

- Programmation Orientée Objet -

Introduction au projet

For fun and profit

TD5

3^e année ESIEA - INF3034

A. Gademer

2013 - 2014

Avant propos

*Il est temps de passer aux choses sérieuses et de se lancer dans votre propre projet : **vous allez réaliser un jeu, entièrement conçu et développé par VOUS.** Le TD5 servira à vous mettre le pied à l'étrier et vous présenter quelques notions pas encore abordées :*

- L'utilisation de bibliothèques extérieures (et de JBox2D en particulier)
- La gestion de Thread (en particulier avec Swing)

Objectif et condition de rendus

Le projet consistera donc à réaliser un jeu en Java sur vos heures autonomes de projet.

Le choix du jeu est laissé à votre **libre choix**, avec cependant **quelques contraintes** pour vous servir de cadre (et servir de grille de notation).

Ce jeu devra :

- être pleinement **fonctionnel** (mieux vaut quelques chose de simple que quelque chose d'inaboutie)
=> KISS : **Keep It Simple Stupid**,
- posséder une **interface graphique interactive** (bouton, clavier ou souris),
- utiliser la bibliothèque JBox2D comme **moteur physique** (pas de jeux de carte ou d'échec donc)
- donner la possibilité de **sauvegarder sa partie** (ou implémenter la persistance de donnée d'une manière ou d'une autre).
- être **amusant** (tant qu'à faire).

Exemples d'idées de jeux réalisables : Pong (multijoueur au minimum), Breakout, [Shufflepuck](#) (en vue du dessus), [Gorillas](#), Angry Bird, Super Mario, Asteroid, Billard, Flipper, etc.

Etape 0 : Utilisation de bibliothèque externe

La majorité des bibliothèques extérieures pour Java sont fournies sous forme d'archive JAR contenant les byte-codes des classes de la bibliothèque.

Pour inclure cette bibliothèque à votre programme il est essentiel :

- d'**importer** le nom des classes des bibliothèques dans votre programme (notez la succession de packages imbriqués, indiquant la provenance des classes) Exemple :

```
import org.jbox2d.dynamics.World; // JBox2D library
import com.thebuzzmedia.sjxp.XMLParser; // Simple Java XML Parser (SJXP) library
```

- d'**indiquer le chemin** vers les JAR au compilateur et à l'exécution (classpath)

Exemple (si les libraires sont dans le sous-répertoire lib) :

```
javac -cp ./lib/*:. Main.java
java -cp ./lib/*:. Main
```

Etape 1 : Premiers pas avec JBox2D

JBox2D est une bibliothèque de simulation physique (forces, gravité, détection de collisions, rebond, etc.) inspiré de la bibliothèque C++ Box2D.

Son code est disponible à l'adresse suivante :

<https://jbox2d.googlecode.com/files/jbox2d-2.2.1.1.zip>

Le fichier qui nous intéresse est /jbox2d-library/target/jbox2d-library-2.2.1.1.jar

JBox2D s'appuie sur une autre bibliothèque, la Simple Logging Facade for Java (SLF4J), disponible à cette adresse :

<http://www.slf4j.org/dist/slf4j-1.7.5.zip>

Le fichier qui nous intéresse est /slf4j-1.7.5/slf4j-simple-1.7.5.jar

La bibliothèque fonctionne en créant un monde `org.jbox2d.dynamics.World` peuplé d'objets constitué de formes (pour gérer les collisions) et de corps solides (auquel on applique des forces) et de fixations (pour relier les deux objets précédents).

Rien ne vaut un exemple concret :

```
import org.jbox2d.common.*;
import org.jbox2d.dynamics.*;
import org.jbox2d.collision.*;
import org.jbox2d.collision.shapes.*;

/*
 * javac -cp ./lib/*:. TestXX.java
 *
 * java -cp ./lib/*:. TestXX
 */

public class Test01 {

    public Test01() {
        /** Create World */
        Vec2 gravity = new Vec2(0.0f, -9.81f);
        World world = new World(gravity);

        /** Intermediary variables */
        BodyDef bodyDef;
        PolygonShape poly;
        CircleShape circ;
        FixtureDef fixDef;
        /** Object's body */
    }
}
```

```

Body floorBody, ballBody;

/** Create a horizontal polygon (the floor) */
/** Define body properties */
bodyDef = new BodyDef();
bodyDef.type = BodyType.STATIC; // Gravity is not applied to static objects
bodyDef.position.set(0,0);
bodyDef.angle = 0f;
/** Create body from the world and body properties */
floorBody = world.createBody(bodyDef);
/** Define shape as polygon */
poly = new PolygonShape();
poly.setAsBox(200f, 0.01f, new Vec2(0,0), 0);
/** Define the fixture properties */
fixDef = new FixtureDef();
fixDef.shape = poly; // The fixture is linked to the shape
/** Create fixture from the body and fixture properties */
floorBody.createFixture(fixDef);

/** Create a ball (object) */
/** Define body properties */
bodyDef = new BodyDef();
bodyDef.type = BodyType.DYNAMIC; // Gravity is applied to dynamic objects
bodyDef.position.set(0, 10);
bodyDef.angle = 0f;
/** Create body from the world and body properties */
ballBody = world.createBody(bodyDef);
/** Define shape as polygon */
circ = new CircleShape();
circ.setRadius(3);
/** Define the fixture properties */
fixDef = new FixtureDef();
fixDef.shape = circ; // The fixture is linked to the shape
fixDef.restitution=0.4f; // Bouncing ball
/** Create fixture from the body and fixture properties */
ballBody.createFixture(fixDef);

float timeStep = 1/6.0f; // Each turn, we advance of 1/6 of second
int nbTurn = Math.round(2/timeStep); // 2 second of simulation => 12 turn

/** Launch the simulation */
for(int i = 0 ; i < nbTurn ; i++) {
    world.step(timeStep, 8, 3); // Move all objects
    System.out.println(ballBody.getPosition()); // Position of the ball
}

public static void main(String[] args) {
    new Test01();
}
}

```

EXERCICE 1



Recopiez le code précédent dans un fichier Test01.java, compilez-le puis exécutez-le.

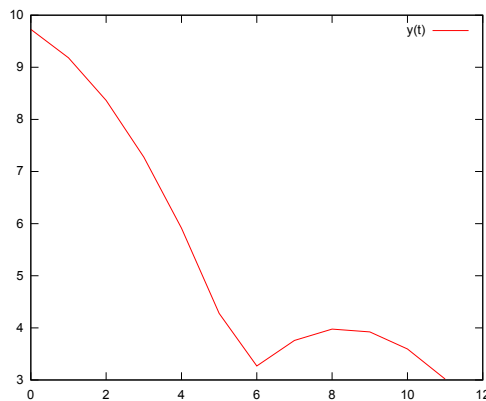
```

$javac -cp ./lib/*:. Test01.java
$java -cp ./lib/*:. Test01
(0.0,9.7275)
(0.0,9.1825)
(0.0,8.365)

```

```
(0.0,7.2749996)
(0.0,5.9124994)
(0.0,4.277499)
(0.0,3.2683754)
(0.0,3.7588754)
(0.0,3.9768753)
(0.0,3.9223752)
(0.0,3.595375)
(0.0,3.018025)
```

En affichant la courbe $y(t)$ avec `gnuplot`, on voit le comportement physique de la balle : **celle-ci tombe, puis rebondit sur le sol** (dans une parfaite vertical, comme le montre la coordonnée x qui reste nulle durant la chute). La vitesse de la chute est liée à la gravité présente sur le monde, à la densité de l'objet et à sa taille (ce qui détermine sa masse). La hauteur de rebond étant liée à l'élasticité de la balle (déterminé par sa propriété restitution).



- ✓ **Objets dynamiques vs objets statique** On notera que JBox2D différencie les objets dynamiques, influencés par les forces (gravité, rebond) des objets statiques qui influence les autres mais sont immobiles (comme collés au support). Ici, nous avons déclaré le sol comme objet statique, sinon il serait "tombé", comme la balle !
- i **Simulation par étape** La simulation du monde avance à chaque appel de la méthode `step`. C'est à l'utilisateur de choisir de combien de temps avance la simulation, par l'intermédiaire de la variable `timeStep`. Les deux autres paramètres décident de la précision des calculs de vitesse et de position. Les valeurs proposées sont un compromis entre précision et temps de calcul.
- ✓ **Et l'affichage ?** JBox2D est un moteur physique, il ne gère absolument pas l'affichage des objets ! Ce sera à vous de les dessiner.

Etape 2 : Affichage des objets

Pour afficher les objets, nous utiliserons la méthode vue au TD3 de détournement d'un composant `JPanel` pour dessiner des formes personnalisés.

Exemple :

```
import org.jbox2d.common.*;
import org.jbox2d.dynamics.*;
import org.jbox2d.collision.*;
import org.jbox2d.collision.shapes.*;
import java.awt.*;
import javax.swing.*;

/*
 * javac -cp ./lib/*:. TestXX.java
 *
 * java -cp ./lib/*:. TestXX
 */

public class Test02 extends JPanel { // CustomPanel
```

```
/* JBox2D World */
World world;
/* Object's body */
private Body floorBody, ballBody;

public Test02() {
    /** Create World */
    Vec2 gravity = new Vec2(0.0f, -9.81f);
    world = new World(gravity);

    /* Intermediary variables*/
    BodyDef bodyDef;
    PolygonShape poly;
    CircleShape circ;
    FixtureDef fixDef;

    /** Create a horizontal polygon (the floor) */
    /* Define body properties */
    bodyDef = new BodyDef();
    bodyDef.type = BodyType.STATIC; // Gravity is not applied to static objects
    bodyDef.position.set(0,0);
    bodyDef.angle = 0f;
    /* Create body from the world and body properties */
    floorBody = world.createBody(bodyDef);
    /* Define shape as polygon */
    poly = new PolygonShape();
    poly.setAsBox(200f, 0.01f, new Vec2(0,0), 0);
    /* Define the fixture properties */
    fixDef = new FixtureDef();
    fixDef.shape = poly; // The fixture is linked to the shape
    /* Create fixture from the body and fixture properties */
    floorBody.createFixture(fixDef);

    /** Create a ball (object) */
    /* Define body properties */
    bodyDef = new BodyDef();
    bodyDef.type = BodyType.DYNAMIC; // Gravity is applied to dynamic objects
    bodyDef.position.set(0, 10);
    bodyDef.angle = 0f;
    /* Create body from the world and body properties */
    ballBody = world.createBody(bodyDef);
    /* Define shape as polygon */
    circ = new CircleShape();
    circ.setRadius(3);
    /* Define the fixture properties */
    fixDef = new FixtureDef();
    fixDef.shape = circ; // The fixture is linked to the shape
    fixDef.restitution=0.4f; // Bouncing ball
    /* Create fixture from the body and fixture properties */
    ballBody.createFixture(fixDef);

    /* Wrapping JFrame */
    JFrame frame = new JFrame("JBox2D GUI");
    frame.setMinimumSize(new Dimension(640,480));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLayout(new BorderLayout());
    frame.add(this, BorderLayout.CENTER); // Add Test02 Panel to the frame
    frame.pack();
    frame.setVisible(true);
}

/** If we put the simulation loop in the constructor, it would be played before showing the
    GUI, so we put it in an external loop */
```

```

public void run() {
    try {

        float timeStep = 1/6.0f; // Each turn, we advance of 1/6 of second
        int nbTurn = Math.round(2/timeStep); // 2 second of simulation => 12 turn
        int msSleep = Math.round(1000*timeStep); // timeStep in milliseconds

        /* Launch the simulation */
        for(int i = 0 ; i < nbTurn ; i++) { // 12 turn = 2 seconds
            world.step(timeStep, 8, 3); // Move all objects
            Thread.sleep(msSleep); // Synchronize the simulation with real time
            this.updateUI(); // Update graphical interface
        }
    } catch (InterruptedException ex) {
        System.err.println(ex.getMessage());
    }
}

@Override
public int getWidth() { // Width of the CustomPanel
    return 640;
}
@Override
public int getHeight() { // Height of the CustomPanel
    return 480;
}
@Override
public Dimension getPreferredSize() { // Dimension of the CustomPanel
    return new Dimension(getWidth(), getHeight());
}
@Override
public void paint(Graphics g) { // Painting the CustomPanel
    super.paint(g);

    float scale = 10; // Ratio between the JBox2D world and the drawings of the CustomPanel

    Vec2 objectPos;
    float radius;
    Point drawingPos, drawingSize;

    /** Custom drawing by method calls to the Graphics instance (see java.awt.Graphics) */
    /** Drawing background */
    g.setColor(Color.BLACK); // Set the color to Black
    g.fillRect(0,0, getWidth(), getHeight()); // Fill a rectangle where top-left corner is at
        (0,0) and of the same size than the component

    /** Drawing the ball */
    g.setColor(Color.YELLOW); // Set the color to Yellow
    objectPos= new Vec2(ballBody.getPosition()); // Position of the ball, in the world
        coordinate, centered
    // We need to take a copy (through the constructor) so we can change the coordinate
        without modifying the simulation
    radius = ballBody.getFixtureList().getShape().getRadius(); // Get Radius of the circle
    // Coordinate conversion from the center to the top-left corner (we know the radius is 3)
    objectPos.x -= radius;
    objectPos.y += radius;
    // Scaling (and rounding) conversion => Drawing will be 10 times bigger
    drawingPos = new Point(Math.round(objectPos.x*scale), Math.round(objectPos.y*scale));
    // For a circle => width = height = diameter = 2*radius
    drawingSize = new Point(Math.round(2*radius*scale), Math.round(2*radius*scale));
    // Reference change from X-Centered, Y-Bottom Positive up (World ref) to X-Left, Y-Top
        Positive down (Drawing ref)
    drawingPos.x += getWidth()/2;

```

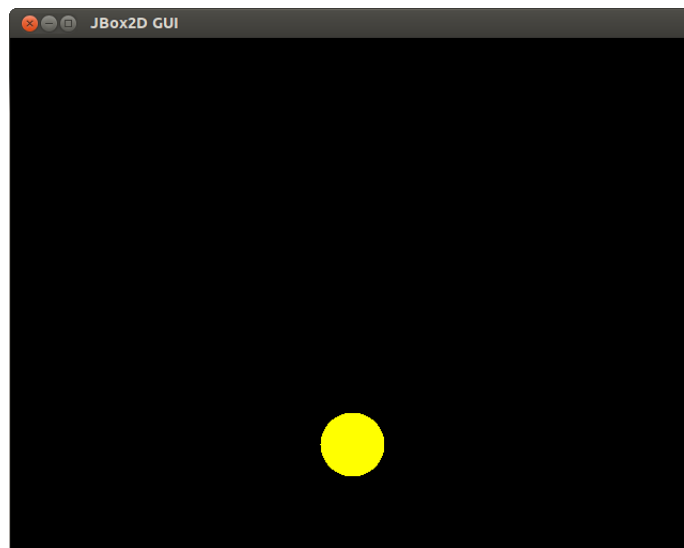
```
        drawingPos.y = getHeight() - drawingPos.y;
        // Actual drawing of the circle
        g.fillOval(drawingPos.x, drawingPos.y, drawingSize.x, drawingSize.y);
    }





    public static void main(String[] args) {
        new Test02().run(); // Create the objet, then call the run method.
    }
}
```

EXERCICE 2



Recopiez le code précédent dans un fichier `Test02.java` (complétez avec les lignes tirées de `Test01.java`), compilez-le puis exécutez-le.



-  **Facteur d'échelle** JBox2D est conçu pour gérer des distances métriques raisonnables (quelques mètres), difficilement compatible directement avec des distances d'affichage en pixel (1280x800 pour un écran moyen). C'est pourquoi on utilise généralement un facteur d'échelle (10 dans l'exemple ci-dessus) qui permet de convertir les positions et les tailles des objets entre le moteur physique et le moteur graphique.
-  **Flottant vers entiers** Le moteur physique gère des grandeurs réelles par des nombres à virgules (**float**). Le moteur graphique de son côté gère des distances en pixels (**int**) qui nécessitent donc une troncature (après application du facteur d'échelle). On utilise pour cela la méthode statique de la classe `Math` : `int Math.round(float a)`.
-  **Changement de repère** Pour simplifier les choses, le repère du moteur physique est généralement centré (le 0 au milieu) avec les ordonnées positives vers le haut. De son côté, le moteur graphique a son origine dans le coin haut gauche et les ordonnées positives vers le bas ! Il est donc nécessaire de faire un changement de repère (comme présenté dans l'exemple ci-dessus).
-  **Interface graphique et déroulement du programme** En laissant la boucle de simulation dans le constructeur, celle-ci devra se terminer avant que l'affichage de la fenêtre se termine. Pour éviter cela, on place la boucle de simulation dans une méthode à part (la méthode **public void run()**) qui est appelé dans le **main** après création de l'objet.

Etape 3 : Détecter les collisions

Utiliser un moteur physique permet de lui confier la gestion des interactions entre les objets (rebond, etc.). Pour autant il y a de nombreux cas où la rencontre entre les objets va être déterminante pour le jeu (collision entre un projectile et une cible ou entre une boule de billard et une poche).

JBox2D vous permet de rajouter un gestionnaire d'événement (un `ContactListener`) signalant les différentes collisions lorsqu'elles se produisent.

Exemple :

```
import org.jbox2d.common.*;
import org.jbox2d.dynamics.*;
import org.jbox2d.collision.*;
import org.jbox2d.collision.shapes.*;
import org.jbox2d.dynamics.contacts.*;
import org.jbox2d.callbacks.*;
import java.awt.*;
import javax.swing.*;

/*
 * javac -cp ./lib/*:. TestXX.java
 *
 * java -cp ./lib/*:. TestXX
 */

public class Test03 extends JPanel implements ContactListener {

    /* JBox2D World */
    World world;
    /* Object's body */
    private Body floorBody, ballBody;

    public Test03() {
        /** Create World */
        Vec2 gravity = new Vec2(0.0f, -9.81f);
        world = new World(gravity);
        /* Add the ContactListener to the World */
        world.setContactListener(this);

        /* Intermediary variables */
        /* ... */
    }
    /* PREVIOUS CODE */
    /* Event when object are touching */
    public void beginContact(Contact contact) {
        System.out.print("Objects are touching ");
        System.out.println(contact.getFixtureA() + " " + contact.getFixtureB());
    }

    /* Event when object are leaving */
    public void endContact(Contact contact) {
        System.out.print("Objects are leaving ");
        System.out.println(contact.getFixtureA() + " " + contact.getFixtureB());
    }

    /* Advanced stuff */
    public void postSolve(Contact contact, ContactImpulse impulse) { }
    public void preSolve(Contact contact, Manifold oldManifold) { }

    public static void main(String[] args) {
        new Test03().run();
    }
}
```

EXERCICE 3



Recopiez le code précédent dans un fichier Test03.java (complétez avec les lignes tirées de exercices précédents, compilez-le puis exécutez-le.

```
$ java -cp ./lib/*:. Test03
```



```
Objects are touching org.jbox2d.dynamics.Fixture@55881a8f org.jbox2d.dynamics.Fixture@5a87ce27
Objects are leaving org.jbox2d.dynamics.Fixture@55881a8f org.jbox2d.dynamics.Fixture@5a87ce27
Objects are touching org.jbox2d.dynamics.Fixture@55881a8f org.jbox2d.dynamics.Fixture@5a87ce27
```

✓ **Contact et rebond** Remarquez comment les deux types d'événements différents se produisent lorsque la balle touche le sol, puis rebondit (quitte le sol), puis retombe.

i **Contact et fixation** La collision se fait entre deux formes, qui sont stockés dans JBox2D au sein des fixations. C'est donc ces fixations que l'on récupère par le biais de l'événement. Pour retrouver les objets concerné il faut comparer ces fixations à celles de nos objets.

```
if(contact.getFixtureA().equals(ballBody.getFixtureList())) {
    System.out.println("A is the ball");
}
```

✓ **Contact sans collision** Comment détecter la collision d'une boule de billard avec une poche sans que celle-ci ne rebondisse dessus ? Cela se fait par l'utilisation de la propriété `setSensor(boolean sensor)` de la fixation de l'objet. Un "sensor" détecte les collisions mais les objets le traverse comme un fantôme.

```
// Sensor declaration at initialization
/* ... */
fixDef.isSensor = true;
/* ... */

// Define as sensor later
ballBody.getFixtureList().setSensor(true);
```

Etape 4 : La voie d'accélération

Afin de vous permettre de vous concentrer sur votre mécanique de jeu, vous aurez à votre disposition un certain nombre de classes pour vous simplifier l'utilisation de la bibliothèque JBox2D.

Vous trouverez ces classes dans une archive disponible sur learning.esiea.fr.

i **Package** Ces classes appartiennent au package (fait maison) `fr.atis_lab.physicalworld`, elles doivent donc être dans le sous-répertoire `fr/atis_lab/physicalworld/` (par rapport à vos sources) et doivent donc être importée :

```
import fr.atis_lab.physicalworld.*;
```

✓ **Sources disponibles** Ces classes vous sont volontairement fournies sous forme de source (et non d'archive JAR) afin de vous permettre de les **modifier** en fonction de VOS besoins.

EXERCICE 4



Téléchargez l'archive `physicalworld.zip` sur le site learning.esiea.fr.
Vous pouvez compiler la documentation dédiée avec la commande suivante :

```
javadoc fr/atis_lab/physicalworld/*.java -author -version -link http://docs.oracle.com/javase/7/docs/api/ -d doc/
```

4.1 PhysicalWorld

Cette classe est au cœur de l'extension proposée : elle encapsule l'instance de `World` et permet l'ajout d'objets (`Body/Shape` et `Fixture`) manière simplifiée. C'est aussi cette classe qui vous permettra de sauver de manière intuitive l'état de votre simulateur en intégrant les méthodes de sérialisation de JBox2D (qui utilise un protocole différent de ce qui était vu au TD3).

✓ **Entre les murs** Par défaut, le `PhysicalWorld` est délimité par quatre murs, mais n'hésitez pas à les commenter dans le code source, le cas échéant.

À leur création, les `PhysicalObject` sont automatiquement ajouté au `PhysicalWorld`.

4.2 Sprite

Pour dessiner nos objets et les identifier de manière simple, nous allons utiliser une classe `Sprite` qui contiendra :

- un nom (identifiant unique de nos objets),
- un numéro de calque (les calques sont dessinés dans l'ordre croissant),
- une éventuelle couleur (lorsque l'on dessine notre objet sous forme géométrique)
- et éventuelle image (lors que l'on désire afficher un image représentant notre objet).

Les `Sprite` sont liés aux `Body` de `JBox2D` par le champ `userData` (prévu pour étendre les capacités des `Body`).

Comme le champ `userData` est générique, on récupère l'instance de `Sprite` par le biais de la méthode statique :

```
Sprite s = Sprite.extractSprite(body);
```

4.3 DrawingPanel

Cette classe sert de zone de dessin pour notre `PhysicalWorld`. Le `DrawingPanel` définit un facteur d'échelle (entre le simulateur physique et le dessin) ainsi qu'une taille de composant. Cette dernière n'est pas forcément celle du monde multiplié par l'échelle, le `DrawingPanel` fonctionnant comme une « fenêtre » pouvant se déplacer (ou zoomer) sur un centre d'intérêt particulier.

Enfin, on peut associer soit une couleur de fond, soit une image de fond au `DrawingPanel` pour une immersion complète.

4.4 Exemple d'utilisation des classes `PhysicalWorld`, `Sprite`, `DrawingPanel`

```
import org.jbox2d.common.*;
import org.jbox2d.dynamics.*;
import org.jbox2d.collision.*;
import org.jbox2d.collision.shapes.*;
import org.jbox2d.dynamics.contacts.*;
import org.jbox2d.callbacks.*;
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.io.*;
import fr.atis_lab.physicalworld.*;

/*
 * javac -cp ./lib/*:. TestXX.java
 *
 * java -cp ./lib/*:. TestXX
 */

public class Test04 implements ContactListener, Serializable {

    /* PhysicalWorld => contains the World and walls (as PhysicalObject) */
    private PhysicalWorld world;
    /* PhysicalObject => contains the Body, Shape & Fixture */
    //private PhysicalObject ball, ramp, door;
    private Body ball, ramp, door;
    /* Custom Panel for drawing purpose */
    private DrawingPanel panel;

    public Test04() {

        /* Allocation of the PhysicalWorld (gravity, dimensions, color of the walls */
        world = new PhysicalWorld(new Vec2(0, -9.81f), -48, 48, 0, 64, Color.RED);
        /* Add ContactListener to the PhysicalWorld */
        world.setContactListener(this);

        try {
```

```

/* Allocation of the ball : radius of 3, position (0, 10), yellow, with an Image */
/* PhysicalObject are automatically added to the PhysicalWorld */
ball = world.addCircularObject(3f, BodyType.DYNAMIC, new Vec2(0, 10), 0, new Sprite("
    ball", 1, Color.YELLOW, new ImageIcon("./img/emosmile.png")));
/* Changing the restitution parameter of the PhysicalObject */
ball.getFixtureList().setRestitution(0.4f);

/* Complex polygon should be set as a list of points in COUNTERCLOCKWISE order */
/* Here, we want a simple triangle. */
/* Note that the (0,0) coordinate correspond to the rotation center */
LinkedList<Vec2> vertices = new LinkedList<Vec2>();
vertices.add(new Vec2(0,0));
vertices.add(new Vec2(10,0));
vertices.add(new Vec2(0,5));
/* We create a PhysicalObject based on this polygon. It is a static object in (-5, 0),
    green with an image */
/* In case of invalid polygon (less than 3 points, or too much) an
    InvalidPolygonException is raised */
try {
    ramp = world.addPolygonalObject(vertices, BodyType.STATIC, new Vec2(-5, 0), 0, new
        Sprite("ramp", 0, Color.GREEN, new ImageIcon("./img/triangle.png")));
} catch (InvalidPolygonException ex) {
    System.err.println(ex.getMessage());
    System.exit(-1);
}

/* Simple rectangle, the Door is a sensor (the other object can go through, but are
    detected) */
door = world.addRectangularObject(1f, 6f, BodyType.STATIC, new Vec2(14, 3), 0, new
    Sprite("door", 2, Color.BLUE, null));
door.getFixtureList().setSensor(true);

} catch (InvalidSpriteNameException ex) {
    ex.printStackTrace();
    System.exit(-1);
}

/* Allocation of the drawing panel of size 640x480 and of scale x10 */
/* The DrawingPanel is a window on the world and can be of a different size than the
    world. (with scale x10, the world is currently 960x640) */
/* The DrawingPanel panel is centered around the camera position (0,0) by default */
this.panel = new DrawingPanel(world, new Dimension(640,480), 10f);
/* Setting the color of the background */
this.panel.setBackground(Color.BLACK);
/* Setting an image as background */
this.panel.setBackgroundIcon(new ImageIcon("./img/paysage.png"));

/* Wrapping JFrame */
JFrame frame = new JFrame("JBox2D GUI");
frame.setMinimumSize(this.panel.getPreferredSize());
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setLayout(new BorderLayout());
frame.add(this.panel, BorderLayout.CENTER); // Add DrawingPanel Panel to the frame
frame.pack();
frame.setVisible(true);
}

/*
 * If we put the simulation loop in the constructor, it would be played before showing the
    GUI
 * So we put it in an external loop

```

```
*/
public void run() {
    try {
        float timeStep = 1/60.0f; // Each turn, we advance of 1/60 of second
        int nbTurn = Math.round(5f/timeStep); // 5 second of simulation => 300 turn
        int msSleep = Math.round(1000*timeStep); // timeStep in milliseconds
        world.setTimeStep(timeStep); // Set the timeStep of the PhysicalWorld

        /* Launch the simulation */
        for(int i = 0 ; i < nbTurn ; i++) { // 300 turn = 5 seconds
            world.step(); // Move all objects
            panel.setCameraPosition(ball.getPosition().add(new Vec2(0,20))); // The camera
                will follow the ball
            Thread.sleep(msSleep); // Synchronize the simulation with real time
            this.panel.updateUI(); // Update graphical interface
        }
        System.out.println(world);
    } catch (InterruptedException ex) {
        System.err.println(ex.getMessage());
    }
}

/* Event when object are touching */
public void beginContact(Contact contact) {
    System.out.println("Objects are touching "+Sprite.extractSprite(contact.getFixtureA()).
        getBody().getName() +" "+Sprite.extractSprite(contact.getFixtureB().getBody()).
        getName() );
}

/* Event when object are leaving */
public void endContact(Contact contact) {
    System.out.println("Objects are leaving "+Sprite.extractSprite(contact.getFixtureA()).
        getBody().getName() +" "+Sprite.extractSprite(contact.getFixtureB().getBody()).
        getName() );
}

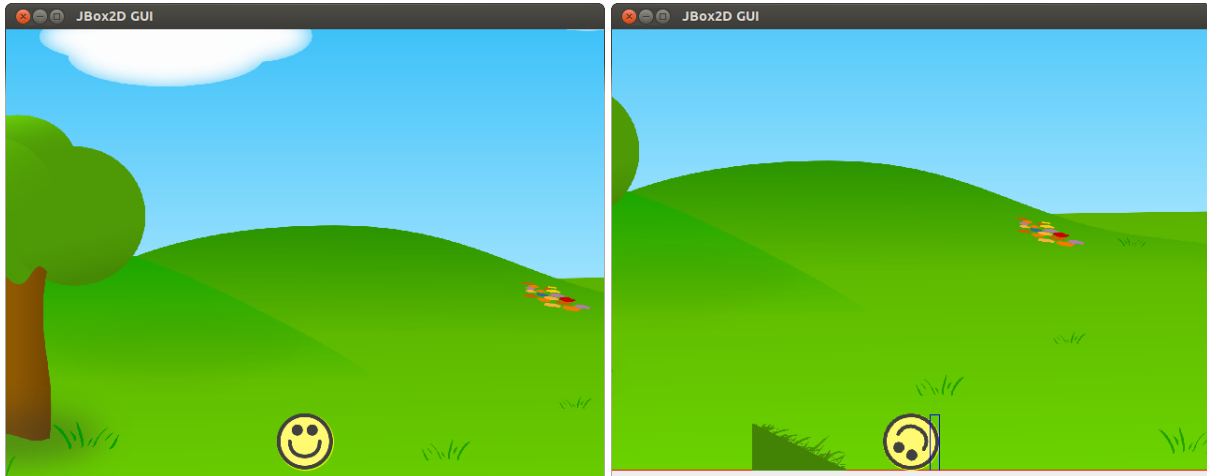
/* Advanced stuff */
public void postSolve(Contact contact, ContactImpulse impulse) {}
public void preSolve(Contact contact, Manifold oldManifold) {}

public static void main(String[] args) {
    new Test04().run();
}
}
```

EXERCICE 5



Recopiez le code précédent dans un fichier `Test04.java`, compilez-le puis exécutez-le.



Collision entre la balle et la porte Notez comment la balle traverse la porte, mais comment la collision avec celle-ci est bien détectée.

```
Objects are touching Ramp Ball
Objects are leaving Ramp Ball
Objects are touching Ramp Ball
Objects are leaving Ramp Ball
Objects are touching Ramp Ball
Objects are leaving Ramp Ball
Objects are touching Ramp Ball
Objects are touching bottomWall Ball
Objects are leaving bottomWall Ball
Objects are leaving Ramp Ball
Objects are touching bottomWall Ball
Objects are leaving bottomWall Ball
Objects are touching bottomWall Ball
Objects are touching Door Ball
Objects are leaving Door Ball
Time : 5 s
- door (14, 0) 0°
- ramp (-5, 0) 0°
- ball (18,67, 3,1) -338,71°
- rightWall (48, 0) 0°
- leftWall (-48, 0) 0°
- topWall (-48, 64) 0°
- bottomWall (-48, 0) 0°
```

Etape 5 : Quelques pistes de travail

5.1 Sauvegarder l'état du monde

Considérons une classe `MyGame` contenant une boucle de jeu, une instance de `PhysicalWorld` et des références sur des `Body` de `JBox2D`.

Les références sur les `Body` sont là pour nous permettre d'interagir plus facilement avec nos objets.

Néanmoins, si l'on désire sauvegarder l'état de cette classe par le biais de la classe `Serializer` vu au TD3.

```
MyGame myGame = new MyGame();
try {
    Serializer.saveToFile("MyGame.bak",myGame);
}catch(IOException ex) {
    ex.printStackTrace();
    System.exit(-1);
}
```

On obtient le message suivant :

```
java.io.NotSerializableException: org.jbox2d.dynamics.Body
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1183)
  at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1547)
  at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1508)
  at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1431)
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1177)
  at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:347)
  at fr.atlas_lab.physicalworld.Serializer.saveToFile(Serializer.java:27)
```

qui nous informe que la classe Body (comme tous les classes de JBox2D) n'implémente pas l'interface Serializable !

Il faut alors déclarer nos champs **transient**, c'est-à-dire invisible du point de vu de la sérialisation. Et il nous faut rajouter les méthodes suivante à notre classe :

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException
```

Pour un exemple complet, regardez le code source de la classe PhysicalWorld.

Dans notre cas, nous ne voulons pas vraiment sauvegarder ces variables qui sont **juste des références** sur des objets existants au sein du PhysicalWorld. Au contraire, nous risquerions de créer des objets doublons. Nous n'écrirons donc pas la méthode writeObject.

En revanche il est important de recréer les références vers nos objets une fois le monde rechargé depuis le fichier. C'est ce que nous faisons dans la méthode readObject :



ContactListener Le lien entre l'instance de World et la classe implémentant le ContactListener à aussi de forte chance d'être perdu, il faut donc le réinitialiser aussi.

```
/* import ... */
public class MyGame implements ContactListener, Serializable {

    private PhysicalWorld world;
    private DrawingPanel panel;

    /* Temporary reference to the objects */
    private transient Body ball, ramp, door;

    private JFrame frame;

    public MyGame() {

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

    private void readObject(java.io.ObjectInputStream in) throws IOException,
        ClassNotFoundException {
        // Read all non-transient fields => Load 99% of the instance state
        in.defaultReadObject();
        // Relink the current instance with the JBox2D World
        world.setContactListener(this);
        // Relink the objects reference with the JBox2D Bodies
        try {
            ball = world.getObject("ball");
            ramp = world.getObject("ramp");
            door = world.getObject("door");
        } catch (ObjectNameNotFoundException ex) {
            ex.printStackTrace();
            System.exit(-1);
        }
    }
}
```

}

5.2 Téléporter ou détruire un objet en réaction à une collision

Rapidement vous voudrez changer arbitrairement la position ou l'orientation d'un objet, voir le détruire en réaction à un événement du jeu (généralement une collision).



Monde verrouillé durant la résolution de l'étape de simulation Durant la résolution de l'étape de simulation (et par extension lors des événements de collisions) il est interdit de modifier l'état interne des objets pour éviter une perte de cohérence.

Il faut donc utiliser un système de drapeau (**boolean** needToTeleportBall) ou de liste (LinkedList<Body> toBeDestroyed) qui seront activés lors de la détection des collisions puis appliqués dans la boucle principale.

Exemple :

```
public class Test08 implements ContactListener, Serializable {

    /* PhysicalWorld => contains the World and walls (as PhysicalObject) */
    private PhysicalWorld world;
    /* PhysicalObject => contains the Body, Shape & Fixture */
    //private PhysicalObject ball, ramp, door;
    transient private Body ball, ramp, door;
    /* Custom Panel for drawing purpose */
    private DrawingPanel panel;

    /* Logic Flag */
    private boolean needToTeleportBall = false;

    public Test08() {

        /* ... */
    }

    /*
     * Simulation loop
     */
    public void run() {
        try {
            float timeStep = 1/60.0f; // Each turn, we advance of 1/60 of second
            int nbTurn = Math.round(5f/timeStep); // 5 second of simulation => 300 turn
            int msSleep = Math.round(1000*timeStep); // timeStep in milliseconds
            world.setTimeStep(timeStep); // Set the timeStep of the PhysicalWorld

            /* Launch the simulation */
            for(int i = 0 ; i < nbTurn ; i++) { // 300 turn = 5 seconds
                world.step(); // Move all objects

                /* Apply the game logic AFTER the simulation step */
                if(needToTeleportBall) {
                    // Change the position and orientation of the object
                    ball.setTransform(new Vec2(0,10), 0);
                    // Change the velocity of the object
                    ball.setLinearVelocity(new Vec2(0,0));
                    needToTeleportBall = false;
                }
                panel.setCameraPosition(ball.getPosition().add(new Vec2(0,20))); // The camera
                    will follow the ball
                Thread.sleep(msSleep); // Synchronize the simulation with real time
                this.panel.updateUI(); // Update graphical interface
            }
            System.out.println(world);
        }
    }
}
```

```

    } catch (InterruptedException ex) {
        System.err.println(ex.getMessage());
    }
}

/* Event when object are touching */
public void beginContact(Contact contact) {
    System.out.println("Objects are touching "+Sprite.extractSprite(contact.getFixtureA().
        getBody()).getName() +" "+Sprite.extractSprite(contact.getFixtureB().getBody()).
        getName() );

    /* Detect game logic */
    String nameA = Sprite.extractSprite(contact.getFixtureA().getBody()).getName();
    String nameB = Sprite.extractSprite(contact.getFixtureB().getBody()).getName();
    if(nameA.equals("door") && nameB.equals("ball")) {
        needToTeleportBall=true;
    }
}
}

```

5.3 Réagir au clavier

Pour récupérer les touches clavier on utilise un gestionnaire d'événement de type `KeyListener` (associé à votre `JPanel` ou `DrawingPanel`)



Focus Pour qu'un `JPanel` détecte les événements clavier, il faut que ce dernier ai le focus. C'est pour-quoi il est essentiel de le demander après avoir afficher la fenêtre.

```

/* ... */
frame.setVisible(true);
/* Request focus for panel MUST BE LAST LINE*/
this.panel.requestFocus();

```

Exemple :

```

public class Test09 implements KeyListener, Serializable {

    /* PhysicalWorld => contains the World and walls (as PhysicalObject) */
    private PhysicalWorld world;
    /* PhysicalObject => contains the Body, Shape & Fixture */
    //private PhysicalObject ball, ramp, door;
    transient private Body ball, ramp, door;
    /* Custom Panel for drawing purpose */
    private DrawingPanel panel;

    public Test09() {

        /* ... */
        /* Changing the friction parameter of the PhysicalObject */
        ball.getFixtureList().setFriction(10f);

        /* ... */
        /* Allocation of the drawing panel of size 640x480 and of scale x10 */
        /* The DrawingPanel is a window on the world and can be of a different size than the
           world. (with scale x10, the world is currently 960x640) */
        /* The DrawingPanel panel is centered around the camera position (0,0) by default */
        this.panel = new DrawingPanel(world, new Dimension(640,480), 10f);
        /* Setting the color of the background */
        this.panel.setBackground(Color.BLACK);
        /* Setting an image as background */
        this.panel.setBackgroundIcon(new ImageIcon("./img/paysage.png"));

        /* Add KeyListener */
    }
}

```



```
this.panel.addKeyListener(this);

/* Wrapping JFrame */
JFrame frame = new JFrame("JBox2D GUI");
frame.setMinimumSize(this.panel.getPreferredSize());
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setLayout(new BorderLayout());
frame.add(this.panel, BorderLayout.CENTER); // Add DrawingPanel Panel to the frame
frame.pack();
frame.setVisible(true);
/* Request focus for panel MUST BE LAST LINE*/
this.panel.requestFocus();
}

/*
 * When a key is pressed
 * CTRL+Q : Quit the program
 * UP : Jump
 * LEFT : Roll left
 * RIGHT : Roll right
 */
public void keyPressed(KeyEvent e) {
    System.out.println("keyPressed "+ e.getKeyCode());
    boolean ctrl_pressed = false;
    if ((KeyEvent.CTRL_MASK & e.getModifiers()) != 0) { // Detect the CTRL modifier
        ctrl_pressed = true;
    }
    switch (e.getKeyCode()) {
        case KeyEvent.VK_Q:
            if(ctrl_pressed) { // If Q AND CTRL are both pressed
                System.out.println("Bye bye");
                System.exit(-1);
            }
            break;
        case KeyEvent.VK_UP:
            if(ball.getContactList() != null) { // If the ball is touching something
                ball.applyForceToCenter(new Vec2(0,3000)); // Apply a vertical force of 3000 Newton to the ball
            }
            break;
        case KeyEvent.VK_RIGHT:
            ball.setAngularVelocity(-20); // Directly set the angular velocity
            break;
        case KeyEvent.VK_LEFT:
            ball.setAngularVelocity(20); // Directly set the angular velocity
            break;
    }
}

public void keyTyped(KeyEvent e) {
}

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

5.4 Position relative à d'autres objets

Si vous voulez positionner une objet par rapport à un autre ou projeter un élément dans l'alignement d'un autre, il faut utiliser les méthodes de la classe Body :

```
// Get the world coordinates of a point given the local coordinates.
public Vec2 getWorldPoint(Vec2 localPoint)
```

```
// Get the world coordinates of a vector given the local coordinates.
public Vec2 getWorldVector(Vec2 localVector)
```



Collisions à la création Lorsque vous avez un couple tireur/tiré, faite attention à ne pas créer l'objet tiré à l'intérieur de l'objet tireur à moins que celui-ci ne soit à capteur (sensor) !

Exemple :

```
/* import ... */
public class Test10 implements Serializable {

    /* PhysicalWorld => contains the World and walls (as PhysicalObject) */
    private PhysicalWorld world;
    /* PhysicalObject => contains the Body, Shape & Fixture */
    //private PhysicalObject ball, ramp, door;
    transient private Body canon, ball1, ball2;
    /* Custom Panel for drawing purpose */
    private DrawingPanel panel;

    public Test10() {

        /* Allocation of the PhysicalWorld (gravity, dimensions, color of the walls */
        world = new PhysicalWorld(new Vec2(0,-9.81f), -42.5f, 42.5f, 0, 64, Color.RED);

        try {

            /* The canon is a static rectangular shape that will rotate and throw bullet from both
               sides */
            canon = world.addRectangularObject(2f, 10f, BodyType.STATIC, new Vec2(0, 22),
                MathUtils.PI/4, new Sprite("canon", 0, Color.BLACK, null));

            /* The balls are simple static circular sensors that will follow each ends of the
               rectangle */
            ball1 = world.addCircularObject(1f, BodyType.STATIC, canon.getWorldPoint(new Vec2(0,5)
                ), 0, new Sprite("ball1", 1, Color.RED, null));
            ball1.getFixtureList().setSensor(true);
            ball2 = world.addCircularObject(1f, BodyType.STATIC, canon.getWorldPoint(new Vec2
                (0,-5)), 0, new Sprite("ball2", 1, Color.GREEN, null));
            ball2.getFixtureList().setSensor(true);

        } catch (InvalidSpriteNameException ex) {
            ex.printStackTrace();
            System.exit(-1);
        }

        /* Allocation of the drawing panel of size 640x480 and of scale x10 */
        /* The DrawingPanel is a window on the world and can be of a different size than the
           world. (with scale x10, the world is currently 960x640) */
        /* The DrawingPanel panel is centered around the camera position (0,0) by default */
        this.panel = new DrawingPanel(world, new Dimension(640,480), 7.5f);
        /* Setting the color of the background */
        this.panel.setBackground(Color.BLACK);
        /* Setting an image as background */
        this.panel.setBackgroundIcon(new ImageIcon("./img/paysage.png"));
        this.panel.setCameraPosition(new Vec2(0,32));

        /* Wrapping JFrame */
        JFrame frame = new JFrame("JBox2D GUI");
        frame.setMinimumSize(this.panel.getPreferredSize());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());
        frame.add(this.panel, BorderLayout.CENTER); // Add DrawingPanel Panel to the frame
```

```

        frame.pack();
        frame.setVisible(true);
    }

    /*
     * Simulation loop
     */
    public void run() {
        try {
            float timeStep = 1/30.0f; // Each turn, we advance of 1/60 of second
            int msSleep = Math.round(1000*timeStep); // timeStep in milliseconds
            world.setTimeStep(timeStep); // Set the timeStep of the PhysicalWorld
            int i = 0;

            /* Launch the simulation */
            while(true) { // Infinite loop

                // Manually rotate the static canon
                // You can adjust speed as you want :)
                canon.setTransform(canon.getPosition(), canon.getAngle()+0.01f);

                // Manually translate the balls at the ends of the rectangle
                // As the rectangle is of size 2x10 and that the origin is centered (1,5)
                // the extrema are (0, 5) and (0, -5) in the object referential
                // To have their position in the world referential, we use the
                // getWorldPoint method.
                ball1.setTransform(canon.getWorldPoint(new Vec2(0,5)),0);
                ball2.setTransform(canon.getWorldPoint(new Vec2(0,-5)),0);

                // To avoid to outrun the processor, we will throw a ball every 4 step (you can
                // change this)
                if(world.getStepIdx()%4==0) {

                    // Create two bullets
                    try {
                        // First bullet (watchout for the unique name condition)
                        // We place the bullet at the extrema of the canon (but outside the canon, to
                        // avoid collision !)
                        Body bullet = world.addCircularObject(1f, BodyType.DYNAMIC, canon.getWorldPoint(
                            new Vec2(0,6)) , 0, new Sprite("bullet"+(i++), 1, Color.YELLOW, null));
                        // We set the speed of the bullet to be 30, colinear to the canon
                        // Colinearity can be seen as the vertical in the object referential
                        // So we use the method getWorldVector to have the vector in the World
                        // referential
                        bullet.setLinearVelocity(canon.getWorldVector(new Vec2(0,30)));
                        // Second bullet, from the other side, with inverted speed
                        bullet = world.addCircularObject(1f, BodyType.DYNAMIC, canon.getWorldPoint(new Vec2
                            (0,-6)) , 0, new Sprite("bullet"+(i++), 1, Color.BLUE, null));
                        bullet.setLinearVelocity(canon.getWorldVector(new Vec2(0,-30)));

                        } catch (InvalidSpriteNameException ex) {
                            System.err.println(ex.getMessage());
                            System.exit(-1);
                        }

                    world.step(); // Move all objects
                    Thread.sleep(msSleep); // Synchronize the simulation with real time
                    this.panel.updateUI(); // Update graphical interface
                }
            } catch (InterruptedException ex) {
                System.err.println(ex.getMessage());
            }
        }
    }

```

```
    }  
}  
  
public static void main(String[] args) {  
    new Test10().run();  
}  
}
```



Have fun !