements liés aux touches du clavier.

KeyEvent et KeyListener

Afin de pouvoir procéder à l'interception et au traitement des évènements clavier, nous allons implémenter dans notre classe principale Game l'interface KeyListener du JDK.

```
public class MonProgrammeWinAndLoop2 extends TestGame
    implements KeyListener {

    @Override
    public void keyTyped(KeyEvent e) {

    }

    @Override
    public void keyPressed(KeyEvent e) {
    }

    @Override
    public void keyReleased(KeyEvent e) {
    }
}
```

Et nous souhaitons faire un premier traitement lorsque la touche ESCAPE est appuyée : nous souhaitons mettre fin à l'exécution de notre programme.

```
public class MonProgrammeWinAndLoop2 extends TestGame
    implements KeyListener {
    //...
    @Override
    public void keyReleased(KeyEvent e) {
        if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
            this.requestExit();
        }
    }
}
```

Ainsi lorsque la touche **ESCAPE** est relâchée, nous demandons au programme de procéder à la sortie de la boucle principale, et ainsi mettre fin au programme.

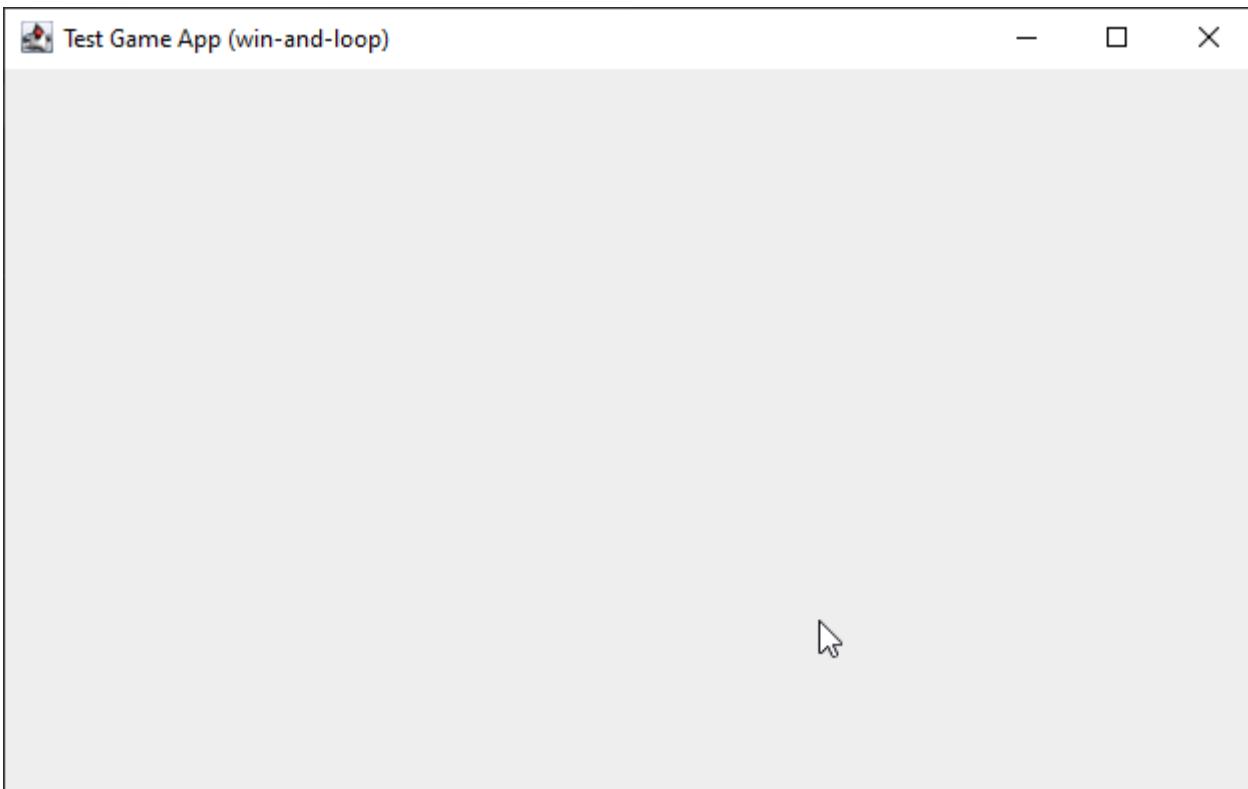


Figure 5. La capture de la fenêtre

Le buffer

Maintenant que nous avons une fenêtre, nous allons enfin pouvoir passer à la partie artistique !

Comme indiqué précédemment, nous souhaitons réaliser un rendu typé "old-school" avec des pixels assez visibles. Pour éviter des post-traitements compliqués pour le propos de ce livre, nous allons utiliser une technique assez simple permettant de dessiner dans un calque ayant une définition réduite, puis copier le contenu de ce calque vers la fenêtre, procédant ainsi à la mise à l'échelle de la fenêtre.

Nous dessineras donc dans un buffer un mémoire, puis à chaque rafraîchissement, nous copierons ce buffer sur la fenêtre cible.

Commençons par créer ce buffer de travail (ou calque) :

```
public class MonProgrammeWinAndLoop3 extends TestGame implements KeyListener {  
    //...  
    private JFrame window;  
    ①    private BufferedImage renderingBuffer;  
    //...  
    public void initialize() {  
        //...  
    ②        createBuffer();  
    }  
    //...
```

```

③
private void createBuffer() {
    Dimension renderBufferSize = config.get("app.render.buffer.size");
    renderingBuffer = new BufferedImage(
        renderBufferSize.width, renderBufferSize.height,
        BufferedImage.TYPE_INT_ARGB);
}

public void loop() {
    int loopCount = 0;
    int frameTime = 1000 / (int) (config.get("app.render.fps"));
    while (!isExitRequested() && ((testMode && loopCount < maxLoopCount) ||
!testMode)) {
        input();
        update();
        render();
        loopCount++;
}

④
        waitTime(frameTime);
    }
    System.out.printf("=> Game loops %d times%n", loopCount);
}

//...
private void render() {
    //<(6)>
    Graphics2D g = renderingBuffer.createGraphics();
    // clear rendering buffer to black
    g.setColor(Color.BLACK);
    g.fillRect(0, 0, renderingBuffer.getWidth(), renderingBuffer.getHeight());

    // draw something
    g.dispose();

    //...
}
}

```

En prenant dans l'ordre :

1. On ajoute le buffer de rendu
2. On délègue l'initialisation du buffer
3. La taille du buffer est définie par l'entrée `app.render.buffer.size`
4. on modifie la méthode d'attente pour la prochaine frame en calculant le temps en fonction du taux de rafraîchissement cible défini dans `app.render.fps`.
5. Enfin, on efface le buffer avec la couleur noir, et plus tard, nous pourrons dessiner dessus !



Figure 6. La capture de la fenêtre avec le buffer

Dessinons !

Nous avons maintenant de quoi faire un premier test d'animation.

MISSION : Nous allons donc dessiner un carré bleu de 16 pixels de côté, au centre de notre fenêtre, et à l'aide des touches de direction, nous pourrons le déplacer, en restant dans la limite de la fenêtre.

Nous devrons donc définir quelques variables pour gérer la position, intercepter les touches pressées et modifier la position en conséquence, tout en restant dans l'espace défini par la taille de jeu.

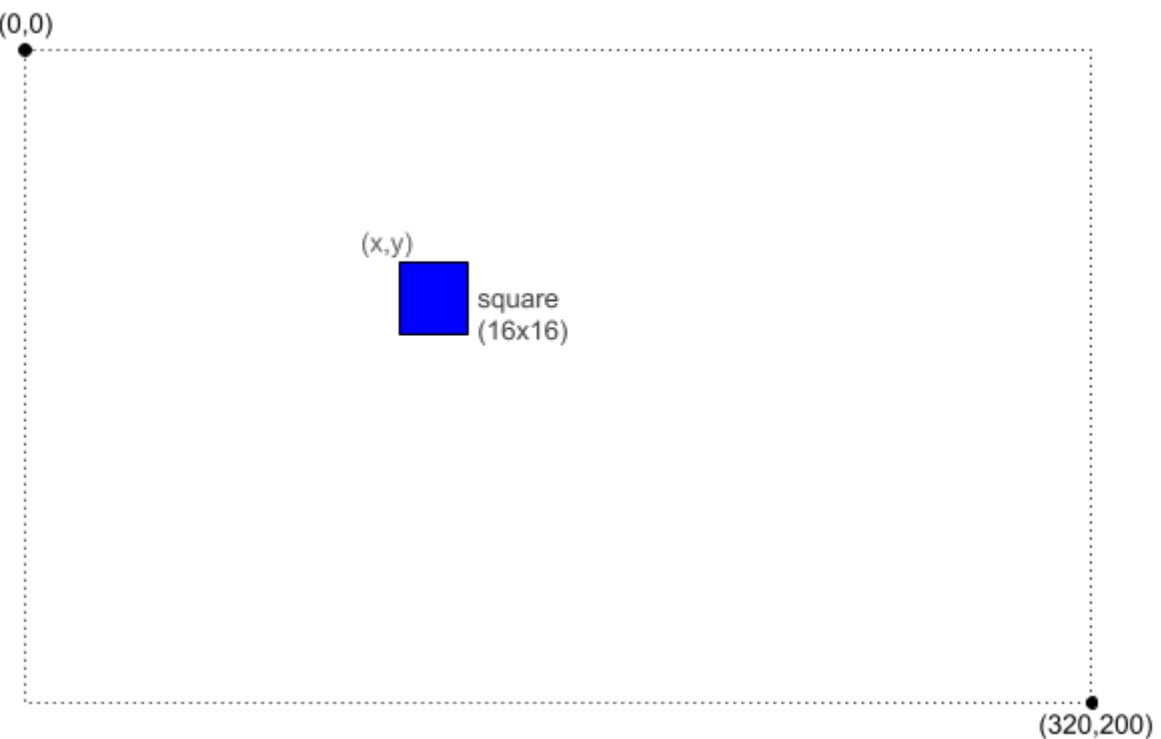


Figure 7. Un carré dans l'espace (de jeu)

Passons au code.

Tout d'abord, positionnons notre carré, et dessinons-le.

```
public class MonProgrammeDemo1 extends TestGame implements KeyListener {
    //...
    ① private int x, y;
    //...

    public void initialize() {
        //...
    ②     // blue square position initialization.
        x = (int) ((renderingBuffer.getWidth() - 16) * 0.5);
        y = (int) ((renderingBuffer.getHeight() - 16) * 0.5);
    }

    //...
    private void render() {
        Graphics2D g = renderingBuffer.createGraphics();
        //...

        // draw something
    ③     g.setColor(Color.BLUE);
        g.fillRect(x, y, 16, 16);

        g.dispose();
    }
}
```

```
//...
}

}
```

1. Nous définissons une position de départ sur un vecteur (x, y) où $x=0$ et $y=0$,
2. Nous définissons la position centrale sur le buffer de rendu,
3. Et nous dessinons ce fameux carré bleu de 16x16 pixels à la position (x, y).

```
public class MonProgrammeDemo1 extends TestGame implements KeyListener {
    //...

①    private boolean[] keys = new boolean[1024];

    //...
    private void input() {
        ②        if (keys[KeyEvent.VK_LEFT]) {
            x = Math.max(x - 2, 0);
        }
        if (keys[KeyEvent.VK_RIGHT]) {
            x = Math.min(x + 2, renderingBuffer.getWidth()-16);

        }
        if (keys[KeyEvent.VK_UP]) {
            y = Math.max(y - 2, 0);
        }
        if (keys[KeyEvent.VK_DOWN]) {
            y = Math.min(y + 2, renderingBuffer.getHeight()-16);
        }

    }
    //...

    @Override
    public void keyPressed(KeyEvent e) {
        ③        keys[e.getKeyCode()] = true;
    }

    @Override
    public void keyReleased(KeyEvent e) {
        ④        keys[e.getKeyCode()] = false;
        if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
            this.requestExit();
        }
    }
}
```

```
}
```

1. Ensuite, nous créons un cache d'état pour l'ensemble des touches de clavier,
2. En fonction des touches de direction appuyée, nous déplaçons notre position sur x ou y de 2 pixels dans la direction correspondante,
3. Nous définissons l'état de la touche pressée dans le buffer à vrai,
4. Nous définissons l'état de la touche relâchée dans le buffer à faux.

Vitesse

Nous pouvons améliorer ce code en décomposant entre l'action de touches et la mise à jour de la position en séparant le code entre `input()` et `update()`. C'est le moment d'introduire une notion de vitesse avec un vecteur (`dx,dy`).

C'est ce que nous nous proposons de faire sur notre seconde démo.

C'est dans la méthode `input()` que nous traiterons les évènements claviers en définissant une vitesse sur les 2 axes en fonction des touches pressées, et dans la méthode `update()` que nous calculerons la nouvelle position en s'assurant que notre carré bleu ne sorte pas de la zone visible de l'écran.

```
import utils.Config;

import javax.swing.*;
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.image.BufferStrategy;
import java.awt.image BufferedImage;

public class examples.MonProgrammeDemo2 extends TestGame implements KeyListener {
    private String configFilePath = "/demo2.properties";
    //...
    private int dx, dy;

    public void initialize() {
        //...
        dx=0;
        dy=0;
    }

    //...
    private void input() {
        if (keys[KeyEvent.VK_LEFT]) {
            dx = -2;
        }
        if (keys[KeyEvent.VK_RIGHT]) {
            dx = +2;
        }
    }
}
```

```

        }
        if (keys[KeyEvent.VK_UP]) {
            dy = -2;
        }
        if (keys[KeyEvent.VK_DOWN]) {
            dy = +2;
        }
        dx *= 0.5;
        dy *= 0.5;
    }

    private void update() {
        x += dx;
        y += dy;
        x = Math.min(Math.max(x, -8), renderingBuffer.getWidth()-8);
        y = Math.min(Math.max(y, -8), renderingBuffer.getHeight()-8);
    }
    //...
}

```

Si nous lançons l'exécution de [MonProgrammeDemo2](#), nous verrons que le comportement visuel reste identique, mais les traitements sont maintenant séparés, facilitant ainsi l'évolution du code et son développement.

Un peu de Physique du mouvement

Nous pouvons pousser encore un peu plus loin notre première approche de code en intégrant un peu de physique du mouvement.

Passons les couples de variables (*x,y*) et (*dx,dy*) en double plutôt qu'en int, et nous pourrons jouer un peu sur une simulation de déceleration.

```

public class examples.MonProgrammeDemo3 extends TestGame implements KeyListener {
    //...
    private double x, y;
    private double dx, dy;
    //...
}

```

Et modifions un peu le code de la méthode update :

```

public class examples.MonProgrammeDemo3 extends TestGame implements KeyListener {
    //...
    private void update() {
        ①
        x += dx;
    }
}

```

```

    y += dy;
②
    x = Math.min(Math.max(x, -8), renderingBuffer.getWidth() - 8);
    y = Math.min(Math.max(y, -8), renderingBuffer.getHeight() - 8);
③
    dx *= 0.98;
    dy *= 0.98;
}
//...
}

```

1. Nous calculons la future position en fonction de la vitesse courante, et appliquant la simple formule $p1=p0+dv$ où $p1$ est la future position, $p0$ la position actuelle et dv la vitesse actuelle,
2. Nous nous assurons que le centre du carré ne sorte pas de la zone délimitée par la fenêtre (en fait, ici, nous utilisons la taille du buffer de rendu),
3. Et ensuite, nous appliquons un facteur de réduction sur la vitesse, afin que le carré décélère lorsque qu'aucune touche du curseur n'est pressée.

Pour rendre les choses plus paramétrables, nous définissons un nouvel attribut de configuration nommé 'app.entity.player.speed' qui est fixé par défaut à 2.0.

```

# Physic
app.physic.entity.player.speed=2.0

```

Et notre code d'analyse de valeur évolue de la façon suivante :

```

public class Config extends HashMap<String, Object> {
//...
    public Config(GameInterface app) {
        //...
①
        put("app.physic.entity.player.speed", 2);
        //...
    }
    private void parseAttributes(List<Entry<Object, Object>> collect) {
        collect.stream()
            .forEach(e -> {
                switch (e.getKey().toString()) {
                    //...
②
                    case "app.physic.entity.player.speed" -> {
                        put("app.physic.entity.player.speed", Double.parseDouble(props
                            .getProperty("app.physic.entity.player.speed")));
                    }
                    //...
                }
            });
    }
}

```

```

    }
}

//...
}

```

1. Une valeur par défaut est créée dans la map,
2. La valeur issue du fichier properties est convertie en Double et stockée dans la map.

Ajout du rebond

Comme nous sommes joueur, nous pouvons ajouter un nouveau paramètre permettant une simulation de physique un peu plus amusante : le rebond lié à un facteur d'élasticité.

En effet, lors de la collision avec le rebord de notre buffer délimitant la zone de jeu, nous pouvons appliquer un facteur d'élasticité à notre carré bleu pour affecter sa vitesse.

Ajoutons ce nouveau paramètre dans notre jeu et rendons le configurable via notre fichier de propriétés.

```

public class examples.MonProgrammeDemo3 extends TestGame implements KeyListener {
    //...
    private double elasticity = 1.0;
    private double friction = 1.0;
    //...
    private void initialize() {
        elasticity = (double)config.get("app.physic.entity.elasticity");
        friction = (double)config.get("app.physic.entity.friction");
    }
    //...
    private void update() {
        // calcul de la position en fonction de la vitesse courante.
        x += dx;
        y += dy;

        // application du rebond si collision avec le bord de la zone de jeu
        if (x < -8 || x > renderingBuffer.getWidth() - 8) {
            dx = -dx * elasticity;
        }
        if (y < -8 || y > renderingBuffer.getHeight() - 8) {
            dy = -dy * elasticity;
        }

        // repositionnement dans la zone de jeu si nécessaire
        x = Math.min(Math.max(x, -8), renderingBuffer.getWidth() - 8);
        y = Math.min(Math.max(y, -8), renderingBuffer.getHeight() - 8);

        // application du facteur de friction
        dx *= friction;
        dy *= friction;
    }
}

```

```

    }
    //...
}

```

Ajoutons dans notre fichier de propriétés les deux nouvelles entrées :

```

app.physic.entity.player.elasticity=0.75
app.physic.entity.player.friction=0.98

```

Et enfin, dans la classe Config, ajoutons la lecture de ces 2 valeurs en type **double** :

```

public class Config extends HashMap<String, Object> {
//...
    public Config(GameInterface app) {
        //...
①        put("app.physic.entity.player.elasticity", 1.0);
        put("app.physic.entity.player.friction", 1.0);
        // ...
    }
    private void parseAttributes(List<Entry<Object, Object>> collect) {
        collect.stream()
            .forEach(e -> {
                switch (e.getKey().toString()) {
                    //...
②                    case "app.physic.entity.player.elasticity" -> {
                        put("app.physic.entity.player.elasticity", Double.parseDouble(props.getProperty("app.physic.entity.player.elasticity")));
                    }
                    case "app.physic.entity.player.friction" -> {
                        put("app.physic.entity.player.friction", Double.parseDouble(props.getProperty("app.physic.entity.player.friction")));
                    }
                    // ...
                }
            });
    }
//...
}

```

Ajoutons une trace du vecteur vitesse appliquée à notre carré bleu afin de rendre plus visuel l'effet des paramètres de frictions et d'élasticité.

```

private void render() {
    //...
}

```

```

// draw something
// ...
g.setColor(Color.YELLOW);
g.drawLine(
    (int) x+8, (int) y+8,
    (int) (x+8 + dx * 4), (int) (y+8 + dy * 4));
g.dispose();
// ...
}

```

Exécutons la classe `MonProgrammeDemo3`, en déplaçant le carré bleu, nous constatons qu'en cas de contact avec le bord, il rebondit dans la direction opposée.



Figure 8. Un peu de physique dans notre GameLoop

Conclusion

Nous avons passé nos premières ligne de code à explorer un premier programme java, en introduisant plusieurs concepts :

1. La boucle de jeu ou "GameLoop" décomposant le traitement des jeux en 3 principales étapes : `input()`, `update()` et `render()`.
2. La creation d'une premiere fenêtre et le traitement des évènements issus du clavier avec la classe `JFrame` et l'interface `KeyListener` proposes par le JDK,
3. L'affichage d'une forme géométrique, ici un carré bleu, que nous animons en fonction des touches directionnelles pressées, le rendu étant assuré par l'API `Graphics2D` du JDK,
4. Un peu de physique du mouvement, simplifie, permettant d'appliquer une vitesse (`speed` dans

notre code) sur notre carre bleu, tout en tenant compte de 2 nouveaux paramètres dans le calcul de la vitesse appliquée que sont l'élasticité et la friction (`elasticity` et `friction` dans le code).

Il est grand temps de prendre un peu de recul et de voir un peu plus grand.

Entité et ennemis

Nous avons maintenant la structure de base pour notre boucle de jeu, nous pouvons essayer de passer à l'étape suivante : passer à un plus grand nombre d'objets affichés à l'écran !

Pour cela, nous allons devoir faire un peu de factorisation de notre code, afin de le rendre plus facile à modifier, et plus facile à gérer.

C'est le moment d'introduire le concept d'Entity.

Ajoutons un ennemi

Notre premier objet visuel est un carré bleu. Nous allons ajouter un rond rouge pour lui tenir compagnie !

Nous utiliserons un emplacement choisi aléatoirement sur l'espace de jeu comme position de départ.

Notre ennemi sera en position `(ex,ey)` avec une vitesse `(edx,edy)` et ayant les caractéristiques physiques `eElastivity` et `eFriction`. La vitesse appliquée à chaque update sera gérée à travers la variable `eSpeed`.

Ajoutons un premier ennemi dans notre programme.

```
public class MonProgrammeEnnemi1 extends TestGame implements KeyListener {  
    //...  
  
    // variables pour le player  
    private double x, y;  
    private double dx, dy;  
    private double elasticity = 0.75;  
    private double friction = 0.98;  
    private double speed = 0.0;  
  
    ①    // variable pour l'ennemi  
    private double ex, ey;  
    private double edx, edy;  
    private double eElasticity = 0.75;  
    private double eFriction = 0.98;  
    private double eSpeed = 0.0;  
  
    public void initialize() {  
        //...  
  
    ②    // Position de départ de l'ennemi rouge  
    ex = (int) (Math.random() * (renderingBuffer.getWidth() - 16));  
    ey = (int) (Math.random() * (renderingBuffer.getHeight() - 16));  
    eSpeed = 1.0;
```

```

        eElasticity = 0.96;
        eFriction = 0.99;

    }

//...
private void input(){
//...
// Simulation pour l'ennemi qui suit le player
if (x+8 != ex+5) {
    edx = Math.min(Math.signum((x+8) - (ex+5)) * 0.5 * (1 - (eSpeed / ((x+8)
- (ex+5)))),2.0);
}
if (y != ey) {
    edy = Math.min(Math.signum((y+8) - (ey+5)) * 0.5 * (1 - (eSpeed / ((y+8)
- (ey+5)))),2.0);
}
}

private void update() {
//...
// calcul de la position du player bleu en fonction de la vitesse courante.
//...

// calcul de la nouvelle position de l'ennemi rouge en fonction de la vitesse
courante.
ex += edx;
ey += edy;

// application du rebond si collision avec le bord de la zone de jeu
if (ex < -5 || ex > renderingBuffer.getWidth() - 5) {
    edx = -edx * eElasticity;
}
if (ey < -5 || ey > renderingBuffer.getHeight() - 5) {
    edy = -edy * eElasticity;
}

// repositionnement dans la zone de jeu si nécessaire
ex = Math.min(Math.max(ex, -5), renderingBuffer.getWidth() - 5);
ey = Math.min(Math.max(ey, -5), renderingBuffer.getHeight() - 5);
// application du facteur de friction
edx *= eFriction;
edy *= eFriction;
}

private void render() {
Graphics2D g = renderingBuffer.createGraphics();
// clear rendering buffer to black
g.setColor(Color.BLACK);
g.fillRect(0, 0, renderingBuffer.getWidth(), renderingBuffer.getHeight());
// draw player
g.setColor(Color.BLUE);
g.fillRect((int) x, (int) y, 16, 16);
}

```

```

        g.setColor(Color.YELLOW);
        g.drawLine((int) x+8, (int) y+8, (int) (x+8 + dx * 4), (int) (y+8 + dy * 4));
        // draw Ennemi
        g.setColor(Color.RED);
        g.fill(new Ellipse2D.Double((int) ex, (int) ey, 10, 10));
        g.setColor(Color.YELLOW);
        g.drawLine((int) ex+5, (int) ey+5, (int) (ex+5 + edx * 4), (int) (ey+5 + edy * 4));
        //...
    }
//...
}

```

Si nous lançons l'exécution nous découvrons un nouvel acteur, un petit point rouge qui track inlassablement le player bleu.

Maintenant si nous souhaitons avoir 10 ennemis rouges qui suivent notre objet player blue, nous allons nous confronter à une complexité de code bien plus élevé et une duplication de celui-ci ! En effet, nous allons avoir besoin d'autant de variable qu'il y a d'ennemi.

Cela reste envisageable, mais tout changement de comportement souhaité de l'un des ennemis apportera une difficulté accrue sur la modification de code.

C'est là qu'intervient la notion d'entité.

En effet, que ce soit notre player bleu ou notre ennemi rouge, ils obéissent tous deux au même règle de traitement. Nous pouvons donc envisager sérieusement de créer un objet spécialisé pour les entités animées à l'écran.

La classe Entity

Il est maintenant temps de repenser notre code pour le rendre plus compact et surtout introduire un composant réutilisable pour nos 2 types d'entités, ennemi et player.

Nous allons pour cela méditer une classe Entity qui contiendra toutes les variables nécessaire pour un objet à animer à l'écran et ainsi nous pourrons créer autant d'instance de cette objet pour chacun d'eux.

```

public class Entity {
①
    // conteur interne d'entité
    private static long index=0;
    // identifiant unique de l'entité
    private long id=index++;
    // nom de l'entité défini par défaut.
    private String name="entity_%04d".formatted(id);

②
    // position
}

```

```

private double x,y;
// vitesse
private double dx,dy;
// forme et dimension
private Shape shape;

③
// propriétés de matériau
private double elasticity=1.0;
private double friction=1.0;

④
// propriété pour le rendu
private Color color = Color.WHITE;
private Color fillColor=Color.BLUE;
}

```

Notre classe comporte plusieurs groupes d'attributs :

1. Le groupe d'attribut permettant l'identification de notre instance d'`Entity`, l'`index`, un compteur interne permettant d'alimenter l'identifiant à la création de l'instance, `id` l'identifiant unique de l'entité dans le jeu et enfin, un nom `name`, permettant de retrouver plus facilement une entité dans le jeu,
2. Un groupe permettant de définir `position`, vitesse (`vélocity`), forme et dimension (`shape`) de l'instance de la classe `Entity`,
3. Un troisième groupe définissant certains paramètres physiques de la matière composant l'Entity, l'élasticité (`elasticity`) et la `friction`.
4. Et le dernier groupe permet de définir les couleurs, `color` et `fillColor`, utilisées lors du rendu graphique de l'entité.

Si nous souhaitons créer une nouvelle instance, nous aurons besoin d'un créateur, mais aussi, et nous utiliserons une nouveauté, une API fluent permettant de créer des entités en chainant les setters.

Les setters Fluent

```

public class Entity {
    //...
    // <1> getter
    public double getElasticity() {
        return elasticity;
    }
    // <2> fluent setter
    public Entity setElasticity(double elasticity) {
        this.elasticity = elasticity;
        return this;
    }
    //...
}

```

```
}
```

Le setter fluent, il définit la valeur de l'attribut, puis retourne l'entité modifiée. Cela permet de chainer les appels de setters lors de la création d'objet :

Dans l'exemple ci-dessous, nous créons une instance nommée 'player' et définissons l'ensemble des valeurs de ses attributs avec des setters "fluent" :

Création d'une Entity player

```
// Création du player bleu
Entity player = new Entity("player")
    .setPosition(
        ((renderingBuffer.getWidth() - 16) * 0.5),
        ((renderingBuffer.getHeight() - 16) * 0.5))
    .setElasticity((double) config.get(
"app.physic.entity.player.elasticity"))
    .setFriction((double) config.get("app.physic.entity.player.friction"))
    .setFillColor(Color.BLUE)
    .setShape(new Rectangle2D.Double(0, 0, 16, 16));
```

Nous allons maintenant procéder à l'adaptation des traitements de mise-à-jour et de rendu.

Ajout des nouvelles variables dans le programme principal :

```
public class MonProgrammeEntity2 extends TestGame implements KeyListener {
//...
private Map<String, Entity> entities = new HashMap<>();
```

Nous allons modifier maintenant le code de mise à jour de la position et de la vitesse de toutes les entités déclarées dans la demo :

Calcul de la position et de la vitesse de chaque entité

```
public class MonProgrammeEntity2 extends TestGame implements KeyListener {
//...
private void update() {
    // parcours de l'ensemble des entités
    entities.values().stream().forEach(e -> {
        // calcul de la position du player bleu en fonction de la vitesse
        // courante.
        e.setPosition(e.getX() + e.getDx(), e.getY() + e.getDy());
        // application du rebond si collision avec le bord de la zone de jeu
        if (e.getX() < -8 || e.getX() > renderingBuffer.getWidth() - 8) {
            e.setVelocity(-e.getDx() * e.getElasticity(), e.getDy());
        }
        if (e.getY() < -8 || e.getY() > renderingBuffer.getHeight() - 8) {
            e.setVelocity(e.getDx(), -e.getDy() * e.getElasticity());
        }
    })
}
```

```

    // repositionnement dans la zone de jeu si nécessaire
    e.setPosition(Math.min(Math.max(e.getX(), -8), renderingBuffer.getWidth()
- 8),
        Math.min(Math.max(e.getY(), -8), renderingBuffer.getHeight() - 8)
);

    // application du facteur de friction
    e.setVelocity(e.getDx() * e.getFriction(), e.getDy() * e.getFriction());
}
}
}

```

La méthode `render` doit-elle aussi être adaptée :

Dessin de toutes les entités.

```

public class MonProgrammeEntity2 extends TestGame implements KeyListener {
    //...
private void render() {
    Graphics2D g = renderingBuffer.createGraphics();
    // ...

①    entities.values().forEach(e -> {
②        g.translate((int) e.getX(), (int) e.getY());
③        g.setColor(e.getFillColor());
        g.fill(e.getShape());
        g.setColor(e.getColor());
        g.drawLine((int) (e.getShape().getBounds().width * 0.5), (int) (e.
getShape().getBounds().height * 0.5),
            (int) (e.getShape().getBounds().width * 0.5 + e.getDx() * 4), (int)
(+e.getShape().getBounds().height * 0.5 + e.getDy() * 4));
④        g.translate((int) -e.getX(), (int) -e.getY());
    });
    g.dispose();

    // copy buffer to window.
    //...
}
}

```

1. Nous parcourons l'ensemble des entités déclaré dans la map,
2. Nous déplaçons le curseur de dessin à la position de l'instance d'`Entity` en cours,
3. Nous procédons au dessin de la forme (`shape`) de l'entité à la position demandée, nous dessinons également le vecteur 2D de la vitesse à la même position,

4. Nous ramenons le curseur à la position de départ pour tracer l'entité suivante.

L'ensemble du traitement ayant été modifié pour supporter nos nouveaux objets, nous modifions maintenant la création de nos objets :

Creation des nouvelles entités au démarrage

```
public class MonProgrammeEntity2 extends TestGame implements KeyListener {  
    //...  
    public void initialize() {  
        //...  
        // Création du player bleu  
        Entity player = new Entity("player")  
            .setPosition(  
                ((renderingBuffer.getWidth() - 16) * 0.5),  
                ((renderingBuffer.getHeight() - 16) * 0.5))  
            .setElasticity((double) config.get(  
"app.physic.entity.player.elasticity"))  
            .setFriction((double) config.get("app.physic.entity.player.friction"))  
            .setFillColor(Color.BLUE)  
            .setShape(new Rectangle2D.Double(0, 0, 16, 16));  
        add(player);  
  
        // Création de l'ennemi rouge  
        Entity enemy1 = new Entity("enemy_1")  
            .setPosition((Math.random() * (renderingBuffer.getWidth() - 16)),  
(Math.random() * (renderingBuffer.getHeight() - 16)))  
            .setElasticity(0.96)  
            .setFriction(0.99)  
            .setFillColor(Color.RED)  
            .setShape(new Ellipse2D.Double(0, 0, 10, 10));  
        add(enemy1);  
    }  
}
```

Et enfin, nous devons modifier la méthode input pour retrouver l'objet nommé "player" et lui appliquer les vitesses adhoc en fonction des touches de directions pressées :

Modification de la méthode input()

```
public class MonProgrammeEntity2 extends TestGame implements KeyListener {  
    //...  
    private void input() {  
        ①        Entity player = entities.get("player");  
        double speed = (double) config.get("app.physic.entity.player.speed");  
        ②        if (keys[KeyEvent.VK_LEFT]) {  
            player.setVelocity(-speed, player.get_dy());  
        }  
        if (keys[KeyEvent.VK_RIGHT]) {
```

```

        player.setVelocity(speed, player.getDy());
    }
    if (keys[KeyEvent.VK_UP]) {
        player.setVelocity(player.getDx(), -speed);
    }
    if (keys[KeyEvent.VK_DOWN]) {
        player.setVelocity(player.getDx(), speed);
    }
}
}

```

1. Récupération de l'entité "player"
2. Application des vitesses en fonction des touches UP, DOWN, LEFT et RIGHT pressées.

Si nous exécutons le nouveau programme "examples.MonProgrammeEntity2", nous verrons apparaître les mêmes entités que précédemment.

Une différence de taille, si nous souhaitons créer 10 entités ennemis, il nous suffit de créer autant d'entité que nécessaire :

Création de 10 ennemis avec délégation à la méthode createScene()

```

public class MonProgrammeEntity2 extends TestGame implements KeyListener {
    //...
    public void initialize() {
        testMode = config.get("app.test");
        maxLoopCount = (int) config.get("app.test.loop.max.count");
        System.out.printf("# %s est initialisé%n", this.getClass().getSimpleName());
        createWindow();
        createBuffer();
    }
    ①
    createScene();
}
②
private void createScene() {
    // Création du player bleu
    Entity player = new Entity("player")
        .setPosition(
            ((renderingBuffer.getWidth() - 16) * 0.5),
            ((renderingBuffer.getHeight() - 16) * 0.5))
        .setElasticity((double) config.get("app.physic.entity.player.elasticity"))
        .setFriction((double) config.get("app.physic.entity.player.friction"))
        .setFillColor(Color.BLUE)
        .setShape(new Rectangle2D.Double(0, 0, 16, 16))
    ③
        .setAttribute("max.speed", 2.0);
    add(player);

    // Création de l'ennemi rouge
    for (int i = 0; i < 10; i++) {

```

```

Entity enemy = new Entity("enemy_%d".formatted(i))
    .setPosition((Math.random() * (renderingBuffer.getWidth() - 16)),
(Math.random() * (renderingBuffer.getHeight() - 16)))
    .setElasticity(Math.random())
    .setFriction(Math.random())
    .setFillColor(Color.RED)
    .setShape(new Ellipse2D.Double(0, 0, 10, 10))

④     .setAttribute("max.speed", (Math.random() * player.getAttribute(
"max.speed", 2.0) * 0.90));
        add(enemy);
    }
}

//...
}

```

1. On délègue à la création de notre scène à la nouvelle méthode `createScene()`,
2. La création de la scène consiste en l'ajout des Entity's à notre jeu,
3. On utilise une nouvelle fonctionnalité des Entity (voir ci-après) que sont les attributs d'entité, qui permet d'ajouter autant d'attribut avec une valeur que l'on veut, ce qui permettra de définir des valeurs temporaires, utilisées dans les calculs et leurs limitations),
4. La vitesse de chaque ennemi est redéfinie en fonction de la différence de position entre l'ennemi et le player, et un facteur de correction aléatoire est appliqué afin de moduler cette vitesse entre 50% et 110% de la vitesse précédente, dans une limite définie ici à 90% de la vitesse du player, et dans tous les ne pouvant excéder 2 pixels/frame.

Les attributs d'une entité

Les attributs pour une entité, sont stockés dans une Map supportant l'accès concurrent : `attributes`.

Le type d'un attribut n'est pas contrôlé à sa création et peut donc convenir à n'importe quel usage.

Un attribut sera ajouté via la méthode `Entity#setAttribute(attrKey:String, attrValue:Object)`

Il peut être récupéré à l'aide de la méthode `Entity.getAttribute(attrKey:String, defaultAttrValue:Object)`.

Si la clé définie par `attrKey` n'existe pas dans la map des attributs, la valeur `defaultAttrValue` sera utilisée par défaut.

Revenons à nos énemis !

Il nous faut ensuite adapter le programme pour la simulation de mouvement de tous ces nouveaux ennemis :

Animation de 10 ennemis en fonction de la position de l'objet "player"

```
public class MonProgrammeEntity2 extends TestGame implements KeyListener {
```

```

//...
private void input() {
    //...
    // On parcourt les entités en filtrant sur celles dont le nom commence par
    "enemy_"
①
    entities.values().stream()
        .filter(e -> e.getName().startsWith("enemy_"))
        .forEach(e -> {
            // new speed will be only a random ratio of the current one (from 50%
            to 110%)
            double eSpeed = (0.5 + Math.random() * 1.1);

            // Simulation pour les ennemis qui suivent le player sur l'axe X,
            // but limited to 'max.speed' attribute's value
            double centerPlayerX = player.getX() + player.getShape().getBounds()
                .width * 0.5;
            double centerEnemyX = e.getX() + e.getShape().getBounds().width * 0.5;
            double directionX = Math.signum(centerPlayerX - centerEnemyX);
            if (directionX != 0.0) {
                e.setVelocity(
                    Math.min(directionX * eSpeed * e.getAttribute("max.speed",
                        2.0),
                        e.getAttribute("max.speed", 2.0)),
                    e.getDy());
            }

            // Simulation pour les ennemis qui suivent le player sur l'axe Y,
            // but limited to 'max.speed' attribute's value
            double centerPlayerY = player.getY() + player.getShape().getBounds()
                .width * 0.5;
            double centerEnemyY = e.getY() + e.getShape().getBounds().width * 0.5;
            double directionY = Math.signum(centerPlayerY - centerEnemyY);
            if (directionY != 0.0) {
                e.setVelocity(
                    e.getDx(),
                    Math.min(directionY * eSpeed * e.getAttribute("max.speed",
                        2.0),
                        e.getAttribute("max.speed", 2.0)));
            }
        });
    //...
}

```

1. Nous parcourons l'ensemble des entités dont le nom commence par "enemy_",
2. Pour chaque entité, nous calculons la vitesse sur l'axe horizontal (X) pour assurer une poursuite de l'entité "player",

3. Nous calculons ensuite la vitesse sur l'axe vertical (Y) pour assurer cette même poursuite.

Nous obtenons une dizaine d'entités "enemy_%" qui vont tenter de rattraper le "player", en ayant une vitesse limitée par calcul.

CAUTION

Si la vitesse maximum autorisée pour chaque entité n'est pas définie dans ses attributs par une valeur pour `max.speed`, elle sera limitée par code à 2 pixels / frame.

Nous avons maintenant un objet `Entity` qui permet d'encapsuler toutes les variables nécessaires à la description d'entités animées dans notre scène.

TIP

Dans un chapitre ultérieur, nous découvrirons comment coder des comportements spécifiques et les appliquer à plusieurs entités, sans pour autant changer le code de l'une des étapes internes de la boucle de jeu, ce via la future interface `Behavior`.

Conclusion

WARNING

TODO rédiger la conclusion de ce chapitre avant d'exposer la suite.

Définir une Scène

Les jeux sont par défaut définis par leur game play.

Le jeu en lui-même a son propre gameplay, basé sur les entrées saisies par le joueur via le clavier, la souris ou un gamepad, et ainsi commandant à l'objet dirigé par le joueur les actions à réaliser.

Le jeu ne se résume pas qu'à cette phase de jeu, il y a souvent un inventaire, l'affichage d'une carte, un écran d'options, permettant la sauvegarde et le chargement de partie, la configuration de différents éléments du jeu comme l'affichage, le son, etc...

Tous ces différents types d'affichage sont en fait chacun une Scene, décrivant un ensemble d'objets visuels et de contrôles.

C'est ce que la classe Scene portera comme concept : la définition des éléments interagissant au sein d'un écran, ainsi que les contrôles nécessaires à assurer le gameplay de cet écran.

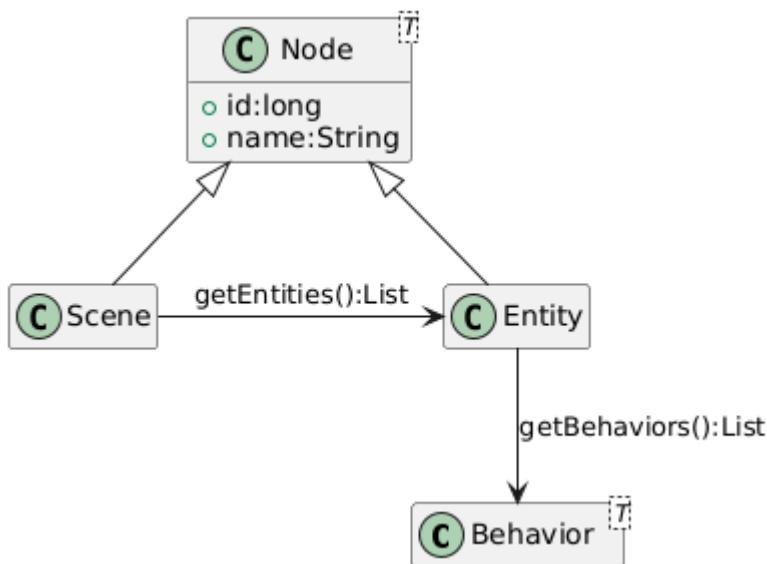


Figure 9. La scene fait partie d'un arbre à nœuds

Nous allons commencer par créer notre objet **Scene**, et nous agencerons les **Scene** et **Entity** à travers un bel arbre à **Node** plus tard.

La Scène

Une scene est un receptacle qui contiendra tous les objets à animer pour assurer un gameplay. Il est nécessaire d'avoir la liste des objets à animer, un petit nom pour pouvoir retrouver une scene parmi d'autres (et oui, plusieurs gameplay implique plusieurs scènes).

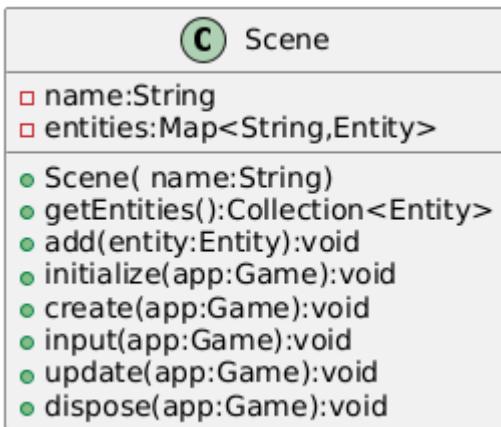


Figure 10. Le diagramme de classe de la Scene

Cela se traduit par la classe java suivante :

Notre classe Scene en java

```

import java.util.concurrent.ConcurrentHashMap;

public class Scene {
    private String name="";
    private Map<String,Entity> entities = new ConcurrentHashMap<>();

    public Scene(String name){
        this.name=name;
    }

①     public void add(Entity entity){
        this.entities.put(e.getName(),entity);
    }

②     public Collection<Entity> getEntities(){
        return entities.values();
    }

③     public void initialize(Game g){}

④     public void initialize(Game g){}

⑤     public void input(Game g){}

⑥     public void update(Game g){}

⑦     public void dispose(Game g){}
}

```

1. La méthode **add** permet d'ajouter une entité à la scene,
2. La méthode **getEntities** retournera une collection contenant toutes les entités de la scène.
3. **getEntity** retrieve an entity from the **Scene** map,

4. `initialization` est la méthode où l'on peut préparer les ressources nécessaires à la scène, par exemple charger l'image qui sera afficher en fond d'écran, la police de caractère pour afficher de beaux textes, etc...
5. `create` crée les entités nécessaires pour la scène,
6. `input` comme dans notre programme existant, nous implémenterons le traitement des évènements du clavier,
7. `update` sert à implementer le traitement spécifique de mise-à-jour de la Scene,
8. `dispose` permet de restituer toutes les éventuelles resources qui auront été chargées lors du traitement de l'`initialization`.

Nous allons revoir notre programme existant afin d'implémenter notre première Scene en intégrant le code existant.

La scène de jeu

Notre scène sera la classe `PlayScene` étendant la classe `Scene`.

```
public class PlayScene extends Scene{

    public PlayScene(String name){
        super(name);
    }

    public void create(Game app){
        // Création du player bleu
        Entity player = new Entity("player")
            .setPosition(
                ((renderingBuffer.getWidth() - 16) * 0.5),
                ((renderingBuffer.getHeight() - 16) * 0.5))
            .setElasticity((double) config.get("app.physic.entity.player.elasticity"))
            .setFriction((double) config.get("app.physic.entity.player.friction"))
            .setFillColor(Color.BLUE)
            .setShape(new Rectangle2D.Double(0, 0, 16, 16))
            .setAttribute("max.speed", 2.0);
        add(player);

        // Création de l'ennemi rouge
        for (int i = 0; i < 10; i++) {
            Entity enemy = new Entity("enemy_%d".formatted(i))
                .setPosition((Math.random() * (renderingBuffer.getWidth() - 16)),
                (Math.random() * (renderingBuffer.getHeight() - 16)))
                .setElasticity(Math.random())
                .setFriction(Math.random())
                .setFillColor(Color.RED)
                .setShape(new Ellipse2D.Double(0, 0, 10, 10))
                .setAttribute("max.speed", (Math.random() * player.getAttribute("max.speed") * 0.90));
            add(enemy);
        }
    }
}
```

```

        }
    }
    //...
}

```

Nous pouvons ajouter le traitement des touches de direction :

Implementation du traitement de la méthode input

```

public class PlayScene extends Scene{
    //...
    public void input(Game app){
        Entity player = entities.get("player");
        double speed = (double) config.get("app.physic.entity.player.speed");

        if (app.isKeyPressed(KeyEvent.VK_LEFT)) {
            player.setVelocity(-speed, player.getDy());
        }
        if (app.isKeyPressed(KeyEvent.VK_RIGHT)) {
            player.setVelocity(speed, player.getDy());
        }
        if (app.isKeyPressed(KeyEvent.VK_UP)) {
            player.setVelocity(player.getDx(), -speed);
        }
        if (app.isKeyPressed(KeyEvent.VK_DOWN)) {
            player.setVelocity(player.getDx(), speed);
        }

        // on parcourt les entités en filtrant sur celles dont le nom commence par
        "enemy_"
        getEntities().filter(e -> e.getName().startsWith("enemy_"))
        .forEach(e -> {
            // new speed will be only a random ratio of the current one (from 50% to
            110%)
            double eSpeed = (0.5 + Math.random() * 1.1);

            // Simulation pour les ennemis qui suivent le player sur l'axe X,
            // but limited to 'max.speed' attribute's value
            double centerPlayerX = player.getX() + player.getShape().getBounds().width
            * 0.5;
            double centerEnemyX = e.getX() + e.getShape().getBounds().width * 0.5;
            double directionX = Math.signum(centerPlayerX - centerEnemyX);
            if (directionX != 0.0) {
                e.setVelocity(
                    Math.min(directionX * eSpeed * e.getAttribute("max.speed", 2.0),
                            e.getAttribute("max.speed", 2.0)),
                    e.getDy());
            }

            // Simulation pour les ennemis qui suivent le player sur l'axe Y,
            // but limited to 'max.speed' attribute's value
        })
    }
}

```

```

        double centerPlayerY = player.getY() + player.getShape().getBounds().width
* 0.5;
        double centerEnemyY = e.getY() + e.getShape().getBounds().width * 0.5;
        double directionY = Math.signum(centerPlayerY - centerEnemyY);
        if (directionY != 0.0) {
            e.setVelocity(
                e.getDx(),
                Math.min(directionY * eSpeed * e.getAttribute("max.speed", 2.0),
                    e.getAttribute("max.speed", 2.0)));
        }
    });
}
//...
}

```

Par contre, nous n'avons aucune raison de déplacer le traitement des entités, l'application des lois de la physique sera bien la même quelque que soit la **Scene**.

Modifions MonProgramme

Il est temps de connecter notre nouvelle Scene avec le programme principal. Nous allons ajouter une liste de scenes ainsi qu'une scene courante.

Initialisation de la Scene dans le MonProgrammeScene1

```

public class MonProgrammeScene1 extends TestGame implements Game {
    //...
①    private Map<String,Scene> scenes = new ConcurrentHashMap<>();
②    private Scene currentScene;

    public initialize(){
        //...
        createWindow();
        createBuffer();
③        addScene(new PlayScene("play"));
④        switchScene("play");
    }
    //...
}

```

1. Nous avons besoin d'une map pour stocker les différentes scenes de notre jeu,
2. Ensuite, nous déclarons un attribut qui servira à stocker l'instance de Scene en cours (la scène active quoi !),
3. Nous ajoutons la ou les scènes pour notre jeu (ici, une seule, la scène **PlayScene**),

4. Et nous demandons à activer la scène souhaitée (ici "play").

Ensuite, nous allons changer la façon de créer la scène, en déléguant cela à la scene elle-même au sein de la méthode `createScene`.

1. Initialisation et création de la scène

```
public class MonProgrammeScene1 extends TestCase implements KeyListener, Game {  
    //...  
    public void switchScene(String name) {  
        ① if (Optional.ofNullable(currentScene).isPresent()) {  
            currentScene.dispose(this);  
        }  
        ② currentScene = scenes.get(name);  
            // Initialise et crée la Scene courante.  
        ③ currentScene.initialize(this);  
        ④ currentScene.create(this);  
    }  
    //...  
}
```

1. Si une scene est déjà active, nous la désactivons,
2. Nous récupérons l'instance de scene demandée depuis la collection (qui est une Map),
3. Nous commençons par initialiser la scene courante,
4. Nous demandons la création de toutes les entités.

Enfin, nous allons intégrer les traitements liés à la Scene dans la boucle principale.

```
public class MonProgrammeScene1 extends TestCase implements KeyListener, Game {  
    //...  
    private void input() {  
        ①     currentScene.input(this);  
    }  
    //...  
    private void update() {  
        // calcul de la position du player bleu en fonction de la vitesse courante.  
        //...  
        ②     currentScene.update(this);  
    }  
    //...  
    private void render() {  
        //...
```

```

    // draw entities
    currentScene.getEntities().forEach(e -> {
        //...
    });

③
    currentScene.draw(this, g);
    g.dispose();
    // copy buffer to window.
    //...
}

//...
private void dispose() {
④
    currentScene.dispose(this);
    window.dispose();
    //...
}
//...
}

```

1. Nous commençons par traiter les inputs de la scène,
2. Nous déléguons l'appel à la mise-à-jour de la scene,
3. Ensuite, si la scène le nécessite, nous pouvons la laisser dessiner ce qu'il faut,
4. Enfin, on peut lors de la cloture du jeu, procéder à la cloture de la scène.

WARNING

Vous aurez remarqué que nous utilisons dans notre scene une référence à un objet **Game**. En effet, comme tous nos programmes de démonstration ont un nom changeant, nous avons dû trouver un moyen d'avoir un point commun pour nos futures instances de **Scene**. Nous avons recours ici à une nouvelle interface mimant l'ensemble des méthodes implementée dans nos porgrammes, l'interface **Game**.

L'interface Game utilisée dans les scènes

```

public interface Game {
    void requestExit();
    void setDebug(int i);
    int getDebug();
    boolean isDebugGreater Than(int debugLevel);
    boolean isNotPaused();
    void setPause(boolean p);
    void setExit(boolean e);
    boolean isExitRequested();
    BufferedImage getRenderingBuffer();
    Config getConfig();
    boolean isKeyPressed(int keyCode);
}

```

Je ne détaillerai pas les méthodes, elles sont déjà présentes dans la class `MonProgrammeScene1`.

Node ?

Les entités et la scène sont en fait tous des éléments d'un gameplay. Nous allons les agencer dans une structure organisée, où les objets auront toutes une relation parent-enfant entre la scene et les objets.

Cela s'inscrit dans l'utilisation d'un arbre à nœuds, nous allons pour cela implémenter une nouvelle classe dont hériteront Entity et Scene.

```
public class Node<T> extends Rectangle2D.Double {  
    ①    private static long index = 0;  
    protected long id = index++;  
    protected String name = "node_" + (id);  
    ②    private Node<?> parent = null;  
    ③    private List<Node<?>> children = new ArrayList<>();  
    //...  
  
    ④    public void add(Node<?> c) {  
        c.parent = this;  
        this.children.add(c);  
    }  
    ⑤    public List<Node<?>> getChildren() {  
        return children;  
    }  
}
```

1. Nous allons profiter de l'occasion pour déplacer certaines fonctionnalités communes à la classe `Scene` et à la classe `Entity`; l'id, le nom et le compteur interne,
2. Enfin comme c'est un arbre, il est nécessaire d'avoir une racine pour le départ,
3. Nous aurons besoin d'ajouter des éléments dans l'arbre, la méthode `add` sera notre arme,
4. Enfin, nous aurons besoin de récupérer les enfants d'un nœud, comme les entités d'une scene par exemple.

```
public class Entity extends Node<Entity>{  
    //...  
}  
  
public class AbstractScene extends Node<Scene> implements Scene {  
    public abstract class AbstractScene extends Node<Scene> implements Scene {
```

```

public AbstractScene(String name) {
    super(name);
}

①
public Collection<Entity> getEntities() {
    return getChildren().stream()
        .filter(Entity.class::isInstance) // Filtrer les objets de type Entity
        .map(Entity.class::cast)          // Les convertir en Entity
        .collect(Collectors.toList());
}

②
public Entity getEntity(String name) {
    return (Entity) getChildren().stream().filter(c -> c.getName().equals(name))
        .findFirst().get();
}

}

```

L'AbstractScene doit implementer quelques surcouches à la classe **Node** :

1. La recherche des Entités de la scene passe maintenant par la collection héritée de la classe **Node**,
2. Enfin, la précédente implementation reposait sur une **Map** d'entité, nous n'avons plus cela à notre disposition, aussi, nous allons simuler la fonction **get** de la map en implémentant grâce aux lambdas java un filtre sur le nom des entités présentes dans la collection child.

Cela simplifie nos classes et aussi contribue à les structurer d'une façon intéressante.

Pour reprendre le diagramme de classes du début :

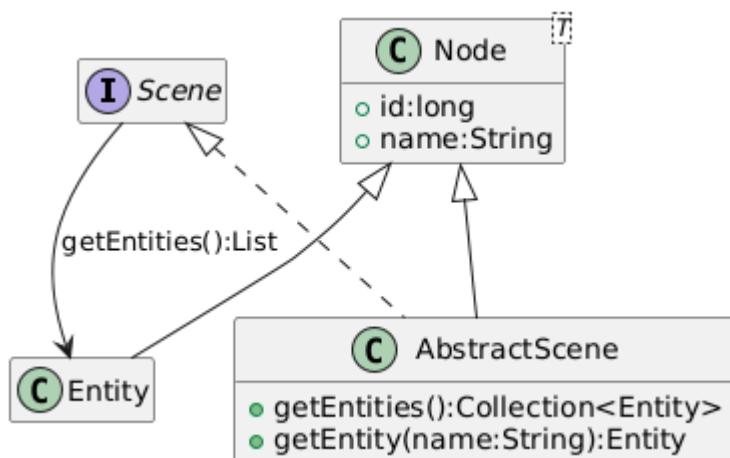


Figure 11. Node, Scene, AbstractScene et Entity dans un arbre

Pour reprendre le diagramme de classes du début :

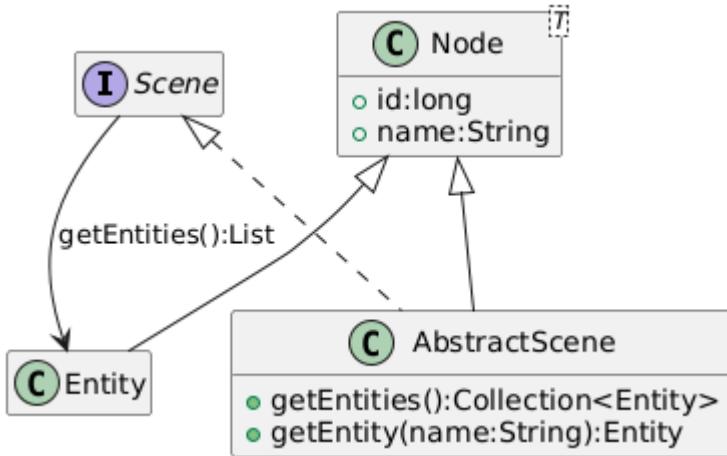


Figure 12. *Node, Scene, AbstractScene et Entity dans un arbre*

WARNING

TODO terminer le chapitre `Node` et l'affichage de la scene en mode arbre composé de nœuds.

Conclusion

Nous avons finalement obtenu un système qui permet une gestion fine des différents gameplay ou phases d'un jeu. La scene est maintenant organisée pour contenir, sous forme d'arbre, l'ensemble des entités qui la constitue.

Il est possible d'activer une autre scene, simplement en appelant la méthode `switchScene(String)` qui prend en charge son initialization, la création de ses entités, mais aussi auparavant vérifie si une autre est déjà activée, et procède à sa désactivation.

Dans le prochain chapitre, nous allons nous intéresser aux comportements et à l'interface `Behavior`.

Comportement dans les entités

Dans ce chapitre, nous allons nous concentrer sur la gestion des comportements au sein de la classe `MonProgrammeBehavior1`.

Qu'est-ce qu'un comportement

Dans le contexte du développement de jeux vidéo 2D, un comportement (`Behavior`) attaché à une entité (`Entity`) constitue un module de logique qui définit comment cette entité se comporte dans le jeu. Voici une explication détaillée de ce concept.

Un comportement est un ensemble de règles ou de logique de programmation qui détermine les actions d'une entité dans certaines conditions. Il peut représenter une compétence, une réaction à un événement ou un aspect fonctionnel de l'entité.

Les comportements sont conçus pour être modulaires, ce qui signifie qu'ils peuvent être facilement ajoutés, retirés ou modifiés indépendamment des autres aspects de l'entité. Cela permet aux développeurs d'étendre et de personnaliser les capacités des entités sans altérer leur structure de base.

Exemples de Comportements

- **Déplacement** : Un comportement qui permet à une entité de se déplacer à l'écran en réponse aux inputs du joueur ou à une IA.
- **Collision** : Gère la réponse de l'entité lors des interactions avec d'autres objets, tels que rebondir sur les murs ou subir des dommages.
- **Animation** : Modifie l'apparence de l'entité pour afficher des animations en fonction de son état ou de ses actions (marcher, sauter, attaquer).

Intégration avec l'Entité

- Chaque entité peut avoir plusieurs comportements associés. Par exemple, un personnage joueur pourrait avoir des comportements de déplacement, de saut, et d'attaque, chacun gérant un aspect différent de l'interaction de l'entité avec le monde du jeu.
- Ces comportements sont souvent implémentés comme des classes ou des méthodes qui sont appelées périodiquement, par exemple dans la boucle de jeu pour mettre à jour l'état de l'entité.

Avantages

- **Flexibilité** : Les comportements permettent de changer facilement la manière dont une entité se comporte sans avoir à toucher au code des autres parties du jeu.
- **Réutilisabilité** : Un même comportement peut être réutilisé par plusieurs entités différentes, favorisant ainsi une économie de code.
- **Maintenance Facilitée** : Puisqu'un comportement encapsule une fonctionnalité spécifique, il est plus facile de déboguer et d'entretenir le code.

Passons au code

Notre interface Behavior

L'interface `Behavior` propose un ensemble de méthodes pour implémenter des traitements spécifiques sur certain moment de la vie d'une entité.

```
public interface Behavior {  
    default void init(Entity e) {}  
    default void create(Entity e) {}  
    default void input(Entity e) {}  
    default void update(Entity e) {}  
    default void draw(Graphics2D g, Entity e) {}  
    default void dispose(Entity e) {}  
}
```

Nous utiliserons cette interface pour déplacer certains traitements au plus près des entités; par exemple dans le traitement des commandes entrées par le joueur pour faire déplacer l'entité "player":

```
public class MonProgrammeBehavior1 extends JPanel implements KeyListener, Game {  
    //...  
    public void create(Game app) {  
        // Création du player bleu  
        Entity player = new Entity("player")  
        //...  
        ① .add(new Behavior() {  
            ② @Override  
            public void input(Entity player) {  
                double speed = (double) app.getConfig().get("app.physic.entity.player.speed");  
                ③ if (app.isKeyPressed(KeyEvent.VK_LEFT)) {  
                    player.setVelocity(-speed, player.getDy());  
                }  
                if (app.isKeyPressed(KeyEvent.VK_RIGHT)) {  
                    player.setVelocity(speed, player.getDy());  
                }  
                if (app.isKeyPressed(KeyEvent.VK_UP)) {  
                    player.setVelocity(player.getDx(), -speed);  
                }  
                if (app.isKeyPressed(KeyEvent.VK_DOWN)) {  
                    player.setVelocity(player.getDx(), speed);  
                }  
            }  
        });  
    }
```

```

        add(player);
    }
}

```

1. Ajoutons une implementation en ligne de la **Behavior**,
2. Dans lequel nous ajoutons l'implémentation de **Behavior#input(Entity)**
3. Pour traiter l'appui des touches,

Nous allons devoir maintenant modifier notre programme principal pour appeler ces nouveaux traitements.

Délégation aux Comportements dans les phases du cycle de jeu

1. Initialisation (**init**)

- Lors de la phase d'initialisation de la scène contenant l'entité implémentant le **Behavior**, la méthode **init(Entity)** vous permettra de charger d'éventuelle(s) ressource(s) nécessaire(s) au traitement de ce **Behavior**.

2. Création de l'Entity parente (**create**)

- si vous devez initialiser des valeurs de variables lors de chaque création de la scène c'est dans cette méthode que vous devrez le faire.

3. Finaliser en restituant les ressources (**dispose**):

- Si dans la phase d'initialisation de l'implémentation du Behavior nous réservions des resources, c'est dans cette phase que nous pourrons les restituer.

Dans la méthode principale de la boucle **loop()** qui est responsable de l'exécution continue du cycle de jeu, le cycle est généralement constitué des étapes suivantes :

- gestion des entrées utilisateur (**input()**),
- mise à jour des états de jeu (**update()**),
- et rendu graphique (**render()**).

Voyons ensemble en détail ces étapes :

1. Gestion des Entrées Utilisateur (**input**):

- La méthode **input()** est appelée au début de chaque cycle de la boucle. Cette méthode délègue le traitement des interactions utilisateur aux comportements des entités en invoquant la méthode **input(this)** pour la scène actuelle, et ensuite pour chaque entité de la scène via **e.getBehaviors().forEach(b → b.input(e))**.
- Ce mécanisme permet à chaque comportement de déterminer quelle action doit être entreprise en réaction aux entrées (comme les appuis sur les touches du clavier), influençant directement le comportement de l'entité correspondante.

2. Mise à Jour des Etats (**update**):

- La méthode `update()` gère la logique de jeu à chaque itération. Pour chaque entité, des calculs sont effectués pour ajuster la position et la vitesse, en prenant en compte la physique et les interactions avec les limites de l'écran.
- Les comportements sont essentiels ici car ils peuvent influencer les décisions prises durant cette étape, par exemple, en intégrant des règles spécifiques de rebond, de friction, ou autres modifications dynamiques des propriétés de l'entité en fonction de son état actuel.

3. Rendu Graphique (`render`):

- Bien que le rendu principalement concerne l'affichage visuel, les comportements des entités peuvent influencer les données sur lesquelles le rendu se base, en modifiant les états ou les apparences avant cette phase.

Le code modifié pour exécuter les Behavior's

```
public class MonProgrammeBehavior1 extends JPanel implements KeyListener, Game {
    //...
    ①
    private void createScene() {
        //...
        currentScene.getEntities().forEach(e -> e.getBehaviors().forEach(b -> b.init(
e)));
        currentScene.getEntities().forEach(e -> e.getBehaviors().forEach(b -> b.
create(e)));
    }
    //...
    ②
    private void input() {
        currentScene.input(this);
        currentScene.getEntities().forEach(e -> e.getBehaviors().forEach(b -> b.input(
e)));
    }
    //...
    ③
    private void update() {
        // calcul de la position du player bleu en fonction de la vitesse courante.
        currentScene.getEntities().forEach(e -> {
            //...
            e.getBehaviors().forEach(b -> b.update(e));
        });
        //...
    }
    //...
    ④
    private void render() {
        //...
        // draw entities
        currentScene.getEntities().forEach(e -> {
            //...
            e.getBehaviors().forEach(b -> b.draw(g, e));
        });
    }
}
```

```

    //...
}

//...

⑤ private void dispose() {
    currentScene.getEntities().forEach(e -> e.getBehaviors().forEach(b -> b
.dispose(e)));
    //...
}
//...
}

```

1. Lors de la création de la scène, on appelle l'implémentation de `Behavior#create(Entity)`,
2. Sur le traitement des entrées réalisées au clavier, on appelle l'implémentation de `Behavior#input(Entity)`,
3. Lors du rendu des entités, on procède à l'appel de `Behavior#draw(Entity)`,
4. Sur le traitement principal de la mise-à-jour, nous devons traiter les `Behavior#update(Entity)`,
5. Enfin lors de la cloture de la scene et du jeu, on doit traiter la dernière méthode proposée par l'API: `Behavior#dispose(Entity)`,

Nous allons faire de même avec le comportement des ennemis :

La méthode create dans la nouvelle scène PlayBehaviorScene adaptée

```

public class MonProgrammeBehavior1 extends JPanel implements KeyListener,Game {
    //...
    public void create(Game app) {
        // Création du player bleu
        //...
        double maxEnemySpeedRatio = app.getConfig().get(
"app.physic.entity.enemy.max.speed.ratio");
        // Création de l'ennemi rouge
        for (int i = 0; i < 10; i++) {
            Entity enemy = new Entity("enemy_%d".formatted(i))
                .setPosition((Math.random() * (app.getRenderingBuffer().getWidth()
- 16)), (Math.random() * (app.getRenderingBuffer().getHeight() - 16)))
                .setElasticity(Math.random())
                .setFriction(Math.random())
                .setFillColor(Color.RED)
                .setShape(new Ellipse2D.Double(0, 0, 10, 10))
                .setAttribute("max.speed", (Math.random() * player.getAttribute(
"max.speed", 2.0) * maxEnemySpeedRatio))
            // On ajoute le comportement de suivi de l'instance d'Entity
            "player"
                .add(new Behavior() {
                    @Override
                    public void input(Entity enemy) {
                        Entity player = getEntity("player");
                        double eSpeed = (0.5 + Math.random() * 1.1);

```

```

        // Simulation pour les ennemis qui suivent le player sur
        // l'axe X,
        // but limited to 'max.speed' attribute's value
        double centerPlayerX = player.getX() + player.getShape()
        .getBounds().width * 0.5;
        double centerEnemyX = enemy.getX() + enemy.getShape()
        .getBounds().width * 0.5;
        double directionX = Math.signum(centerPlayerX -
        centerEnemyX);
        if (directionX != 0.0) {
            enemy.setVelocity(
                Math.min(directionX * eSpeed * enemy
        .getAttribute("max.speed", 2.0),
                enemy.getAttribute("max.speed", 2.0)),
            enemy.getDy());
        }

        // Simulation pour les ennemis qui suivent le player sur
        // l'axe Y,
        // but limited to 'max.speed' attribute's value
        double centerPlayerY = player.getY() + player.getShape()
        .getBounds().width * 0.5;
        double centerEnemyY = enemy.getY() + enemy.getShape()
        .getBounds().width * 0.5;
        double directionY = Math.signum(centerPlayerY -
        centerEnemyY);
        if (directionY != 0.0) {
            enemy.setVelocity(
                enemy.getDx(),
                Math.min(directionY * eSpeed * enemy
        .getAttribute("max.speed", 2.0),
                enemy.getAttribute("max.speed", 2.0)))
        };
    }
}
add(enemy);
}
}
// ...
}

```

Nous voilà fin prêt à lancer notre programme modifié :

Exécution des codes d'exemples dans le répertoire src/test

```
javac -d target/test-classes $(find src/test -name "*.java") && find src/test -name
"*.properties" -exec cp --parents {} target/test-classes \; && java -cp target/test-
classes MonProgrammeBehavior1
```

NOTE

Je vous propose de créer un petit script *bash* `runex.sh` pour exécuter n'importe quel exemple:

Script 'run.sh' d'exécution de la compilation et d'exécution d'un exemple

```
#!/bin/bash
# Executing example code from the src/test path
# Copyright 2024 - Frederic Delorme<frederic.delorme@gmail.com>
export SRC=src/test
export classpath=target/test-classes
```

echo START Execute "\$1" code example.

```
javac -d $classpath \ $(find $SRC -name ".java") && \ find $SRC -name ".properties" -exec cp -parents {} $classpath \; && \ java -cp $classpath "$1"
```

echo END Ending execution of "\$1".

Ce qui sera exécuté par :

```
./runex.sh MonProgrammeBehavior1
```

Vous obtiendrez quelques chose de similaire à la copie dcran ci-dessous (ici capturée sous linux Ubuntu 24.04) :



Figure 13. Le programme de démonstration des "Behavior"s.

Conclusion

Les comportements se comportent comme des modules autonomes attachés à chaque entité. Ils reçoivent les événements de la boucle de jeu et agissent en conséquence pour modifier l'état des entités. Cette structure permet d'ajouter, de modifier, ou de supprimer facilement des fonctionnalités au niveau des entités sans alourdir la logique centralisée de la boucle de jeu. Cela favorise une architecture claire et extensible, où de nouvelles logiques peuvent être ajoutées simplement en définissant de nouveaux comportements et en les associant aux entités pertinentes.

Camera, Monde et Aire de jeu

La caméra est le moyen le plus simple et bizarrement le plus imagé, pour suivre une autre entité à l'écran.

En effet, comme au cinéma la caméra va suivre le héros dans ses actions, dans le jeu, elle suivra le personnage dirigé par le joueur, comme lors d'une scène d'action dans un film.

Nous allons donc devoir implementer un nouvel objet qui se comportera comme une caméra de cinéma, à savoir qu'elle devra suivre une cible (**target**), qu'elle aura un cadre précis défini (**viewport**) et qu'elle suivra la cible avec une certaine souplesse, définie par un facteur numérique (**tweenFactor**).

Elle fera partie, comme les autres entités, de la scène. Plusieurs caméras pourront être définies et activées à la demande chacune leur tour.

Ainsi pourra être fait un jeu "d'angles de vues" pour rendre une action plus dynamique quand cela est nécessaire : par exemple faire le focus l'espace de quelques secondes sur un élément important de la scène, ou sur un ennemi, un objet particulier.

L'objet Camera

Avec tous les éléments décrits précédemment, nous pouvons maintenant imaginer notre object **Camera**.

Les attributs de notre Camera

```
public class Camera extends Entity{
    ①    private Entity target=null;
    ②    private Rectangle2D viewport=new Rectangle2D.Double(0,0,320.0,200.0);
    ③    private double tweenFactor=1.0;
}
```

1. **target** : On retrouve donc la cible pointée par la caméra,
2. **viewport** : le cadre de la caméra,
3. **tweenFactor** : le facteur de vitesse de suivi de la cible.

Nous ajoutons un constructeur qui appelle le constructeur parent :

Le constructeur unique

```
public class Camera extends Entity{
    //...
    public Camera(String name){
        super(name);
```

```
    }
    //...
}
```

Et nous ajouterons bien sûr les indispensables getters et setters respectant le concept de Fluent API.

Les getters et les Fluent API setters

```
public class Camera extends Entity{
    //...
    public Entity getTarget() {
        return target;
    }

    public Camera setTarget(Entity target) {
        this.target = target;
        return this;
    }

    public double getTweenFactor() {
        return tweenFactor;
    }

    public Camera setTweenFactor(double tweenFactor) {
        this.tweenFactor = tweenFactor;
        return this;
    }

    public double getRotation() {
        return rotation;
    }

    public Camera setRotation(double rotation) {
        this.rotation = rotation;
        return this;
    }

    public Rectangle2D getViewport() {
        return this;
    }

    public Camera setViewport(Dimension vp) {
        this.setRect(x, y, vp.width, vp.height);
        return this;
    }
    //...
}
```

Et enfin, il est important de mettre à jour la position de la caméra en fonction de la cible visée. Nous allons donc ajouter une méthode **update** "maison" :

Mise à jour de la position de la Camera en fonction de la position de la cible

```
public class Camera extends Entity{  
    //...  
    public void update(double elapsed) {  
  
        this.x = this.x + (((target.x - this.x) - (this.getBounds2D().getWidth() - target.width) * 0.5) * tweenFactor * elapsed);  
  
        this.y = this.y + (((target.y - this.y) - (this.getBounds2D().getHeight() - target.height) * 0.5) * tweenFactor * elapsed);  
    }  
    //...  
}
```

On positionne le centre du **viewport** de la caméra à la position du centre de la cible (**target**), mais avec un ratio **tweenFactor** de la distance restante en fonction du temps passé depuis le précédent appel, ce qui donne un effet plus doux sur le déplacement.

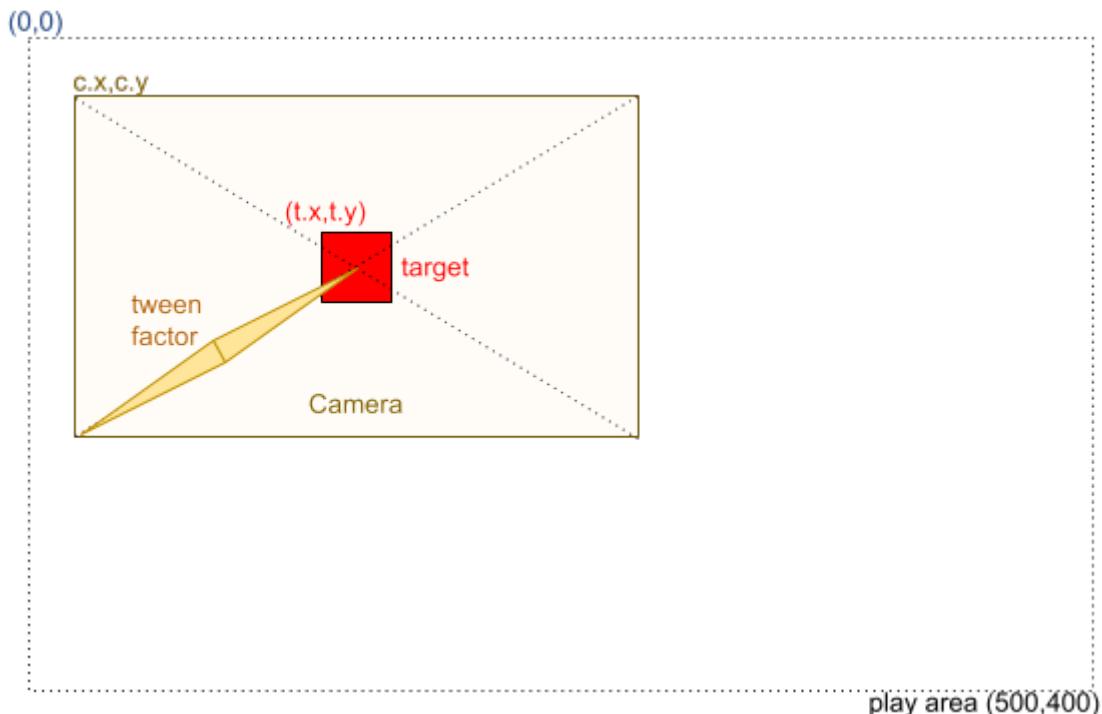


Figure 14. Illustration de la caméra fixant une cible ayant un cadre et un facteur de suivi.

Sur l'illustration ci-dessus, imaginer que le **tweenFactor** est un ressort avec amortisseur, reliant la cible et la caméra.

Plus le **tweenfactor** est élevé (proche de 1) et moins il y a d'amortissement et de douceur.

Plus le **tweenFactor** s'approche de 0.0 et moins la propension à la promptitude de la caméra à suivre la cible est forte. En clair et en décrypté : il aura donc un délai plus long à suivre la cible avec une valeur proche de zéro.

Intégrons la camera dans la Scene.

TODO

Modifions le process de mise-à-jour

Comme la caméra est ajoutée comme toute autre entité à la scène, nous devons en début de chaque mise à jour, appliquer le mécanisme standard d'update à tous les objets qui ne sont pas une caméra.

Ensuite, nous appellerons la méthode spéciale de mise à jour de l'objet caméra pour se positionner en fonction de la nouvelle position de l'entité suivie par celle-ci.

Processus de mise-à-jour de la caméra considérée comme active.

```
private void update() {  
    ①    currentScene.getEntities().stream().filter(e -> !(e instanceof Camera)).forEach(e  
    -> {  
        //...  
    });  
    ②    Optional<Entity> cam = currentScene.getEntities().stream().filter(e -> e instanceof  
    Camera).findFirst();  
    cam.ifPresent(entity -> ((Camera) entity).update(16.0));  
  
    currentScene.update(this);  
}
```

1. Nous filtrons tous les objets qui ne sont pas des caméras, afin d'appliquer le process standard,
2. le premier objet de type Camera que nous trouvons sera la camera active, et appelons sa mise à jour de position.

NOTE

Ce procédé sera changer avec l'adaptation de la Scene, lorsque nous ajouterons d'autres attributs à celle-ci.

Modifions le rendu graphique

Une fois l'instance de la **Camera** repérée dans les objets de la **Scene**, nous allons utiliser la position de son cadre (**viewport**) pour déplacer le point de vue de rendu.

Rendu du point de vue de la Camera.

```
private void render() {  
    Graphics2D g = renderingBuffer.createGraphics();  
    // clear rendering buffer to black  
    g.setColor(Color.BLACK);
```

```

g.fillRect(0, 0, renderingBuffer.getWidth(), renderingBuffer.getHeight());
①
Optional<Entity> cam = currentScene.getEntities().stream().filter(e -> e
instanceof Camera).findFirst();

// draw entities
currentScene.getEntities().stream()
.filter(e -> !(e instanceof Camera))
.forEach(e -> {
②
    if (cam.isPresent()) {
        Camera camera = (Camera) cam.get();
        g.translate((int) -camera.getX(), (int) -camera.getY());
    }
③
    drawEntity(e, g);
④
    if (cam.isPresent()) {
        Camera camera = (Camera) cam.get();
        g.setColor(Color.gray);
        g.draw(camera.getShape());
        g.translate((int) camera.getX(), (int) camera.getY());
    }
}

// Exécuter les comportements de dessin pour cette instance d'Entity.
e.getBehaviors().forEach(b -> b.draw(g, e));
});

currentScene.draw(this, g);
//...
}

```

1. Nous isolons la première instance de **Camera** dans la scene active,
2. Nous déplaçons le point de vue à la position du cadre de la camera (viewport),
3. Nous procédons au rendu de l'entité en cours,
4. Nous ramenons la position du point de vue à sa place initiale pour passer à l'entité suivante.

Si nous exécutons maintenant notre programme de test de la caméra :

```
./runex.sh MonProgrammeCamera1
```

Nous obtiendrons un rendu similaire à ce que présenta la capture d'écran ci-dessous :

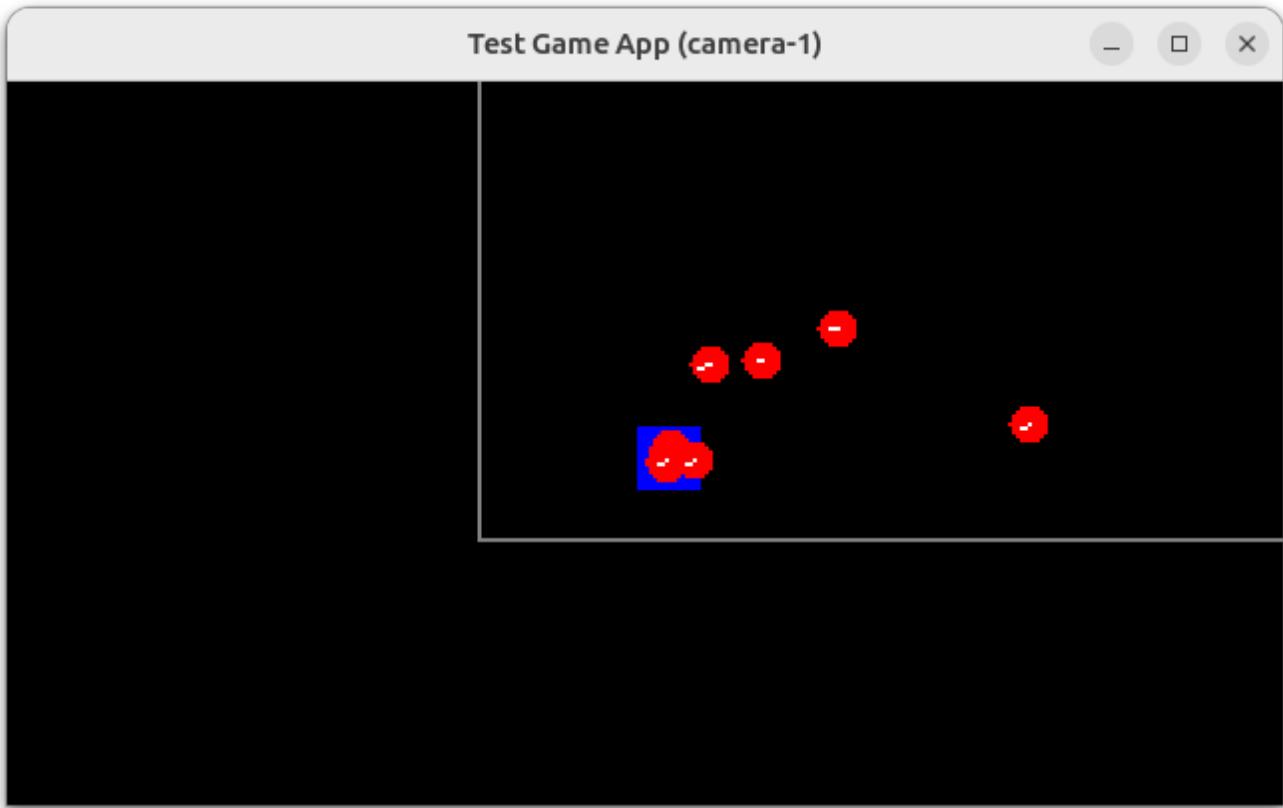


Figure 15. Capture la démo MonProgrammeCamera1

Allons un peu plus loin

Dans cette capture d'écran le cadre gris délimite la zone actuelle jeu.

Ajoutons un rectangle jaune délimitant le cadre du viewport.

Ajout du dessin d'information visuel de débogage de la caméra

```
private static void drawDebugCamera(Graphics2D g, Camera camera) {  
    g.setFont(g.getFont().deriveFont(9.0f));  
    g.setColor(Color.yellow);  
    Rectangle2D drawCamera = new Rectangle2D.Double(  
        camera.getBounds2D().getX() + 20,  
        camera.getBounds2D().getY() + 30,  
        camera.getBounds2D().getWidth() - 40,  
        camera.getBounds2D().getHeight() - 40);  
    g.draw(drawCamera);  
    g.drawString(  
        "#%d:%s".formatted(camera.getId(), camera.getName()),  
        (int) (camera.getBounds2D().getX() + 20  
            + camera.getBounds2D().getWidth() * 0.70),  
        (int) (camera.getBounds2D().getY()  
            + camera.getBounds2D().getHeight() - 14));  
}
```

Et dans la méthode `render` principale:

Modification de la méthode de rendu

```
private void render() {
    Graphics2D g = renderingBuffer.createGraphics();
    //...
①    Optional<Entity> cam = currentScene.getEntities().stream().filter(e -> e
 instanceof Camera).findFirst();
    Camera camera = cam.isPresent() ? (Camera) cam.get() : null;
②    if (cam.isPresent() != null) {
        g.translate((int) -camera.getX(), (int) -camera.getY());
    }
    drawWorldLimit(g, currentScene.getWorld(), 16, 16);
    if (cam.isPresent() != null) {
        if (isDebugGreater Than(1)) {
            drawDebugCamera(g, camera);
        }
        g.translate((int) camera.getX(), (int) camera.getY());
    }
    // draw entities
    currentScene.getEntities().stream()
        .filter(e -> !(e instanceof Camera))
        .forEach(e -> {
            //...
        });
    //...
}
```

Si nous relançons maintenant notre programme de test :

```
runex.sh MonProgrammeCamera1
```

Nous verrons s'afficher la fenêtre suivante, proposant un contenu approchant :

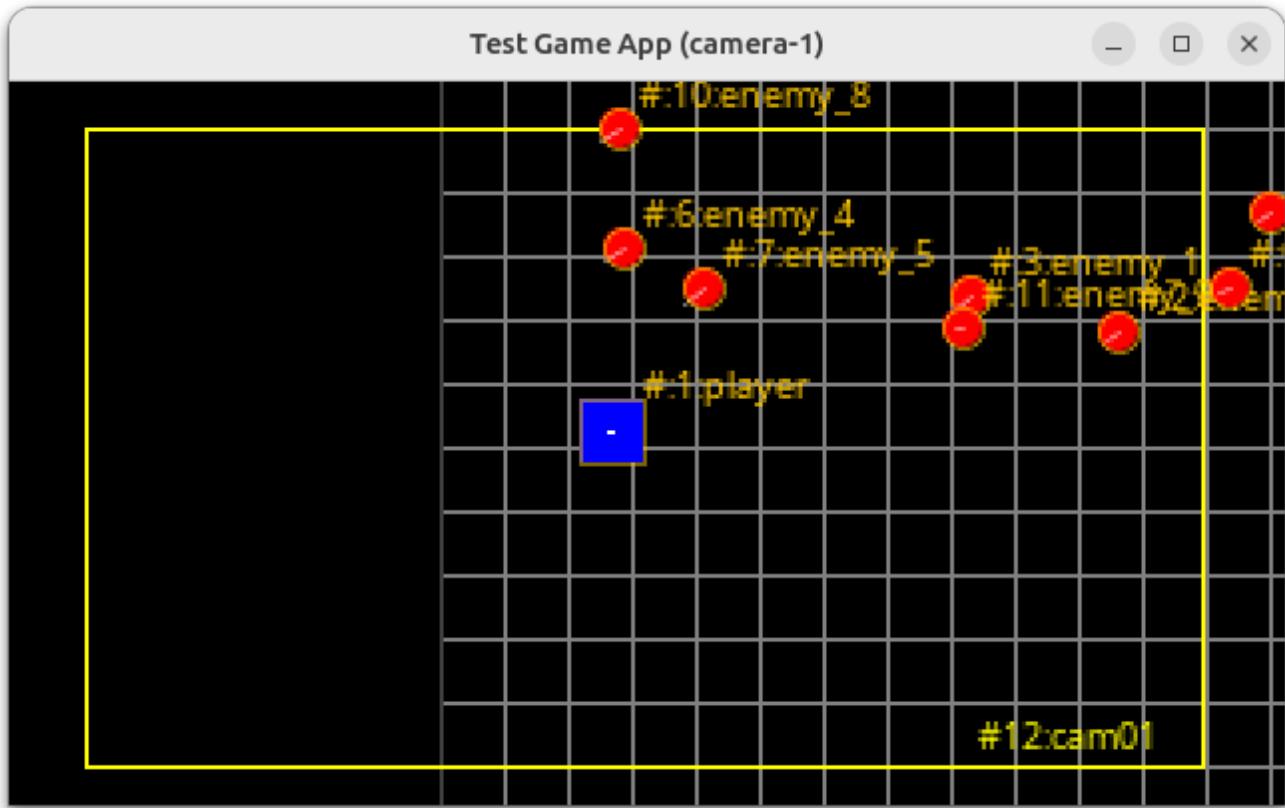


Figure 16. Affichage d'information visuel de débogage sur l'objet Camera nommé "cam01".

Si nous pressons la combinaison de touche **CTRL + D**, nous verrons changer le niveau d'information de débogage affichée à côté de chaque entité.

World, un nouveau monde

Ajoutons un nouvel objet à notre set de jeu, le monde ! Afin de mieux gérer le contexte pour l'ensemble des entités évoluant dans notre jeu, nous allons ajouter une classe qui nous permettra de définir certains traits communs.

La classe **World** servira à définir :

1. Les dimensions de l'espace de jeux dans un **Rectangle2D**,
2. La gravité (**gravity**), nouvelle venue dans notre moteur de jeux, servira à appliquer une force définie à l'ensemble des entités contenues dans l'espace de jeu correspondante à la gravité de notre environnement. Par défaut à zéro, elle pourra être définie à $9,81m/s^2$, soit l'accélération liée à **la gravité terrestre**.

Ainsi, dans le schéma ci-dessous, nous pouvons voir plusieurs entités (carrés jaunes) évoluant dans la même zone de jeu (cadre vert), contraintes dans l'espace défini par celle-ci, et subissant la gravité (flèche bleue orientée vers le bas).

TIP

TODO Ajouter un schéma montrant quelques entités évoluant dans une zone de jeux et en subissant les contraintes

Commençons par créer un nouvel exemple en se basant sur le programme précédent :

Copions la classe précédente `MonProgrammeCamera1` vers une nouvelle classe `MonProgrammeCamera2`.

Ensuite, nous allons créer la classe World :

La classe World définissant notre monde.

```
import java.awt.geom.Rectangle2D;
public class World extends Rectangle2D.Double{
    private Point2D gravity;

    public World(){
        gravity = new Point2D.Double(0.0,0.0);
        this.setRect(new Rectangle2D.Double(0,0,640,400));
    }
    public World(Rectangle2D playArea, Point2D g){
        this.gravity=g;
        this.setRect(playArea);
    }
    public World setGravity(Point2D g){
        this.gravity=g;
        return this;
    }
    public World setPlayArea(Rectangle2D pa){
        this.setRect(pa);
        return this;
    }
}
```

Nous devons adapter notre Scene pour qu'elle définisse un objet World:

Notre Scene PlayCameraScene2 adapté à l'objet World

```
public class PlayCameraScene2 extends AbstractScene {

    public PlayCameraScene2(String name) {
        super(name);
    }

    @Override
    public void create(Game app) {
        ①
        setWorld(
            new World(
                new Point2D.Double(0, -0.981),
                new Double(0, 0, 30*16, 20*16)));
        //...
    }
    //...
}
```

En utilisant l'interface `setWorld(World)` définie par `AbstractScene`, nous pouvons déclarer une instance de la classe `World` qui définit le monde de notre jeu pour l'ensemble des entités de la `Scene`.

Ici, nous définissons un monde délimitant une zone de jeu de **480** points en largeur et **320** points en hauteur, ainsi qu'une gravité définie via une instance de `Point2D` de `(0.0,-0.981)`, soit la gravité terrestre de **9,81m/s²**.

NOTE Nous utilisons ici un object de type `Point2D`, fourni par le JDK afin de définir un pseudo vecteur. Nous aurons dans un prochain chapitre l'opportunité d'implémenter notre propre classe `Vector2D` permettant les opérations sur les vecteurs en 2 dimensions quand nous travailler sur une simulation physique un peu plus poussée.

Nous allons aussi modifier les méthodes `render` et `update` de notre programme de démonstration pour qu'elles prennent en compte notre nouvelle classe.

Render

Modifions la méthode de rendu en intégrant l'attribut `World` de notre `PlayCameraScene2`

```
public class MonProgrammeCamera2 extends TestGame implements KeyListener, Game {
    //...
    private void render() {
        Graphics2D g = renderingBuffer.createGraphics();
        // ...
        if (camera != null) {
            g.translate((int) -camera.getX(), (int) -camera.getY());
        }
    }

    ①
    drawWorldLimit(g, currentScene.getWorld(), 16, 16);

    if (camera != null) {
        if (isDebugGreaterThan(1)) {
            drawDebugCamera(g, camera);
        }
        g.translate((int) camera.getX(), (int) camera.getY());
    }
    //...
}

//...
```

②

```
private void drawWorldLimit(Graphics2D g, World world, int tileSize, int
tileHeight) {
    // draw the world limit.
    g.setColor(Color.GRAY);
    for (int ix = 0; ix < world.getWidth(); ix += tileSize) {
        for (int iy = 0; iy < world.getHeight(); iy += tileHeight) {
            g.drawRect(ix, iy,
                tileSize, tileHeight);
```

```

        }
    }
    g.setColor(Color.DARK_GRAY);
    g.draw(world);
}
//...
}

```

1. Dans la méthode `render`, nous passons maintenant l'objet `World` issue de la scène : `currentScene.getWorld()`
2. La méthode `drawWorldLimit` prendra maintenant un objet `World` comme attribut afin de dessiner une représentation de cet objet.

Update

Ajoutons également la mise à jour des positions de nos entités présentes dans la scène active en prenant en compte notre nouvel objet `World`:

```

public class MonProgrammeCamera2 extends TestGame implements KeyListener, Game {
    //...
    private void update() {
        currentScene.getEntities().stream().filter(e -> !(e instanceof Camera))
        .forEach(e -> {
            ①
            World world = currentScene.getWorld();
            ②
            e.setPosition(
                e.getX() + e.getDx() - (world.getGravity().getX()),
                e.getY() + e.getDy() - (world.getGravity().getY()));

            ③
            if (!world.contains(e)) {
                applyBouncingFactor(world, e);
            }
            ④
            e.setPosition(
                Math.min(Math.max(e.getX(), world.getX()), world.getWidth() -
e.getWidth()),
                Math.min(Math.max(e.getY(), world.getY()), world.getHeight() -
e.getHeight()));
            }
            //...
        });
        // ...
    }

    ⑤
    private void applyBouncingFactor(World world, Entity e) {
        // application du rebond si collision avec le bord de la zone de jeu
        if (e.getX() < world.getX()
            || e.getX() + e.getWidth() > e.getWidth() + world.getWidth()) {

```

```

        e.setVelocity(-e.getDx() * e.getElasticity(), e.getDy());
    }
    if (e.getY() < world.getY()
        || e.getY() + e.getHeight() > world.getHeight()) {
        e.setVelocity(e.getDx(), -e.getDy() * e.getElasticity());
    }
}
//...
}

```

1. Nous récupérons l'objet **World** de la scène,
2. Nous calculons la nouvelle position de l'entité en lui appliquant sa vitesse ET la gravité issue de l'objet World,
3. Nous appliquons l'effet de rebond si nécessaire (voir (5)),
4. Nous repositionnons l'entité dans la zone de jeu si sa nouvelle position sort du cadre de la zone de jeu,
5. Et enfin, nous échangeons l'objet **Rectangle2D** pour un objet **World** dans la signature de la méthode de gestion du rebond.

Conclusion

Nous avons ajouté à notre moteur quelques éléments importants qui serviront de base à d'autre principes que nous allons développer dans les chapitres suivants, comme les systèmes de particules qui feront appel à l'utilisation des comportements spécifiques (**Behaviors**), ainsi qu'à la structure en arbres

Délégation de la boucle de jeu

Afin de vous plonger dans l'ambiance ad hoc, je pense qu'il serait utile de se refaire une courte histoire de la technologie des années 80-90, qui a imposé un "moule" de fabrication des jeux.

Un peu d'histoire

En effet, les processeurs des ordinateurs de l'époque, où le PC n'était pas encore le maître absolu, une très grande variété de machines différentes, appelées "ordinateurs personnels" ou "Personal Computer" (PC), peuplaient le marché vidéoludique naissant.

Du Commodore 64, au ZX Spectrum en passant les exotiques Sinclair et TI 994/A, les machines d'alors proposaient toutes des architectures et des capacités différentes. Elles avaient aussi toutes deux choses en commun : la basse fréquence de leur processeur, et un affichage sur des écrans dits de technologie cathodique, et PAS de carte graphique,

Seuls les gestionnaires d'entrées/sorties plus ou moins performant, donnaient lieu à des qualités d'affichages allant du gris-vert, de la nuance de gris sur 16 niveaux, jusqu'à la palette de 256 couleurs pour les plus avancés.



Figure 17. L'écran cathodique.

Cela a imposé une façon de faire qui est devenue une norme, forcée par les limitations et la conception d'alors : la GameLoop.

Entrons un peu plus dans les détails, car c'est de là que vient la structure de code que nous allons appliquer.

L'écran cathodique, aussi appelé CRT (cf. figure 1), est l'ancienne technologie d'affichage de la télévision (TV) et de l'ordinateur.

Cette technologie est basée sur la création d'un faisceau d'électrons concentré, qui est dirigé vers une surface phosphorescente, le tout au sein d'un environnement sous vide, dans une grosse ampoule en verre résistant à la pression négative.

Pardon, vous dites ?

Oui cette grosse ampoule, comme les ampoules que nous utilisions pour nous éclairer avant l'avènement de la LED contenant un filament, en un peu plus robuste, car elle doit résister à une pression négative d'une atmosphère (-1 ATM).

À la base de cette ampoule, se cache un émetteur d'électron constitué d'une anode et d'une cathode.

L'une émet des électrons, la deuxième, présente en plusieurs parties, permet de transformer le flux d'électron en un fin faisceau. Une fois dirigé à l'aide de plaques de déviation, ira frapper la substance photo fluorescente présente en couche sur la partie intérieure de l'ampoule servant d'affichage (figure 2).

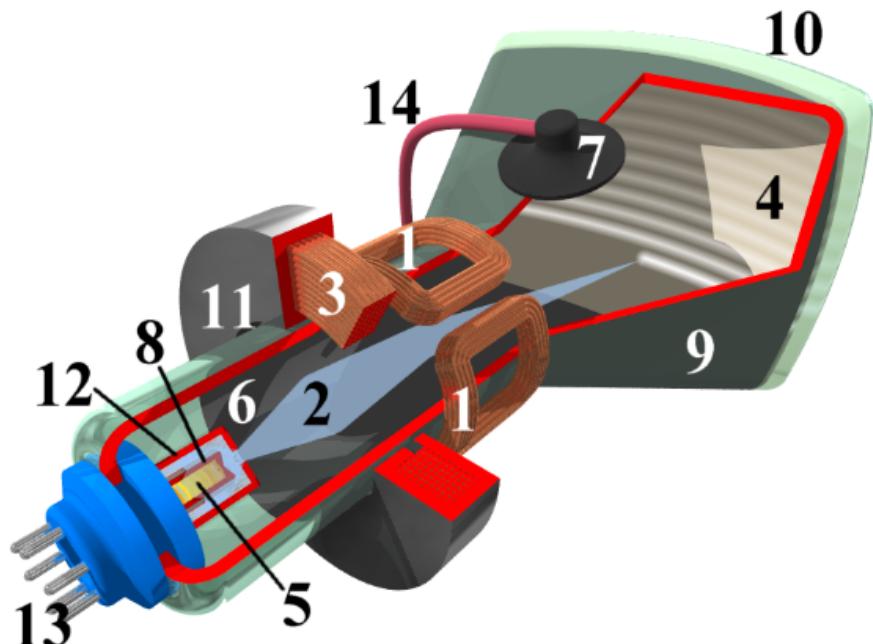


Figure 18. Éclaté technique de l'écran cathodique

1. Bobines de déflexion
2. Faisceau d'électrons
3. Bobine de concentration
4. Couche phosphorescente
5. Filament de chauffe de la cathode
6. Couche de graphite sur l'intérieur du tube
7. Joint en caoutchouc pour l'alimentation de l'anode
8. la Cathode
9. Le "Tube" ou Ampoule résistant à la pression
10. L'écran
11. Collier en fer de la Bobine de concentration
12. Electrode de contrôle de la puissance pour gérer la luminosité du faisceau
13. Connecteur de puissance pour le faisceau
14. Fil le tension de l'anode.

C'est quoi le rapport ?

Ok, mais POURQUOI diable vous parle-je de cela ? En quoi cela nous force une façon de coder ? J'y viens, patience.

Donc le faisceau ainsi généré vient littéralement peindre la surface intérieure du tube cité écran, en passant au travers une très fine grille dont les trous sont finement réalisés avec un espacement savamment calculé. Ces petits trous sont les constituants de l'image informatique que nous connaissons tous : les pixels ! Maintenant, je pense que vous comprenez où je veux en venir. Ce pinceau balaye l'image, ligne par ligne, puis revient à sa position de départ. Il réalise ce parcours à une vitesse assez élevée pour que l'œil et le cerveau humain ne puissent pas voir le faisceau, mais l'ensemble des points illuminés par celui-ci, cette suite de point constituant ainsi une IMAGE !

La fréquence en deçà de laquelle le cerveau et l'œil bénéficie de cette rémanence est de 24 images par seconde. Pour des raisons techniques liées à la fréquence d'alimentation de ces écrans qui varie en fonction des continents et des pays de 50 à 60 Hz.

L'image est constituée par la superposition de DEUX balayages consécutifs de lignes paires, puis de lignes impaires.

À cette fréquence, soit 50 ou 60 images/ secondes à raison d'une "demi" image à chaque passage, la rémanence créée par l'œil et le cerveau humain fait que nous ne voyons finalement qu'une seule image constituée de pixels !

Passons au code

Durant le parcours de retour du faisceau entre ces demi-images, le faisceau est éteint, afin de ne pas voir un gros trait en travers de l'écran.

Vous savez, celui que vous faites en tirant un trait à l'aide d'une règle et d'un crayon HB alors que la règle glisse subitement vers le bas !

Toute l'astuce du CODE d'un jeu consistait à effectuer les opérations de calculs de l'ensemble des éléments du jeu durant ce court laps de temps où le faisceau est éteint. Durant ce laps de temps, le processeur de l'ordinateur pouvait être pleinement utilisé pour la mécanique de jeu, et non pas pour gérer les entrées sorties du gestionnaire graphique chargé de piloter l'écran. Il en résultait une boucle de code sereinement appelée la "Game-Loop" où l'on traitait les différentes informations nécessaires :

Début de la boucle

- |_ la capture des actions du joueur sur le clavier ou la manette de jeu,
- |_ mettre à jour l'ensemble des objets interagissant dans la scène du jeu,
- |_ réaliser les opérations d'affichage!

Fin de la boucle

C'est ici la boucle que nous avons précédemment implémentée. Nous nous proposons cette fois de déléguer la boucle à une classe spécialisée qui nous permettra dans un second temps de proposer des alternatives à cette fameuse boucle.

Nous commencerons par une interface définissant le cadre de nos implementations : [GameLoop](#).

```
public interface GameLoop{
```

```

void process(Game game);
void input(Scene scene);
void update(Scene scene, double elapsed);
void render(Scene scene);
void setExit(boolean exitRequest);
void setPause(boolean pauseRequest);
}

```

1. `process` est le point d'entrée d'exécution de la boucle de jeu, c'est elle qui est chargée d'appeler les autres méthodes de traitement de la boucle,
2. `input` délègue le traitement des entrées utilisateurs (joueurs),
3. `update` lance la mise-à-jour de la scène active,
4. `render` procède au rendu de la scène active,
5. `setExit` permet de définir une demande de sortie de l'exécution de la boucle,
6. `setPause` demandera la mise en pause de l'exécution de la méthode `update`.

Voilà ce qui attendu par notre nouvelle interface de traitement de la boucle de jeu.

L'intégration de cette interface dans notre programme de démonstration `MonProgrammeGameLoop1` se fait en modifiant principalement la méthode `loop` présente dans la classe.

Intégration de GameLoop de MonProgrammeGameLoop1

```

public class MonProgrammeGameLoop1 extends TestGame implements KeyListener, Game {
    //...
    public void loop() {
        ① StandardGameLoop standardGameLoop = new StandardGameLoop(this);
        ② standardGameLoop.loop();
    }
    //...
}

```

1. Nous instancions l'implémentation désirée de l'interface `GameLoop`, ici, la classe `StandardGameLoop` dont nous verrons le code juste après,
2. L'exécution de la boucle est déléguée à la classe instanciée via la méthode `GameLoop.loop()`

Si nous regardons dans le détail le diagramme d'appel ci-dessous, nous voyons que la boucle est réellement exécutée dans l'instance de `GameLoop`, appelant en contre-partie les traitements nécessaires dans la classe parente (ici symbolisée par `Game`).

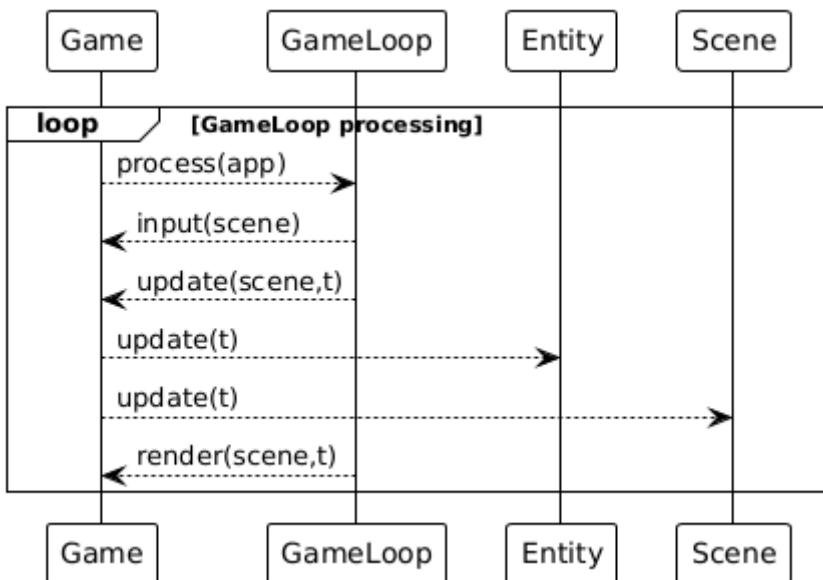


Figure 19. Delegation de la Boucle de jeu à l'interface spécialisée GameLoop

Passons à une implementation correspondante à ce que nous avions précédemment, mais en utilisant cette fameuse interface.

Implementation de StandardGameLoop

En partant des signatures de l'interface GameLoop, nous nous allons de réaliser notre premiere implémentation de la boucle de jeu, proposant ici une version assez standard, permettant de calculer le temps écoulé entre dans chaque boucle.

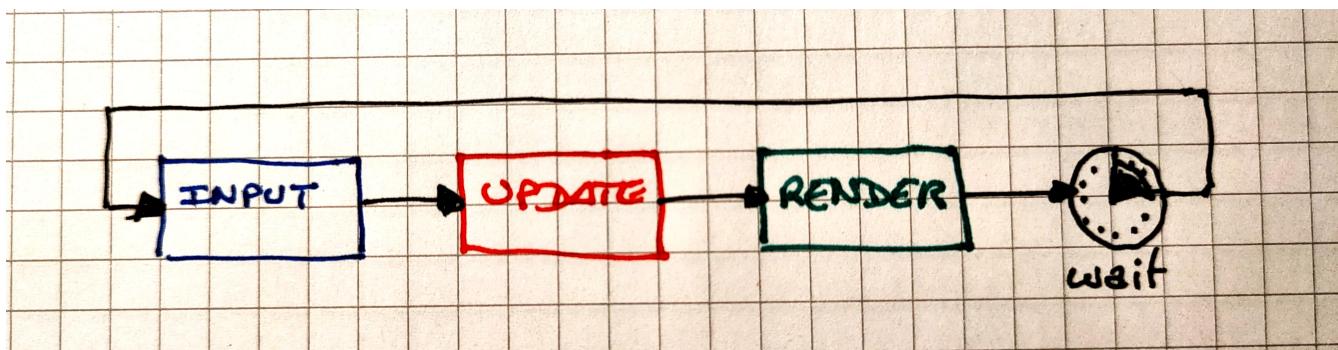


Figure 20. La boucle de jeu standard (image inspirée du livre "Game Programming Patterns" de Robert Nystrom.)

Nous allons aussi reprendre les fonctionnalités existantes dans la demo, à savoir, un mode de test permettant de limiter le nombre de boucles à exécuter, les valeurs nécessaires étant définies dans le fichier configuration.

```

public class StandardGameLoop implements GameLoop {
    private final MonProgrammeGameLoop1 game;
    ①
    public StandardGameLoop(MonProgrammeGameLoop1 monProgrammeGameLoop1) {
        this.game = monProgrammeGameLoop1;
    }
}

```

```

② public void process(Game game) {
    Scene scene = game.getCurrentScene();
    int loopCount = 0;
    int frameTime = 1000 / (int) (game.getConfig().get("app.render.fps"));
    long elapsed = 0;
    long startLoop = System.currentTimeMillis();
    long endLoop = startLoop;
    while (!game.isExitRequested()
        && ((game.isTestMode()
            && loopCount < game.getMaxLoopCount()) || !game.isTestMode())))
    {
        elapsed = endLoop - startLoop;
        startLoop = endLoop;
        input(scene);
        update(scene, elapsed);
        render(scene);
        loopCount++;
        waitTime(frameTime);
        endLoop = System.currentTimeMillis();
    }
    System.out.printf("=> Game loops %d times%n", loopCount);
}

③ public void waitTime(int delayInMs) {
    try {
        Thread.sleep(delayInMs);
    } catch (InterruptedException e) {
        System.err.println("Unable to wait 16 ms !");
    }
}

@Override
public void input(Scene scene) {
}

④     game.input(scene);
}

@Override
public void update(Scene scene, double elapsed) {
}

⑤     game.update(scene);
}

@Override
public void render(Scene scene) {
}

⑥     game.render(scene);
}

@Override
public void setExit(boolean exitRequest) {
}

```

```

    ...
}

@Override
public void setPause(boolean pauseRequest) {
    ...
}
}

```

1. Initialisation de l'implémentation de la boucle de jeu (**StandardGameLoop**) en lui passant l'instance parente du jeu (**Game**),
2. C'est dans la méthode **process(Game game)** est le cœur de la boucle, ici l'implémentation est basée sur le respect d'un temps de "frame",
3. La méthode **wait** donne le 'la' pour déterminer le temps de refresh de la boucle,
4. La gestion des entrées (**input**) est déléguée à l'instance du jeu,
5. La mise-à-jour (**update**) de la scene et de ses entités est déléguée à l'instance du jeu,
6. Le rendu visuel de l'ensemble des composants de la scene (**render**) est également déléguée à l'instance du jeu.

Les autres méthodes **setXxx()** font le pont entre l'instance du jeu et l'implémentation de la **GameLoop** afin de gérer les demandes de sortie du jeu ou de mettre l'exécution de la boucle en mode pause. Dans ce dernier cas, seule la mise-à-jour (**update**) est mise en pause.

Conclusion

Nous avons ici délégué le traitement de la boucle à une classe dite spécialisée, ce qui permettra plus tard de proposer d'autre implémentation permettant par exemple de privilégier le traitement des updates vis-a-vis du rendu, permettant ainsi d'éviter l'apparition de problèmes de collision, de calcul physique, etc.

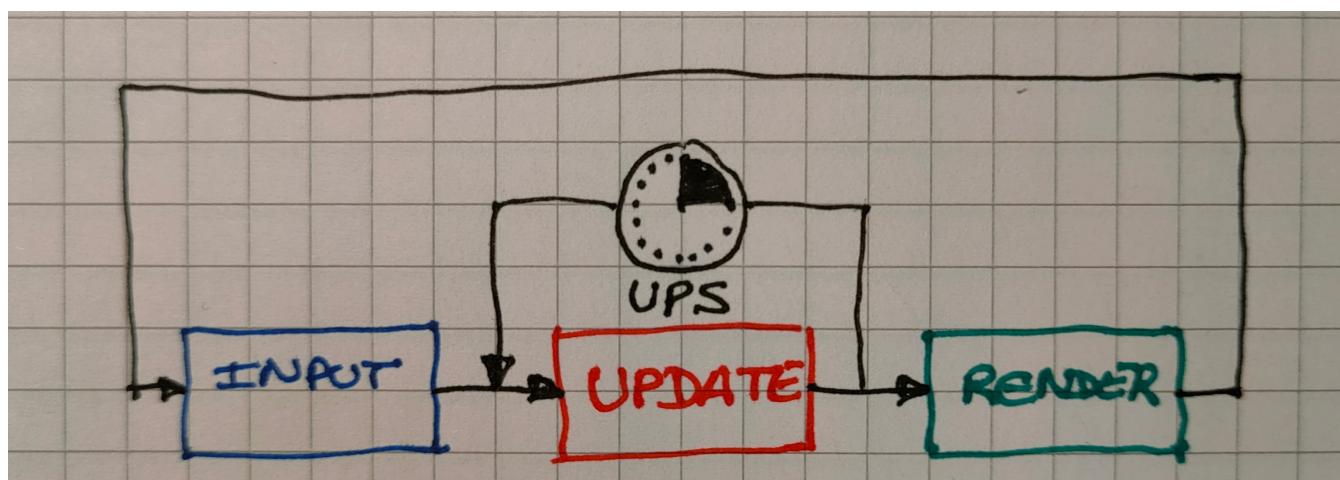


Figure 21. Exemple de boucle de jeu alternative basée sur le principe l'"update over rendering".

Dans le prochain chapitre, nous allons passer à la détection des collisions entre entités, et au traitement de celles-ci.

Collision et réponse

Nous avons réalisé un certain nombre d'évolutions sur la gestion et l'animation des entités. Et si nous ajoutons un peu d'interaction entre les entités ? Il est temps de détection les possibles collisions et de gérer leur réponse. Nous allons créer un nouvel objet dont la spécialisation consistera à détecter les collisions entre ceux-ci, et de déclencher le comportement souhaité.

La Détection

Si nous nous souvenons bien, notre classe `Entity` est une extension de la classe `Rectangle2D` issue du JDK, depuis l'introduction de la notion de `Node`. Nous allons donc de nouveau profiter de cet héritage pour détecter l'intersection entre 2 entités, cela permettant de lever un nouvel événement appelé `CollisionEvent`.

Pour cela, l'algorithme est assez simple, il consiste à comparer chaque entité aux autres, et s'il y a intersection, cela signifie qu'il y a collision.

Ajout de la détection de collision

```
public class MonProgrammeCollision1 extends TextGame implements KeyListener, Game{  
    //...  
    private void update() {  
        currentScene.getEntities().stream().filter(e -> !(e instanceof Camera))  
.forEach(e -> {  
            // Calcul de la nouvelle position en tenant compte de l'objet World issue  
            de la scène active.  
            //...  
        });  
        ① detectCollision(currentScene);  
  
        Optional<Entity> cam = currentScene.getEntities().stream().filter(e -> e  
instanceof Camera).findFirst();  
        cam.ifPresent(entity -> ((Camera) entity).update(16.0));  
  
        currentScene.update(this);  
    }  
    //...  
}
```

1. La méthode ici appelée, `detectCollision(currentScene)` possède l'implémentation suivante

Détection de collision, approche simple.

```
private void detectCollision(Scene scene) {  
    collisions.clear();  
    ① scene.getEntities().forEach(e1 -> {  
        scene.getEntities().forEach(e2 -> {
```

```

②     if (!e1.getName().equals(e2.getName())) {
③         if (e1.intersects(e2)) {
            CollisionEvent ce = new CollisionEvent(e1, e2);
④             collisions.add(ce);
        }
    );
}

```

1. Propose le parcours dans une double boucle imbriquée de toutes les entités,
2. En prenant garde à écarter les cas où les 2 entités sont en fait le même objet,
3. Et de détecter toute possible intersection entre les 2 entités issues de chacune des boucles,
4. Enfin le cas échéant, ajouter un **CollisionEvent** impliquant les 2 entités à la liste des collisions.

L'objet **CollisionEvent** est un record, sorte de classe immuable une fois créée :

Le record CollisionEvent

```

public record CollisionEvent(Entity e1, Entity e2) {
}

```

Après l'exécution d'une itération de boucle, nous obtenons une liste de collision. Si celle-ci n'est pas vide, nous pouvons traiter les réponses désirées.

Reponse aux collisions

```

private void resolveCollision(List<CollisionEvent> collisions) {
    collisions.forEach(ec->{
        Entity e1 = ec.e1();
        Entity e2= ec.e2();

    });
}

```

Nouvelle Scene

Nous allons maintenant créer à partir de la scène précédente PlayGameLoop1 une nouvelle scène permettant d'expérimenter la nouvelle sorte de **CollisionBehavior**:

```

public class PlayCollisionScene1 extends AbstractScene {
    @Override
    public void create(Game app) {
        //...
        // Création du player bleu
        Entity player = new Entity("player")
            .setPosition(
                ((getWorld().getWidth() - 16) * 0.5),
                ((getWorld().getHeight() - 16) * 0.5))
        //...
        ①   .add((CollisionBehavior) new CollisionBehavior() {
            @Override
            public void collide(Entity e1, Entity e2) {
                ②       if(e2.getName().startsWith("enemy")){
                    e2.setFillColor(Color.ORANGE);
                }
            }
        });
        add(player);
        //...
    }
    //...
}

```

1. Ajoutons une nouvelle implementation de l'interface **Behavior**, une déclaration en ligne de **CollisionBehavior** sur l'`Entity` nommée "player".
2. L'implémentation permet de traiter la collision avec une entité dont le nom commence par "enemy", le cas échéant, la couleur de remplissage de l'entité "enemy" est passée en ORANGE.

Ainsi, lors de l'exécution de notre classe **MonProgrammeCollision1**, lors de chaque collision entre le "player" et un "ennemi", ce dernier se voit alors coloré de orange.

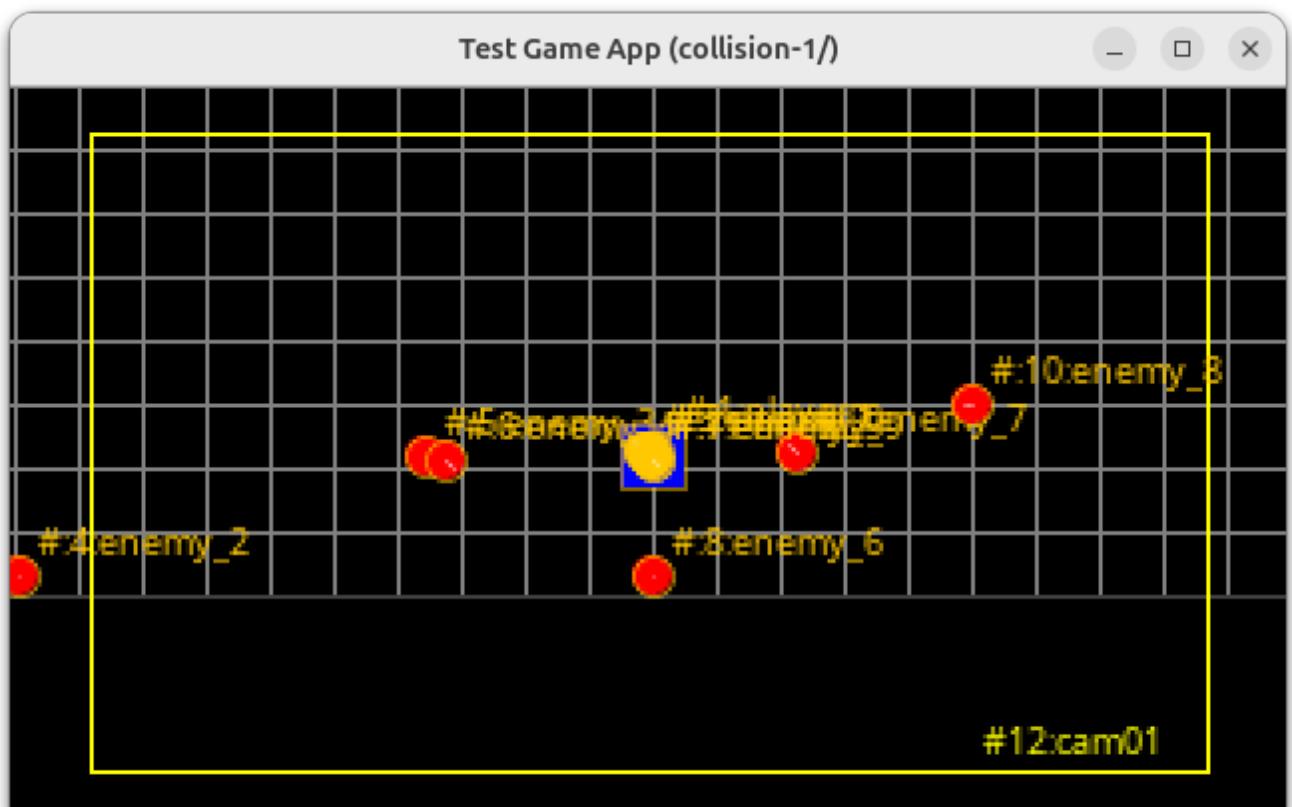


Figure 22. Lors d'une collision entre 'player' et 'ennemi', celui-ci devient orange.

Ajoutons un peu de Physique

Dans un jeu 2D, certes le gameplay est important, mais l'aspect simulation de la réalité l'est tout autant. Aussi, apporter un peu de réalité physique peut largement apporter au gameplay, mais aussi permettre quelques nouveaux effets dans celui-ci, comme influencer des objets quand ils pénètrent une zone particulière de la zone de jeux, pour leur faire subir, du vent, une contrainte magnétique, etc. Votre seule limite, alors, sera votre imagination sur comment modifier les paramètres de la simulation pour apporter de nouveau gameplay, comme on a pu l'observer dans les jeux Mario ces dernières années.

Contexte

Faisons en sorte que nos objets soient animés de façon un peu plus réaliste. Rendons plus vivante notre scène en ajoutant quelques éléments de la physique du mouvement dont Newton et sa pomme ont été les premiers contributeurs.



Figure 23. Portrait de Sir Isaac Newton, 1689 (source wikipedia)

Voici résumé les 3 lois de Newton :

1. **Première loi de Newton** : Tout objet reste au repos ou en mouvement rectiligne uniforme sauf si une force nette agit sur lui.
2. **Deuxième loi de Newton** : La force appliquée sur un objet est proportionnelle à son accélération, et inversement proportionnelle à sa masse.
3. **Troisième loi de Newton** : Pour chaque action, il y a une réaction égale et opposée.

Voyons comment convertir ces belles lois via un peu de math à du code Java !

Un peu de théorie

Dans un monde en deux dimensions, la vitesse (\vec{v}) d'un objet peut être exprimée en fonction de l'accélération (\vec{a}) et du temps (t) à l'aide de la formule suivante :

$$\vec{v} = \vec{v}_0 + \vec{a} * t$$

où :

- (\vec{V}_0) est la vitesse initiale de l'objet,
- (\vec{a}) est le vecteur d'accélération,
- (t) représente le temps écoulé.
- Cette formule indique que la vitesse à un moment donné est égale à la vitesse initiale plus la variation de vitesse causée par l'accélération sur une période. En deux dimensions, les vecteurs peuvent être décomposés en composantes (x) et (y) pour chaque variable.

Dans un monde en deux dimensions, l'accélération (\vec{a}) d'un objet peut être calculée à partir des forces appliquées en utilisant la deuxième loi de Newton, qui s'exprime par la formule suivante :

$$\vec{F}_{\text{résultante}} = m * \vec{a}$$

où :

- ($\vec{F}_{\text{résultante}}$) est la force résultante agissant sur l'objet,
- (m) est la masse de l'objet,
- (\vec{a}) est l'accélération de l'objet.

Pour calculer l'accélération, vous pouvez suivre ces étapes :

1. Déterminer les forces appliquées : Identifiez toutes les forces agissant sur l'objet. Cela peut inclure des forces telles que la gravité, la friction, la tension, etc.
2. Calculer la force résultante : Additionnez toutes les forces vectorielles. Si vous avez des forces (\vec{F}_1), (\vec{F}_2), etc., la force résultante est donnée par :

$$\vec{F}_{\text{résultante}} = \vec{F}_1 + \vec{F}_2 + \dots + \vec{F}_n$$

1. Appliquer la deuxième loi de Newton : Une fois que vous avez la force résultante, vous pouvez calculer l'accélération en réarrangeant la formule :

$$\vec{a} = \vec{F}_{\text{résultante}} / m$$

Cette formule vous donnera l'accélération (\vec{a}) de l'objet en fonction des forces appliquées et de sa masse (m).

La position résultante pour l'objet en mouvement sera alors :

$$\vec{P} = \vec{P}_0 + 0.5 * \vec{V} * t$$

où :

- \vec{P}_0 est la position précédente connue
- \vec{V} est la vitesse résultante
- t le temps écoulé depuis le précédent calcul.

Un peu de code Java

Nous allons voir, étape par étape, classe par classe, une solution possible d'implémentation d'un moteur de physique qui remplira le rôle principal d'animation et de coordination des mouvements des entités au sein d'une scène.

Nous commencerons par modéliser un objet censé bouger sur notre écran de jeu, puis, point par point, nous construirons notre moteur physique, en implémentant, fonction après fonction, l'ensemble des opérations nécessaires pour obtenir une simulation suffisamment précise pour l'emploi dans un jeu de plateforme 2D.

Une Entité

Si maintenant, nous souhaitons modéliser nos objets animés, nous devons créer un certain nombre d'attributs permettant de représenter ces vecteurs et forces, ainsi que quelques attributs permettant d'identifier facilement les objets.

Voici une première proposition :

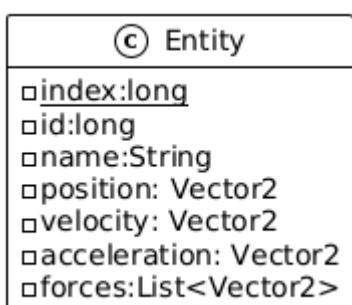


Figure 24. Le modèle UML pour la classe Entity

Ce qui se traduira par le code suivant :

```
public class Entity extends Rectangle2D {
    private static long index = 0;
    private long id = index++;
    private String name = "entity_" + id;

    private Vector2 position = new Vector2();
    private Vector2 velocity = new Vector2();
    private Vector2 acceleration = new Vector2();
    private List<Vector2> forces = new ArrayList<>();

    public Entity(String name) {
        this.name = name;
    }

    public void update(double time) {
        setRect(position.x, position.y, width, height);
        // TODO
    }
}
```

```
    }  
}
```

Cette classe Entity hérite de la classe `Rectangle2D` du JDK, ce afin de faciliter l'implémentation à venir de certains contrôles et comparaison. pour que cela fonctionne, nous utiliserons une méthode `update` qui synchronisera la position du `Rectangle2D` avec celle issue du `Vector2` position.

Notre classe devra également proposer quelques accesseurs pour définir les différentes valeurs des attributs. Nous ne les aborderons pas ici, je vous invite à aller voir le code source. Cependant, il est à noter que nous proposerons une implémentation que l'on appelle communément `Fluent Interface` permettant la création facile d'entité, passant par le principe de `Method Cascading`.

Nous avons la base de nos entités.

Afin de satisfaire la seconde loi, nous ajouterons également la masse, et bien sûr, ses accesseurs :

```
public class Entity {  
    //...  
    List<Vector2> forces = new ArrayList<>();  
    private double mass = 1.0;  
    //...  
}
```

IMPORTANT Afin d'éviter tout futur problème de calcul lié à la possible division par zéro, nous prenons la valeur 1.0 par défaut.

Nous pourrons ajouter d'autres attributs plus tard via la notion de "matériel" pour jouer sur les paramètres de friction et d'élasticité de nos entités.

Regardons d'un peu plus près maintenant l'implementation du moteur physic qui supervisera les calculs.

Le service PhysicEngine

Ce que nous savons à travers les lois de Newton, c'est que le mouvement de notre Entité sera dirigé par les forces qui lui seront appliquées et du temps écoulé.

Commençons par calculer l'accélération résultante de ces forces :

```
public class PhysicEngine {  
  
    public PhysicEngine() {  
    }  
  
    public void update(Entity e, elapsed time) {  
        // Calculons la somme des forces appliquées pour obtenir l'accélération
```

```

résultante
    e.setAcceleration(e.getAcceleration().addAll(e.getForces()).divide(e.getMass()
)));
    // La vitesse et le résultat l'effet de l'accélération en fonction du temps
    // écoulé
    e.setVelocity(e.getVelocity().add(e.getAcceleration().multiply(time)));
    // la position résultante est calculée en fonction de la vitesse et du temps
    // écoulé.
    e.setPosition(e.getPosition().add(e.getVelocity().multiply(0.5).multiply(time
)));
    // on supprime toutes les forces appliquées en attendant le prochain cycle
    // dans la boucle de jeu.
    e.getForces().clear();
}
}

```

Et pour l'appliquer à l'ensemble des entités actives de la **Scene**:

```

public class PhysicEngine {
    //...

    public void update(Scene s, elapsed time) {
        scene.getEntities().values().stream()
            .filter(Entity::isActive)
            .forEach(e -> {
                // apply Physic rules
                update(e, time);
                // update the position in inherited Rectangle2D from Entity.
                e.update(time);
            });
    }
}

```

Ce code peut être décrit sommairement via UML avec ce diagramme d'activités :

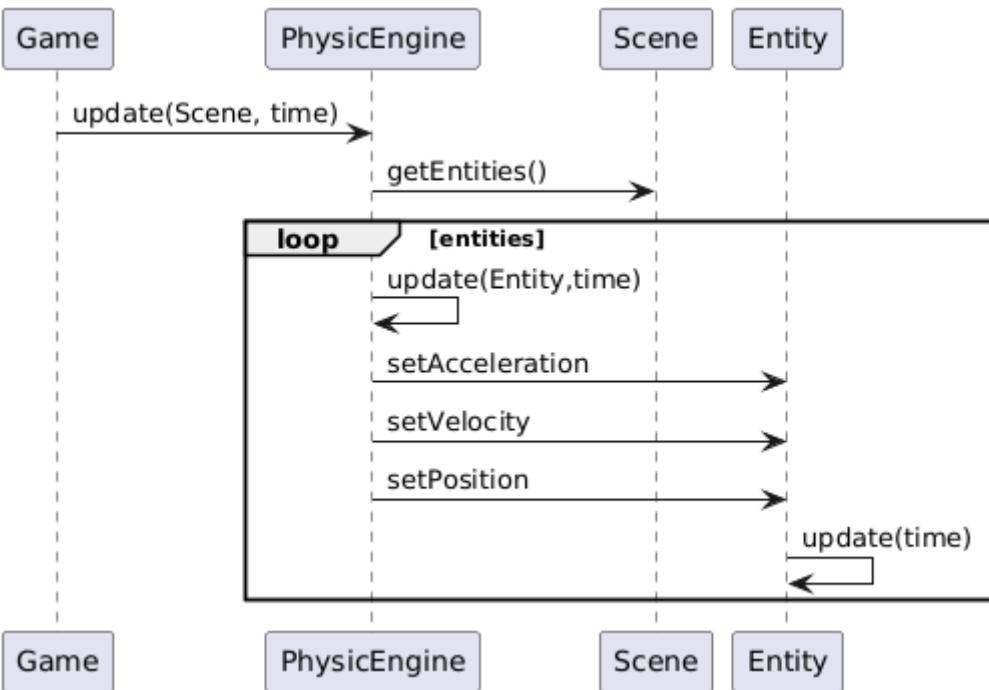


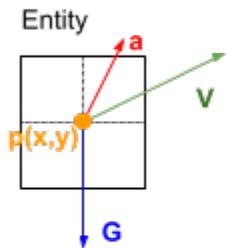
Figure 25. Calculs dans le moteur physique pour l'ensemble des entités d'une scène.

Nous avons le fondement de notre moteur de calcul. Il est temps de mettre quelques contraintes, afin de garder les entités dans un espace visible, et dans des limites de vitesse et d'accélération contrôlées.

Les limites liées au jeu

Dans l'absolu, la proposition d'implémentation pourrait suffire, mais dans la réalité, la fenêtre par laquelle nous regardons notre espace de jeu est limitée.

Ce sera notre première limite à définir : garder les entités de notre scene dans l'espace du monde de notre jeu.



World.playArea (400x200)

Figure 26. Notre Entité soumise à un ensemble de forces et limitée dans l'espace

Nous allons donc passer par un autre objet qui sera attaché à notre scene, et qui définira cette limite.

La classe World

Notre nouvel object sera défini par une class World, permettant dans un premier temps de définir la zone de jeu dans laquelle les entités de la scène évolueront.

```
import java.awt.geom.Rectangle2D;

public class World {
    private Rectangle2D playArea;

    public World() {
        playArea = new Rectangle2D.Double(0, 0, 320, 200);
    }
}
```

Par défaut, et pour à nouveau éviter des erreurs de calcul ou tout problème de valeur nulle, nous initialisons la zone de jeu fin définir une zone minimum de 320 par 200.

NOTE La taille de cette zone de jeu correspond à la taille minimum par défaut de la fenêtre d'affichage de notre jeu.

Nous pouvons donc faire évoluer notre moteur physique en lui ajoutant une méthode permettant de contenir toute entité dans la zone de jeu :

```

public class PhysicEngine {
    //...

    public void update(Scene s, elapsed time) {
        scene.getEntities().values().stream()
            .filter(Entity::isActive)
            .forEach(e -> {
                //...
                keepEntityInWorld(scene.getWorld(), e);
            });
    }

    public void keepEntityInWorld(World w, Entity e) {
        if (!world.getPlayArea().contains(e)) {
            if (!w.contains(e) || w.intersects(e)) {
                if (e.x < w.x) {
                    e.x = w.x;
                }
                if (e.x + e.width > w.width) {
                    e.x = w.width - e.width;
                }
                if (e.y < w.y) {
                    e.y = w.y;
                }
                if (e.y > w.height - e.height) {
                    e.y = w.height - e.height;
                }
            }
        }
    }
}

```

Dans ce code, nous pouvons constater que nous profitons des capacités héritées de `Rectangle2D` ici, pour une première comparaison afin de détecter si l'instance de notre `Entity` est contenue par l'objet `World`. Si ce n'est pas le cas, nous repositionnons l'instance `Entity` dans la limite de l'espace de jeu du monde.



Figure 27. Les limites du monde imposées à une instance d'Entity

Nous avons ainsi corrigé la position de notre entité, mais les vitesses sur les deux axes sont toujours actives. Il est préférable, pour des facilités de calculs, de les ramener à zéro sur l'axe où se produit la collision avec la zone de jeu :

```
public class PhysicEngine {
    //...

    public void keepEntityInWorld(World w, Entity e) {
        if (!world.getPlayArea().contains(e)) {
            if (!w.contains(e) || w.intersects(e)) {
                if (e.x < w.x) {
                    e.x = w.x;
                    e.getVelocity().setX(0.0);
                }
                if (e.x + e.width > w.width) {
                    e.x = w.width - e.width;
                    e.getVelocity().setX(0.0);
                }
                if (e.y < w.y) {
                    e.y = w.y;
                    e.getVelocity().setY(0.0);
                }
                if (e.y > w.height - e.height) {
                    e.y = w.height - e.height;
                    e.getVelocity().setY(0.0);
                }
            }
        }
    }
}
```

Voilà un moteur de physique permettant le mouvement des entités d'une scène dans un espace limité et contrôlé. Nous pouvons apporter un peu plus de réalisme en introduisant d'autres composantes dans le calcul.

l'effet Material

Afin de simuler au mieux les comportements de nos objets en mouvement, nous nous proposons d'ajouter de nouvelles notions liées à la physique du mouvement, à savoir la friction pour appliquer une résistance sur les déplacements en contact avec une surface, ainsi qu'une élasticité qui permettra de calculer le rebond lors de collision.

La classe **Material** sera notre object de définition des valeurs et une instance de celle-ci sera ajouté à la classe **Entity** en tant qu'attribut **material**

```
public class Material {  
    private String name = "default";  
    private double density = 1.0;  
    private double elasticity = 1.0;  
    private double friction = 1.0;  
  
    public Material(String name, double d, double e, double f) {  
        this.name = name;  
        this.density = d;  
        this.elasticity = e;  
        this.friction = f;  
    }  
}
```

Une petite amélioration permettra d'affecter bien plus rapidement un **Material** : la définition d'une liste de matériaux par défaut.

Name	Density	Elasticity	Friction
Default	1.0	1.0	1.0
Wood	1.1	0.3	0.7
Glass	1.3	0.5	1.0
Ice	1.1	0.4	1.0
Water	1.0	0.4	0.3
Bouncing ball	1.0	0.999	1.0

Matériaux qui seront implémentés par l'intermédiaire de variables finales dans la classe :

```
public class Material {  
    public final Material DEFAULT = new Material("default", 1.0, 1.0, 1.0);  
    public final Material BOUNCING_BALL = new Material("default", 1.1, 0.999, 1.0);  
    //...
```

```
}
```

Occupons-nous maintenant des calculs dans le moteur physique. Nous devons, afin de savoir quand appliquer la friction, si l'Entity est en contact avec autre chose. Dans notre premier exemple, le seul contact que nous pouvons détecter est celui avec le bord de la zone de jeux. Aussi, modifions Entity avec l'ajout d'un flag **contact** et ajoutons le code nécessaire.

```
public class Entity extends Rectangle2D {  
    //...  
    private boolean contact = false;  
  
    //...  
    public boolean getContact() {  
        return this.contact;  
    }  
  
    public Entity setContact(boolean c) {  
        this.contact = c;  
        return this;  
    }  
}
```

Appliquons dans un premier temps le facteur d'élasticité afin de calculer la nouvelle vitesse suite à une collision :

```
public class PhysicEngine {  
    //...  
  
    public void keepEntityInWorld(World w, Entity e) {  
        e.setContact(false);  
        if (!world.getPlayArea().contains(e)) {  
            if (!w.contains(e) || w.intersects(e)) {  
                Material m = e.getMaterial();  
                if (e.x < w.x) {  
                    e.getPosition().setX(0.0);  
                    e.getVelocity().setX(e.getVelocity().getX() * -m.getElasticity());  
                    e.setContact(true);  
                }  
                if (e.x + e.width > w.width) {  
                    e.getPosition().setX(w.width - e.width);  
                    e.getVelocity().setX(e.getVelocity().getX() * -m.getElasticity());  
                    e.setContact(true);  
                }  
                if (e.y < w.y) {  
                    e.getPosition().setY(w.y);  
                    e.getVelocity().setY(e.getVelocity().getY() * -m.getElasticity());  
                    e.setContact(true);  
                }  
            }  
        }  
    }  
}
```

```

        if (e.y > w.height - e.height) {
            e.getPosition().setY(w.height - e.height);
            e.getVelocity().setY(e.getVelocity().getY() * -m.getElasticity());
            e.setContact(true);
        }
    }
}
}
}

```

Ensuite, si le contact est persistant, appliquons le facteur de friction dans le calcul de la vitesse :

```

public class PhysicEngine {

    public PhysicEngine() {

    }

    public void update(Entity e, elapsed time) {
        // Calculons la somme des forces appliquées pour obtenir l'accélération
        résultante
        e.setAcceleration(e.getAcceleration()
            .addAll(e.getForces())
            .divide(e.getMass()));

        // La vélocité et le résultat l'effet de l'accélération en fonction du temps
        écoulé
        e.setVelocity(e.getVelocity()
            .add(e.getAcceleration()
                .multiply(time)
                .multiply(
                    e.getContact()
                        ? e.getMaterial().getFriction()
                        : 1.0);

        // la position résultante est calculée en fonction de la vitesse et du temps
        écoulé.
        e.setPosition(e.getPosition()
            .add(e.getVelocity()
                .multiply(0.5)
                .multiply(time)));

        // on supprime toutes les forces appliquées en attendant le prochain cycle
        dans la boucle de jeu.
        e.getForces().clear();
    }

}

```

Les autres facteurs issus de la classe Material seront utilisés ultérieurement dans d'autres fonctions.

Nous pouvons continuer d'améliorer notre moteur en proposant d'autres possibilités. Nous pouvons ajouter quelques éléments de simulation comme les effets que sont le vent, le courant de l'eau, le magnétisme. Nous allons donc ajouter de nouvelles capacités à notre class World pour définir des zones d'interaction dans notre zone de jeu.

Les WorldArea

La class World telle qu'elle existe ne définit qu'une chose, la taille de la zone de jeu. Nous allons lui adjoindre de nouveaux attributs pour étendre ses effets sur les entités d'une scène.

Imaginons une Scene d'automne, où le vent souffle, et l'eau de la rivière est soumise à un fort courant.

Nous allons matérialiser ces zones de vent et de courant dans la classe World à travers la définition de la nouvelle classe **WorldArea**.

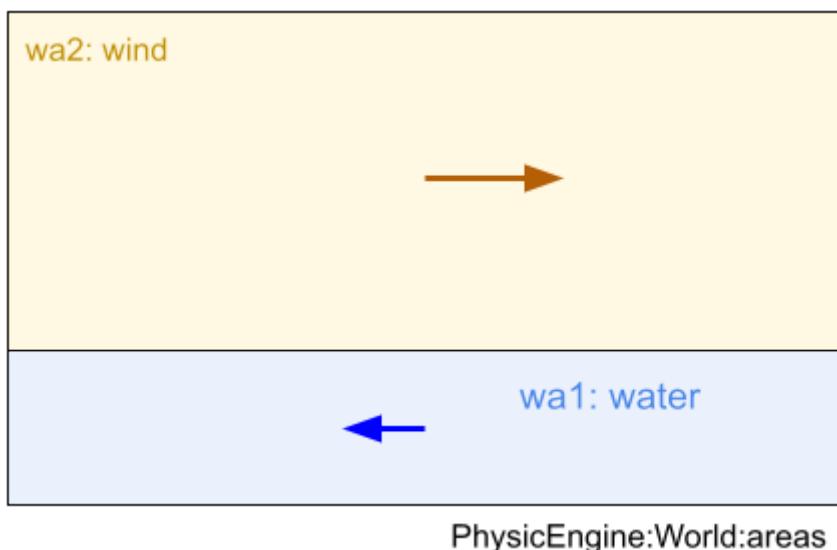


Figure 28. Définissons une zone de vent et une zone de courant.

Nous pouvons maintenant définir ce qu'est une **WorldArea**, une zone d'influence pour toute Entity qui sera contenue par celle-ci.

Cet objet partage des caractéristiques avec l'**Entity** : une position, une taille, une ou plusieurs forces qui peuvent lui être appliquées, elle peut aussi contenir un **Material** définissant des attributs physique comme la friction et la densité, il paraît judicieux de la faire hériter de la class **Entity** :

```
public class WorldArea extends Entity {  
    public WorldArea(String name) {  
        super(name);  
    }  
}
```

Si nous mettons en place une mécanique d'héritage en place, les fonctions de fluent interface offerte par `Entity` deviennent problématiques, car la création d'une `WorldArea` via les setters "fluent" retournera une `Entity` et non une `WorldArea`.

Aussi, il est nécessaire de modifier un peu notre `Entity` pour permettre de paramétriser la nature de l'objet de retour des setters :

```
① public class Entity<T> extends Node<T> {
    //...
    List<Point2D> forces = new ArrayList<>();

    //...
    public T setPosition(double x, double y) {
        this.position.setX(x);
        this.position.setY(y);
        super.setRect(x, y, width, height);
    }
    ②     return (T) this;
}

public T setPosition(Vector2 p) {
    this.position = p;
    super.setRect(p.x, p.y, width, height);
}
③     return (T) this;
}
//...
}
```

Nous pouvons voir que l'object retourné en <2> et <3> est le parametre T défini en <1>.

Notre classe `Entity` reçoit maintenant un paramètre, la classe cible, permettant une instanciation correcte de nos `WorldArea`.

```
public class WorldArea extends Entity<WorldArea> {
    public WorldArea(String name) {
        super(name);
    }
}
```

Modifions l'objet World

Nous allons définir la liste de zones d'influence dans l'objet extant. Ajoutons donc une liste à cet effet :

```
public class World {
    //...
```

```

private List<WorldArea> areas = new ArrayList<>();
//...

public World add(WorldArea wa) {
    this.areas.add(wa);
    return this;
}

public List<WorldArea> getWorldAreas() {
    return this.areas;
}
}

```

Nous pouvons maintenant facilement ajouter des zones d'influence sur notre monde lors de la création de la scène (voir chapitre précédent pour la Scene) :

```

public class SceneDemo {
    public void create() {
        World world = new World();
        world.add(
            new WorldArea("water")
                .setPosition(0, 280)
                .setSize(320, 40)
                .addForce(new Vector2(0, 0.2)));
    }
}

```

Appliquons les effets

Il est maintenant temps de procéder au calcul des effets de ces zones sur nos Entity dans le moteur physique.

Pour chaque entité de la scène, nous devons vérifier pour chaque zone si celle-ci est en collision avec l'entité. Si oui, on applique les forces de ladite zone sur l'entité AVANT de lancer les calculs physique pour l'entité.

```

public class PhysicEngine {
    //...

    public void update(Scene s, elapsed time) {
        scene.getEntities().values().stream()
            .filter(Entity::isActive)
            .forEach(e -> {
                // apply World constraints
                applyWorldConstraints(s.getWorld(), e, time);
                // apply Physic rules
                update(e, time);
            });
    }
}

```

```

        keepEntityInWorld(scene.getWorld(), e);
        // update the position in inherited Rectangle2D from Entity.
        e.update(time);
    });

}

public void applyWorldConstraints(World w, Entity e, double time) {
    w.getWorldAreas().filter(wa -> wa.contains(e) || e.intersects(wa)).forEach(wa
-> {
    e.getForces().addAll(wa.getForces());
});
}
}
}

```

Ainsi, lorsque qu'un objet **Entity** pénétrera dans une zone définie par un objet **WorldArea**, toutes les forces décrites dans celui-ci seront appliquées à l'entité contenue.

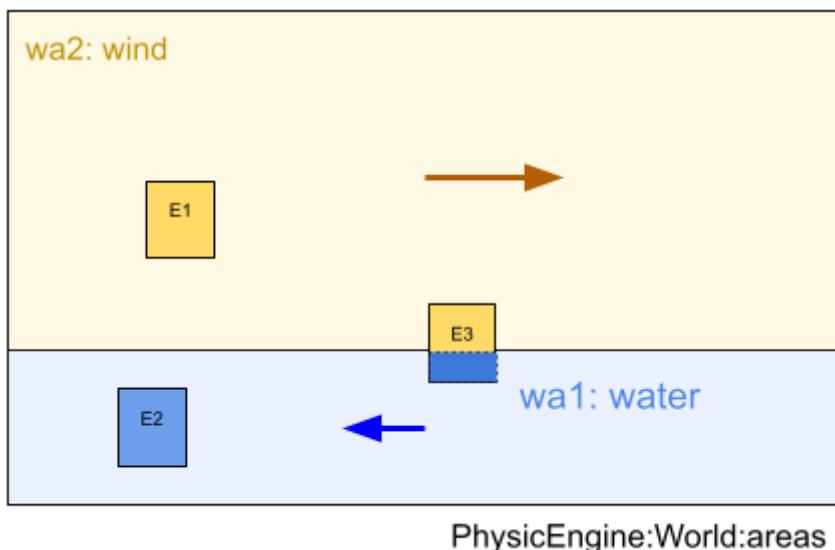


Figure 29. Effets de zone d'influence sur les Entités

Les entités sur l'image ci-dessus subissent les forces comme suit :

- l'entité **E1** est soumise au vent de la WorldArea "**wind**",
- l'entité **E2** est quant à elle soumise à l'influence de l'objet **WorldArea** "Effets de zone d'influence sur les Entités",
- alors que l'objet **E3** est lui soumis à l'influence des 2 zones que sont "**water**" et "**wind**".

Ajout de Systems et de leur manager

TODO

Thank

Thanks to all people who contributed to making this book exist.

Index

D

deuxieme loi de Newton
2eme loi de Newton, [93](#)

N

Newton
lois de Newton, [93](#)

P

physique
mouvement, [93](#)
premiereloi de Newton
1ere loi de Newton, [93](#)

T

troisieme loi de Newton
3eme loi de Newton, [93](#)