CS-107 : Mini-projet 2
Jeux avec moteur physique

J. Berdat, B. Goullet, J. Sam, B. Jobstmann

Version 1.4

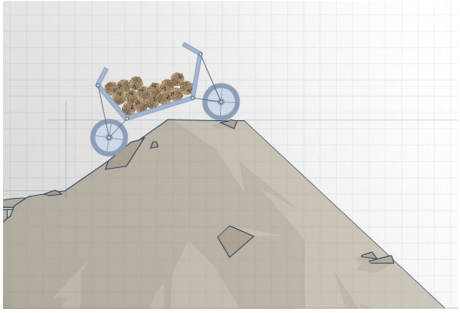
Table des matières

1 Présentation

La notion de moteur physique vous est peut être déjà familière. De nombreux jeux vidéo y ont en effet recours. Ce projet a pour objectif de vous faire programmer une structure logicielle permettant de développer des petits jeux basés sur un moteur physique (en l'occurrence il s'agira de JBox2D).

Voici ce que le resultat pourrait être :

(a) https://www.youtube.com/watch?v=073Dg2g40yY



(b) https://www.youtube.com/watch?v=8tkDMry_0X8

FIG. 1 : Exemples de jeux avec simulation physique

Au vu des temps impartis, nous nous contenterons évidemment de déclinaisons simples tel le « Bike Game » de la figure 2 où un héroïque cycliste tentera de réaliser un parcours accidenté et franchir une ligne d'arrivée sans tomber de son engin.

Une fois la structure mise en place, vous pourrez développer des petits jeux selon vos envies.

Outre son aspect ludique, ce mini-projet vous permettra de mettre en pratique de façon naturelle les concepts fondamentaux de l'orienté-objet. Vous aurez également à expérimenter l'utilisation d'une API existante, une compétence fondamentale et nécessaire à tout développeur de nos jours.

Dans un premier temps, vous vous familiariserez avec l'API relative au moteur physique au travers d'un petit tutoriel, puis vous mettrez en place une architecture orientée objet permettant de programmer des petit jeux utilisant cette API. Le but sera de vous placer au bon niveau d'abstraction et de créer des liens adaptés entre les composants.

Le projet comporte quatre étapes :

- Étape 1 (« Découverte du moteur physique ») : il s'agira de découvrir les fonctionnalités essentielles du moteur physique, basé sur JBox2D, par le biais d'un petit tutoriel.
- Étape 2 (« Noyau de base ») : au terme de cette étape vous disposerez d'une architecture de base pour créer des jeux à base de physique.
- Étape 3 (« Bike Game ») : au terme de cette étape vous aurez développé une instance simple mais concrète de jeu ; une variante du « Bike Game » de notre figure 2 basée sur le noyau de l'étape précédente.

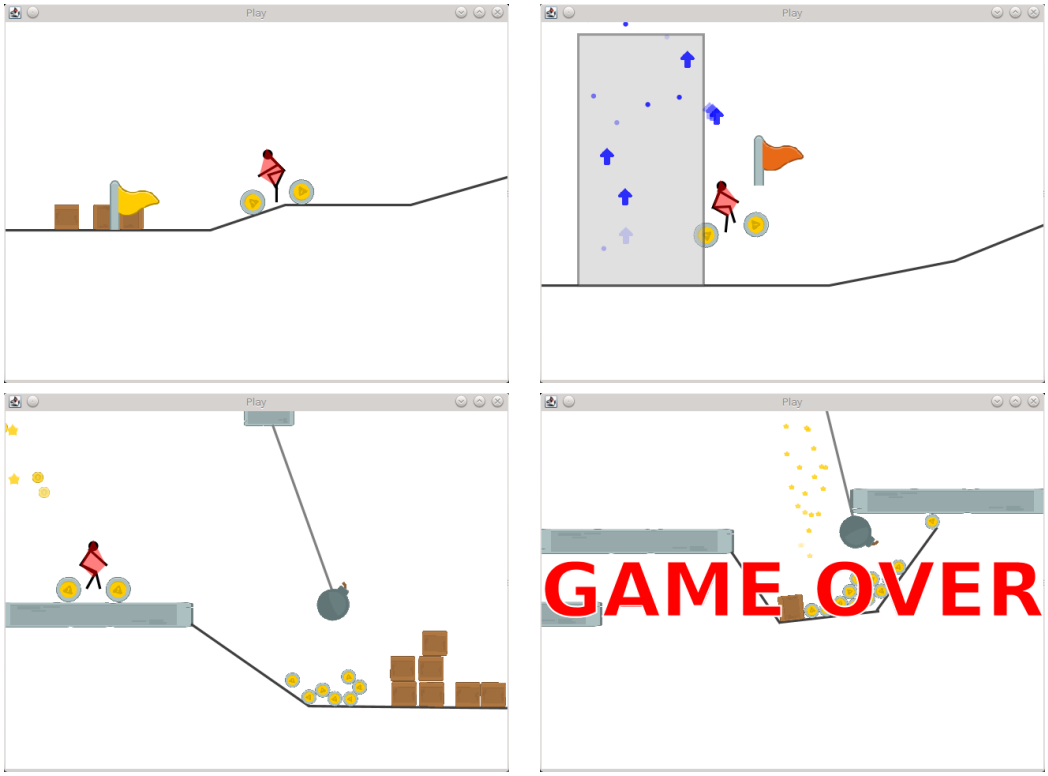


FIG. 2 : « Bike Game » élémentaire

- Étape 4 (« Extensions ») : durant cette étape, diverses extensions vous seront proposées (par exemple, des ajouts de composants et d'interactions plus complexes ou des améliorations visuelles) et vous pourrez créer un jeu de votre propre invention.

> Coder quelques extensions (à choix) fait partie des objectifs du projets.

La première étape est volontairement très guidée. Il s'agira essentiellement de prendre en main l'API du moteur physique, de bien comprendre les problématiques soulevées à chaque fois et comment il est possible d'y répondre.

La seconde devra répondre à un schéma de spécification précis (que vous aurez néanmoins la possibilité de modifier). Vous aurez d'avantage de liberté (et de responsabilités) lors des deux dernières étapes.

Points importants :

- Veillez à documenter votre code par le biais de commentaires.
- Si vous souhaitez faire des ajouts/modifications à l'API fournie, faites-en la demande en envoyant un message à `cs107@epfl.ch` (en donnant les raisons qui vous font souhaiter ces modifications et en le documentant dans votre fichier `CONCEPTION`[a]).

[a] Référez-vous à la page descriptive du mini-projet pour plus d'informations sur ce fichier ainsi que sur les recommandations pour le commentaire du code

Remarque : dans tous les exemples d'affichage donnés, le fond de la fenêtre est blanc. Il devrait être noir lorsque vous exécutez le code avec la matériel fourni (vous pouvez changer cela à la ligne 163 de `window/swing/SwingWindow.java`).

2 Présentation de JBox2D

Un moteur physique est une bibliothèque logicielle permettant de simuler de façon approximée des systèmes physiques. Ils peuvent être conçus pour simuler des « mondes » en deux ou trois dimensions. JBox2D (`www.jbox2d.org`) est un moteur physique en 2D permettant de simuler la dynamique des *corps rigides*.

Il s'agit typiquement de faire chuter des corps de façon réaliste, de permettre de leur appliquer des forces, de reproduire des effets cinétiques et de simuler des collisions entre objets.

Résumées de façon simple, les entités fondamentales mises à disposition par cette bibliothèque et qui vous seront utiles dans le cadre du projet sont les suivantes :

- « Body » : modélise un *corps physique rigide*, pas forcément matériel, par le biais de ses propriétés caractéristiques (masse, vitesse, position etc.)
- « Fixture » : modélise *une propriété matérielle* que l'on peut associer à un « Body » ; typiquement sa forme géométrique qui permettra de gérer les collisions avec d'autres objets. Seuls les « Body » comportant une forme matérielle pourront entrer en collision avec d'autres objets.
- « Shape » : modélise les formes géométrique que l'on peut associer aux « Body » par le biais de « Fixture ».
- « Joint » : modélise une *contrainte* à appliquer entre des corps. Ceci permettra de lier des corps entre eux par le biais de cordes, poulies, axes de rotation etc.
- « World » : le monde physique contenant un ensemble de « Body » dont il faut simuler le comportement.

JBox2D met principalement à disposition un simulateur permettant de faire évoluer le `World` par unité de temps `dt` grâce à la fonction principale `step`. Par exemple, et pour faire simple, si un corps est en train de chuter selon les lois de la gravité, et s'il occupe à un moment donné la position `p`, la fonction `step` permettra de calculer son état (nouvelle position, vitesse, impacts des collisions etc) après écoulement d'une unité de temps `dt`. La fonction `step` simule donc les déplacements en tenant compte des contraintes entre objets. Elle travaille à minimiser l'erreur dans l'application des contraintes. Par exemple, si un corps est attaché à une corde et qu'il chute, le corps devra rester le plus près possible de la corde.

Note : JBox2D est relativement peu documentée est difficile d'accès pour un programmeur débutant. Nous avons donc pris le parti de vous fournir une API un peu plus simple, encapsulant ses fonctionnalités de base. Nous vous fournissons aussi quelques éléments indispensables relatifs à l'interface graphique et à la gestion des touches.

Voici une vue d'ensemble du matériel fourni :

- Le répertoire `math` : contient l'API simplifiée liée au moteur physique. Il contient notamment tout ce qui permet de représenter et construire les corps rigides (`Entity` et `EntityBuilder`), les Fixtures (`Part` et `PartBuilder`), les contraintes et les classes qui en permettent la construction (comme `RopeConstraint` et `RopeConstraintBuilder` par exemple et l'abstraction `Constraint`), les formes géométriques (`Shape`) et le monde physique (`World`). Ce répertoire contient aussi tout ce qui permet de modéliser un objet doté d'une position (`Positionable`, `Attachable`, `Transform` et `Node`, voir la section 9.2 pour des compléments à ce sujet).
- Le répertoire `window` : fournit les abstractions `Window` (fenêtre), `Canvas` (zone de dessin), `Mouse` (souris), `Keyboard` (clavier) modélisant les éléments de base de liés à l'interface graphique. La classe `SwingWindow` du répertoire `swing` est une réalisation concrète de la notion de fenêtre basée sur les composants Java Swing. Il ne vous est pas demandé de consulter ce matériel dans le détail.
- Le répertoire `actor` qui contient une abstraction de la notion de jeu, tel qu'il sera perçu par le monde extérieur (interface `Game`). la répertoire `actor` contient aussi des éléments graphiques spécifiques (comme `ImageGraphics` permettant de dessiner une image). Une première ébauche d'une classe à compléter est fournie dans dans `actor.tutorial`.
- Un programme principal `Program.java` qui va lancer un jeu spécifique (ligne 34). La boucle principale (ligne 42) appelle en boucle la simulation du jeu (`update`, ligne 65) et son rendu graphique (`draw`, ligne 68) tant que la fenêtre dans laquelle s'exécute le jeu n'a pas été fermée.

Note : C'est dans le répertoire `actor` que vous allez coder l'essentiel de vos contributions relative au « Bike Game ».

Le matériel fourni est documenté dans le code et nous vous invitons à l'examiner.

Quelques compléments utiles sont également fournis dans la section 9. **Nous vous recommandons de commencer par parcourir ces compléments dans les grandes lignes, pour avoir une idée de leur contenu**.

Pour suivre le tutoriel de la section suivante, il vous suffit de savoir que :

- La classe `Entity` représente la notion de corps rigide (« Body »).
- La classe `Part` représente des « Fixtures » associées à une `Entity`.
- La hiérarchie de `Shape` modélises diverses formes géométriques.
- La classe `World` représente le monde physique simulé.

Ces classes sont définies dans le répertoire `math`.

La première étape de votre mini-projet consiste à découvrir concrètement l'utilisation de cette API au travers d'un petit tutoriel.

3 Découverte du moteur physique (étape 1)

Cette partie est délibérément très guidée. Vous allez y apprendre à créer des instances basiques de `Game` simulant des mondes physiques très simples.

3.1 Première entité physique (rôle de `Entity`)

Nous avons vu plus haut que la classe fournie `Entity` représente un corps, pas forcément matériel, évoluant dans un monde physique : c'est un « Body » appartenant à un « World » de `JBox2D`. A une telle entité, il est possible d'associer une représentation graphique, permettant de l'afficher de façon concrète dans un programme. Notre premier objectif est l'affichage d'un simple bloc fixe.

Pour cela, ouvrez la coquille de programme `HelloWorldGame` fournie dans le répertoire `tutorial/`. Cette classe implémente l'interface `Game` et il s'agit d'y compléter les méthodes `begin` est `update`. Les autres méthodes qu'il est nécessaire de redéfinir pour rendre `HelloWorldGame` instantiable sont déjà remplies pour vous car elles n'impliquent aucun traitement particulier vu la simplicité du « jeu » (qui affichera simplement un bloc).

Vous remarquerez qu'un `HelloWorldGame` contient le moteur physique en charge de sa simulation (un `World`) et le corps physique unique qui va être simulé, un objet `body` de type `Entity`.

La méthode `begin` permet l'initialisation du monde simulé.

Commencez par compléter cette méthode en lui faisant créer le moteur physique à utiliser :

```
// Create physics engine
world = new World();
```

Pour permettre la gestion de la gravité, initialisez ensuite la constante gravitationnelle qui sera utilisée par votre moteur physique :

```
// Note that you should use meters as unit
world.setGravity(new Vector(0.0f, -9.81f));
```

(A propos de mètres et autres unités de mesures, voir la section 9.1).

Il est temps maintenant de créer notre premier corps physique. Ceci se fait en ayant recours à une classe utilitaire `EntityBuilder`, permettant de créer *par étape*[1] une `Entity`.

[1] Vous apprendrez au second semestre que cette façon de construire les objets obéit à un schéma de conception courant appelé « builder pattern »

Le protocole à suivre est le suivant (et répondra toujours au même schéma pour tous les corps physiques que vous souhaiterez simuler) :

```
// To create an object, you need to use a builder
EntityBuilder entityBuilder = world.createEntityBuilder();

// Make sure this does not move
entityBuilder.setFixed(true);

// This helps you define properties, like its initial location
entityBuilder.setPosition(new Vector(1.f, 1.5f));

// Once ready, the body can be built
body = entityBuilder.build();
```

La première et la dernière ligne de ce code devront toujours être utilisées. Les lignes intermédiaires dépendent des caractéristiques dont vous souhaitez doter votre corps physique. Si vous consultez le code de `EntityBuilder`, vous verrez d'ailleurs qu'il existe d'autres propriétés que l'on peut associer au corps physique lors de sa création (vitesse, position angulaire etc.) et que par défaut un corps sera créé comme mobile (non-fixe, si l'on n'invoque pas la méthode `setFixed`).

Le corps physique étant construit, on peut maintenant lui associer une représentation graphique qui nous permettra de l'afficher concrètement. L'API fournie propose dans le répertoire `actor`, deux types d'objets graphiques qui peuvent être associés à une `Entity` : des `ImageGraphics` (images dessinables) ou des `ShapeGraphics` (formes géométriques dessinables). Nous allons utiliser ici un `ImageGraphics` mais le schéma d'utilisation est analogue pour les autres types d'objets graphiques.

A ce stade, il est naturel de considérer que les représentations graphiques utilisées par un jeu pour dessiner ses corps font partie de ses caractéristiques (attributs). Déclarez donc dans `HelloWorldGame` un attribut permettant de stocker la représentation graphique associée au corps `body` :

```
// graphical representation of the body
private ImageGraphics graphics;
```

et initialisez-le dans la méthode `begin` en l'associant au corps physique :

```
graphics = new ImageGraphics("stone.broken.4.png", 1, 1);
graphics.setParent(body);
```

`"stone.broken.4.png"` est le nom du fichier contenant l'image. Les fichiers d'images sont disponibles dans le sous-répertoire `resources/`. Les deux derniers paramètres représentent la largeur et la hauteur de l'image (jetez un oeil à la documentation de la classe `ImageGraphics`).

Notez qu'il existe des moyens plus bas niveaux de faire des dessins en les plaçant aux bons endroits (voir à ce sujet les sections 9.2 et 9.3).

Vous avez à ce stade écrit toutes les lignes de code nécessaires à l'initialisation de l'unique corps (`Entity`) à simuler et vous lui avez attribué une représentation graphique. La méthode `begin` est complète pour ce que nous voulons réaliser.

Passons maintenant à la méthode `update`. Pour rappel, cette dernière va être appelée en boucle par le programme principal `Program` ; chaque appel permettant de simuler l'évolution du monde par unité de temps `deltaTime`. La méthode `update` va donc pour l'essentiel :

- Implémenter les logiques spécifiques au jeu (par exemple, doit-on être réceptif à un événement du clavier ?) : pour notre cas, il n'y a rien à faire.
- Simuler l'évolution du monde physique sur une unité de temps `deltaTime` (en faisant appel au moteur physique) : pour notre cas le moteur physique va être invoqué mais rien ne va se passer de visible car nous n'avons qu'un corps fixe.
- Effectuer le rendu graphique au terme de cette évolution.

Ces trois étapes se traduisent ici par les lignes de code suivantes (à placer dans `update`) :

```
// Game logic comes here
// Nothing to do, yet

// Simulate physics
// Our body is fixed, though, nothing will move
world.update(deltaTime);

// We can render our scene now,
graphics.draw(window);
```

Vous pouvez maintenant lancer l'application principale `Program` ; vous observerez qu'à la ligne 34, c'est bien une instance de votre `HelloWorldGame` qui se lance. Vous devriez voir s'afficher ... une fenêtre vide. Ce n'est pas exactement ce que vous aviez en tête n'est-ce pas ?

En fait, dans la fenêtre graphique qui s'affiche, vous n'allez voir dans le cas général qu'une (petite) partie du monde simulé (voir à ce sujet la section 9.2). Il faut donc indiquer au programme comment placer la vue à l'endroit adéquat pour observer ce qui nous intéresse.

Reprenez votre méthode `HelloWorldGame.update` et juste avant l'instruction effectuant le dessin, ajoutez les lignes suivantes :

```
// we must place the camera where we want
// We will look at the origin (identity) and increase the view size
    a bit
window.setRelativeTransform(Transform.I.scaled(10.0f));
```

Exécutez à nouveau votre programme et là, les choses devraient mieux se passer. Vous devriez voir apparaître notre fameux petit bloc fixe comme sur la Figure 3.

Analysons ce qu'il se passe à l'exécution :

- `Transform.I` veut dire que l'on place la vue à l'origine (centre).

FIG. 3 : Premier affichage : un simple bloc

- `scaled(10.0f)` veut dire que nous nous plaçons à une échelle 10 (il va être possible de voir 10 objets juxtaposés de taille graphique 1x1 (ce qui est la taille choisie pour notre `ImageGraphics`)

Conseil : tentez de faire varier la position de votre image et le facteur d'échelle pour bien comprendre comment ces méthodes fonctionnent.

En faisant preuve d'un peu de curiosité, vous decouvrirez que l'API de `ImageGraphics` offre la possibilité de jouer sur le niveau de transparence de l'image ainsi que sa profondeur. En jouant sur la profondeur, vous pourrez placez une représentation graphique derrière une autre.

Par exemple, dans notre cas, vous pouvez ajouter les lignes suivantes, invoquant les setters utiles pour jouer sur la transparence et la profondeur :

```
graphics = new ImageGraphics("stone.broken.4.png", 1, 1);

// Transparency can be chosen for each drawing (0.0 - transparent,
    1.0 - opaque)
graphics.setAlpha(1.0f);

// Additionally, you can choose a depth when drawing
// Therefore, you could draw behind what you have already done
graphics.setDepth(0.0f);

graphics.setParent(body);
```

En fait, rien n'empêche d'associer à un corps physique plusieurs représentations graphiques. Faites en sorte qu'à notre attribut `body` soient désormais associés deux `ImageGraphics`.

Le premier étant celui existant et le second un autre se superposant à lui et ayant pour nom de fichier associé *"bow.png"*. Experimentez avec différentes profondeurs d'images pour obtenir l'affichage suivant (Figure 4) :

FIG. 4 : Images superposées : plusieurs objets graphiques associés à un même corps physique.

Par ailleurs, en jouant sur les profondeurs vous devriez pouvoir expérimenter des situations où le cube couvre l'arc et le rend invisible.

Fichiers à rendre : le fichier `HelloWorldGame` du répertoire `tutorial` permettant de faire les affichages décrits ci-dessus fait partie des fichiers à rendre à la fin du projet.

3.2 Première simulation (rôle de `Part`)

Notre moteur physique n'a été que peu mis à l'épreuve jusqu'à présent. Il est temps d'y remédier et de rendre le monde simulé un peu plus dynamique.

En vous inspirant de ce que vous avez fait dans `HelloWorldGame`, créez un nouveau jeu `SimpleCrateGame` dans le répertoire `tutorial`. L'objectif du jeu sera de faire tomber une caisse sur un bloc fixe. `SimpleCrateGame` aura donc deux attributs de type `Entity` : un attribut `block` (similaire au `body` de `HelloWorldGame`) et un attribut `crate` représentant la caisse.

Donnez les positions initiales `(1.0f, 0.5f)` à `block` et `(0.2f, 4.0f)` à `crate`. Concernant les représentations graphiques, vous pouvez par exemple utiliser `box.4.png` pour la caisse. Commencez par initialiser ces deux entités comme fixes. Remplacez `HelloWorldGame` par `SimpleCrateGame` dans `Program`[2] et lancez ce dernier. Vous devriez observer un

[2]Ctrl-Shift-O est votre ami dans Eclipse (voir le formulaire Eclipse sur le site du cours)

affichage similaire à celui de la Figure 5. Pour faire en sorte que `crate` obéisse à la

Fig. 5 : Caisse essayant de chuter sur un bloc mais restant suspendue en l'air.

gravité et tombe sur le bloc, modifiez ensuite `SimpleCrateGame` de sorte à ce que `crate` ne soit plus fixe. Pour ce faire, modifiez la ligne suivante :

```
entityBuilder.setFixed(false); // ICI
crate = entityBuilder.build();
```

En lançant `Program`, vous devriez alors voir `crate` tomber et traverser `block` sans faire cas de sa présence.

Que s'est il passé ?

Premièrement, la ligne `world.update(deltaTime);` de la méthode `update` a commencé à avoir un effet (faire chuter `crate`). Cette méthode appelle à son tour la méthode `step` de JBox2D, et le moteur physique peut ainsi simuler l'évolution du système physique.

Ensuite, pourquoi les lois physiques visibles sont-elles uniquement celles de la gravité et qu'aucune collision n'est détectée entre `block` et `crate` ?

Comme vu dans notre introduction sur JBox2D, un « Body » (ou `Entity` pour nous) n'a *a priori* pas de propriétés géométriques (formes, dimensions) ; or, ces propriétés sont requises pour simuler les collisions entre objets.

Rappel : Une fois un corps physique construit, il est possible de lui affecter ces propriétés, ainsi que d'autres, au moyen de ce que JBox2D appelle des « Fixtures ».

Dans l'API simplifiée que nous vous proposons, la gestion des « fixtures » se fait au moyen des classes `Part` et `PartBuilder`.

Pour attribuer un forme géométrique à `block`, vous ajouterez les lignes suivantes juste après sa création (c'est à dire, juste après l'appel à la méthode `build`).

```
// At this point, your body is in the world, but it has no geometry
// You need to use another builder to add shapes
PartBuilder partBuilder = block.createPartBuilder();
// Create a square polygon, and set the shape of the builder to
   this polygon
Polygon polygon = new Polygon(
          new Vector(0.0f, 0.0f),
          new Vector(1.0f, 0.0f),
          new Vector(1.0f, 1.0f),
          new Vector(0.0f, 1.0f)
      );
partBuilder.setShape(polygon);

// Finally, do not forget the following line.
partBuilder.build();

// Note : we do not need to keep a reference on partBuilder
```

Dans l'exemple ci-dessus, nous avons attribué un carré de taille `1x1` comme forme géométrique à notre corps physique `block`. Notez que le répertoire fourni `math` fournit des classes modélisant des formes géométriques simples (`Polygon`, `Circle`, `Polyline`).

Associez maintenant, de la même façon, la même forme géométrique (vous utiliserez la même variable `polygon`) au corps `crate`.

Si vous lancez maintenant la simulation vous devriez voir `crate` tomber sur `block` et entrer en collision. La gestion des collisions est devenue possible car nos deux corps physiques ont désormais une forme géométrique associée.

D'autres « fixtures » peuvent être attachés aux corps physiques par le même procédé (regardez l'API de `PartBuilder` du répertoire `math` pour voir lesquels sont prévues).

Par exemple, si on souhaite associer un coefficient de friction au bloc on peut ajouter l'instruction

```
partBuilder.setFriction(0.5f);
```

avant l'appel `partBuilder.build()`.

Finalement, notez qu'il est possible d'affecter plusieurs `Part` à une même `Entity`.

Important : Il faut ici bien dissocier la représentation graphique de l'objet de sa forme géométrique réelle dans le monde physique simulé : ce n'est pas parce que nos corps ont comme représentation graphique un carré 1x1 qu'ils ont cette forme au niveau physique. Bien sûr, pour la cohérence du rendu, il est souhaitable que les deux niveaux coïncident au mieux.

Pour vous convaincre de la remarque précédente, agrandissez le polygone associé à `block` et `crate` en le faisant passer à une taille 2x2 mais en gardant en 1x1 les tailles des

représentations graphiques associées. Vous devriez constatez une situation visuellement étrange comme celle de la Figure 6.

FIG. 6 : Le bloc est physiquement plus grand qu'il ne parait : la caisse reste bloquée dessus comme en suspension.

Dans votre code, faites en sorte que des variables soient utilisées de façon cohérente pour fixer la forme géométrique et les dimensions graphiques (et n'ayez donc plus recours à des valeurs littérales codées « en dur » et recopiées à différents endroits).

Fichiers à rendre : le fichier `SimpleCrateGame` du répertoire `tutorial` permettant de simuler la chute d'une caisse sur un bloc fixe fait partie des fichiers à rendre à la fin du projet.

3.3 Contraintes

Les entités vues jusqu'ici sont autonomes. Dans certaines situations, il est nécessaire d'établir des contraintes entre entités (comme les attacher entre elles). La dynamique de l'ensemble doit alors changer en conséquence. Par exemple, un bloc attaché à un autre va être entraîné par une chute de ce dernier.

Le répertoire `math` fournit plusieurs classes permettant de modéliser des contraintes :

`RopeConstraint`, `WeldConstraint`, `RevoluteConstraint` et `WheelConstraint` etc.

Cette section a pour but de vous familiariser avec l'utilisation de certaines d'entre elles, le principe général restant le même (et la documentation des classes est à votre disposition).

3.3.1 Premier exemple : les `RopeConstraint`

Ce type de contraintes permet d'attacher deux corps entre eux au moyen d'une corde.

Dans le répertoire `tutorial`, créer un nouveau jeu `RopeGame` de même nature que `SimpleCrateGame`.

Pour varier un peu l'expérience, remplacez `crate` par une `Entity` appelée `ball` de forme circulaire. La forme géométrique associée sera alors donnée par quelque chose comme :

```
Circle circle = new Circle(ballRadius);
```

(donnez la valeur `0.6f` à `ballRadius`.

Pernez comme position de départ de votre balle le `Vector (0.6f, 4.0f)` (position du centre de la balle).

L'objet graphique associé sera construit comme suit :

```
ballGraphics = new ShapeGraphics(circle, Color.BLUE, Color.RED,
    .1f, 1.f, 0);
```

(regardez l'API de la classe `ShapeGraphics` et `Circle` pour comprendre le rôle des paramètres)

Si vous lancez le programme, vous devriez voir une balle tomber sur le bloc fixe, heurter son coin puis poursuivre sa chute après avoir vu sa trajectoire déviée par le choc.

Il s'agit maintenant de lier le bloc et la balle par une contrainte qui garantit qu'une distance maximale est observée entre les deux entités. Pour cela, ajouter les lignes de code suivantes juste avant le `return true ;` de la méthode `begin` :

```
RopeConstraintBuilder ropeConstraintBuilder =
    world.createRopeConstraintBuilder();
ropeConstraintBuilder.setFirstEntity(block);
ropeConstraintBuilder.setFirstAnchor(new Vector(blockWidth/2,
    blockHeight/2));
ropeConstraintBuilder.setSecondEntity(ball);
ropeConstraintBuilder.setSecondAnchor(Vector.ZERO);
ropeConstraintBuilder.setMaxLength(6.0f);
ropeConstraintBuilder.setInternalCollision(true);
ropeConstraintBuilder.build();
```

La construction de la contrainte répond encore une fois au « builder pattern » avec lequel vous devriez commencer à être familiarisé.

Ces lignes de code lient nos deux entités `block` et `ball` par une corde invisible de taille `6.0f`. Les points d'ancrage de la corde sur chaque entité sont leur centre respectifs (`blockWidth` et `blockHeight` sont ici la largeur et auteur du bloc, valant selon nos exemple tous deux 1.0f). Le point d'ancrage est ainsi fixé au milieu du bloc. Le second point d'ancrage est le centre de la balle.

Vous remarquerez que les points d'ancrage sont donnés **en coordonnées locales** (relatives à l'objet), voir à ce propos la section 9.2.

En relançant le programme vous devriez voir la balle se balancer comme un pendule au bout de la corde après sa chute.

La méthode `setInternalCollision` permet d'indiquer si les deux entités connectées peuvent collisionner entre elles (essayez de mettre la valeur à `false` pour observer la différence de comportement). A propos de cette méthode, voir aussi la section 9.4.

Fichiers à rendre : Le fichier `RopeGame` du répertoire `tutorial` permettant de simuler la chute d'une balle liée à un bloc par une corde fait partie des fichiers à rendre à la fin du projet.

3.3.2 Deuxième exemple : les `RevoluteConstraint`

En vous inspirant de ce que vous avez fait jusqu'ici créez dans le répertoire `tutorial` un jeu `ScaleGame` où une balle tombe sur une bascule constituée d'un planche attachée à à un bloc fixe par un pivot (invisible) autour duquel elle peut basculer (comme sur la Figure 7).

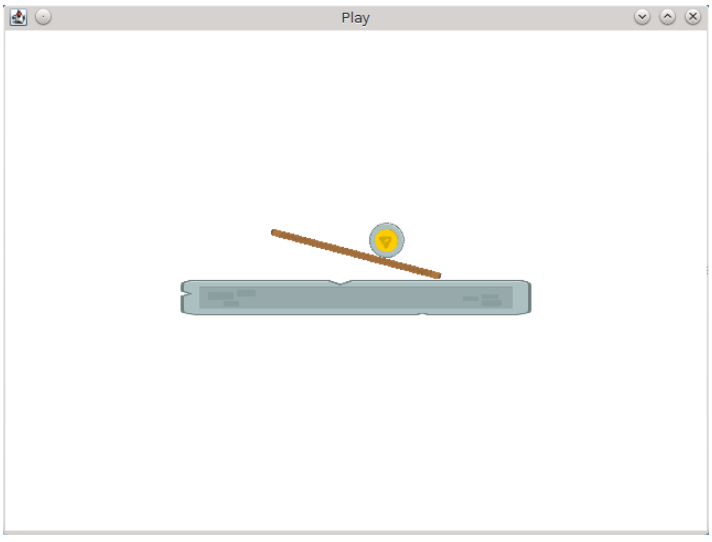


FIG. 7 : Balle chutant sur une bascule

Pour attacher la planche et le bloc fixe par le biais d'un pivot de rotation, vous utiliserez une `RevoluteConstraint` mise en place comme ceci :

```
RevoluteConstraintBuilder revoluteConstraintBuilder =
    world.createRevoluteConstraintBuilder();
```

```
revoluteConstraintBuilder.setFirstEntity(block);
revoluteConstraintBuilder.setFirstAnchor(new Vector(blockWidth/2,
    (blockHeight*7)/4));
revoluteConstraintBuilder.setSecondEntity(plank);
revoluteConstraintBuilder.setSecondAnchor(new Vector(plankWidth/2,
    plankHeight/2));
revoluteConstraintBuilder.setInternalCollision(true);
revoluteConstraintBuilder.build();
```

Cette portion de code fixe la planche (`plank`) au bloc immobile (`block`). Les points d'ancrage respectifs du lien sont au milieu de chaque entité, **toujours en coordonnées locales à l'entité**.

`setInternalCollision` est à `true` car on veut que les collisions soient détectées entre le bloc et la planche ; ce, afin que cette dernière ne traverse pas le bloc sur lequel elle repose.

Créez le bloc en `(-5.0f, -1.0f)` et attribuez lui les dimensions `10x1` et la représentation graphique du fichier *`"stone.broken.4.png"`*. Créez la planche en `(-2.5f, 0.8f)` et attribuez lui les dimensions `5x0.2` et la représentation graphique du fichier *`"wood.3.png"`*. Enfin, créez la balle en `(0.5f, 4.f)` et attribuez lui le rayon `0.5f` et la représentation graphique du fichier *`"explosive.11.png"`*.

> Attention, pour « coller » une image sur un objet circulaire, il faut prendre une précaution supplémentaire qui n'avait pas été nécessaire pour le bloc (voir la section 9.3).

Si vous lancez ce jeu, vous devriez observer la balle chutant sur la planche et la faisant basculer vers la droite avant de poursuivre sa chute dans le vide.

3.4 Contrôles

Dans la méthode `update`, nous avons jusqu'ici pu voir en action la simulation opérée par le moteur physique (appel à `world.update`) ainsi que le rendu graphique des corps impliqués dans la simulation. Le troisième volet, à savoir la logique du jeu, est resté jusqu'ici inactif.

Nous allons nous intéresser à la réaction à des événements extérieurs, comme l'intervention d'un joueur avec la souris ou le clavier.

Reprenons notre exemple du `ScaleGame`. Supposons que nous souhaitions y ajouter le fait qu'appuyer sur les flèches gauche et droite permette de diriger la balle pour la garder en équilibre sur la planche.

Ceci peut se faire tout simplement en ajoutant les lignes suivantes avant l'appel à `world.update()` dans la méthode `update` :

```
if (window.getKeyboard().get(KeyEvent.VK_LEFT).isDown()) {
  ball.applyAngularForce(10.0f);
```

```
} else if (window.getKeyboard().get(KeyEvent.VK_RIGHT).isDown()) {
  ball.applyAngularForce(-10.0f);
}
```

`KeyEvent.VK_LEFT` et `KeyEvent.VK_RIGHT` représentent respectivement les flèches gauche et droite du clavier \footnote{Les différents noms associés aux touches sont disponibles ici \url{https ://docs.oracle.com/javase/8/docs/api/java/awt08/event/Key
Les fonctionnalités qu'il est possible d'invoquer sur une touche (comme \lstinlineisDown ! ici qui teste si la touche est enfoncée) sont données par l'API de la classe `Button` du répertoire `window`.

La réaction à cette touche est ici l'application d'une force angulaire[3] d'amplitude `10.0` à `ball`.

Examinez les fonctionnalités dont le nom commence par `apply` dans la classe `Entity`, pour voir ce que l'API du moteur physique vous permet d'appliquer comme forces aux entités.

Comme événement extérieur, il est bien sûr aussi possible d'intercepter ceux émanant de la souris. Par exemple, pour tester si le bouton gauche de la souris a été appuyé, on écrirait :

```
if (window.getMouse().getLeftButton().isPressed() {...}
```

L'API de l'interface `Mouse` du répertoire `window` est à consulter pour voir comment se formulent les autres tests possibles.

Fichiers à rendre : Le fichier `ScaleGame` du répertoire `tutorial` permettant de simuler la bascule et de contrôler le déplacement de la balle fait partie des fichiers à rendre à la fin du projet.

3.5 Contacts

Certaines logiques de jeux peuvent exiger que des corps réagissent de façon particulière lorsqu'ils entrent en collision avec d'autres corps : un personnage peut être assommé et devenir hors-jeu s'il reçoit un objet sur la tête, par exemple. Pour qu'un corps puisse réagir en cas de collision, il doit pouvoir être à « l'écoute » d'éventuelles collisions. Ceci se fait concrètement en le mettant en relation avec un « listener » à l'écoute de contact (`ContactListener`).

[3] https://fr.wikipedia.org/wiki/Force_centrip%C3%A8te

Pour vous familiariser avec ces concepts, codez un jeu `ContactGame` dans le répertoire `tutorial` constitué d'une balle bleue tombant sur un bloc fixe.

Donnez à la balle (appelons la `ball` dans le code) le rayon `0.5f` et la position `(0.0f, 2.0f)`, par exemple. L'objet graphique associé peut être construit comme suit :

```
ballGraphics = new ShapeGraphics(circle,Color.BLUE, Color.BLUE, .1f,
    1, 0);
```

Donnez au bloc la position `(-5.0f, -1.0f)` et les dimensions `10x1`, ainsi que la représentation graphique de votre choix.

Nous souhaitons maintenant que la balle soit *sensible au contact* et devienne rouge dès qu'elle a heurté le bloc.

Pour ceci, il faut :

- Déclarer dans `ContactGame` un écouteur de contact :

  ```
  private BasicContactListener contactListener;
  ```

- Ensuite, dans la partie d'initialisation du jeu (méthode `begin`), il faut mettre `ball` en relation avec cet écouteur :

  ```
  contactListener = new BasicContactListener();
  ball.addContactListener(contactListener);
  ```

  Enfin, dans la partie `update`, au moment du dessin, il faut tester si l'écouteur a détecté des contacts et réagir en conséquence, ce qui peut se faire par exemple comme suit (ce n'est pas la seule façon de faire, nous en reparlerons) :

  ```
  // contactListener is associated to ball
  // contactListener.getEntities() returns the list of entities in
     collision with ball
  int numberOfCollisions = contactListener.getEntities().size();

  if (numberOfCollisions > 0){
     ballGraphics.setFillColor(Color.RED);
   }

  ballGraphics.draw(window);
  ```

  On suppose ici que `ballGraphics` est le `CircleGraphics` utilisé pour dessiner `ball`.

Une fois ceci implémenté, vous devriez voir une balle bleue tomber sur le bloc fixe et devenir rouge dès qu'elle le heurte.

Pour comprendre plus en details ce qu'il se passe, explorons l'API proposée et définie dans le répertoire `math` :

- la classe `Contact` modélise un contact entre deux corps. Basée sur des fonctionnalités de JBox2D, elle donne accès aux `Part` respectives de chaque corps effectivement en contact.
- l'interface `ContactListener` défini ce que doit fournir un objet à l'écoute de contacts. Toute classe implémentant cette interface doit concrètement fournir la définition des méthodes `beginContact` et `endContact`. La première met en oeuvre ce qui doit être fait lorsqu'un contact donné est détecté et la seconde lorsque le contact en question n'a plus lieu.
- la classe `BasicContactListener` est une implémentation simple d'objets à l'écoute de contacts. Cette classe permet de récupérer les entités qui sont en collision au moyen de la méthode `getEntities`.

Pour qu'une entité soit à l'écoute de collisions, il faut donc créer un écouteur de collision et ajouter l'écouteur en question à sa liste d'écouteurs (elle peut en avoir plusieurs).

Nous aurons l'occasion d'y revenir dans la suite du projet.

Fichiers à rendre : Le fichier `ContactGame` complété conformément à la description ci-dessus fait partie des fichiers à rendre à la fin du projet.

4 Mise en place de l'architecture (étape 2)

Lors de l'étape précédente, nous nous sommes essentiellement préoccupés de comprendre les outils offerts par le moteur physique. Vous avez sans doute remarqué que certaines parties du code étaient un peu lourdes et répétitives.

Par ailleurs, la modélisation du monde à simuler s'y est faite de façon très basique ne permettant de mettre en oeuvre que les comportements physiques des objets impliqués.

Votre première tâche lors de cette étape va être d'affiner le modèle orienté-objet de sorte à se placer à un niveau d'abstraction plus élevé. Sur la base de cette architecture, il vous sera ensuite demandé de coder un « Bike Game » très simple répondant à une spécification donnée.

Nous vous suggérons d'utiliser une conception basée sur trois classes `Actor`, `ActorGame` et `GameEntity`, décrites ci-dessous. Vous devrez coder ces classes dans le répertoire `actor`. Vous êtes libres de rediscuter cette conception et d'en proposer une autre, à condition qu'elle soit raisonable **et** justifiée et que vous documentiez vos choix dans votre fichier `CONCEPTION`.

Nous essayerons d'adhérer au mieux au principe suivant :

> *"Minimize the accessibility of classes and members.*
> *The rule of thumb is simple : make each class or member as inaccessible as possible. In other words, use the lowest possible access level consistent with the proper functioning of the software that you are writing."* [5]

En d'autres termes, il faut privilégier les méthodes et attributs *privés*, de sorte de minimiser la quantité d'attributs et méthodes *publics*. Le droit d'accès *protégé* sera utilisé à bon escient pour certaines méthodes mais pas pour les attributs.

Note : il est difficile de faire en sorte que votre programme soit dénué de toute faille d'encapsulation. Ceci impliquerait trop de travail pour les temps impartis et/ou l'utilisation de concepts ou approches non encore abordés dans le cadre de ce cours. Nous pointerons dans la suite de l'énoncé les situations où des failles seront tolérées (et les éventuels problèmes que cela peut poser).

4.1 Actor

Tous les jeux que nous avons mis en oeuvre ont recours à un moteur physique (`World`), et tous les corps (`Entity`) qui y évoluent sont systématiquement initialisés par des séquences d'instructions telles que :

```
EntityBuilder entityBuilder = world.createEntityBuilder();
entityBuilder.setFixed(true);
```

```
entityBuilder.setPosition(new Vector(1.0f, 0.5f));
block = entityBuilder.build();
```

qui sont répétées dans chaque programme et pour chaque corps. Ces répétitions (attribut `World` dans chaque classe, recours récurrent au schéma d'initialisation via `EntityBuilder`) suggèrent que ce code peut être modularisé/abstrait dans une fonction.

En programmation, si vous devez copier-coller des lignes de code, c'est souvent signe qu'il vaut mieux mettre les mettre dans une fonction (ou une classe) et réutiliser cette fonction (ou classe) aux endroits nécessaires. Cela permet notamment de produire du code plus facile à maintenir en assurant la cohérence des modifications.

Plus fondamentalement, `Entity` matérialise un corps tel que perçu par la moteur physique. Or, les jeux que nous pouvons imaginer vont mettre en scène des acteurs qui auront des caractéristiques pouvant aller bien au delà de leur représentation dans le monde physique. Par exemple un coffre rempli d'accessoires peut se comporter comme un simple bloc au niveau physique (lorsqu'il chute ou glisse), mais avoir toute sorte d'autres propriétés intéressantes dans la logique du jeu (comme son contenu ou une clé associée).

Pour dissocier les aspects physiques et logiques des intervenants d'un jeu, il est ici naturel d'introduire les deux concepts suivants :

- `Actor` permettant de modéliser un acteur intervenant dans un jeu conformément à une certaine logique de jeu.
- `ActorGame` permettant de modéliser un jeu faisant évoluer des `Actor` dans un monde physique.

Le concept d'`Actor` représente un acteur du jeu au sens logique ; par exemple un vélo qui peut être sensible à la présence d'une ligne d'arrivée. Pour détacher ce concept de toute implémentation concrète, le liant par exemple à un corps physique comme une `Entity`, il vous est suggéré de le coder sous la forme d'une *interface*.

Tout ce que l'on peut dire à ce stade d'un acteur est qu'il va évoluer au cours du temps, et que sa disparition implique potentiellement de prendre certaines mesures (comme faire disparaître des corps physiques qui lui seraient associés). Le contenu de l'interface se bornera donc à quelque chose ressemblant à ceci :

```
/**
 * Simulates a single time step.
 * @param deltaTime elapsed time since last update, in
    seconds, non-negative
 */
public default void update(float deltaTime) {
    // By default, actors have nothing to update
}
public default void destroy(){
    // By default, actors have nothing to destroy
```

```
}
```

Il est raisonnable de concevoir un `Actor` comme un objet doté d'une position dans l'espace et représentable graphiquement. L'interface `Actor` étendra donc les interface fournies `Positionable` et `Graphics` (consultez les sections 9.2 et 9.3)

4.2 ActorGame

Le concept `ActorGame` sera par contre plutôt matérialisé par une classe abstraite, implémentant l'interface fournie `Game`.

Cette dernière, fournie dans le répertoire `actor`, représente tout jeu au sens logique, tel qu'il sera perçu par la monde extérieur. Si vous jetez un oeil au programme principal fourni (`Program.java` à la racine du projet), vous verrez que les seules fonctionnalités qui lui sont utiles pour faire tourner le jeu sont celles spécifiées par l'interface `Game`. Cette dernière ne nous donne qu'une vue logique du jeu, et ne se préoccupe pas d'implémentations concrètes à base d'un moteur physique par exemple.

> Vous noterez que les interfaces sont à cet égard un puissant outil d'encapsulation : `ActorGame` et les acteurs auront besoin de se connaître mutuellement, ce qui implique de leur part d'ouvrir l'accès à certaines informations (failles d'encapsulations potentielles). Cependant, si en tant qu'utilisateur, on s'astreint à la discipline de ne voir d'un jeu que sa logique d'utilisation édictée par `Game` (comme c'est le cas de `Program` par exemple), alors les accès sensibles ne sont plus exposés.

`ActorGame` est précisément une classe de base permettant d'implémenter des réalisations concrètes de jeux à base de moteur physique. Un `ActorGame` sera caractérisé par :

- La liste d'`Actor` qu'il fait intervenir.
- Le monde physique dans lequel ils évoluent (un `World`).
- Des données externes permettant de l'initialiser en tant que `Game` (voir la ligne 35 de `Program`) : c'est à dire la fenêtre dans laquelle il va s'afficher (`Window`) et le système de fichiers qui lui servira à accéder à des ressources, comme des images (`FileSystem`).
- La partie visible du monde simulé.

La dernière partie d'un `ActorGame` concerne la caméra et vous est donnée :

```
// Viewport properties
private Vector viewCenter;
private Vector viewTarget;
private Positionable viewCandidate;
private static final float VIEW_TARGET_VELOCITY_COMPENSATION =
    0.2f;
```

```
private static final float VIEW_INTERPOLATION_RATIO_PER_SECOND =
    0.1f;
private static final float VIEW_SCALE = 10.0f;
```

Nous aurons pour objectif de centrer la caméra sur un `Positionable` donné (typiquement le vélo dans notre « Bike Game »). C'est le `viewCandidate` dans les attributs ci-dessus. Si l'on re-centre la caméra directement sur `viewCandidate` à chaque `update`, le rendu peut être saccadé. Pour cette raison, on utilisera un algorithme qui permettra, par interpolation, d'obtenir une transition plus fluide vers le `viewCandidate`. Les variables `viewCenter` et `viewTarget` ainsi que les constantes ci-dessus sont utilisées par cet algorithme, qui vous sera fournit un peu plus bas. `VIEW_SCALE` est le facteur d'échelle, tel que déjà utilisé dans la partie tutoriel.

La classe `ActorGame` doit évidemment fournir dans son API des méthodes permettant d'ajouter ou de supprimer un acteur de la liste des acteurs. Elle autorisera également l'accès au clavier et à la fenêtre d'affichage utilisée (ses acteurs en auront besoin) mais par le biais de getter ayant cette allure :

```
public Keyboard getKeyboard(){
    return window.getKeyboard();
}

public Canvas getCanvas(){
    return window;
}
```

Question 1

On aurait pu à la place ne définir que le getter

```
public Window getWindow(){
    return window;
}
```

En quoi le choix suggéré est-il meilleur ?

La classe `ActorGame` doit aussi fournir une version de base des méthodes exigées par l'interface `Game`, à savoir :

- `begin` : qui se chargera de l'initialisation de la fenêtre d'affichage, du système de fichiers, du moteur physique et de la gravité associée (comme vous l'avez fait dans les tutoriels) et qui positionnera la vue (`ViewCenter` et `ViewTarget`) à `Vector.ZERO`. Pour être conforme à l'interface d'utilisation prévue par `Game`, nous ne prévoyons aucun constructeur. C'est la méthode `begin` qui se charge de gérer tout ce qui doit l'être au démarrage d'un `Game`, y compris les initialisations d'objets.
- `end` qui ne fait rien de particulier à ce stade.
- et `update` qui se codera de façon similaire à ce que vous avez fait dans les parties tutoriel. Elle comportera donc les étapes :
  1. Simulation du monde physique.

2. Pour chaque acteur le faire évoluer sur une unité de temps `deltaTime` selon sa méthode `update`.
3. Calculer la position de la caméra après cette évolution.
4. Dessiner tous les acteurs ; prenez note du fait que un `Actor` est un `Graphics` est doit donc fournir une méthode `draw` qui dessinera dans la fenêtre du jeu.

Le code vous est fourni pour le positionnement de la caméra :

```
// Update expected viewport center
if (viewCandidate != null) {
    viewTarget =
        viewCandidate.getPosition().add(viewCandidate.getVelocity()
                                        .mul(VIEW_TARGET_VELOCITY_COMPENSATION));
}

// Interpolate with previous location
float ratio = (float)Math.pow(VIEW_INTERPOLATION_RATIO_PER_SECOND,
    deltaTime);
viewCenter = viewCenter.mixed(viewTarget, ratio);

// Compute new viewport
Transform viewTransform =
    Transform.I.scaled(VIEW_SCALE).translated(viewCenter);
window.setRelativeTransform(viewTransform);
```

où `window` est l'attribut de type `Window`.

Nous vous suggérons d'ajouter la possibilité de modifier l'attribut `viewCandidate` au moyen d'un « setter » `setViewCandidate`

4.3 `GameEntity`

Pour que la simulation physique de `ActorGame.update` fasse quelques chose, il faut naturellement que les acteurs aient une représentation physique. Concrètement, à chaque acteur seront associés une ou plusieurs `Entity`. Il s'agira cependant d'entités un peu particulières puisqu'elles ont la particularité d'être simulées dans un jeu et non pas uniquement au niveau physique (ce qui peut avoir une incidence sur leur comportement aussi).

Le concept de `GameEntity` est précisément dédié à faire le pont entre les aspects physiques et la logique du jeu. Il s'agira d'une classe abstraite représentant une `Entity` évoluant dans un `ActorGame` (elle aura donc un attribut de type `Entity` et un attribut de type `ActorGame`). Cette classe sera dotée pour le moment :

- D'un constructeur `GameEntity(ActorGame game, boolean fixed, Vector position)`.
- D'une sucharge de ce constructeur `GameEntity(ActorGame game, boolean fixed)`
- D'une méthode `destroy` qui dans sa version de base, permet de détruire l'entité physique associé (méthode `destroy` de `Entity`).

FIG. 8 : les 3 caisses tombent vers le bas et disparaîssent

Ces constructeurs initialiseront le jeu associé à l'entité. Ils initialiseront aussi l'entité elle-même en la construisant dans le monde physique du jeu auquel elle appartient. Cette construction se fera selon le schéma vu dans les tutoriels (« builder pattern »).

Dans l'entête des constructeurs, `position` est la position de l'entité et le booléen indique si l'entité est fixe ou pas.

Question 2

Comment créer l'entité dans le monde physique du jeu sans fournir l'accesseur trop intrusif que serait `ActorGame.getWorld()` ? (et d'ailleurs, pourquoi cet accesseur est-il intrusif ?)

Question 3

Est-il nécessaire de faire une copie défensive du `ActorGame` passé en paramètre ? (Ceux qui ont pris de l'avance et qui n'ont pas attendu la semaine 11 pour aller au delà de la partie tutoriel pourront revenir à cette question plus tard)

Tous les attributs de `GameEntity` devront être `private`.

Vous êtes autorisé, en guise de simplification, à définir des getter `getEntity` (accès à la représentation physique d'une `GameEntity`) et `getOwner` (accès au jeu à laquelle appartient la `GameEntity`). Il est cependant vivement recommandé de les définir en `protected` et non en `public`. Ceci signifie par exemple que toutes les `GameEntity` et toutes les classes du paquetage `actor` ont le droit d'accéder à la couche physique des autres `GameEntity`, mais pas le reste du monde.

Question 4

Quels sont les avantages et inconvénients de `protected` ici ?

4.4 Gestion des erreurs

Les constructeurs d'objets, principalement ceux des `GameEntity` et de ses sous-classes à venir devront lancer des exceptions en cas de paramètres invalides. Typiquement :

- des `NullPointerException` en cas de paramètres indispensables valant `null` (par exemple le `ActorGame` associé à une `GameEntity` ou encore sa position) ;
- des `IllegalArgumentException` en cas de paramètres invalides (par exemple une largeur négative ou null pour une caisse)

Si la gestion des exceptions ne vous dit rien, attendez le cours 12 et revenez à vos constructeurs pour les finaliser à ce moment là.

4.5 Test de l'architecture

Les classes et interfaces `Actor`, `ActorGame` et `GameEntity` vous ont été suggérées ici avec un contenu minimal. Elles seront amenées à évoluer par la suite (si vous gardez ce modèle d'architecture).

Pour comprendre comment elles vont être utilisées concrètement, il vous est demandé de coder dans le répertoire `actor/crate`[4] :

- Un premier acteur `Crate` héritant de `GameEntity` (et implémentant `Actor`). Cette classe représente une caisse en tant qu'acteur logique d'un jeu (le `CrateGame` ci-dessous). Vous vous inspirerez de ce que vous avez fait dans la partie tutoriel pour la représentation physique de la caisse. Les dimensions de la caisse, et l'image utilisée pour la dessiner pourront typiquement être données à la construction.
- Un jeu `CrateGame` héritant de `ActorGame` mettant en scène 3 caisses (mais qui ne fait rien d'autre que les créer au démarrage). Vous pouvez utiliser les positions `(0.0f, 5.0f)`, `(0.2f, 7.0f)` et `(2.0f, 6.0f)` pour les caisses.

Le lancement de votre `CrateGame` devrait vous permettre de voir trois caisses tombant vers le bas, comme suggéré par la Figure 8.

Fichiers à rendre : Le fichier `CrateGame` du répertoire `actor/crate` codant le jeu suggéré ci-dessus fait partie des fichiers à rendre à la fin du projet.

[4] Pour créer ce répertoire dans Eclipse : Clic droit sur le répertoire `actor`, puis `new > package` et ajouter `crate` à la suite : `ch.epfl.cs107.play.game.actor.crate`

Important : l'acteur `Crate` peut être amené à être détruit (imaginez que vous introduisiez un laser qui fait voler la caisse en éclat). Dans ce cas, il faut qu'il disparaisse de la simulation et que l'entité physique qui lui est associée soit détruite (faute de quoi vous pourriez avoir une caisse fantôme qui continue à être simulée par le moteur physique même si elle n'est plus visible). La méthode `destroy` de `Actor` a donc un rôle à jouer ici. **Ce même raisonnement doit en fait s'appliquer à tout acteur que vous serez amené à coder.**

5 « Bike Game » (étape 3)

Nous allons maintenant essayer de faire un peu mieux que de faire tomber des caisses dans le vide. Notez que cette partie du projet est délibérément beaucoup moins guidée.

Il vous est demandé de programmer un `BikeGame` élémentaire constitué :

- D'un vélo monté par un cycliste (nous faisons pour commencer le choix que les deux sont indissociables, et qu'il ne fait pas sens de donner une existence au cycliste en dehors de son véhicule).
- D'un terrain sur lequel roule le vélo et qui sera jonché de caisses qui seront autant d'obstacles à surmonter.
- D'une ligne d'arrivée.

Le vélo a une position de départ dans le jeu. Des touches du clavier permettent de le contrôler pour l'orienter (il peut rouler vers la gauche ou la droite du terrain), l'arrêter ou lui appliquer des forces qui lui permettront par exemple de monter sur un obstacle.

Si le vélo bascule et que le cycliste touche le sol, la partie est perdue. Si par contre le vélo franchit la ligne d'arrivée sans chuter la partie est gagnée. Ce qui se traduira par l'affichage d'un message approprié dans chaque cas (l'affichage des messages est décrit dans la section 5.3).

Vous commencerez par créer un `BikeGame`, inspiré du `CrateGame` précédent.

Un `BikeGame` va naturellement contenir dans sa liste d'acteurs, des acteurs tels que « vélo », « ligne d'arrivée » ou « terrain » (ainsi que tout autre `Actor` qui va y évoluer, comme les caisses servant d'obstacles). Quelques indications vous sont fournies dans ce qui suit concernant la modélisation de ces acteurs.

Vous considérerez que la position de départ du vélo est une caractéristique du `BikeGame`, initialisable au moyen d'un setter. Initialisez la par exemple à `(4.0f, 5.0f)`. La caméra sera positionnée sur cette position de départ (rappelez-vous de `ActorGame.setViewCandidate`).

Certains `Actor` ne seront pas spécifiques à un jeu de type `BikeGame` (terrain, caisse, et même roue par exemple). Il est suggéré que vous les codiez dans un sous-répertoire `actor/general` . D'autres `Actor`, par contre, auront parfois besoin de savoir qu'ils évoluent spécifiquement dans un `BikeGame` et non dans un `ActorGame` quelconque (par exemple le vélo ou encore la ligne d'arrivée pour réagir spécifiquement au passage de ce dernier). Nous vous suggérons de coder ce qui est spécifique au `BikeGame` dans le sous-répertoire `actor/bike`.

Important : dans JBOx2D, la nature des formes géométriques a un incidence sur le comportement physique. Par exemple, plus la surface d'une forme est grande plus l'objet correspondant sera lourd. Vous êtes libres de jouer avec les grandeurs physiques et les formes, mais il faudra bien être conscient de l'impact que cela peut avoir (cela peut avoir des effets déroutants sur la simulation). Nous vous suggérons ci-dessous des valeurs précises pour les grandeurs physiques. Il peut être raisonnable de commencer par utiliser ces valeurs jusqu'à aboutir à quelque choses de jouable puis de rediscuter ces choix par la suite si vous le voulez.

5.1 Le terrain

Il vous est suggéré de modéliser un `Terrain` comme une `GameEntity` jouant le rôle d'`Actor` dans un `BikeGame`. Il prendra la forme d'une `Polyline` (définie dans `math`). Cette forme devrait pouvoir être paramétrable à la construction du terrain.

Voici un exemple de comment pourrait être construite cette forme :

```
new Polyline(
            -1000.0f, -1000.0f,
            -1000.0f, 0.0f,
            0.0f, 0.0f,
            3.0f, 1.0f,
            8.0f, 1.0f,
            15.0f, 3.0f,
            16.0f, 3.0f,
            25.0f, 0.0f,
            35.0f, -5.0f,
            50.0f, -5.0f,
            55.0f, -4.0f,
            65.0f, 0.0f,
            6500.0f, -1000.0f
        )
```

Créer la classe `Terrain` et utilisez là pour ajouter un terrain au `BikeGame`. Posez ensuite trois caisses sur le terrain du `BikeGame` (par exemple aux mêmes positions que dans `CrateGame` si vous avez choisi `(4.0f, 5.0f)` comme position de départ du vélo.

5.2 Le vélo

Le vélo monté du cycliste (classe `Bike`) est un `Actor` de `BikeGame`. Il peut être modélisé comme une `GameEntity`.

Un modèle possible (il y en a d'autres) consiste à considérer que la `GameEntity` est l'abstraction du couple vélo-cycliste qui nous servira à tester s'il y a chute ou non sur le terrain (« hitbox ») :

FIG. 9 : L'abstraction vélo-cycliste prendra la forme d'une « hitbox » approximative (la boîte orange ici).

à cette `GameEntity` seront associées des roues motrices (attributs du `Bike`).

En guise de forme associée à cette `GameEntity` vous pourrez prendre par exemple :

```
Polygon polygon = new Polygon(
            0.0f, 0.5f,
            0.5f, 1.0f,
            0.0f, 2.0f,
            -0.5f, 1.0f
        );
```

L'entité physique associée à `Bike` et qui a donc la forme physique de la « hitbox », sera considérée comme « fantôme » (voir à ce propos, `setGhost` dans l'API de `PartBuilder` ainsi que la section 9.4) : en clair on ne veut pas que la « hitbox » collisione avec le reste du dispositif, notamment les roues.

Une constante `MAX_WHEEL_SPEED` (avec la valeur `20.0f`) pourra être utilisée pour fixer la vitesse seuil en dessous de laquelle il faut motoriser les roues pour les faire bouger.

Enfin, comme le cycliste est orienté, un attribut indiquant s'il regarde vers la droite ou vers la gauche est sans doute une bonne idée.

5.2.1 Les roues motrices

Vous avez eu l'occasion de jouer avec la notion de « contrainte physique » dans la partie tutoriel (section 3.3). Nous y revenons ici pour créer les roues du vélo.

L'idée est de créer une `GameEntity`, `Wheel`, modélisant une roue. Il est raisonnable de considérer qu'une `Wheel` peut aussi jouer le rôle d'un `Actor` de `BikeGame` en tant que telle. Elle sera caractérisée par la position de son centre et son rayon.

Le `Bike` aura donc deux attributs `Wheel` qu'il faudra lui attacher physiquement au moyen d'une contrainte.

Seule la roue arrière sera motrice. Il nous faut donc le moyen de repérer la roue arrière de la roue avant. Comme le cycliste est orienté (il regarde soit vers la droite soit vers la gauche) il est plus simple de considérer qu'il y a une roue gauche et une roue droite (voir

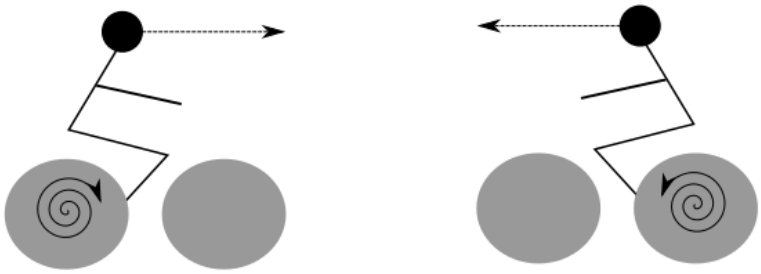


FIG. 10 : Si le cycliste regarde vers la droite, la roue gauche est la roue arrière et donc la roue motrice. S'il regarde vers la gauche, c'est la roue droite qui est motrice.

la figure 10).

La `WheelConstraint` fournie dans le répertoire `math` permet d'attacher une roue motorisée à une entité. C'est cette contrainte que vous aller utiliser pour lier les roues au `Bike`.

Elle doit pouvoir être attachée à une autre entité pour la faire rouler. Nous vous suggérons de fournir dans l'API de `Wheel` une méthode telle que :

```
public void attach(Entity vehicle, Vector anchor, Vector axis)
```

Cette méthode aura typiquement pour vocation de créer une `WheelConstraint` entre la roue et `vehicle` par du code ressemblant à cela :

```
WheelConstraintBuilder constraintBuilder = ...;
constraintBuilder.setFirstEntity(vehicle);
// point d'ancrage du véhicule :
constraintBuilder.setFirstAnchor(anchor);
// Entity associée à la roue :
constraintBuilder.setSecondEntity(wheelEntity);
// point d'ancrage de la roue (son centre) :
constraintBuilder.setSecondAnchor(Vector.ZERO);
// axe le long duquel la roue peut se déplacer :
constraintBuilder.setAxis(axis);
// fréquence du ressort associé
constraintBuilder.setFrequency(3.0f);
constraintBuilder.setDamping(0.5f);
// force angulaire maximale pouvant être appliquée
//à la roue pour la faire tourner :
constraintBuilder.setMotorMaxTorque(10.0f);
constraint = constraintBuilder.build();
```

Les points d'ancrages sont toujours donnés en coordonnées relatives.

La roue sera ainsi attachée au véhicule et pourra se déplacer sur la droite `anchor + axis * t`, la position `t` étant attaché à un ressort (suspension) (voir la figure 11).

Ainsi attacher les roues à l'`entity` d'un vélo ayant la forme géométrique suggérée plus haut pourrait se faire par des appels tel que :

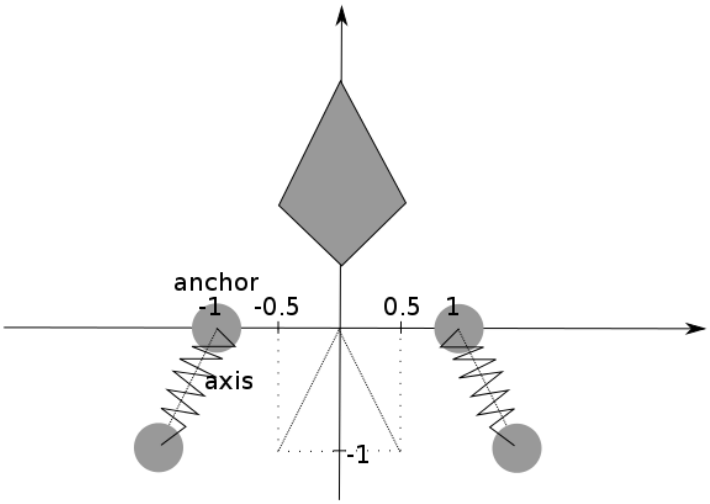


FIG. 11 : `anchor` est le point d'ancrage au véhicule, `axis` donne l'axe sur lequel la roue peut se déplacer en étant attachée à un ressort (suspensions)

```
leftWheel.attach(entity, new Vector(-1.0f, 0.0f), new
    Vector(-0.5f, -1.0f));
rightWheel.attach(entity, new Vector(1.0f, 0.0f), new
    Vector(0.5f, -1.0f));
```

En guise de simplification, vous êtes autorisé ici à faire en sorte que `ActorGame` fournisse des « constraint builders » de son monde physique (ici spécifiquement des `WheelConstraintBuilder`).

Question 5

En toute rigueur, il serait beaucoup plus propre conceptuellement d'attacher une `GameEntity` à une autre `GameEntity` et non pas une `Entity`. Quelle est le problème lorsque l'on remplace `Entiy` par `GameEntity` dans :

```
public void attach(Entity vehicle, Vector anchor, Vector
    axis)
```

Comment pourrait-on y remédier ? (il ne vous est pas demandé de le faire)

Pour pouvoir agir sur la roue, les méthodes suivantes sont préconisées dans la classe `Wheel` :

- `public void power(float speed)`

  qui permet d'activer le moteur associé à la roue (pour la faire tourner, `speed` est la vitesse de rotation du moteur (utilisez pour cela `setMotorEnabled` de l'API des `WheelConstraint`).

Immobiliser une roue `wheel` reviendra donc à écrire :

```
wheel.power(0.0f);
```

- ```
  public void relax()
  ```

  qui permet de désactiver le moteur.

- ```
  public void detach()
  ```

  qui permet de détruire la contrainte liant le véhicule à la roue (les contraintes ont une méthode `destroy`).

- ```
  /**
  @return relative rotation speed, in radians per second
  */
  public float getSpeed()
  ```

  qui retourne différence entre la vitesse angulaire de la roue et celle du véhicule auquel elle est éventuellement attachée.

Avec les grandeurs suggérées pour le `Bike`, vous pouvez prendre `0.5f` comme rayon des roues et placer ces dernières aux positions `p + (-1.0f, 0.f)` et `p + (1.0f, 0.f)` où `p` désigne la position du `Bike`.

Vous pouvez utiliser une image circulaire quelconque pour le dessin d'une `Wheel` (par exemple *`"explosive.11.png"`* ou simplement un cercle coloré).

Enfin, n'oubliez pas qu'une roue doit pouvoir être détruite et disparaître de la simulation. Une roue détruite sera bien sûr détachée (méthode `detach`) du véhicule auquel elle est éventuellement attachée.

5.2.2 Le dessin

Si on se bornait à dessiner la « hitbox » et les roues pour dessiner le vélo, le visuel ne serait pas très plaisant.

Pour agrémenter la représentation, la méthode de dessin du vélo devra se charger de dessiner le cycliste par dessus la « hitbox ».

Vous êtes libre de choisir la représentation que vous voulez pour le dessin, comme par exemple le petit personnage de la figure 12.

Nous vous suggérons de modulariser le dessin au moyen de « getters » et de raisonner dans le référentiel local du vélo (voir la figure 13) selon les exemples suivants :

```
// Draw head
Circle head = new Circle(0.2f, getHeadLocation());
```

où `getHeadLocation` retourne la position de la tête en coordonnées locales :

```
// Head location, in local coordinates
private Vector getHeadLocation() {
```
```

FIG. 12 : Dessin du cycliste au moyen de lignes pour le corps et d'un cercle pour la tête.

```
    return new Vector(0.0f, 1.75f);
}
```

et pareil pour le reste du corps :

```
// Draw arm
Polyline arm = new Polyline(
    getShoulderLocation(),
    getHandLocation());
```

Cela vous donnera plus de facilité si par la suite vous voulez animer la représentation et donner l'illusion que le cycliste lève les bras ou pédale (suggéré dans les extensions).

Rappelez-vous que `ActorGame.getCanvas` fournit l'abstraction de la zone de dessin associée au jeu.

5.2.3 Les contrôles

La méthode `update` du `Bike` devra mettre en oeuvre les contrôles permettant d'agir sur le vélo. On rappelle que seule la roue arrière est motrice.

L'algorithme suggéré est le suivant :

1. Prendre en charge le contrôle (barre d'espace) permettant d'inverser l'orientation du cycliste. Par exemple, le fait d'appuyer sur la barre d'espace permet de l'orienter vers la droite s'il l'était vers la gauche et vice-versa.
2. Désactiver par défaut la motorisation des deux roues.
3. Prendre en charge le contrôle permettant de bloquer les roues (flèche bas pressée).
4. Prendre en charge le contrôle permettant de faire rouler le vélo (flèche haut pressée) : si la vitesse est inférieure à `MAX_WHEEL_SPEED` on va motoriser les roues pour les faire rouler à cette vitesse. Sinon, il faut désactiver le moteur (qui n'est plus nécessaire car les roues roulent seules, par exemple en raison d'une pente). Si le cycliste regarde vers la droite, la roue motrice est la roue gauche et elle tourne à l'envers des aiguilles d'une montre ; et donc, le fait de ne pas avoir atteint la vitesse maximale se teste de la sorte : `leftWheel.getSpeed()> -MAX_WHEEL_SPEED)`, si le cycliste regarde vers la gauche, par `rightWheel.getSpeed()< MAX_WHEEL_SPEED`.

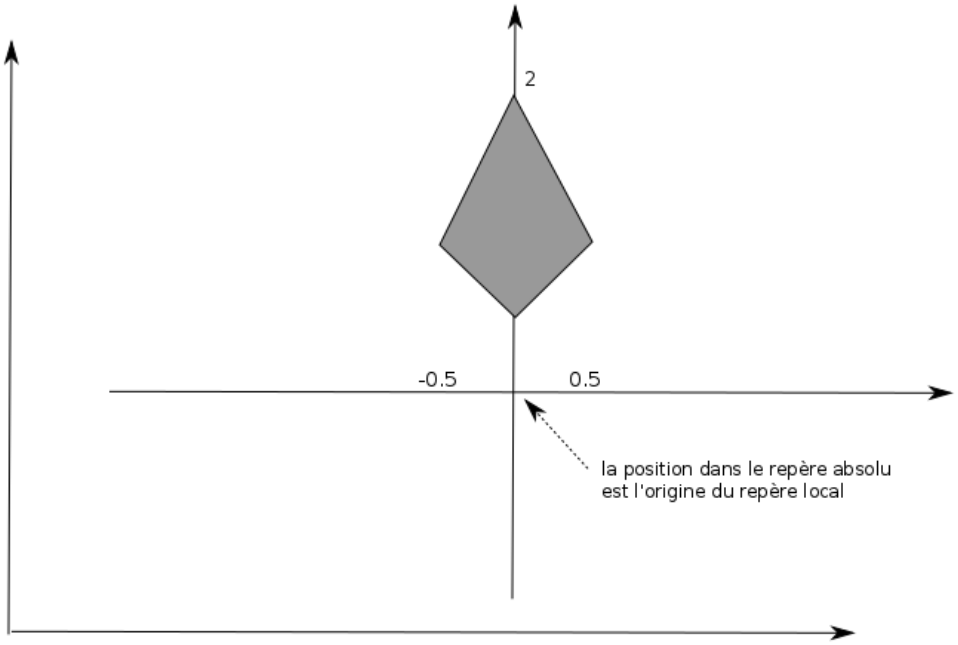


FIG. 13 : Il est plus facile de raisonner dans le référentiel local du vélo pour dessiner le cycliste.

5. Prendre enfin en compte le contrôle (flèches gauche et droite) permettant d'appliquer une force au vélo, par exemple au moyen de :

   ```
   entity.applyAngularForce(10.0f);
   ```

   si on veut faire monter le vélo ou

   ```
   entity.applyAngularForce(-10.0f);
   ```

   si on veut le faire descendre. `entity` représente l'`Entity` associée au `Bike`

Rappelez-vous que `ActorGame` fournit l'abstraction du clavier associé au jeu.

5.2.4 Gestion de la chute

Lorsque le cycliste chute sur le sol, la partie est perdue et le vélo doit être détruit. Nous nous occuperons de la gestion de la fin de partie un peu plus loin. Pour l'heure nous nous intéressons à détecter la chute. Pour rappel, la `GameEntity`, `Bike` est l'abstraction du couple vélo-cycliste dont la forme physique (« hitbox ») a justement pour but de détecter des situation du chute : si la « hitbox » est en contact avec un objet du monde extérieur alors il y a potentiellement chute du cycliste.

De façon analogue à ce que nous avons vu dans le tutoriel (section 3.5), la `GameEntity Bike` doit être à l'écoute de contacts éventuels avec les objets extérieurs et il faut par conséquent lui associer un `ContactListener`. Ici la situation est un peu particulière :

- Si l'on anticipe l'existence d'objets traversables (comme une ligne d'arrivée), le contact avec ces derniers ne doit pas être assimilé à une chute.
- Les éventuels contacts de la « hitbox » avec les roues, doivent être négligés.

Le `ContactListener` devrait donc enregistrer les entités non « ghosts » (voir à ce propos le complément de la section 9.4) et exclure les roues. Il est donc plus spécifique que celui précédemment utilisé.

Il est en fait possible d'ajouter au `Bike` un `ContactListener` qui lui spécifique selon la syntaxe suivante :

```
ContactListener listener = new ContactListener() {
     @Override
     public void beginContact(Contact contact) {
         if (contact.getOther().isGhost())
             return;
        // si contact avec les roues :
             return;
         hit = true;
     }
     @Override
     public void endContact(Contact contact) {}
 };

 addContactListener(listener);
```

On définit ici une classe anonyme qui redéfinit `beginContact` et `endContact` de façon spécifique. La variable `listener` est une instance de cette classe anonyme. L'avantage est que la classe anonyme a accès à toutes les variables de la classe dans laquelle elle est définie. Elle peut par exemple directement accéder à un attribut `hit` de la classe `Bike` (et aux roues aussi !). Ceci nous épargne des soucis au niveaux de l'encapsulation.

Question 6

Comment proposez-vous d'étendre l'API de `GameEntity` pour garantir que sa méthode `getEntity` puisse rester protégée, en dépit des traitements nécessaires pour filtrer le contact avec les roues ?

Important : Dans `Bike.update`, la détection de la chute doit primer sur les autres traitements liés à la logique du jeu.

5.3 La ligne d'arrivée

Pour modéliser la ligne d'arrivée vous pouvez utiliser un acteur `Finish` qui sera aussi une `GameEntity`. La forme géométrique de cette dernière (par exemple un cercle) servira à détecter les collisions avec le cycliste. Il s'agit typiquement d'un cas de `Part` fantôme (section 9.4). Pour le dessin de la ligne d'arrivée vous pouvez choisir une image telle que *`"flag.red.png"`*.

`Finish` devra être à l'écoute de collisions (voir le tutoriel de la section 3.5). Une des problématiques qui se pose ici est que seule la collision avec le cycliste doit faire réagir la ligne d'arrivée. Celle-ci doit alors signifier au jeu que la partie est gagnée et disparaître du jeu.

Le jeu réagira à la notification de la ligne d'arrivée pour afficher un message de victoire par exemple.

Remarque : Le cycliste est le « personnage » central du jeu. Il est courant dans la conception des jeux de traiter une telle entité de façon particulière et d'en autoriser l'accès : `ActorGame` peut fournir une référence au personnage central `getPayload()` et permettre aussi sa mise à jour `setPayload`. Cet accès est potentiellement nocif pour l'encapsulation mais l'abstraction `Game` est un bon garde-fou. Le programmeur utilisateur d'un jeu (`Program` pour nous) va travailler avec un `Game` (qui cache les accès intrusifs) et non avec l'implémentation `ActorGame` (qui autorise ces accès). Les programmeurs du moteur de jeu (`ActorGame` et tous les acteurs) doivent par contre faire preuve de responsabilité et veiller à ce que le partage des informations nécessaire se fasse sans corrompre les objets partagés.

Les messages textuels (pour signifier la victoire par exemple), peuvent être affichés au moyen de la classe `TextGraphics`.

Un attribut `message` de ce type peut être associé au `BikeGame` et initialisé comme suit :

```
message = new TextGraphics("", 0.3f, Color.RED, Color.WHITE, 0.02f,
    true, false, new Vector(0.5f, 0.5f), 1.0f, 100.0f);
message.setParent(getCanvas());
message.setRelativeTransform(Transform.I.translated(0.0f, -1.0f));
```

La méthode `setText` de `TextGraphics` peut être utilisée pour changer le contenu textuel du message.

Pour comprendre le rôle de `setRelativeTransform`, consultez le complément sur le positionnement des objets (section 9.2, en particulier à propos de la classe `Attachable`).

5.4 Gestion des fins de partie

Lorsque la méthode `update` de `Bike` détecte une chute, le vélo doit disparaître et la partie doit se terminer sur un « Game Over ».

La méthode `destroy` de `Bike` doit être invoquée car il faut détruire toutes les entités physiques associées au `Bike` (sinon elles continueront à être simulée par le moteur physique !).

Par ailleurs, le jeu auquel appartient le vélo doit être informé de la disparition de ce dernier et doit le supprimer de sa liste d'acteurs.

> Attention : modifier le contenu d'une collection pendant que l'on itère dessus (à l'aide d'une `for` loop) peut conduire au lancement de `ConcurrentModificationException`. Il faut donc modifier un peu la classe `ActorGame` de sorte à ce que l'ajout et la suppression d'acteurs se fasse avant ou après l'itération sur cette collection.

Il est judicieux à ce stade de prévoir une fonctionnalité de « reset » permettant de relancer le jeu en cas victoire ou d'échec. On peut prévoir pour cela la gestion d'un contrôle (par exemple via la touche `'R'`).

Conceptuellement, il est plus naturel de placer la gestion de ce contrôle au niveau du `update` du jeu (et non pas de celui du vélo). Par exemple, on pourrait avoir des abstractions `Controller` qui seraient en charge de la gestion des contrôles. On pourrait donc associer des `Controller` aux acteurs ou aux jeux. Notez que vous n'êtes pas obligés de procéder de la sorte.

6 Extensions (étape 4)

Pour atteindre la note maximale, il vous est demandé de coder quelques extensions librement choisies parmi celles suggérées ci-dessous. Vous devrez cumuler 25 points au moins. La mise en oeuvre est libre. Seules des suggestions et indications vous sont données ci-dessous. Une estimation de barème pour les extensions suggérées est donnée dans la section 6.7.

Comme résultat final du projet, il vous est demandé de rendre (au moins) un jeu que l'on peut lancer dans `Program` et qui fait intervenir tous les composants que vous avez codé, extensions comprises.

Une petit bonus sera attribué si vous faites preuve d'inventivité dans la conception du jeu (par exemple, placement des composants aux bons endroits pour créer des petit challenges, comme le fait que le vélo doive prendre de la vitesse pour faire tomber un pilier et ainsi se créer un passage vers la ligne d'arrivée etc.).

Vous prendrez soin de commenter soigneusement dans votre `README`, le nom de vos jeux (et/ou niveaux) et les modalités de jeu qu'ils impliquent. Nous devons notamment savoir quels contrôles utiliser et avec quels effets sans aller lire votre code.

Il est attendu de vous que vous choisissiez quelques extensions et les codiez jusqu'au bout (ou presque). L'idée n'est pas de commencer à coder plein de petits bouts d'extensions disparates et non aboutis pour collectionner les point nécessaires ;-).

6.1 Niveaux de jeux

Il est rapidement tentant de faire en sorte que le vélo passe à un autre niveau de jeu (avec d'autres terrains/objets physiques etc.) en fonction de certains critères, par exemple, lorsqu'il franchit la ligne d'arrivée.

Un moyen simple de mettre cela en place est de créer un Actor `Level` (qui ne serait qu'un acteur logique sans représentation physique), et dont le rôle est de créer les objets du monde à simuler. Il disposerait par exemple d'une méthode :

```
public abstract void createAllActors();
```

en charge de créer tous les objets voulus dans un niveau donné.

Des sous-classes de `Level`, donneraient alors des définitions concrètes à la méthode `createAllActors()`, avec à chaque fois à la clé un monde différent.

Un jeu avec des niveaux aurait donc comme attribut une liste de niveaux possibles qu'il peut créer par une méthode telle que :

```
protected List<Level> createLevelList() {
```

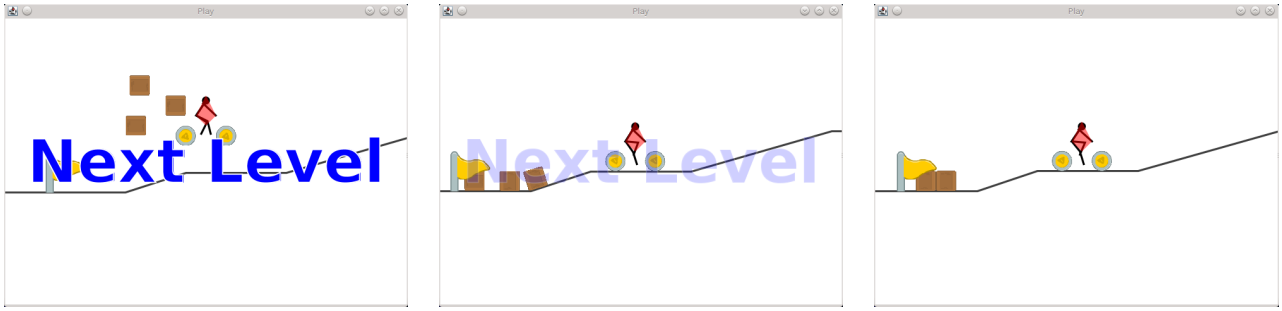


FIG. 14 : L'acteur en charge du niveau permet les affichages signalant la transition. On joue sur les transparences dans `ImageGraphics` pour faire s'estomper le message au cours du temps. Lorsque le message a disparu, le niveau peut commencer et l'acteur `Level` peut être supprimé de la simulation.

```
        return Arrays.asList(
            new BasicBikeGameLevel(..),
            ...
            new CrazyEpicLevel(..)
        );
    }
```

et c'est à lui de gérer les transitions entre niveaux. Le passage à un nouveau niveau implique notamment de :

- Détruire tous les acteurs courants.
- Invoquer la méthode `createAllActors` du nouveau niveau.
- Ajouter le nouveau niveau aux acteurs du monde.

Le dernier point est intéressant si l'on souhaite faire en sorte que le passage à un nouveau niveau entraîne des actions visibles dans le monde (`update` et `draw` du `Level` impliqué), comme par exemple un message de transition à un autre niveau qui s'estompe au cours du temps (voir la Figure 14).

Ici le rôle de l'acteur `Level` serait uniquement d'afficher ce message de transition. L'acteur peut alors disparaître du monde lorsqu'il a fini d'afficher son message.

Du point de vue de la conception, un `Level` va typiquement hériter de `Node` afin de situer les messages (ou autres affichages) qui lui sont liés dans un repère. Il est aussi intéressant de spécifier les fonctionnalités liées à un jeu avec niveaux dans une interface `GameWithLevels` qui indiquerait les fonctionnalités typiques attendues comme :

```
// gère ce qui se passe lorsque la transition au niveau suivant doit
    se faire :
nextLevel();
// gère ce qui se passe lorsque l'on veut recommencer le niveau
    courant :
resetLevel();
```

La décision de passer à un autre niveau dépend de la logique du jeu souhaitée. Vous pouvez simplement faire en sorte que ce soit le franchissement de la ligne d'arrivée qui cause le passage à un autre niveau (qui sollicite le `nextLevel()`). Il existe d'autres façons de faire, comme utiliser un sélecteur de niveau (un niveau spécial permettant de lancer à la main d'autres niveaux, quelques indications sont données dans la section « Aller plus loin »).

Dans un jeu à plusieurs niveaux, la notification de victoire peut intervenir seulement à la fin de tous les niveaux (à vous de voir quelle logique vous voulez mettre en place, mais documentez-la dans votre `README`).

A la fin des niveaux possibles on peut revenir au premier niveau pour recommencer tout le jeu.

6.2 Edition de niveaux

Il peut être intéressant d'ajouter des composants à la main, dynamiquement en cours de jeux. Par exemple, pouvoir cliquer pour créer des caisses, ou bien ajouter des points d'une polyline pour créer un bout de terrain.

Ici, avoir deux niveaux de contrôles (celui au niveau du jeu et celui au niveau des acteurs) devient particulièrement utile. On peut en effet au niveau du jeu créer des contrôles pour : mettre la simulation en pause (à vous de voir comment !), créer les nouveaux objets dans ce monde à l'arrêt, puis repasser ensuite en mode simulation.

Les problèmes soulevés alors sont :

- Comment déplacer la caméra : il est en effet probablement bien utile de déplacer la caméra dans différentes directions pour pouvoir faire les ajouts aux endroits où on le souhaite. Vous pouvez pour cela introduire un `shiftedViewCenter` en relation avec le `viewCenter` dans `ActorGame`, et écrire des choses ressemblant à ceci :

```
if (pause){ // on teste si le jeu est en mode pause
    if (getKeyboard().get(KeyEvent.VK_RIGHT).isDown()){
     shiftedViewCenter = shiftedViewCenter.add((new Vector(0.1f,
         0.0f)));
    }
// Compute shifted viewport
Transform shiftViewTransform =
    Transform.I.scaled(VIEW_SCALE).translated(shiftedViewCenter);
  window.setRelativeTransform(shiftViewTransform);
}
```

- Comment gérer la persistance des objets créés : idéalement on aimerait pouvoir retrouver les objets créés dynamiquement dans un jeu ou sur un niveau. Cela nous pousserait cependant bien au delà des exigences de ce projet et touche à des concepts qui ne sont pas encore présentés (sérialisation d'objets, entrée-sorties). Ceci n'est donc pas attendu de vous.

6.3 Triggers

La ligne d'arrivée codée à l'étape précédente préfigure une classe d'acteurs physiques plus généraux : ceux ne réagissant pas aux collisions physiquement mais capable de les prendre en compte pour entreprendre certaines actions (par exemple décréter la victoire du joueur).

Cette extension consiste à coder une classe `Trigger` plus générale dont hériterait l'acteur `Finish`. Cette classe peut être mise à profit pour créer d'autres acteurs réagissant au contact du joueur sans entrer en collision physiquement avec lui (`Checkpoint`, ou objets à collectionner donnant un score au joueur).

Pour pouvoir utiliser cette classe de façon plus générale, par exemple pour pouvoir en dériver des sous-classes d'objets à collecter, il est intéressant de lui associer des « timer ». Un `Trigger` pourrait, grâce à une gestion appropriée de ces « timer », disparaître momentanément lorsqu'il a été touché, puis réapparaitre un peu plus tard. Si le `Trigger` disparaît pendant un moment, il doit être considéré comme inactif (sans influence sur le reste du monde et n'y apparaissant pas).

> Astuce : en utilisant judicieusement la surcharge des constructeurs et en assignant dans certains cas des valeurs très grandes à certains « timer » (comme `Float.POSITIVE_INFINITY` pour un temps de réapparition « infini » par exemple), on pourra par exemple créer des `Trigger` qui ne réapparraîssent jamais après avoir été touchés.

Cette extension consiste donc à faire en sorte que `Finish` hérite d'une classe plus générale `Trigger` puis à ajouter d'autres objets de type `Trigger` ; comme des `Checkpoint`, ou des objets à collecter (`Pickup`) qui ont une certaine influence sur le joueur (attribution de points de scores par exemple).

6.4 Particules

Pour améliorer l'aspect visuel du jeu, vous pouvez introduire des éléments de décors animés. Exemples :

- des papillons qui suivent le joueur pendant un moment
- des étincelles ou des particules de poussières qui apparaîssent quand le cycliste freine/collisionne etc.

Il est pour cela raisonnable de concevoir une classe `Particle` pour représenter une particule et une autre classe pour représenter un émetteur (créateur) de particules, `Emitter`.

Une particule est ici considéré comme un élément graphique plutôt que comme un acteur (il s'agira typiquement d'un `Graphics` et d'un `Positionable`). Elle pourra être modélisée au moyen de divers caractéristiques permettant de la faire accélérer, tourner sur elle même, s'effacer graduellement au cours du temps jusqu'à disparaitre, etc. Il faudra donc

typiquement lui associer des attributs qui permettront de jouer avec sa position, vitesse et/ou accélération, comme :

```
private Vector position; // dans le repère absolu
private Vector velocity;
private Vector acceleration;

private float angularPosition;
private float angularVelocity;
private float angularAcceleration;
```

La transformée associée aurait alors cette allure :

```
@Override
public Transform getTransform() {
   return
      Transform.I.rotated(angularPosition).translated(position);
}
```

Pour faire en sorte qu'une particule puisse causer la génération d'autres particules identiques (copies polymorphiques), on peut imaginer une méthode telle que :

```
public abstract Particle copy();
```

La méthode `update` d'une particule va donc faire des mises à jours des positions et vitesse selon un schéma tel que :

```
velocity = velocity.add(acceleration.mul(deltaTime));
position = position.add(velocity.mul(deltaTime));
angularVelocity += angularAcceleration * deltaTime;
angularPosition += angularVelocity * deltaTime;
```

Comme instances concrètes de particules, on peut imaginer des particules dessinables comme des formes ou des particules comme des images. Vous pourrez créer des sous-classes `ImageParticle` et `ShapeParticle` répondant à cette spécification (l'une ou l'autre suffira).

Conseil : vous pouvez vous inspirer de `ImageGraphics` et `ShapeGraphics` pour la mise en oeuvre de `ImageParticle` et `ShapeParticle` du point de vue du dessin.

Emetteurs de particules : Un `Emitter` serait quant à lui un acteur doté d'une forme (visible ou pas) à l'intérieur de laquelle seraient générées des particules à des positions aléatoire (avec une limite supérieure du nombre de particules). Il doit être possible de créer des émetteurs permanents ou qui disparaissent de la simulation au bout d'un moment. Par ailleurs, un émetteur peut être capable d'émettre différents types de particules.

Vous noterez que la hiérarchie de `Shape` offre une méthode `sample` permettant de générer des points aléatoires uniformément distribués dans une forme donnée. Ainsi, si `area`

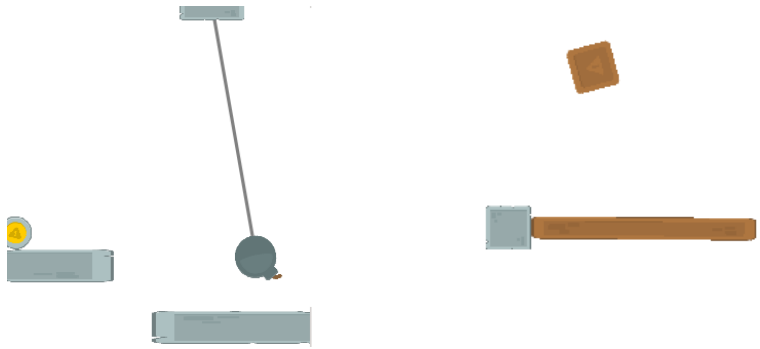


FIG. 15 : Exemple de nouveaux composants : pendule et tremplin

est la forme associée au `Emitter`, alors `getTransform().onPoint(area.sample());` permettra de positionner les particules dans le repère absolu (voir à ce propos la section 9.2).

6.5 Nouveaux acteurs/composants physiques

De très nombreux acteurs/composants physiques peuvent être envisagés (deux exemples simples sont donnés dans la figure 15). Nous vous suggérons les suivants, et d'autres idées sont données dans la section « Aller plus loin ».

- Bascules, tremplins, pendules qui oscillent et que le vélo doit éviter pour pouvoir passer (à coder sur la base de ce qui a été vu dans la partie tutoriel).
- Caisses explosives, qui explosent selon certains critères et ayant un impact sur l'entourage dans un certain rayon.
- Des pics qui immobilisent le vélo s'il passe dessus (crevaison, avec divers scénarios pour la suite : perte de score, et remise en service du vélo après un temps de latence par exemple).
- Terrain avec d'autres propriétés physiques (comme des terrains glissants, il faut alors jouer sur le paramètre de friction).
- Puits de gravité (à associer avec un système de particules pour en améliorer le visuel) : il s'agira d'une zone caractérisée par une forme. Tous les corps entrant dans cette zone subiront une accélération ; qui annulera la force de gravité et pourra les projeter vers le haut par exemple. Notez que les `Entity` sont dotés d'une méthode `applyForce`.

6.6 Animation du cycliste

En gérant le dessin de façon plus sophistiquée, la figurine du cycliste peut être animée. Vous pouvez faire en sorte qu'il :

- lève les bras en signe de succès lorsqu'il atteint un checkpoint ;

- pédale lorsqu'il avance ;
- etc.

Ceci se fera typiquement en utilisant de façon appropriée des getters/setters sur les points qui représentent les différentes parties du corps du cycliste : fin du bras, coude, genoux, arrière-train, pieds (fin de l'avant-jambe) et en faisant en sorte que la position de ces points évolue aussi en cours de jeu. Le fait de lever les bras est plutôt facile à mettre en oeuvre. Par contre le fait de le faire pédaler nécessitera plus d'efforts (et le recours à vos souvenirs de trigonométrie). La position angulaire de la roue motrice sera utilisée pour déterminer l'angle entre les jambes et avant-jambes.

6.7 Barème

Une estimation de barème est donnée ci-dessous pour les différentes idées suggéreés.

- Gestion de niveaux de jeu (avec au moins deux niveaux) : 8 points
- Edition de niveau (avec ajout dynamiques de bouts de terrains, de caisses et de blocs) : 8 points
- `Trigger` + `Finish` + `Checkpoint` : 6 points
- `Trigger` + `Finish` + Système d'objet « collectable » avec impact sur le gameplay (par exemple score du joueur) : 8 points
- `Particle` + `Emitter` + mise en situation dans le jeu : entre 10 et 13 points
- Pendule ou bascule : 2 points par acteur
- Tremplin : 4 à 6 points (seule, cette extension n'apporte pas beaucoup au jeu, il faudrait la combiner avec l'édition de niveaux et/ou le fait que le vélo puisse sauter).
- Puit de gravité (sans particules) : 3 points
- Terrains glissants : 3 points
- Caisse explosive avec impact sur l'environnement et mise en situation dans le jeu (créer les conditions pour qu'elle explose) : 5 points pour une caisse qui explose en liaison avec un « timer » et impacte l'environnement, 10 points si l'explosion de la caisse dépend de la force d'un choc, par exemple la force du choc avec le vélo.
- Pics immobilisant le vélo + mise en situation : 4 points
- Animation du cycliste : signe de victoire (bras levé) 3 points, pédalage 6 points (c'est presque plus des mathématiques que de la programmation ;-)).

Si vous envisagez d'autres extensions, y compris celle suggérées dans la partie « Aller plus loin » vous pouvez envoyer un message à cs107@epfl.ch pour que nous vous donnions une estimation du barème.

Plus d'images Nous vous avons fournis un ensemble d'images, conçues et aimablement mises à disposition par le studio Kenney. Leur site propose de nombreuses autres images dans le même style, garantissant une certaine unité pour le jeu. Toutefois, libre à vous d'utiliser d'autres images, qu'elles soient de votre création ou collectées sur la toile.

Il est alors indispensable d'en citer l'origine !

7 Concours

Les personnes qui ont terminé le projet avec un effort particulier sur le résultat final (gameplay intéressant, richesse de niveaux de jeu, effets visuels soignés, extensions nombreuses etc.) peuvent concourir au prix du « meilleur jeu du CS107 ».[5]

Si vous souhaitez concourir, vous devrez nous envoyer d'ici au **12.12 à midi** un petit "dossier de candidature" par mail à l'adresse **cs107@epfl.ch**. Il s'agira d'une description de votre jeu et des extensions que vous y avez incorporées (sur 2 à 3 pages en format .pdf avec quelques copies d'écran mettant en valeur vos ajouts).

8 « Aller plus loin »

La base que vous avez codée jusqu'ici peut être enrichie selon vos envies. Si vous êtes motivés, laissez parler votre imagination, et essayez vos propres idées. Pour ne pas y passer trop de temps (pensez aussi aux autres matières !), vous pouvez aussi les laisser en friche et y travailler comme passe-temps à l'inter-semestre pour garder de bon réflexes en programmation ;-)

Si vous abordez cette partie facultative, vous pouvez bien évidemment aussi participer au concours.

Vous trouverez ci-dessous quelques suggestions en vrac. S'il vous vient une idée originale qui vous semble différer dans l'esprit de ce qui est suggéré et que vous souhaitez l'implémenter pour le rendu ou le concours, il faut la faire valider avant de continuer[6]. Pour cela, envoyez un message à cs107@epfl.ch en décrivant brièvement ce que vous souhaitez faire. Exemples d'idées :

- Sélection de niveau plus sophistiquée : un niveau particulier qui pourrait se présenter comme sur la figure 16 et qui permettrait de sélectionner un niveau en cliquant sur une des boites.
- Editeurs de niveaux plus complexes : on peut faire afficher un menu graphique pour sélectionner des éléments à placer à la souris par exemple.
- Nouveaux acteurs/composants physiques : bombes qui se déclenchent selon certains critères. Projectiles lancés par le cycliste (laser, boules de feu, flèches, missiles), passerelles constituées de planches liées par des cordes. Utiliser `PointConstraint` (et

[5] Pour le Wall of Fame ;-) et il faudra qu'on trouve autre chose que du chocolat.

[6] L'an passé, certains se sont lancés dans la programmation d'un jeu en réseau. Cela sort définitivement de la portée de ce cours et nous n'aurions pas validé l'idée ;-)

FIG. 16 : Exemple de sélecteur de niveaux

fixed rotation) pour faire un élévateur, possiblement attaché à un système de bouton ou plaque de pression. Autres véhicules, ou par exemple ajouter une remorque ou un petit chariot à pousser.

- Diverses améliorations relatives au cycliste/personnage :
  1. lui associer des ”pouvoirs” supplémentaires : un saut (uniquement depuis le sol), un double saut (pouvoir sauter une deuxième fois après un saut), une invulnérabilité temporaire, lancer des projectiles etc.
  2. qu'il tombe et rebondisse du vélo en cas de ”mort” (impliquant de créer une class `Ragdoll`, indépendante de `Bike`) ou qu'il y ait création de débris ou d'un cadavre.
  3. que s'affichent des petites bulles de dialogues, avec des commentaires caustiques
  4. jouer avec le zoom de la caméra ou la faire trembler pour mettre en valeurs certains événements (par exemple tremblement de la caméra quand le cycliste va vite)
- Possibilité de multijoueur (local), ce qui implique de pouvoir spécifier les touches qui contrôlent le vélo, pour que chacun ait sa moitié du clavier. Cela amène à des questions relatives à la position de la caméra, qui doit être la moyenne interpolée des vélos, avec gestion correct du zoom (qui n'est pas si évident).

  Un complément au multi-vélos est de définir des filtres de collision, dans le but que les deux vélos ne se collisionnent pas. Sinon c'est injouable ;)
- toute sortes d'extensions du « gameplay » : comme introduire des adversaires (fixes ou mobiles), et de fournir des armes au cycliste (le but n'est cependant pas forcément de tendre trop vers l'aspect jeu de plateforme). Cela peut se combiner avec le but initial du `PointConstraint`. Quand on clique sur un objet, cela crée une contrainte et l'on peut déplacer les objets à la souris, y compris les lancer en relâchant au bon moment.
- etc.

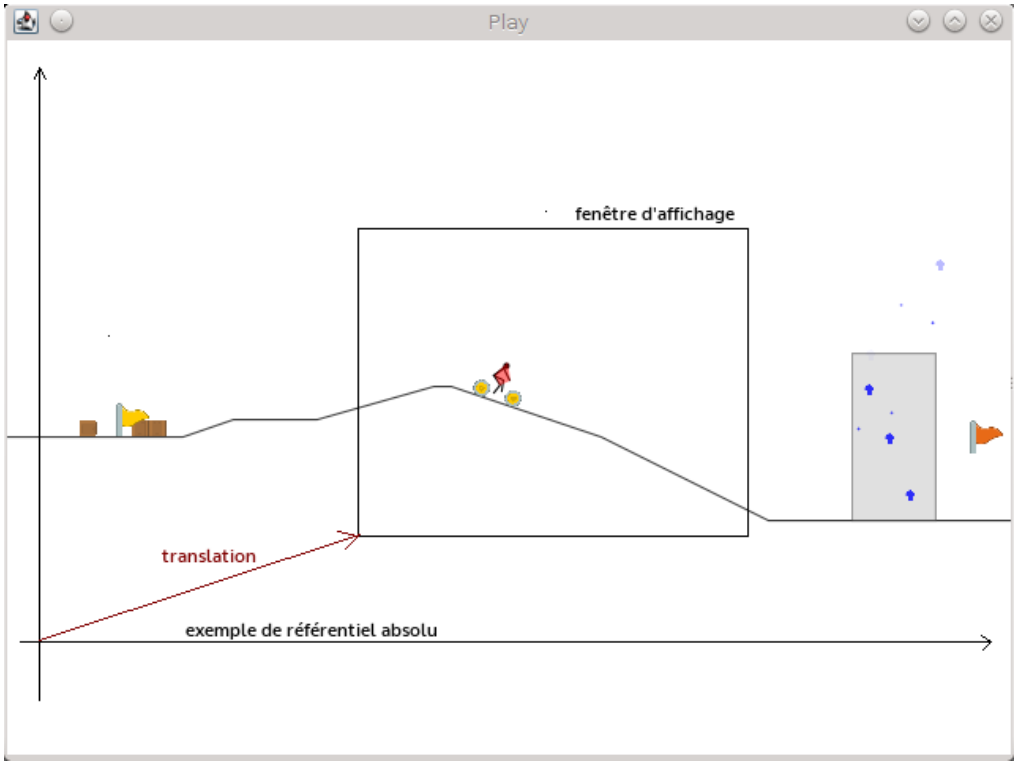


FIG. 17 : La vue sur le monde s'obtient par une transformation affine (ici une simple translation)

9 Divers compléments

9.1 Grandeurs et mesures

JBox2d utilise les unités : mètres, kilogrammes et secondes. Pour son bon fonctionnement, les objets mobiles auront typiquement entre 0.1 et 10 mètres. Les formes statiques peuvent être plus grandes. Tout corps rigide (`Entity`) a une masse finie non nulle. S'il a des « Fixtures » (`Part`), sa masse est le total des masses des `Part`, et son centre de gravité est la moyenne de ceux de ses `Part` (pondérée par les masses de ces dernières).

9.2 Objets « positionnables » et transformées

Le positionnement et l'affichage des éléments simulés dans la fenêtre de simulation sont évidemment des points fondamentaux.

La première remarque à faire à ce propos est que pour positionner les objets simulés il n'est pas commode de raisonner en pixels : cela nous rend dépendant de la taille de la fenêtre ce qui est contre-intuitif ; nos univers simulés seront probablement plus grands que ce que l'on souhaite afficher.

Nous allons donc exprimer toutes nos grandeurs relatives aux positions, dimensions etc. dans les échelles de grandeurs du monde physique simulé et non pas en terme de pixels

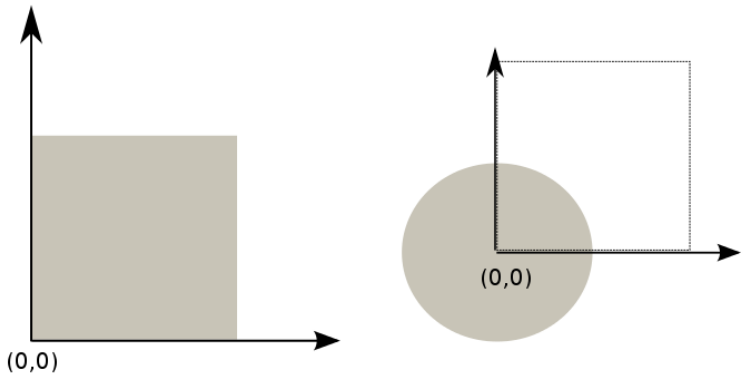


FIG. 18 : Exemples de référentiel local pour un bloc (gauche), ou un cercle (droite)

dans la fenêtre. Nos éléments vont ainsi se situer dans le référentiel absolu de notre monde physique. Néanmoins comme ce monde va potentiellement être grand, notre "fenêtre" d'affichage va subir des transformations affines (translation, zoom etc.) pour nous permettre de nous focaliser sur une partie spécifique du monde (voir la Figure 17).

La fenêtre d'affichage est un exemple typique d'élément nécessitant d'être placé/modifié dans le repère absolu par le biais de transformations. En fait tous les éléments à positionner dans le repère absolu, peuvent l'être selon le même procédé (par exemple les formes ou les images à dessiner ou les entités physiques elles mêmes).

En raison de ce besoin, l'API fournie met à disposition les éléments suivants :

- l'interface `Positionable` qui décrit un objet dont on peut obtenir la position absolue par le biais d'une transformation affine (classe `Transform`). Une `Entity` est typiquement un `Positionable`.
- l'interface `Attachable` qui décrit un `Positionable` que l'on peut attacher à un autre (son parent). Ceci se fait au moyen de la méthode `setParent`. Il est caractérisé par une transformée relative, qui indique comment l'objet sera positionné dans le référentiel de son parent (ou dans l'absolu si elle n'a pas de parent).
- la classe `Node` qui est une implémentation concrète simple de l'interface `Attachable`.

La méthode `getTransform()` appliquée à un `Positionable` permet en fait de se situer dans son référentiel local/relatif (voir la figure 18 pour des exemples). Pour "coller" une image sur une `Entity`, `block`, on peut par exemple indiquer à la méthode de dessin par quelle transformée placer l'image dans le référentiel du block :

```
// fait le dessin de l'image dans le référentiel local de block
    (application de la transformée block.getTransform)
window.drawImage(window.getImage("box.4.png"), block.getTransform(),
    1.0f, 0.0f)
```

D'ailleurs, même si l'on travaille avec des objets se plaçant dans un référentiel absolu, il est souvent utile de pouvoir raisonner dans le référentiel local, relatif à l'objet lui même.

Par exemple, pour donner les coordonnées de la tête du cycliste, il est plus simple de se positionner dans le référentiel local du vélo plutôt que dans le référentiel absolu.

```
/* getHeadLocation() retourne la position de la tête dans le
 * repère local du bike
 */
Circle head = new Circle(0.2f, getHeadLocation());
// il faut alors faire le dessin dans le référentiel du vélo :
window.drawShape(head, bike.getTransform(), Color.YELLOW, null,
    0.0f, 1.0f, 1.0f);
```

La position absolue de la tête peut être trouvée au moyen de la méthode `onPoint` de la classe `Transform` :

```
headAbsolutePosition =
    bike.getTransform().onPoint(getHeadLocation());
```

9.3 Objets graphiques

La section précédente montre comment faire un dessin en le situant dans le bon référentiel par le biais de transformées.

Vous noterez, que l'API fournie met à disposition des classes de plus haut niveau `ImageGraphics` et `ShapeGraphics` qui implémentent la notion d'objets « dessinables » (`Graphics`). Un `Graphics` peut être attaché à une `Entity` par le biais de la méthode `setParent`. Le dessin peut alors se faire de façon simple sans référence explicite aux transformées employées :

```
// on attache une image à block
ImageGraphics blockGraphics = new
    ImageGraphics("stone.broken.4.png", blockWidth, blockHeight);

// setParent fait que blockGraphics aura la même transformée
//que block
blockGraphics.setParent(block);
```

puis :

```
  // le dessin se fait dans le référentiel du parent (block)
  blockGraphics.draw(window);
```

L'image *"stone.broken.4.png"* sera positionnée de sorte à ce que son coin inférieur gauche coincide avec l'origine du référentiel local de l'entité. La méthode `ImageGraphics.draw` se charge d'adapter la taille de l'image à la taille de l'entité par l'application d'une transformation.

Il est toutefois nécessaire parfois de préciser le point d'ancrage de l'image sur l'entité (c'est à dire de combien l'image doit être décalée de l'origine pour se superposer proprement à l'entité). Par exemple, pour un cercle, vu sa position dans le référentiel local, il est nécessaire de décaler l'image d'une demi-largeur et d'une demi-hauteur pour être proprement centré (voir la figure 18) :

```
ImageGraphics ballGraphics = new
   ImageGraphics("explosive.11.png", 2.0f*ballRadius, 2.0f *
   ballRadius, new Vector(0.5f, 0.5f));
```

9.4 Gestion et détection des collisions, `Part` fantômes

Les collisions entre corps rigides sont détectées par le moteur physique. La notion de `ContactListener` permet des définir des objets à l'écoute des collisions.

`BasicContactListener` est une implémentation de base de tels types d'objets. Lorsqu'une `Entity` est associée à une `ContactListener`, et que sa simulation par le moteur physique détecte une collision (matérialisée par un objet de type `Contact`), la méthode `begin(Contact)` est invoquée. Pour le `BasicContactListener`, cette méthode se contente de stocker l'entité en contact dans une liste à laquelle on peut accéder. Ceci fournit un outil très rudimentaire qu'il sera nécessaire de compléter dès lors que l'on a besoin de *filtrer* les collisions (une `Entity` peut avoir besoin d'être réactive lors de la collision avec certains objets mais pas avec d'autres par exemple). Voir à ce propos http://www.aurelienribon.com/post/2011-07-box2d-tutorial-collision-filtering par exemple.

Pour qu'une collision entre deux `Entity` soit détectable, il faut que chacune d'elle ait au moins une `Part` lui donnant une forme géométrique. Une `Part` associée à une `Entity` peut être déclarée comme fantôme (méthode `Part.setGhost()`) auquel cas elle peut détecter les collisions sans y réagir : un `Part` fantôme va être traversée par les `Entity` qui collisionnent avec, mais la collision sera quand même détectée.

A noter que les `Part` associées à une `Entity` ne se meuvent pas les unes par rapport aux autres et ne collisionnent pas.

Enfin, pour les `Entity` liées ensemble par des contraintes, il est possible d'indiquer si l'on souhaite qu'elle collisionnent entre elles ou pas. La méthode `setInternalCollision` de la classe `ConstraintBuilder` permet de le spécifier.

10 Références

[1] http://box2d.org/manual.pdf

[2] http://trentcoder.github.io/JBox2D_JavaDoc/

[3] http://www.aurelienribon.com/post/2011-07-box2d-tutorial-collision-filtering

[4] http://info.usherbrooke.ca/ogodin/enseignement/imn428/chapitres/imn428-chap02.pdf

[5] "Effective Java", Joshua Bloch