A simple Scala Framework

2D Animation and Simulation

Fynn Feldpausch Martin Ring Daniel Beßler

Universität Bremen, FB3: Mathematik & Informatik
Fortgeschrittene Anwendungen der funktionalen Programmierung
Lecturer: Christoph Lüth

October 14, 2010 and November 15, 2010

Creating a simple graphical application or simulation in *Java*, *C++* or any other programming language often comes with a lot of overhead even if the environment is limited to two-dimensional space. One often faces the problem of having to write plenty of boilerplate code segments though the final application might be rather small. As soon as basic physical behavior – such as applying forces to objects or detecting collisions – is designated, the task soon becomes quite tricky.

This paper is part of a project accomplished within the lecture course *Fortgeschrittene Anwendungen der funktionalen Programmierung (Advanced Applications in Functional Programming)* during the summer term 2010. The ambition of this project was to simplify the processes mentioned above as well as getting to know the programming language *Scala*, which tries to do the splits between functional and object-oriented programming.

Despite the fact that this project is far from being finished one can get an impression of the concepts and considerations during the process of development. After an overview of the system a couple of short examples are given for an easy start with the framework.

Getting started

At first, a short introduction to Scala shall be given. After that we will provide the essential information to get the framework working.

Scala – a SCAlable LAnguage

Scala is a statically typed pure object-oriented language designed by Martin Odersky. It is influenced by a bunch of other programming languages and among others tries to extend the idea of object-orientation by the positive aspects of functional programming such as a lightweight syntax for defining anonymous functions, support for higher-order functions and supports currying as well as case classes and pattern matching.

The name itself stands for *»scalable language«* and indicates that the languages can be adjusted to suit the individual needs of the user. Scala offers possibilities to add new language constructs and statements so that their usage feels as if they were embedded in the language itself. Beyond that, Scala runs in the *Java Virtual Machine* (*JVM*) which allows the usage of any *Java*-library as well.

Another interesting point is *Scala*'s (or rather *Erlang*'s) concurrency model: the actor model. Actors are a concurrency abstraction that communicate by sending messages to each other. In combination with immutable objects this becomes a powerful attempt in respect of multi-threaded programming.

Not only these facts make *Scala* become an interesting programming language for this project. Note that due to the goal of getting to know the features of *Scala*, in some cases the highest aim of the framework might not be its efficiency.

- ▷ http://www.scala-lang.org/
- ▷ http://www.scala-lang.org/api/

Requirements

The project was realized with the use of *IntelliJ**IDEA*** and *NetBeans* which seemed to have the best support for the programming language. Of course one is free to use an arbitrary IDE as long as the essential plugins (e.g. Scala- or Maven-plugin) are available.

- ▷ http://www.jetbrains.com/idea/
- ▷ http://netbeans.org/

All remaining requirements and dependencies are handled by the build-management-tool *Maven* [1].

[1] http://maven.apache.org/

Architecture

The Project consists of three main libraries: *Geometry* for geometrical calculations in the plane, *Graphics* for graphical display and control of these and *Physics* for the simulation of rigid body dynamics:

- ▷ **Geometry** (edu.fafp.geometry)
- ▷ **Graphics** (edu.fafp.graphics)
- ▷ **Physics** (edu.fafp.physics)
- ▷ Math (edu.fafp.fastmath)
- ▷ Units (edu.fafp.units)

When designing these libraries we always kept in mind that users have different goals when using them and so we tried to make all three independently usable. Hence, the dependencies among the libraries were kept at a minimum level. The blend of possible use cases reaches from simple tasks like finding intersections of line segments or do some two-dimensional vector calculations to full blown interactive physics simulations. To accomplish that we made heavy use of *Scala*'s features like implicit conversions, mixin traits and functional composition.

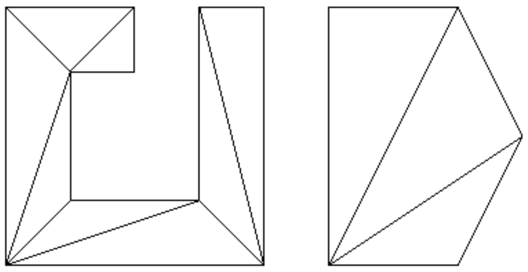
Geometry

This module provides objects and functions for geometrical calculations in two-dimensional space. The case class Vector2D represents points in the plane. All common operations on vectors are already defined: addition, subtraction, multiplication as well as the dot product. Functions like the distance of two vectors (v1↔v2) or the perpendicular of a vector (v⊥) can be intuitively used since we made use of special UTF-8 characters. Vectors can be connected to lines and line segments which in turn implement a couple of functions e.g. the intersection.

The class Shape can be used to describe the shape of geometrical objects such as circles, rectangles, triangles or arbitrary polygons. Functions on shapes include...

- ▷ ...a unique **ID**.
- ▷ ...the calculation of the shape's **area** and it's **centroid**.
- ▷ ...the maximum radius as well as an axis-aligned **bounding-box**.
- ▷ ...a check if the shape is **convex**.
- ▷ ...the calculation of the **convex hull** of the shape.

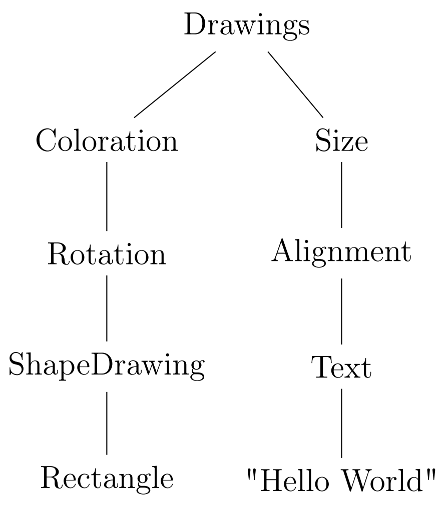
The module provides support for convex as well as for concave polygons. However, polygons must be simple i.e. their edges must not intersect. Complex polygons offer a (lazy evaluated) list of triangles for an easier handling. Therfore, shapes are decomposed with an ear-cutting algorithm:



Triangulation is necessary for a correct display of concave polygons in *Java OpenGL* (*JOGL*) as well as for a lot of calculations in the **Physics**-library. This includes the collision detection and handling as well as the calculation of the moment of inertia of an arbitrary polygon (an approximation is achieved by summing up the moments of inertia of the triangles).

Graphics

The **Graphics** module contains the (very simple) window management, the necessary functions to draw into the window and the handling of the user's input. The main task is the abstraction of the use of *Java OpenGL* (*JOGL*). Since *OpenGL* is a stateful system, a simple tree-like data structure can easily be processed by traversing the tree during the drawing phase and setting the appropriate *OpenGL* commands. That is the why every drawable object is wrapped into a Drawing. This includes not only the drawable shapes and texts but also their modifiers e.g. rotation or translation. Hence the code snippet in first example (▷ Examples) yields the following drawing tree or scene graph:



Every shape or text can not only be translated, rotated or scaled but also colorized. The withColor function can be called on every drawing and accepts a couple of different hex code strings:

- ▷ "#000" A short color hex code with RGB values.
- ▷ "#F000" A short color hex code including the alpha channel.
- ▷ "#000000" A long color hex code with RGB values.
- ▷ "#FF000000" A long color hex code including the alpha channel.

On top of that, a set of 16 RGB standard colors are provided in the package colors including black, white, blue, red and many more.

The GraphicalApplication draws all objects onto the screen, making use of OpenGl's display lists to increase the efficiency. The abstract draw function needs to be implemented by the user for that purpose (▷ A Simple Graphical Application). The class takes over the complete *JOGL* initialization and drawing. On top of that it also supplies a bunch of default settings that can be adjusted by the user (e.g. foreground and background color, view point, window size and title). Additional run time information such as the current frame rate (*fps*) can be accessed by appropriate functions (▷ ScalaDoc).

The GraphicalApplication provides a function *createGUI* for the AWT GUI creation, applications may override this method. For example window menus can be implemented quickly with this function. Finally GraphicalApplication provides also a function *nextState* for processing user input and calculating the new application state.

This function will be called each frame.

Physics

The **Physics**-library basically provides two main classes: the PhysicalObject and the PhysicalWorld. In combination they provide a solid foundation for interactive physics simulations.

Each physical object is responsible for the calculation of it's own next state (after a short time span of Δt). The state of an object is defined by it's physical properties i.e. it's position and it's orientation as well as (among others) the momentum (p) and angular momentum (L) of the object.

The new momentum is calculated as follows using the relative velocity $v^{AB} = v^{AP} - v^{BP}$ (P being the penetration point of the collision), vector r (the vector from the polygons centroid to the penetration point) and the object's elasticity e:

$$\Delta p = j \cdot n$$
$$\Delta L = r_{\perp}^{AP} \cdot j \cdot n$$

where the factor j is calculated by

$$j = \frac{-(1 + e_A \cdot e_B) v^{AB} \cdot n}{n \cdot n \left(\frac{1}{M^A} + \frac{1}{M^B} \right) + \frac{(r_{\perp}^{AP} \cdot n)^2}{I^A} + \frac{(r_{\perp}^{BP} \cdot n)^2}{I^A}}$$

The class PhysicalWorld determines the collisions between objects in a two-part process. First a set of critical objects containing objects that possibly collided is created. For this calculation the maximum radius r_{max} and the position of the objects' centroid c are used so that the collision detection comes down to a simple check if $c^A \leftrightarrow c^B > r_{max}^A + r_{max}^B$. After that a more complex calculation is performed in order to determine Δp and ΔL.

Additionally the **Physics**-library provides elastic and not elastic connection handling. The connection handling is very flexible. There are two different types of connections object-point connections and object-object connections.

With object-point connections you can pin a point, relative to an object, to a dynamic or static point. If you pin an object twice, it position will be fixed at the pin positions. You can also connect a point, relative to an object, to a dynamic or static point on distance. The connector length can be fixed at the specified distance or elastic between both connector points within a specific range.

Obviously object-object connections will connect two objects relative to each other. The connector can have a fixed or elastic length again.

The connectors apply force on the object if they are not in their desired length. They are using a linear force calculation using Hooke's law. You can specify the Hooke law constant D for each connection. If the connectors are at their limit length a collision between the connected objects may occur. If the directly connected objects are connected again some other objects may be involved in the collision too. The **Physics**-library handles the case of objects with multiple connections and will send *Collision* messages to all connected objects as needed.

The offsets are then send to the individual objects. The global time tick as well as the application of force (e.g. gravity) is handled in the same way:

$\text{o ! Collision}(\Delta p, \Delta L)$
$\text{o ! ForceEvent}(gravity \cdot mass)$
$\text{o ! Tick}(\Delta t)$

Additional Libraries

For the convenience of users there are also two small additional libraries for easy handling of physical units and for fast approximations of mathematical functions.

The module *Math* contains approximations of math.sqrt, math.log and math.exp. These run considerably faster while not deflecting too much from exact results. That is accomplished by taking advantage of the nature of the *IEEE* representation of double precision floating point numbers.

The *Units*-library implements easy to use unit conversions for standard physical quantities like length, mass, time and so on. To accomplish that we use an implicit conversion on Double and Int which allows to write something like $36\ \text{km}\ /\ 1\ \text{h}$ which then will automatically be converted to the standard *SI* unit of m/s and result in a Double value of 10. For most units the standard notation is to write the unit symbol as postfix operator like in the previous example but there is one special case when it comes to temperatures and angles. The degree symbol will usually be interpreted as an angle like in $45°$ (which will be converted to the dimensionless radiant number 0.785...) but if a $\text{TemperatureModifier}$ is placed after the °-operator the value will be interpreted as a temperature. So one can type $15°\text{C} + 5°\ \text{F}$ which will (maybe surprisingly) result in the correct scale of 546.3 K.

Examples

The entire framework is documented in *ScalaDoc*. This documentation should be the first source of information when a problem is on hand. But metaphorically following the proverb *»A picture is worth a thousand words«* a short set of examples might help getting to know the framework.

A Simple Graphical Application

To start with an easy task, a simple graphical application is presented below. The result of this tiny code snippet is a single window, containing a *Hello World* text and a rotating red square.

```
1 object Main extends
      GraphicalApplication(
2     title="Square", size=(400,300)) {
3   def draw = Drawings(
4       Text("Hello World")
5         withSize 48 withAlign Top at
              (0,-25),
6       Rectangle(3,3)
7         rotateBy (now) withColor
              "#800000"
8     )
9 }
```

Despite the simplicity of this program one can spot plenty of interesting facts concerning the usage of the framework. For that reason every single line is worth a closer look. As shown in line 1, the application is created simply by extending the GraphicalApplication class. It handles the creation of the window as well as the processing of the contained drawings. In doing so, the framework completely abstracts from the usage of the *Java OpenGL* (*JOGL*)-library or *Java*'s *Abstract Window Toolkit* (*AWT*). On is able to modify the window's title and its size by overwriting the default values of the application's constructor (▷ line 2).

In order to add drawings to the window it is essential to define the abstract draw function (▷ line 3). In the following line, the definition of the Drawings starts. This can either be a text or any shaped object (e.g. a circle, a rectangle or an arbitrary simple polygon). Strings and Shapes will be converted into appropriate Drawings making use of *Scala*'s implicit conversions. Line 4 and 6 show two examples.

Drawings can be manipulated by the following set of functions, each of which can also be applied on a set of Drawings. For an example of how to use these modifiers see line 6 and 8.

- ▷ rotateBy(<Angle>)
- ▷ translateBy(<Vector2D>) | at(<Vector2D>)
- ▷ scaleBy(<Double>)
- ▷ withColor(<Color>)

Since Drawings can be nested this allows a neat and flexible way for the manipulation of Drawings. One should keep in mind though that some of these manipulations are not commutative e.g. rotateBy(π) translateBy(1,0) yields another result as translateBy(1,0) rotateBy(π). Also note that the translation of drawings and text differs with respect to their coordinate system. The translation of drawings uses world coordinates whereas the translation of text uses window coordinates. This allows a simplified usage when it comes to text overlays.

In addition to the functions mentioned

above, a text can be manipulated using the following self-explanatory functions:

- ▷ withSize(<Int>)
- ▷ withAlign(<VAlign> | <HAlign> | (<VAlign>,<HAlign>) | (<HAlign>,<VAlign>))

Getting Physics Envolved

And now for something completely different. Say we want to simulate some violence between geometric primitives. Let's throw a heavy block on a small defenseless ball. To do that we simply need to utilize the Physics-library:

```
 1 object Main extends
       GraphicalApplication(title = "
       Physics Example") {
 2   object BouncingBall extends
         PhysicalObject(
 3     shape = Circle(radius = 30 cm),
 4     position = (3 m, -50 cm),
 5     material = Materials.Rubber
 6   )
 7
 8   object HeavyBlock extends
         PhysicalObject(
 9     shape = Rectangle(width = 2 m,
           height = 90 cm),
10     orientation = 30°,
11     momentum = (10 kNs, 20 kNs),
12     angularMomentum = -1000 Nms,
13     material = Material.Stone
14   )
15
16   var state: PhysicalWorld =
         PhysicalWorld(
17     objects = Seq(BouncingBall,
           HeavyBlock)
18   )
19
20   override def nextState(Δt: Double,
         inputs: Set[Input]) {
21     state = state.nextState(Δt)
22   }
23
24   def draw = state.draw
25 }
```

When you run that code you will see a big rectangle flying towards a ball while both are pulled downwards by gravity.

The code is quite easy to understand but let's go through it. Like in the previous example we derive our main object from GraphicalApplication to get our graphics going.

But now for the interesting new part: First we need to create the ball and the box. That part is pretty straight forward. We simply derive from PhysicalObject (▷ line 2 and 8) and supply the constructor with the information we want to set. All parameters that are not set will be set to the default values which can be found in the *ScalaDoc* of the class PhysicalObject. To provide the physical quantities it comes in handy to utilize the Units-library which allows to write 30 cm (▷ line 3) or 30° (▷ line 10). Apart from that the object declarations are self-explanatory.

Next we need something to store our state in. In our example that is done by declaring a var state (line 16). Our state is a PhysicalWorld which will handle all things concerning the physical motions of our two objects. To let our world know which objects it should care about we need to pass them into the constructor (▷ line 17). Additionally we could specify a value for gravity (which is 9.81 ms^{-2} by default) and the bounds of the world but for the example we are fine with the default values which fit together nicely with the default values of GraphicalApplication.

In this example we override the nextState function from the baseclass. Here we can do user event processing and state transformation using the supplied *inputs* and Δt.

Finally in the draw function we create the

Drawings again. Note that PhysicalWorld itself doesn't know anything about graphics, and thus has no draw function, still state.draw will return a correct result because there are implicit conversions defined in *Graphics* that will draw any list of objects and check for each object if it has implemented one or more of the traits HasShape, HasPosition and HasRotation and then draws each object with the information it has.

The example shows how neatly the Physics-library can be integrated even though Graphics and Physics don't know anything about each other. To dive deeper into the functions the Physics-library provides you can consult the *ScalaDoc*. We hope you enjoy playing around with various physical scenarios and maybe even extend our libraries to your need.