Emini Physics Engine

Version 1.3.4

Alexander Adensamer

Table of Contents

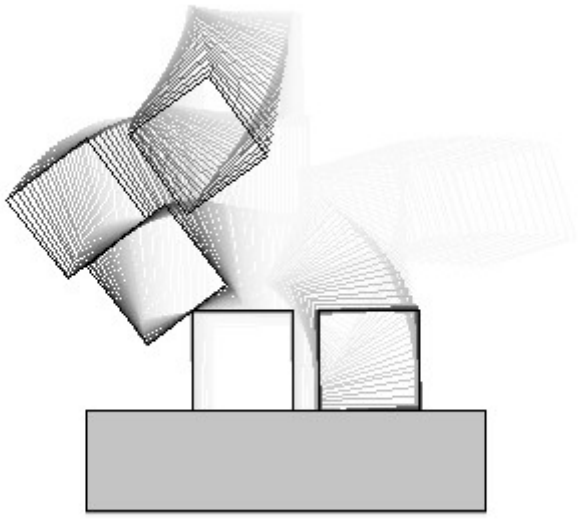
Introduction

Physics simulations is becoming more and more important for applications – in particular games. Physics engines are successful because they are intuitive and realistic. Everybody knows daily life physics and understands how physical objects interact.
In a time when developers strives to create natural and intuitive interfaces, physical effects can make the difference.

The Emini Physics Engine is a 2D rigid body physics library. It is intended to be used as a physics engine in mobile applications. It can be used to create realistic physical simulations. The physical behavior produced by the engine is a simulation of real life physics.

The physics engine covers collision detection and resolution, customizable shapes, friction, elasticity (bounce), gravity, pin joints, springs and motors.

The engine can be used for J2ME and Android. It is available as Fixpoint Math version for devices without Floatingpoint Processing Unit and as a true floating point version.



Physics background

The physics of a rigid body system are described by Newton's laws. These govern the motion of the bodies within the system. In short, these are conservation of momentum, conservation of energy and Actio-Reactio. Conservation of energy ensures that a pendulum released at a given position will reach the same height on the other side.
Conservation of momentum ensures that the momentum (mass times speed) is conserved when two objects collide. This is important when a heavy object hits a light object or vice versa.
Actio-Reactio governs object interactions like collisions. Each action causes a reaction.

However, it is unpractical to solve Newtons equations directly. This would be much too costly in terms of performance. Instead the body interactions are formulated as constraints. Each constraint can be solved directly. Using an iterative algorithm the system of all constraints can be simulated efficiently.

Basic concepts of the Physics Engine

Definitions of commonly used terms

Body – The body is the basic element of a physical simulation. It represents a physical rigid body. It is represented by its physical properties (shape, mass, friction, etc) and its state (position, velocity, etc).

Shape – Each body has a shape that defines the boundaries and physical properties of the object. The same shape can be used for several objects. Shapes are always convex polygons or circles.

Constraints – Constraints modify the behavior of bodies. The system has to comply to all constraints imposed onto it. The most important constraint is non-penetration of objects. Other examples range from pin joints to motors.

World – This term is used throughout the document for the complete physical system. This contains all bodies, constraints, external forces and simulation parameters.

The physical simulation

The real world physics happens in continuous time. The simulation in contrast runs in discrete time steps. An elementary time step is applied for the equations of motion. That implies that events happening between two time steps (most event will not exactly fall on one time step) cannot be measured precisely. This is a restriction common to all physical simulations.
The simulation is constraint based. In particular collisions are resolved by applying constraints to the bodies.

Basic Concepts

The basic concepts are important to understanding the simulation and creating a world. Most of these concepts are represented as classes in the code. The usage is explained in the java docs. The sample code explained in the last section shows this directly.

Shapes

Shapes represent the boundaries and physical properties of the bodies. Each shape represents either a convex polygon or a circle. A convex polygon consists of a line segments so that no recess exists.

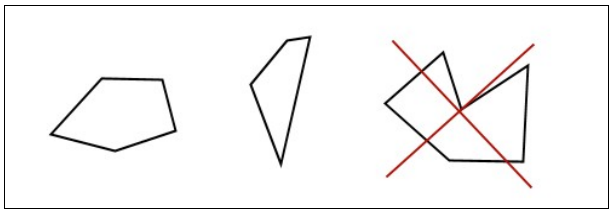


Figure 1: Convex/Non-convex Polygons

The polygon is stored in an array of Vertices (2D vector). To create a shape several convenience factory methods exist.

- Circle
- Rectange
- Regular Polgyon

For further details see the javadoc. The maximum vertex count is 12. For any other shape the

corners have to be supplied directly.

With Engine version 1.3.0 convex shapes are possible as well. A convex shape is created by merging multiple (convex) shapes to a Multi-shape. The resulting shape can be concave. Note that merged shapes affect the performance and should be used only if absolutely necessary.

Important properties for physical objects are the friction factor, elasticity and density. These are stored in the shape object as well. See further details on these properties below.

Friction

The friction factor (between 0 and 1) controls the frictional forces in the system. Frictional forces act on body contacts in tangential direction (sliding). The amount of the force is determined by combining the friction factors of the two concerned bodies. The product of the factors f1 * f2 gives the effective friction for the contact. Two objects with a friction factor of 0 will not feel any frictional force. Two objects with a friction of 1 will feel the maximum frictional force. That means that all translational movement is eliminated. The resulting force also depend on the translational velocity of a contact and the force acting on it.

Elasticity

The elasticity factor (between 0 and 1) governs the impact behavior of two bodies. Similar to the friction, the product of the factors of both involved bodies is used $e_1 \cdot e_2$. An elasticity factor below 1 causes energy to be lost (in the real world it is transformed into deformation energy or heat inside the bodies). For example a bouncing ball will loose height after each consecutive bounce.

Density

The density is an important physical property of each body. The density represents the mass of the body and the inertia. It is however simpler to work with masses, so the interface of the shape class uses mass instead of density. The corresponding inertia is calculated internally and is used for the algorithms as the mass.

Bodies

All objects in the physical environment are represented by bodies.
A body is defined by its physical properties (shape) and the physical state – position, angle and the respective velocities. Bodies can be static or dynamic. Dynamic bodies move according to physical laws.
Static bodies always remain in their original state and do not move. External forces do not work on them, so they are not affected by gravity. The complete impulse (depending on the elasticity) of an impact is reflected. These can be used for landscaping.

Gravity

From daily experiences we are used to see gravity as the force that accelerates all bodies downwards. This is also the default setting for the physics engine. However, gravity can be set directly on the world object and can point in any direction. This can be done even in mid simulation. However, bodies should be allowed to move consistently with respect to each other, so the gravity should only be set between the ticks (see The physical simulation).
An important usage of explicit gravity modification is to set the gravity to zero. This way a 2D system from bird eyes view can be simulated (e.g: Billiard game). The friction between the bodies and the simulation plane can be modeled by using the damping factor.

External Forces

All forces that do not originate the system itself are called external forces and put work (thus energy) into the system. Gravity is the most common external force and thus directly included into the engine. By implementing the External Force interface any additional force can be added easily.

Collisions

During the collision detection step of the simulation all collisions between bodies are calculated. Each contact of a pair of colliding bodies is represented by a contact. The contact can be single if a corner touches a face or double if two faces of the bodies touch. During collision resolution these contacts are treated according to the physical laws in the same manner as the constraints.

In some situations collisions between particular bodies are not desired. For example bodies that are joined together by a pin joint might be allowed to move through each other; one in front of the other. This behavior can be achieved by using the collision bitflags. There are 32 levels for bodies, where no collisions happen. Whenever two bodies that would collide otherwise have the same level activated, they do not interact. Per default a newly created body has no collision level activated (bitflag = 0) so that is interacts with any other body.

Constraints

A constraint represents a force (or pair of forces) in the environment, that constrains the unhindered movement of bodies. The constraint is applied to the velocities of the bodies. Examples of elements that can exert a constraint force on the bodies are springs, joints or motors.

By implementing the interface constraint any type of constraint can be created and used in the simulation. Common constraints like joints, springs and motors are supplied by the library.

Pin Joints

Pin joints attach two bodies together. The joint position of these bodies is fixed relative to each other. The pin joint can be loose or fixed. A loose joint allows relative rotation. A fixed pin joint allows absolutely no relative movement of the involved bodies.

In the case of loose pin joints there should be no collisions between the two involved objects. Otherwise there would be conflicting constraints – the joint keeping the bodies together and the contact pushing them apart. This leads to highly unstable behavior. This problem is solved by setting collision flags to avoid collisions of certain objects. Further details see Collisions.

Springs

Springs are constraints that link two bodies together. A spring is fixed in any position for both bodies (relative to the center) and has a default distance. This is usually the distance that the two objects (precisely: the two anchor positions) have at their creation. The bodies can rotate freely around the anchor position of the spring.

When no spring coefficient is set the constraint acts as a rigid connection, that keeps the exact distance of the pivotal points. If the spring coefficient is set, the constraint acts as spring according to the spring law. The acting force is proportional to the difference of default length and current length of the spring.

Motors

A motor is a constraint that acts on a single body. The body is kept at a designated velocity. Each motor also has a maximum force that it can exert to reach the target velocity. If the force is to small in the given circumstances, the target velocity will not be reached.
A motor can be linear or rotational. If it is linear and one component (x or y axis) is zero, the constraint is not applied to that direction.
The rotational motor applies to the rotational velocity.

Events

The physics engine has a callback mechanism for special events that can occur during the simulation. When an even tlistener is registered on the world object, the listener will be informed about each event.

There are several types of events:

- Body events
    - Body collisions: triggered when the body collides with another body
    - Body position (or angle): triggered when the body is in a given range
    - Body velocity (or angular velocity): triggered when the body moves at a given velocity
- Constraint events: triggered when the force exerted by a constraint lies in a given range.

The events can apply to all bodies of constraints respectively or only to a filtered set (possibly a single body or constraint). These filter criteria have to be fixed when creating the event and adding it to the world.

Particles

The engine covers also the inclusion of particles systems. Particles originate at an emitter, which can be placed in the simulation. The particles move according to their physical settings and sourrounding. They bounce off walls and bodies, but do not interact with each other or exert any force on bodies.

Each particle emitter controls the behaviour of its particle using various parameters. These include how the particle react to gravity or bounce and also the emit parameters like creation rate, particle life time and initial speed. The emitter can be fixed in space or be applied to a body, moving with the body.

Working with the Physics Engine

This section explains how these concepts can be used in an application. For further details and examples see Step by step examples.

Creating a world

The creation of a world consists of several steps. First you should have a clear idea of what you want to create.

1. Create an empty world object.
2. Create all shapes that will be required in the simulation.

3. Create all static bodies, that form the static environment. This can also include a Landscape (see Landscape). Add the bodies (or the landscape respectively) to the world.

4. Create the bodies using the shapes from step1 and place them according to their initial positions. Add the bodies to the world.

5. If joints, springs or motors are required, create them applying them to the already created bodies. Add the constraints to the world as well.

The world is now complete can can be simulated.

Running the simulation

Running the physical simulation is straight forward. Create a thread that calls tick() continuously on the world in the run method of the thread. You should wait between the ticks to ensure constant simulation speed.
The tick handles all physical laws. It performs a collision detection, resolves all collisions, applies all active constraints and applies the external forces.
The simulation can be changed at runtime. In particular bodies can be removed or added and simulation parameter changed. This must happen synchronously between the ticks, to ensure consistency of the simulation.

Advanced concepts

The advanced topics cover useful information to get best performance out of the physics engine. Additional information about the inner workings of the engine can be found here.

Special constructs

Non-convex bodies

The shapes of bodies can only be defined as circles or convex polygons (See Shapes). If for some reason non-convex bodies are required, these can be created by joining two or more bodies using fixed pin joints (see Pin Joints). If different forces act on the two bodies so that large stress is exerted on the pin, displacements of the bodies relative to each other can happen. Should this be the case consider using additional joints in different positions to cover various directions of stress.

Landscape

The landscape is similar to static bodies. It is used to model the border of the world and other fixed environment related obstacles. These can also be modeled by using static bodies, but it is usually more effective to make use of the optimization the landscape offers.
The landscape consists of many line segments, that can form an arbitrary shape. Each segment can also have a facing direction, which is used to obtain better consistency for collision detection. This is not required, but can have a good effect on the simulation in extreme cases.

Similar to a shape the landscape has fricion and elasticity factors that govern the interaction of bodies with the individual segments.

Physical and simulation topics

The stability of the physical simulation depends on several parameters and optimal modeling of the environment. Due to necessary approximations by the constraint solver some aspects have to be considered carefully. Stacking of multiple objects on top of each other is a typical candidate that causes problems in physical simulations. Also interactions of bodies with a high mass ratio (e.g: a very heavy body hitting a very light body or vice versa).

Units of space and time

The spatial unit used is pixel [px] and the temporal unit is seconds [s].

- Positions [px]
- Velocities [px/s]
- Accelerations [px/s^2]

The unit of mass is irrelevant, as bodies act according to mass ratios (a ratio has no unit).

Stacking

Stacking is a typical problem in physical simulations. When a large stack is constructed, the topmost body has only indirect contact to the solid floor. For this reason the upper parts of a stack can become unstable. The number of stacked bodies that remain stable is dependent on the shape. A good rule of thumb is that 10 approximately square bodies are still stable.

Mass ratio

Another typical problem in pyhsical simulations are mass ratios. Impulse and energy is transferred by an impact of two bodies. If a larger collision takes place that involves one or more intermediate objects, these objects have to transfer the energy and impulse. Due to simulation restrictions a light object cannot transfer as much energy and impulse as a heavy object.

In the sample on the right, not the complete energy and impulse can be conveyed from the left to the right heavy object if the mass ratio is too large.

This effect has to be taken in consideration when the world is designed. Elements like levers, see-saws or other interacting bodies should have sufficiently large mass.

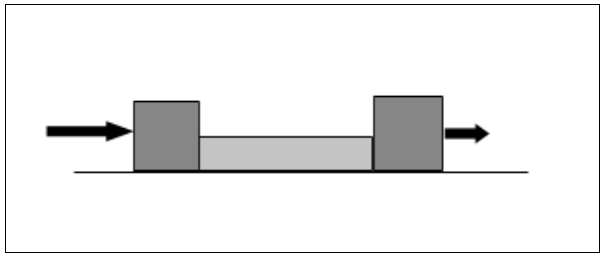


Figure 2: Mass Ratio: Dark objects are heavier

Fast body treatment

Very fast objects can move through other bodies due to the discretized time. In each step every body moves according to its velocity. Then the engine checks for collisions. A fast object could move through another body instead of colliding with it.

The engine has an automatic fast body detection that takes care of this problem. It is solved by checking for potential (future) collisions. The problemn can still happen if rotationg objects with long “arms” are involved.

Usually fast bodies are not desired by design anyway. So it is preferred to check why very fast movement can happen and try to remove the sources. One option is to increase the damping factor (see Damping).

Damping

In the real world every movement is dampened due to air resistance. Swinging around a joint is also dampened by friction in the axle. This effect can be simulated withe physics engine by setting the damping factor of the world. Sensible values are between 0 (no damping) and 0.05 (all moveemnt is dampened by 0.5% every tick (20 times per second).

Damping applies to tranlsation as well as rotation. Damping of both movement types can be set separately.

Overlap

Due to the discrete time steps bodies do not move continuously, but in discrete amounts. This can cause bodies to overlap instead of merely touch. This is normally only a fragment of a pixel and is not noticed. In some special cases this can be more than a pixel and it can be noticed.

The collision lookahead (since v1.3.0) makes sure that this happens only in very special situation. Mostly when fast rotating bodies are involved.

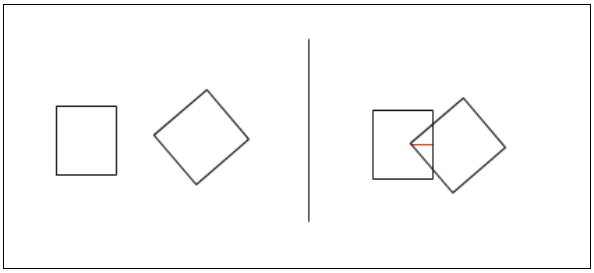


Figure 3: Overlap; left: timestep before collision; right: next timestep bodies overlap

Mathematics

The mathematics used by the physics engine are important for advanced usage and extensions.

Fixpoint arithmetics

For performance and compatibility reasons internally all values are stored as fixpoint values. In contrast to floating point decimals, these numbers use an integer to hold them, but sport fixed decimals. Here is a brief summary how to understand and work with fixpoint values:

For integers each bit represents a power of two starting with 1, 2, 4... ($2^0, 2^1, 2^2$,...). Fixpoint numbers are represented basically the same way. Each bit represents a power of two. However, fixpoint values do not start with 1, but with a negative power. Here we use 12 (DECIMAL) as maximum decimals.

1/4096, 1/2048, 1/1024, … ($2^{-12}, 2^{-11}, 2^{-10}$,...). This way we can represent decimal numbers up to a precision of 1/4096.

Integer representation:

8	4	2	1
0	1	1	0

= 8*0+4*1+2*1+1*0 = 6

Fixpoint representation

2	1	0.5	0.25
0	1	1	0

= 2*0+1*1+0.5*1+0.25*0 = 1.5

To convert a integer number to a fixpoint number the integer has to be shifted by DECIMAL to the left:

```
newFX = y << FXUtil.DECIMAL;
newFX = FXUtil.toFX(y);
new = yFX >> FXUtil.DECIMAL;
new = FXUtil.fromFX(yFX);
```

Addition and subtraction work exactly the same way as for integers:

```
addFX = xFX + yFX;
subFX = xFX - yFX;
```

The multiplication and division of fixpoint values is slightly more complex, as we have to consider that the shift in the decimals.

```
multFX = (int) (((long)aFX * (long) bFX) >> FXUtil.DECIMAL);
multFX = FXUtil.multFX(xFX, yFX);
divFX = (int) (((long)xFX << FXUtil.DECIMAL) / yFX);
divFX = FXUtil.divideFX(xFX, yFX);
```

Optimisations

For large and complex environments it is important to understand the fine tuning of parameters and concepts. This affects simulation quality and consistency as well as performance.

Constraint iterations

Each simulation step creates a situation with a number of contacts and possibly other constraints to be satisfied at the same time. This is represented by a large system of equations. It would be much too time consuming to solve these directly. So there is an constraint iteration that solves the equations iteratively. A iteration parameter governs the number of iterations. The setting of this parameter depends of typical situations that appear during the simulation. Most important for consideration of parameter choice are high stacks, which which require large iteration counts to transfer the energy of the upper elements to the bottom. Also important are situations where bodies with different mass and/or velocity collide.

There is also the option set the parameter to dynamic to allow the engine to determine how many iterations are required for each step.

Simulation Area

At times the simulation is a large area, but only a small section is visible on the screen. In this case a simulation area can be defined on the world, that restricts all physics activity to this area. While this has the disadvantage of halting movement outside the area (objects might stay in the air), it has the advantage of a large performance boost.
An optimal usage of this feature adjusts the window whenever the screen moves and allows some slack beyond the actual screen. This way the objects close to the screen can start to move correctly and be in a consistent state when the screen moves there.

Shape design

The collision detection can work with any kind of convex polygon (see Shapes). The basic costs of colliding two polygons are $O((n+m)^2)$, where *n* and *m* are the number of vertices of the shapes. These costs can be reduced when faces of the polygon are collinear. The costs reduce to $O((n+m)(\tilde{n}+\tilde{m}))$ where $\tilde{n}, \tilde{m}$ are the numbers of non-collinear faces. For rectangles, this reduces the collision effort by factor 2.

Joints placement

The exact placement of joints can increase the correctness of the simulation. If strong forces act on joints they can move apart. For fixed pin joints that fix two bodies together, the joint should be placed such that acting forces become minimal. This is usually close to the center of the body positions. It can makes also sense to place the joint, where it would be placed in the real world using the intuition we have for everyday physics.

An interesting fact in particular for non-fix pin joints is that joints can be placed outside the actual bodies. This can help to remove the need of extra bodies, that would be required as additional connection bars.

Restrictions

Keep in mind that the computer can only simulate real physics. The resulting body movements feel

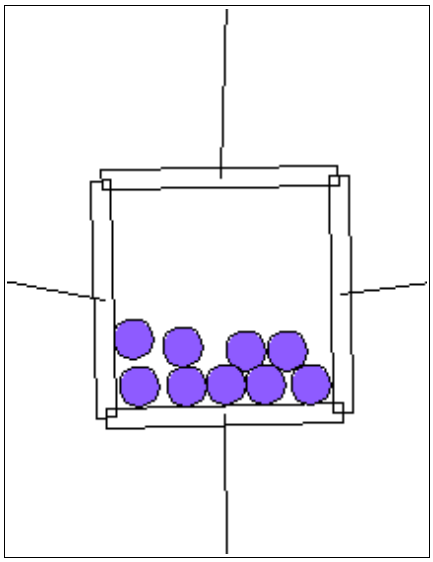
like physical behaviour, but are the results of a model that is close to the real physics. The practical implication is that in some border cases the model might behave differently than the real physics. Some possible cases are presented in the following:

- Mass ratios of bodies (see Mass ratio) can cause inconsistent behaviour of the lighter elements. In particular in multi-body situations, when energy or impulses have to carry from one body to another.
- Fast Bodies (see Fast body treatment). While fast moving bodies are detected and treated correctly, some situations remain, where this problem can resurface. This affects fast rotationg bodies with long “arms”, where collisions might be detected missed.
- Overlap: While we are used from real world physics that rigid bodies do not overlap, this does not happen automatically in the physics engine. This is not an implicit behavior due to some restrictions like data structure, but the result of explicit computations (see Overlap). Bodies do overlap by very small amounts at each step. The contacts make sure that velocities are adapted, so that the overlap disappears. You can never completely rely on the fact that bodies do not overlap.
- Performance issues can happen when the number of bodies becomes large obviously. However, more relevant to the performance is the number of contacts. If the bodies are all at the same spot, a lot of contacts exist between the bodies. Large groups of bodies should be avoided if possible.
- Stacking of objects becomes unstable at some point. The stability of a large stack depends on the properties of the involved bodies. A rule of thumb is that the default settings (20Hz, 10 iterations) allow stable stacking of 10-15 square boxes. This restriction is due to the fact that the constraint equations are not solved directly (which would be too time consuming), but iteratively and the constraints in the stack case are highly dependent of each other.

Step by step examples

This chapter explains the creation of some physical applications for J2ME. It is inteded as a tutorial for the usage of the engine.

Create an application - J2ME



This tutorial explains the creation of the shaker sample application. This covers the world function implementing, class extending and world creation.

The main element is a box containing nine balls. The box is supported by springs, so it stays in the air and has the tendency to move back to the center. The springs are mounted on fix walls outside the screen.

The box can get impulses in any direction using the arrow keys. While the box moves relatively slow and constant, the balls inside jump around when the direction of the box changes. Due to different weight distribution of the balls, the box can tilt. This effect wears off after some timebecause the springs move the box back into the original position.

We start with to create the midlet. Irrelevant code pieces are omitted. We want to create the world and run the simulation.

```
public class Shaker extends MIDlet
{
    private World shakerWorld;
    private Canvas simulationCanvas;

    public Shaker()
    {
        ...
        simulationCanvas = new Canvas();
        createShakerWorld();
    }

    public void startApp()
    {
        ...
    }

    public void destroyApp( boolean unconditional )
    {
        ...
    }

    private void createShakerWorld()
    {
    }
}
```

Now we create the world and set the parameters. The helper constants are here for better readability of the code.

```
    private void createShakerWorld()
    {
        shakerWorld = new World();
        shakerWorld.setDampingFX( (FXUtil.ONE_FX * 995) / 1000);

        final boolean isDynamic = true;
        final boolean isStatic = false;
        final boolean fixedJoint = true;
        final int calculateDistance = -1;

        int width = simulationCanvas.getWidth();
        int height = simulationCanvas.getHeight();
        int centerx = width / 2;
        int centery = height / 2;
    }
```

We define the shapes: The bars for the box, the circles and a small box, which is used as anchor object for the springs. It looks more interesting when the circles jump around a lot, so we set the elasticity to 80%.

```
    private void createShakerWorld()
    {
        ...
        Shape boxRectangle = Shape.createRectangle(width, 10);
        boxRectangle.setMass(10);
        Shape ball = Shape.createCircle(10);
        ball.setElasticity(80);

        Shape smallbox = Shape.createRectangle(2, 2);
    }
```

Now we create the actual bodies. The bars for the box, the circles and the anchor for the springs. The left and right bars have to be rotated by 90 degrees. The position and size of the anchor is irrelevant, because we can put the spring anywhere – even outside the actual body. Note that the bars are 10 times heavier than balls. This allows the box to move relatively constant, while the balls inside jump wildly. Also the box stays much more stable that way.

All bodies are added to the world.

```
    private void createShakerWorld()
    {
        ...
        Body upper = new Body(centerx, centery - width / 2,
                              boxRectangle, isDynamic);
        Body lower = new Body(centerx, centery + width / 2,
                              boxRectangle, isDynamic);
        Body left  = new Body(centerx - width / 2, centery,
                              boxRectangle, isDynamic);
        left.setRotationDeg(90);
        Body right = new Body(centerx + width / 2, centery,
                              boxRectangle, isDynamic);
        right.setRotationDeg(90);

        shakerWorld.addBody(upper);
        shakerWorld.addBody(lower);
        shakerWorld.addBody(left);
        shakerWorld.addBody(right);
```

```
        for( int i = -1; i <= 1; i++)
            for( int j = -1; j <= 1; j++)
                world.addBody(new Body(centerx + i * 20,
                                       centery + j * 20,
                                       ball, isDynamic));

        Body anchor = new Body(0, 0, smallbox, isStatic);
        shakerWorld.addBody(anchor);

    }
```

We have four bars, but we need to put nails into them to build a box. The fixed pin joints will serve as nails. We also need to adjust the collision bitflags to allow the barx of the box some slight overlap.

```
    private void createShakerWorld()
    {
        ...
        Joint upperLeft  = new Joint(upper, left,
                                     FXVector.newVector(- width / 2, 0),
                                     FXVector.newVector(- width / 2, 0),
                                     fixedJoint);
        Joint upperRight = new Joint(upper, right,
                                     FXVector.newVector(  width / 2, 0),
                                     FXVector.newVector(- width / 2, 0),
                                     fixedJoint);
        Joint lowerLeft  = new Joint(lower, left,
                                     FXVector.newVector(- width / 2, 0),
                                     FXVector.newVector(  width / 2, 0),
                                     fixedJoint);
        Joint lowerRight = new Joint(lower, right,
                                     FXVector.newVector(  width / 2, 0),
                                     FXVector.newVector(  width / 2, 0),
                                     fixedJoint);

        shakerWorld.addConstraint(upperLeft);
        shakerWorld.addConstraint(upperRight);
        shakerWorld.addConstraint(lowerLeft);
        shakerWorld.addConstraint(lowerRight);

        upper.addCollisionLayer(1);
        lower.addCollisionLayer(1);
        left.addCollisionLayer(1);
        right.addCollisionLayer(1);
    }
```

The box is finished. Now we need to support it by applying springs that keep it floating in the center of the screen. The springs are fixed at the anchor body. However, the anchor position lies outside the body boundaries. While this may feel awkward at the first moment, this an efficient way of anchoring joints and springs.

```
    private void createShakerWorld()
    {
        ...
        Spring upperSpring = new Spring(anchor, upper,
                                        FXVector.newVector(centerx, 0),
                                        new FXVector(),
```

```
                                        calculateDistance);
        upperSpring.setCoefficient(120);
        Spring lowerSpring = new Spring(anchor, lower,
                                  FXVector.newVector(centerx, height),
                                        new FXVector(),
                                        calculateDistance);
        lowerSpring.setCoefficient(120);
        Spring rightSpring = new Spring(anchor, right,
                                        FXVector.newVector(0, centery),
                                        new FXVector(),
                                        calculateDistance);
        rightSpring.setCoefficient(100);
        Spring leftSpring  = new Spring(anchor, left,
                                   FXVector.newVector(width, centery),
                                        new FXVector(),
                                        calculateDistance);
        leftSpring.setCoefficient(100);

        shakerWorld.addConstraint(upperSpring);
        shakerWorld.addConstraint(lowerSpring);
        shakerWorld.addConstraint(leftSpring);
        shakerWorld.addConstraint(rightSpring);
    }
```

The world is now finished and can be used for the simulation. Now we need to create the thread that ticks the world (see Running the simulation). We extend the Canvas class to create a simulation canvas, so that the canvas handles the drawing as well as the simulation.

When the canvas is shown a new simulation thread is created and started, whenever it is hidden, the thread is removed. The canvas itself is made runnable, so everything happens there. We pass the canvas itself to the newly created thread.

```
public class SimulationCanvas extends Canvas implements Runnable
{
    private World world;
    private Thread thread;

    private boolean stopped = false;

    public SimulationCanvas()
    {
    }

    public void setWorld(World world)
    {
        this.world = world;
    }

    protected void showNotify()
    {
        thread = new Thread( this );
        thread.start();
    }

    protected void hideNotify()
    {
        world = null;
```

```
        thread = null;
    }

    public synchronized void end()
    {
        stopped = true;
    }

    public void run()
    {
    }
}
```

Now we can implement the actual simulation steps. That happens in the run method of the runnable. We create use constant (millis) for the amount of time for each tick. If the tick executed faster than this time, we wait for the remaining time until we execute the next tick.

```
public class SimulationCanvas extends Canvas implements Runnable
{
    private static final int millis = 50;

    ...

    public void run()
    {
        long sleep = 0;
        long start = 0;
        while(world != null && ! stopped)
        {
            start = System.currentTimeMillis();

            world.tick();

            sleep = millis - (System.currentTimeMillis() - start);
            sleep = Math.max(sleep, 0);

            try
            {
                Thread.sleep( sleep );
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
                throw(e);
            }
        }
    }
}
```

The simulation is running now, but we do not see anything. We need to extend the world class. This world will be able to draw itself to a graphics object. We need to draw the bodies as well as the springs. The springs are represented as simple lines. The bodies are defined by their shapes – rectangles and circles in our case.

We iterate over all bodies and paint each of them (the drawBody(...) methods follows directly below. Then we iterate over all constraints and check whether it is a spring. If so we paint the line.

```
public class GraphicsWorld extends World
{

   public GraphicsWorld()
   {
   }

   public void draw(Graphics g)
   {
       int bodyCount = getBodyCount();
       Body[] bodies = getBodies();
       int constraintCount = getConstraintCount();
       Constraint[] constraints = getConstraints();

       g.setColor(255, 255, 255);
       g.fillRect(0, 0, g.getClipWidth(), g.getClipHeight());

       //draw bodies
       g.setColor(0, 0, 0);
       for(int i = 0; i < bodyCount; i++)
       {
           drawBody( g, bodies[i] );
       }

       //draw springs
       g.setColor(0, 0, 0);
       for(int i = 0; i < constraintCount; i++)
       {
           if (constraints[i] instanceof Spring)
           {
               Spring spring = (Spring) constraints[i];
               g.drawLine(spring.getPoint1().xAsInt(),
                          spring.getPoint1().yAsInt(),
                          spring.getPoint2().xAsInt(),
                          spring.getPoint2().yAsInt() );
           }
       }
   }
}
```

We need to add the method to draw a single body. The vertex positions of the boundary of the body can be retrieved with `getVerticesFX()`. We check whether it is a circle by looking at the vertices size, which is a trick to avoid looking up the actual shape. Another way of dealing width shapes will be shown in the second example.

```
public class GraphicsWorld extends World
{
   ...
   public void drawBody(Graphics g, Body b)
   {
       g.setColor(0x000000);

       FXVector[] positions = b.getVerticesFX();
       if (positions.length == 1)
       {
           //draw a circle
           int radius = b.shape().getBoundingRadius();
```

```
            g.drawArc( b.positionFX().xAsInt() - radius,
                       b.positionFX().yAsInt() - radius,
                       radius * 2, radius * 2, 0, 360);
        }
        else
        {
            //draw the polygon
            for( int i = 0; i < positions.length - 1; i++)
            {
                g.drawLine(positions[i].xAsInt(),
                           positions[i].yAsInt(),
                           positions[i + 1].xAsInt(),
                           positions[i + 1].yAsInt());
            }

            //draw the final segment
            g.drawLine(positions[positions.length - 1].xAsInt(),
                       positions[positions.length - 1].yAsInt(),
                       positions[0].xAsInt(),
                       positions[0].yAsInt());
        }
    }
}
```

Now we want to use the new class as well. First in the Midlet itself.

```
public class Shaker extends MIDlet
{
    private GraphicsWorld shakerWorld;

    ...

    private void createShakerWorld()
    {
         shakerWorld = new GraphicsWorld();
         ...
    }
}
```

And we also use the new world class in the SimulationCanvas.

```
public class SimulationCanvas extends Canvas implements Runnable
{
    private GraphicsWorld world;

    public void setWorld(GraphicsWorld world)
    {
        this.world = world;
    }
    ...
}
```

Now we can call the painting method of the world from the thread loop.

```
public class SimulationCanvas extends Canvas implements Runnable
{
    ...
```

```
    public void run()
    {
        while(world != null && ! stopped)
        {
            ...
            world.tick();
            repaint();
            ...
        }
    }

    protected void paint(Graphics g)
    {
        world.draw(g);
    }
}
```

Finally we can change the midlet to use the new Canvas class. When the canvas becomes visible, the simulation will automatically start.

```
public class Shaker extends MIDlet
{
    private GraphicsWorld shakerWorld;
    private SimulationCanvas simulationCanvas;

    public Shaker()
    {
        ...
        simulationCanvas = new SimulationCanvas();
        createShakerWorld();
        simulationCanvas.setWorld(shakerWorld);

        getDisplay().setCurrent( simulationCanvas );
    }

    ...
}
```

We are almost done. The simulation starts and is drawn to the screen, but we still cannot interact. We will add that functionality to the SimulationCanvas class. We need to have references to the box elements. Whenever a key is pressed we apply forces (acceleration) to all these elements. The key determines in what direction the force is applied. *Note*: We could also apply the four times the force to a single box element. Due to the fact that they are fixed together, the constraint would ensure that the effective force will be distributed evenly among the bars. However, it is preferable to avoid unnecessary work for the simulation.

```
public class SimulationCanvas extends Canvas implements Runnable
{
    private final static int force = 64;
    private Vector boxElements = new Vector();
    ...

    public void addBoxElement(Body b)
    {
        boxElements.addElement(b);
    }
```

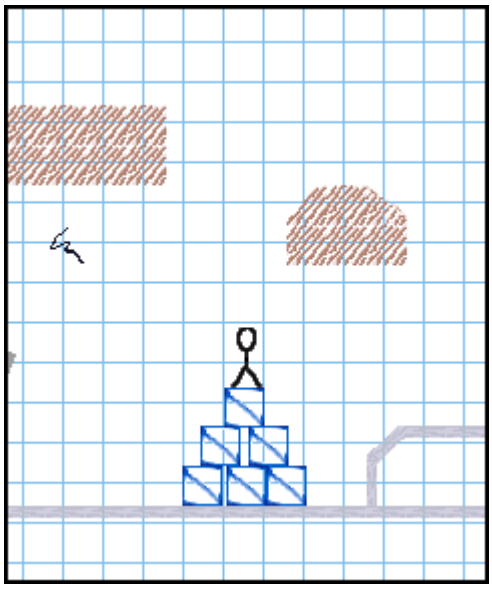
```
    protected void keyPressed(int keyCode)
    {
        int action = getGameAction(keyCode);
        switch( action )
        {
        case UP:
            shake(FXVector.newVector(0, -force));
            break;
        case DOWN:
            shake(FXVector.newVector(0,  force));
            break;
        case LEFT:
            shake(FXVector.newVector( -force, 0));
            break;
        case RIGHT:
            shake(FXVector.newVector(  force, 0));
            break;
        case FIRE:
            break;
        default: break;
        }
    }

    private void shake(FXVector vector)
    {
        for( int i = 0; i < boxElements.size(); i++)
        {
           Body boxElement = (Body) boxElements.elmentAt(i);
           boxElement.applyAcceleration( vector,
                                         World.getTimestepFX());
        }
    }
}
```

We simply have to add the box elements to the simulation canvas. We do this when the world is created.

```
public class Shaker extends MIDlet
{
    ...
    private void createShakerWorld()
    {
        ...
        simulationCanvas.addBoxElement(upper);
        simulationCanvas.addBoxElement(lower);
        simulationCanvas.addBoxElement(right);
        simulationCanvas.addBoxElement(left);
    }
}
```

A slightly different version of this sample can be found in the sample section of the distribution.

Physics Game - J2ME



This tutorial shows how to create a small jump'n'run game. We will walk through the implementation step by step. The world creation features the World Designer, a powerful tool to create complex worlds fast and simple. A slightly more complex version of the code presented here can be found in the demo application.

The game features a sprite, that can move “naturally” by pressing the arrow keys. It moves in a physical environment and can interact with all the objects. The goal is to reach a certain area by manipulation of the environment. That can include pushing crates, using see-saws or levers. An additional aspect is exploring the environment. Think along the lines of different friction of the floors (eg: ice, mud, ...), breaking bridges or levers and buttons.

The actual puzzles are implemented in the world, that is loaded.

We start off with the bare Midlet. Some parts, that are irrelevant to this tutorial are left out. We use similar concepts as in the example above (see Create an application - J2ME) in particular a GraphicsWorld and a SimulationCanvas.

```
public class PhysicsGame extends MIDlet
{
    private GraphicsWorld gameWorld;
    private SimulationCanvas simulationCanvas;

    public PhysicsGame()
    {
        ...
        simulationCanvas = new SimulationCanvas();
        createWorld();
        simulationCanvas.setWorld(gameWorld);

        getDisplay().setCurrent( simulationCanvas );
    }

    public void startApp()
    {
        ...
    }

    public void destroyApp( boolean unconditional )
    {
        ...
    }

    private void createWorld()
    {
    }
}
```

First we create the GraphicsWorld, which is used to extend the World class by drawing capabilities. We want to display the landscape (see Landscape) boundaries and the bodies.

```
public class GraphicsWorld extends World
{

   public GraphicsWorld(World w)
   {
        super(w);
   }

   public void draw(Graphics g, int width, int height)
   {
       int bodyCount = getBodyCount();
       Body[] bodies = getBodies();

       g.setColor(0x000000);
       for(int i = 0; i < bodyCount; i++)
       {
           drawBody(g, bodies[i]);
       }
       drawLandscape(g);
   }
}
```

We have to implement the both draw methods (landscape and body). The landscape simply walks over all elements and draws the corresponding line.

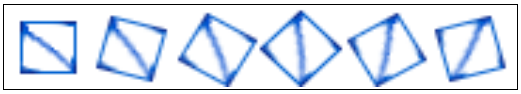
```
public class GraphicsWorld extends World
{
   ...

   private void drawLandscape(Graphics g)
   {
       Landscape landscape = getLandscape();
       g.setColor(0x000000);
       for( int i = 0; i < landscape.segmentCount(); i++)
       {
           g.drawLine(landscape.startPoint(i).xAsInt(),
                      landscape.startPoint(i).yAsInt(),
                      landscape.endPoint(i).xAsInt(),
                      landscape.endPoint(i).yAsInt());
       }
   }
}
```

To draw a body is a bit trickier this time, because we want to display bitmaps. We need to differentiate between the different shapes. This is done by overloading the ShapeSet. The ShapeSet handles the shape ids, which can be used to access images in the overloaded ShapeSet.
A vector stores the images (or null if none was registered). The register method assigns the image to the shape. The order of the shape is exactly the same as in the World Designer tool.

```
public class GraphicsShapeSet extends ShapeSet
{
    private Vector images = new Vector();
```

```
    public void registerImage(Image image, int index)
    {
        if (index < images.size())
        {
            images.setElementAt(image, index);
        }
    }

    public Image getImage(Body b)
    {
        Object image = images.elementAt(b.shape().getId());
        if (image == null) return null;
        return (Image) image;
    }
}
```

The GraphicsWorld uses GraphicShapeSet instead the default ShapeSet.
The actual drawing methods need to consider the rotation to display the correct image from the tileset. If no image is set, nothing is drawn. This will be used for the sprite to draw the correct animation frame after the world drawing. The correct image in the tileset can be determined by multiplying the angle by the number of available tiles (for the full revolution) and dividing the this by `FXUtil.TWO_PI_2FX`. These results can be improved slightly by using an offset of half the required tileset angle (=360/number of images).
The constant *`numImages`* is the number of images in a quarter turn. this code assumes an image file with *`numImages`* tiles.



The image is drawn by calculating the index corresponding to the current rotation and copying the appropriate part of the image to the graphics object.

```
public class GraphicsWorld extends World
{
    public GraphicsWorld(World w)
    {
        super(w);
        shapeSet = new GraphicsShapeSet();
    }

    ...

    private void drawBody(Graphics g, Body b)
    {
        Image image = ((GraphicsShapeSet) shapeSet).getImage(shape);
        if (image != null)
        {
            drawImage(g, image,
                      b.positionFX().xAsInt(),
                      b.positionFX().yAsInt(),
                      b.rotation2FX());
        }
    }

    public static final void drawImage(Graphics g, Image image,
                                       int x, int y, int angleFX)
    {
        int rotFX = angleFX + (FXUtil.PI_2FX / (4 * numImages));
```

```
        if (rotFX > FXUtil.TWO_PI_2FX) rotFX -= FXUtil.TWO_PI_2FX;
        int transformIdx = rotFX / (FXUtil.PI_2FX / 2);
        int index = (rotFX - transformIdx * (FXUtil.PI_2FX / 2)) /
                    (FXUtil.PI_2FX / (2 * numImages));
        int transformation = Sprite.TRANS_NONE;
        switch(transformIdx)
        {
            case 1: transformation = Sprite.TRANS_ROT90; break;
            case 2: transformation = Sprite.TRANS_ROT180; break;
            case 3: transformation = Sprite.TRANS_ROT270; break;
            default: break;
        }

        g.drawRegion(image, index * image.getHeight(), 0,
                     image.getHeight(), image.getHeight(),
                     transformation,
                     x, y,
                     Graphics.HCENTER | Graphics.VCENTER);
    }
}
```

The next step is the simulation canvas. Just like in the example before, the simulation canvas handles the world simulation, actual drawing and user input. We need a reference to the world to draw it and a reference to the sprite, because we want the screen to follow the sprite. The run() method for the thread ticks the world and draws it. When the canvas is shown, the simulation starts and when it is hidden, the simulation is destroyed. After drawing the world, we draw the sprite according to the current animation frame. If no image is registered for the Sprite, it will not be drawn by the routine, so we can draw it at the end.

```
public class SimulationCanvas extends Canvas implements Runnable
{

    private Thread thread;
    private int millis = 10;

    private GraphicsWorld world;
    private Body sprite;

    public SimulationCanvas()
    {
    }

    public void setWorld( GraphicsWorld world)
    {
        this.world = world;
    }

    public void setSprite( Body sprite)
    {
        this.sprite = sprite;
    }

    protected void paint(Graphics g)
    {
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, getWidth(), getHeight());

        g.translate(- (sprite.positionFX().xAsInt()) + getWidth() / 2,
```

```
                    - (sprite.positionFX().yAsInt()) + getHeight()/ 2);

        world.draw(g, getWidth(), getHeight());

        drawSprite();
    }

    protected void showNotify()
    {
        thread = new Thread( this );
        thread.start();
    }

    protected void hideNotify()
    {
        world = null;
        thread = null;
    }

    public void run() {
        long sleep = 0;
        long start = 0;
        while(world != null)
        {
            start = System.currentTimeMillis();

            tick();

            sleep = millis - (System.currentTimeMillis() - start);
            sleep = Math.max(sleep, 1);

            try
            {
                Thread.sleep( sleep );
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }

    public void tick()
    {
        world.tick();
        repaint();
    }

    protected void keyReleased(int keyCode)
    {
    }

    protected void keyPressed(int keyCode)
    {
    }
}
```

Now we create the world and start to simulate it. We load the world from a file, that we will create using the World Designer. That is a much simpler way to create a world. We add the sprite

manually to add it to the simulation canvas easily. the physical shape of the sprite is a rectangle (this could easily be improved to match the bitmap) and we set the sprite to “no rotate”. We assume that a (jump'n'run) character will always manage to stay upright. We could allow rotation, but we would have to implement some logic how the sprite stands up again. This is out of scope for this tutorial.

```
public class PhysicsGame extends MIDlet
{
    static public Shape spriteRectangle =
                        Shape.createRectangle(20, 30);
    private GraphicsWorld gameWorld;
    ...


    private void createWorld()
    {
        spriteRectangle.setFriction(98);
        spriteRectangle.setElasticity(0);

        PhysicsFileReader reader =
            new PhysicsFileReader("game_world_test");

        GraphicsWorld world =
            new GraphicsWorld(World.loadWorld(reader));

        Body sprite = new Body( 100, 100, spriteRectangle, 1, true);
        sprite.setRotatable(false);
        world.addBody(sprite);

        gameCanvas.setWorld(world);
        gameCanvas.setSprite(sprite);
    }
}
```

We also register the images here.

```
public class PhysicsGame extends MIDlet
{
    ...
    private void createWorld()
    {
        ...
        GraphicsShapeSet shapeSet =
                  (GraphicsShapeSet) world.getShapeSet();
        shapeSet.registerImage(crateImage, 0);
    }
}
```

Apart from the actual world file we are missing the controls. This is done by extending the motor constraint in a dynamic way, that can be controlled from outside. The two possible movements walking and jumping are handled differently. Walking is realized as the motor constraint. The jump is implemented on top of that. The methods move() and jump() set the target speeds for the respective action. The jump is executed in the precalculate part. It simply applies an acceleration (or force) to the body in the jumping direction.

```
public class SpriteControl extends Motor
{
    private int targetSpeedFX = 0;
    private int jumpSpeedFX = 0;

    private int spriteForceFX = FXUtil.ONE_FX * 20;

    public SpriteControl(Body body)
    {
        super(body, 0, 0, 0);
        setRotation(false);
    }

    public void move(int target)
    {
        targetSpeedFX = target * spriteForceFX;
        setParameter(targetSpeedFX, 0, false);
    }

    public void jump(int factor)
    {
        jumpSpeedFX = - spriteForceFX * factor;
    }

    public void setForce(int forceFX)
    {
        spriteForceFX = forceFX;
    }

    public void precalculate()
    {
        if (jumpSpeedFX != 0)
        {
            FXVector jump = new FXVector(0, jumpSpeedFX);
            body.applyAcceleration(jump, FXUtil.ONE_FX);
            jumpSpeedFX = 0;
        }
        super.precalculate();
    }

}
```

We have to create this control when creating the world and we have to pass it to the simulation canvas. We have to distinguish between key pressed and key released events to enable/disable the motor of the sprite.

```
public class SimulationCanvas extends Canvas implements Runnable
{
    ...
    private Body sprite;
    private SpriteControl spriteControl;

    ...

    public void setSprite( Body sprite, SpriteControl control)
    {
        this.sprite = sprite;
        this.spriteControl = control;
    }
```

```
    ...

    protected void keyReleased(int keyCode)
    {
        int gameAction = getGameAction( keyCode );
        switch (gameAction )
        {
        case RIGHT:
        case LEFT:
            spriteControl.move(0);
            break;
        default:
            break;
        }
    }

    protected void keyPressed(int keyCode)
    {
        int gameAction = getGameAction( keyCode );
        switch (gameAction )
        {
        case RIGHT:
            spriteControl.move(1);
            break;
        case LEFT:
            spriteControl.move(-1);
            break;
        case UP:
            spriteControl.jump(5);
            break;
        case DOWN:
            break;
        default:
            return;
        }
    }
}
```

Now follows the creation of the control instance.

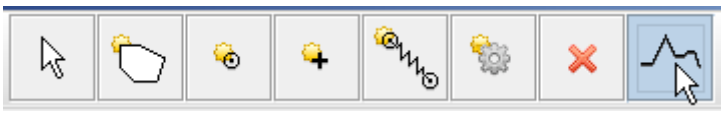
```
public class PhysicsGame extends MIDlet
{
    ...

    private void createWorld()
    {
        ...

        Body sprite = new Body( 100, 100, spriteRectangle, true);
        sprite.setRotatable(false);
        world.addBody(sprite);
        SpriteControl control = new SpriteControl(sprite);

        world.addConstraint(control);

        gameCanvas.setWorld(world);
        gameCanvas.setSprite(sprite, control);
    }
}
```

Now one problem remains with jumping: The character can jump even while it is the air. We have to check if the sprite touches the ground to decide whether it is allowed to jump. *Note*: This jump implementation is a simplified model, that does not follow Newtons third Law (which states, that each action has a reaction). To implement the jump correctly, we would have to model the sprite more realistically. We would need one body for the torsus and one for the foot and a muscle element between. The muscle could be a modified spring, that contracts prior to the jump and is released to jump.

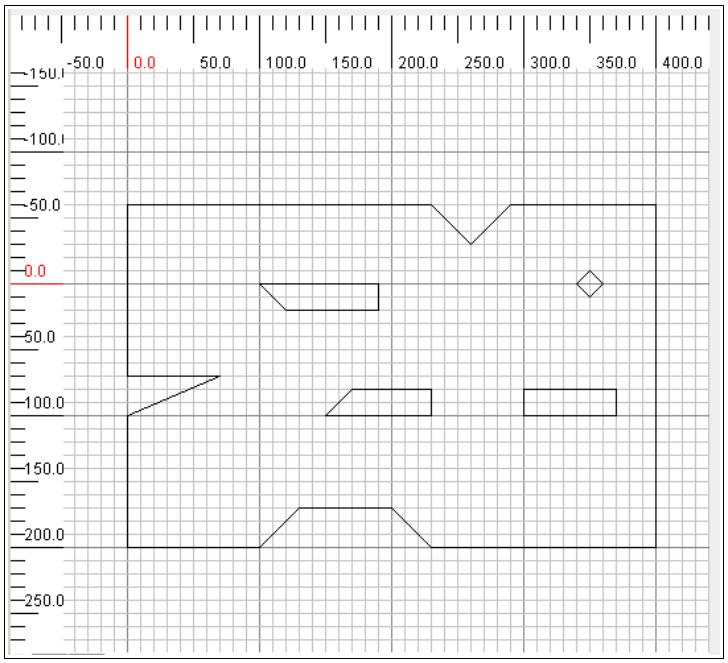
```
public class SpriteControl extends Motor
{
    ...
    public void jump(int factor)
    {
        boolean canJump = false;

        Contact[] contacts = world.getContactsForBody(body);
        for( int i = 0;
             (!canJump && i < contacts[i] != null && contacts.length;
             i++)
        {
            if ( contacts[i].body1() == body)
            {
                canJump = contacts[i].getContactPosition1().yFX >
                          FXUtil.ONE_FX * 12;
            }
            else if ( contacts[i].body2() == body)
            {
                canJump = contacts[i].getContactPosition1().yFX -
                          contacts[i].body1().positionFX().yFX +
                          body.positionFX().yFX >
                          FXUtil.ONE_FX * 12;
            }
        }

        if (canJump)
        {
            jumpSpeedFX = - spriteForceFX * factor;
        }
    }
    ...
}
```

Now we will start to create the world using the World Designer. The main view shows an empty canvas at first. To create the landscape we activate the grid. So we can position the vertices precisely.

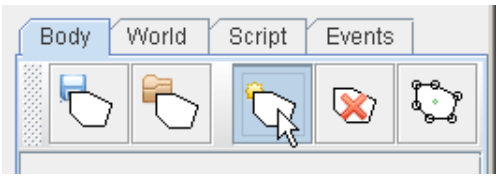
We select the landscape tool from the toolbar.



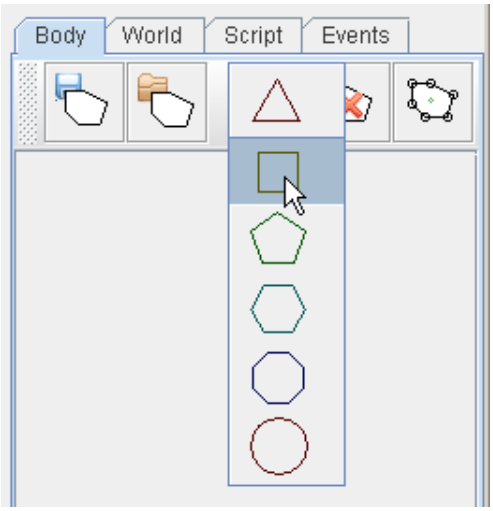
Now we can simply draw the lines forming the landscape of the world environment for our sprite to live in.

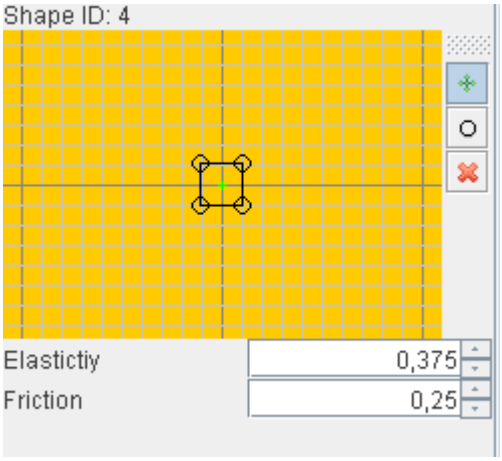


The world is quite empty at the beginning, so we create some dynamic bodies. We need to create a shape first. This is done in the panel to the right.



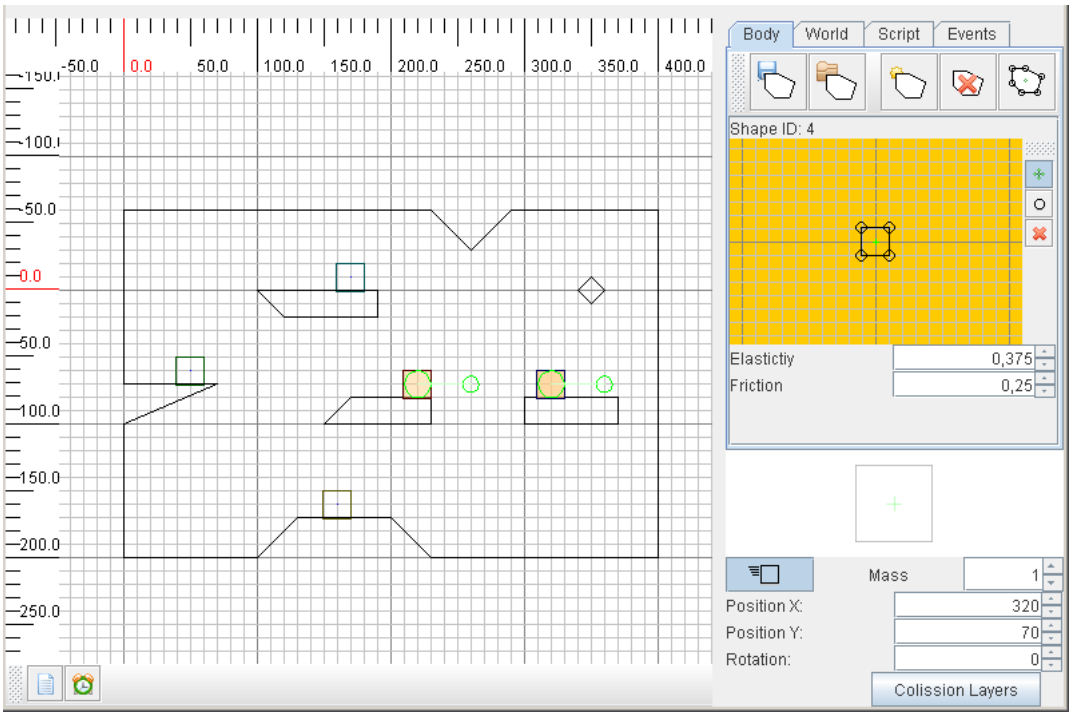
We select the square to work with.



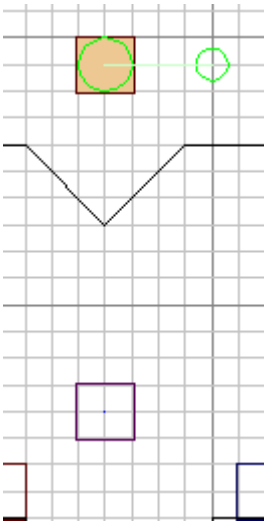


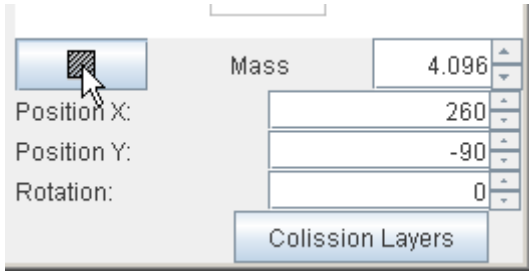
The newly created shape is given an ID, which we can use for checking as mentioned earlier. The we selected the “new body” tool in the toolbar and start to place some bodies.

Now the world should look something like this:

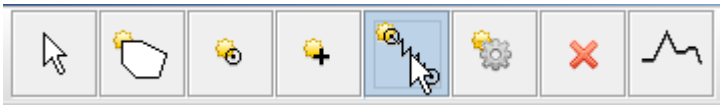


We also want a pendulum hanging from the spike in the ceiling. First we add the pendulum body and an anchor object. This can be positioned anywhere, we just need a static object somewhere to fix our pendulum. When we have the anchor selected, we click on the static button of the body info panel in the bottom right corner.

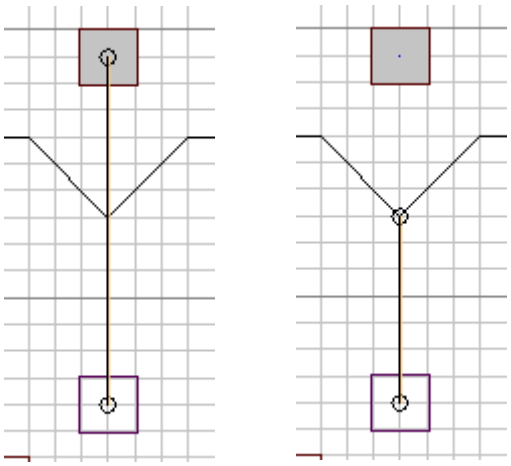


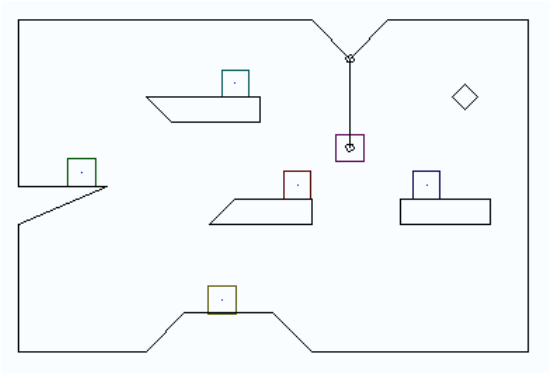


Now we can add a spring. We select the spring tool from the toolbar.



Without setting the spring coefficient it will act as a solid connection. After creating spring we can move the upper vertex to the position we actually want it to be. We simply drag the vertex to the desired spot. *Note*: We have to use that additional anchor body, because anchoring on the landscape is not possible.





Now we can save the final world and load it in the application.

The createWorld method loads the world and puts the sprite in it, so we can now run the game. the sprite will move in the world, push boxes and swing the pendulum.