EDUCATIONAL PEARL

Haskell in Space

An Interactive Game as a Functional Programming Exercise

CHRISTOPH LÜTH

FB 3 — Mathematik und Informatik, Universität Bremen
(*e-mail:* cxl@informatik.uni-bremen.de)

Abstract

This paper describes a practical exercise set to an introductory functional programming course. The exercise is to implement a small game involving a space ship in an asteroids belt, after the fashion of the classic *Asteroids* arcade game. The positive experience suggests that interactive graphics programs of this kind make good and entertaining programming exercises for functional programming courses.

1 Introduction

Good programming exercises are hard to find, since they have to satisfy a number of requirements: they should have an appropriate level of difficulty, should be easily yet precisely explainable, and should motivate students. A good source of programming exercises are interactive games (Aerts & De Vlamnick, 1999). In this paper, we describe a practical exercise fitting the three criteria above, in which a version of the interactive game *Asteroids* had to be implemented. The exercise was set to second-year students at the end of an introductory course to functional programming, and was well received by the students.

In *Asteroids*, the player manoeuvres a space ship through a number of asteroids moving about the screen, which have to be dodged or preferably destroyed with the ship's laser. If the ship hits an asteroid, the game is over. These simple game mechanics are quickly explained and lend themselves well to a functional description. Moreover, the graphical setting of the game is a welcome change from the console-based text interaction that students had become used to throughout the course until then. Thus, the exercise also hopefully serves to convince students that functional programming is about more than sorting lists, and can be useful in practical situations.

The contribution of this paper is twofold: firstly, teachers or students of functional programming may want to use this exercise as it is, or with slight modifications. Secondly, in a wider context it can serve as an example of the kind of programs which are not too hard to write or explain, and which fit in well at the end of a functional programming course. The advantages and drawbacks of this particular game are discussed in more detail below, but we believe the general idea of finishing an introductory course to functional programming with a simple interactive graphical game is widely applicable.

The rest of this paper is structured as follows: we first describe the setting of the practical exercise, the course and its prerequisites. We then introduce the game and its mechanics in more detail, covering also some historical aspects. This is followed by a detailed tour of the implementation. We finish with an evaluation and conclusions.

2 The Practical Exercise

2.1 Course and Prerequisites

The practical exercise described here was set in an introductory functional programming course. The students were in the second year of a four-year *Diplom* course. In their first year, they had become acquainted with imperative and object-oriented programming using Java; in the second year, they had been introduced to functional programming using Haskell. Before this course, students did not have any previous experience with functional or graphical programming.

The exercise was the last of the course; the students had two weeks to complete it. At this point, the students had been taught functional programming for eleven weeks, covering roughly the material in (Thompson, 1999), which was used as a course book. They had seven previous exercise sheets running one or two weeks, which posed several small functional programming exercises.

The exercise was set at the end of winter term 2001/2002 as described here. In this year's run of the course, we have used a slight variation of the exercise (described in the conclusions).

2.2 Setting the Exercise

Designing and implementing a small interactive game like *Asteroids* is a nice short exercise for an experienced programmer, but it is too difficult to do from scratch for second-year undergraduates who have just learnt the basics of functional programming. We have to give them a hand.

In the lectures prior to handing out the exercise sheet, a basic reference system was presented and explained in detail. The reference system allowed to manoeuvre a small "space ship" across the screen (i.e. "space"), without any asteroids or bullets. The architecture of the system had been designed in such a way that it would be easily extendible towards the full game.

This reference system demonstrates basic functional graphics programming using the Hugs Graphics Library (`HGL`), a lightweight portable graphics library usable with Hugs and the Glasgow Haskell Compiler for a variety of operating systems, including Windows, Linux, and Solaris. It also demonstrates how to react to user events, such as button presses, and how to implement animations in `HGL`. All of this could be explained in two lectures of 90 minutes each.

3 The Game

In *Asteroids*, a small space ship has to be manoeuvred through a treacherous asteroid belt. The space ship can be manoeuvred by turning it to the left or right, or by accelerating it using a thruster. The ship's momentum carries it along the direction of its current velocity, which may not be the same as its current orientation, whereas acceleration always operates in the direction of the orientation.

The ship can fire laser bullets to destroy the asteroids. They are fired in the ship's current orientation, always have the same velocity, and travel in a straight line for a certain time, until they disappear.

The asteroids come in three sizes. Initially, there is a random amount between one and five of the largest size. If an asteroid is hit by a bullet, it breaks up into a random number of one to three asteroids of the smaller size. If an asteroid of the smallest size is hit, it disperses into space dust. This way, the player can clear the screen of all asteroids. Once this is achieved, a new level is reached, and another random amount of large asteroids appears.

3.1 A Short Introduction to Space Travel

Mathematically, the ship can be described as follows: it always has a current velocity, which is a vector $\vec{v}$, and an orientation, which is an angle o. If the ship flies straight ahead, velocity and orientation have the same direction (Fig. 1, left). If the ship turns, the direction of the orientation will differ from that of the velocity. Since the ship always accelerates in the direction of its orientation, the new velocity $\vec{v'}$ is given by adding the old velocity $\vec{v}$ and the thrust $\vec{t}$ (Fig. 1, right):

$$\vec{v'} = \vec{v} + \vec{t} \tag{1}$$

The vector $\vec{t} = (\omega, l)$ is given in polar form by the orientation ω and a length l, which is either a constant L if the ship is currently accelerating, or 0 if it is not. Thus, change of direction is achieved by turning the ship into a different direction

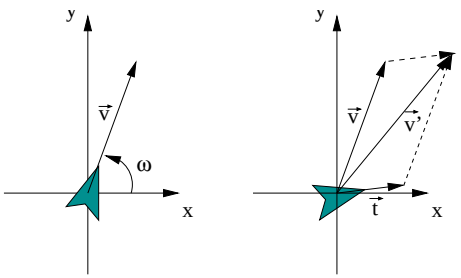

Fig. 1. Determining the ship's movement.

and accelerating, and equally deceleration is achieved by accelerating against the current velocity. Later on, we will be able to translate this mathematical description into a functional program easily.

3.2 A Brief Look at History

The original *Asteroids* arcade game was introduced in 1979 by Atari, Inc. It additionally featured hyperspace (allowing the space ship to relocate instantaneously at a random location), and an enemy spaceship which would appear at random and fire laser bullets at the player. Although we did not require the students to implement these, their addition would make a nice variation on the exercise.

However, the game can be traced back directly to the seminal *Spacewar* game, written in 1961/62 at MIT by Stephen Russell, Peter Samson, Dan Edwards, and Martin Graetz, together with Alan Kotok, Steve Piner, and Robert A. Saunders (Edwards & Graetz, 1962). Spacewar featured two space ships which were operated by turning and accelerating exactly like in Asteroids. However, there were no asteroids, and the game was played by two players trying to destroy the other player's ship with the laser bullets the ships could fire.

Spacewar was originally implemented on the Digital PDP-1 machine (Graetz, 1981). This was one of the first so-called mini-computers, allowing reactive user interaction via a CRT display and keyboard, as opposed to the mainframe computers mainly in use in those days, where user interaction consisted in handing in a batch of data (programs and input) to the operator, and getting back a printout of the results some hours later (or even the next day). Spacewar was one of the first, and one of the most enduringly popular programs making use of the new user interaction capabilities.[1] This corroborates our premise that simple graphical games like Spacewar or Asteroids provide a good motivation to students and programmers.

4 The Implementation

This section gives a guided tour to the implementation of the reference system. Essentially, this is the material that students need to be familiar with in order to successfully finish the exercise; thus, the contents of this section are the contents of the lectures preceding the handing out of the exercise sheet.

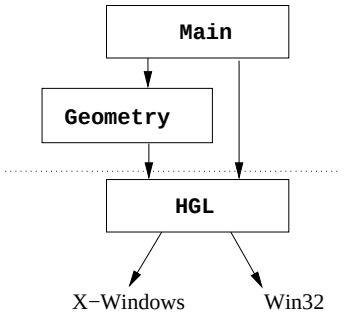

Fig. 2. System Architecture Overview.

[1] The game survives until this day: the project web site at MIT's Media Group `http://lcs.www.media.mit.edu/groups/el/projects/spacewar` offers the original Spacewar as a Java applet.

Fig. 2 shows an overview of the system architecture. In general, we model the reactive animation in a loop, which draws the current state, calculates the next state, and waits until the next iteration. This is the contents of the `Main` module. In order to calculate the state, we use the mathematical description from Section 3.1; the linear geometry required is provided by the `Geometry` module. As can be seen, the Hugs Graphics Library `HGL` completely hides the implementation of the graphics by the operating system (such as X Windows for Unix-based systems, or the relevant libraries provided by Windows operating systems).

4.1 The `Geometry` Module

The `Geometry` module provides utility functions for linear geometry, and abstract datatypes for geometric figures.

The basic types `Angle` and `Point` come from `HGL`, and are defined

```
type Angle = Double                type Point = (Int, Int)
```

The utility functions consider points as vectors from the origin to that point, and provide functions to add vectors, calculate the length, and multiply with a scalar. A fourth function converts polar coordinates into vectors:

```
add   :: Point-> Point-> Point
len   :: Point-> Double
smult :: Double-> Point-> Point
polar :: Double-> Angle-> Point
```

Implementing these functions earlier in the course provides a useful small programming exercise in its own right.

However, the main purpose of the `Geometry` module is to implement two datatypes supplementing `HGL`'s graphical capabilities. `HGL` supports drawing polygons, but not rotating, scaling or translating them. Moreover, when implementing the bullets and asteroids, we will need a way to efficiently check if two such polygons intersect, e.g. to check if the space ship hit an asteroid. As with the basic geometry above, implementing intersection algorithms for two polygons earlier in the course serves as another useful programming exercise.

To this end, two kinds of geometric figures are supported. On the more abstract level, we have geometric figures which can be scaled, rotated, and moved about:

```
data Figure = Polygon [Point]
            | Circle Dimension
            | Translate Point Figure
            | Scale Double Figure
            | Rotate Angle Figure
```

These figures can be translated into an abstract type `Shape`; a shape can be efficiently drawn and checked for intersection with other shapes:

```
type Shape
```

```
shape     :: Figure -> Shape
drawShape :: Figure-> Graphic
contains  :: Shape-> Point-> Bool
intersect :: Shape-> Shape-> Bool
```

Internally, shapes are polygons or circles, with translation, rotation and scaling normalised to absolute coordinates. Since this can be expensive, we only do this once. `Graphic` is `HGL`'s type representing drawable primitives (like polygons or circles).

4.2 The Main Loop and the State

As mentioned above, we model the reactive animation in a loop, which about every 30 ms draws the current state on the screen, calculates the next state, and waits until the next iteration. This allows a clean separation of concerns: on the one hand, the state is given by a data type `State`, which contains the data for all objects on the screen: the ship, and later on the asteroids and the bullets. A function `nextState` calculates the next state, given the user input and the previous state, according to the principles laid out in Sect. 3.1. On the other hand, a function `drawState` renders the state on the screen. Hence, the main loop of the game is:

```
loop :: Window-> State-> IO ()
loop w s =
  do setGraphic w (drawState s)
     getWindowTick w
     evs<- getEvs
     s'<- nextState evs s
     loop w s'
```

The function `getWindowTick` is from `HGL` and waits until the full 30 ms have passed (30 ms being the "tick" here), and the function `getEvs` gets all the events which have occurred while waiting for the next tick.

4.3 The Ship

The state of the ship is a labelled record containing the current position `pos`, velocity `vel`, orientation `ornt`, thrust `thrust` and angular speed `hAcc` (i.e. the speed with which the ship is turning). We also have to keep track the ship's `Shape`, to allow efficient check for intersection with other objects.

```
data State = State { ship  :: Ship }

data Ship = Ship { pos    :: Point,
                   vel    :: Point,
                   ornt   :: Double,
                   thrust :: Double,
                   hAcc   :: Double,
                   shape  :: Shape }
```

`thrust` and `hAcc` are changed whenever the user presses a key to turn left or right (in this case `hAcc` is set to `hDelta` or `-hDelta`, a constant which determines how fast the ship is turning), or releases such a key (in this case `hAcc` is reset to zero), or presses or releases the thrust key (which sets or resets `thrust`). This is implemented by a function

```
procEv :: State-> Event-> State
```

which does a straightforward case distinction on the key press and release events.

The function `nextState` calculates the next state state of the ship, given the current one and a list of events which have occurred in the meantime. It uses `procEv` to process the events, and afterwards the function `moveShip` (shown below) to calculate the ship's new position.

```
nextState :: [Event]-> State-> IO State
nextState evs s =
  s1{ship= moveShip (ship s1)} where
     s1= foldl procEv s evs
```

The function `moveShip` computes the position, velocity and orientation from the current ship state, as described in Sect. 3.1 above.

```
moveShip :: Ship-> Ship
moveShip(Ship{pos= pos, vel= vel,
              hAcc= hAcc, thrust= t, ornt= o}) =
  setShape
    (Ship{pos= add pos vel,
       vel= if l> vMax then smult (vMax/l) vel1 else vel1,
       thrust= t,
       ornt= o+ hAcc,
       hAcc= hAcc}) where
         vel1= add (polar t o) vel
         l   = len vel1
```

The new position is given by adding the velocity to the old position (modulo the size of window; this is omitted above). The new velocity is given by adding the current velocity and the thrust as vectors; however, if the resulting velocity's magnitude exceeds a global constant `vMax`, it is multiplied with a real factor to make its magnitude equal to `vMax`. Thus, the magnitude of the velocity never exceeds `vMax` but the direction can still to change.[2] After moving the ship, a new `shape` has be calculated; this is what the function `setShape` does.

In order to draw the ship we have to calculate its shape. It is given by a global constant, which is rotated by current orientation, and moved to the current position:

[2] Actually, the maximum speed of a ship in space would be given by friction (of whatever little gas particles there in interstellar or interplanetary space) and relativity effects, which would make the magnitude of the velocity approach a maximum asymptotically, not linearly as is the case here.

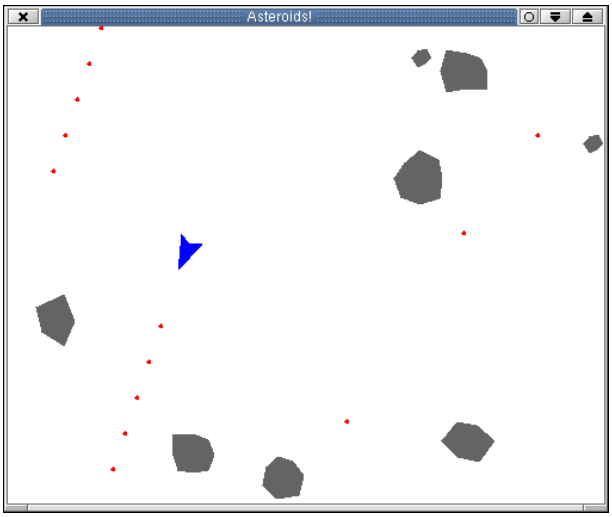

Fig. 3. The finished *Asteroids* game.

```
spaceShip :: Figure
spaceShip = Polygon [(15, 0), (-15, 10),
                     (-10, 0), (-15, -10), (15, 0)]
setShape :: Ship-> Ship
setShape s = s{shape= shape (Translate (pos s)
                            (Rotate (ornt s) spaceShip))}
```

Figure 3 shows a screenshot of the finished game, with asteroids and all; the background is originally black, but has been set to white for better typesetting. The full source code for both the reference system, and a reference solution, can be found at the web page (Lüth, 2001). The reference solution comprised about 220 lines of Haskell, the reference system described here about 90 lines, and the Geometry module 110 lines (excluding comments).

5 Conclusion

The exercise was very popular with the students. Out of 129 finishing the course, 79 returned a questionnaire which was handed out, and from these 32 named this exercise as their favourite exercise sheet (out of a total of eight exercise sheets). Since one of the other exercises had been a small game as well, the popularity of the exercise cannot be attributed to the fact that it was a game alone, but also that it used animated graphics. Some students went well out of their way to embellish the solutions they handed in (a pick of the best can be found at the web page).

It remains open to debate whether the particular game of *Asteroids* has been a good choice. In favour of it are the simple game mechanics, the comparatively easy algorithms, and the geometry algorithms that could be used earlier in the course as exercises. However, as opposed to the games such as the ones presented in (Aerts & De Vlamnick, 1999), the game cannot be extended with strategies or other such

constructs which would particularly benefit from functional programming; in fact, the embellishments mentioned above were all concerned with more fancy graphics.

The style of implementation in this exercise is fairly straightforward, as opposed to the more abstract style of reactive graphical programming proposed in (Hudak, 2000), which allows a very elegant, declarative description of animated objects (the functional animation library `FAL`). However, in our experience this abstraction unfortunately tends to go over the heads of the students in a first introductory course to functional programming. We have used `FAL` in the year before, and set a similar exercise to this one, but the completion rate (both of the course and the exercise) and popularity of that exercise was far below this year's.

We hope that the practical exercise introduced in this small paper can be useful to other teachers (and students) of functional programming languages, either by directly using it, or by developing similar exercises on their own. Variations on this exercise include the original *Spacewar* game, or a game we have invented for this year's rerun of the course: we have a star in the middle of the game, with old space satellites orbiting around it. The player has to collect the old space satellites with his ship by using a tractor beam (which obviously can only be used if you are very close to the ship). The star exerts a strong gravitational force, which complicates the ship's manoeuvring. Optionally, at some point a space ship of a rival space junk collection company appears, and opens up fire on our ship.

To sum up, the exercise was easy to write (a fact only made possible by the existence of the Hugs Graphics Library, showing the usefulness of a light-weight, portable graphics library for teaching), and popular. We hope it can serve as an inspiration for other teachers to develop similar exercises; this will certainly be beneficial to the reception of functional programming at the undergraduate level.

References

Aerts, Kris, & De Vlamnick, Karel. (1999). Games provide fun(ctional programming tasks). *Functional and declarative programming in education, FDPE'99.*

Edwards, D. J., & Graetz, J. M. (1962). PDP-1 plays at Spacewar. *Decuscope*, **1**(1). Available at `http://www.wheels.org/spacewar/decuscope.html`.

Graetz, J. M. (1981). The origin of Spacewar. *Creative computing*, August 1981. Available at `http://www.enteract.com/~enf/lore/spacewar/spacewar.html`.

Hudak, Paul. (2000). *The Haskell school of expression.* Cambridge University Press.

Lüth, Christoph. (2001). *Haskell in Space web page.* `http://www.informatik.uni-bremen.de/~cxl/haskell-in-space`.

Reid, Alastair. *The Hugs Graphics Library.* `http://www.haskell.org/graphics`.

Thompson, Simon. (1999). *Haskell: The craft of functional programming.* Second edn. Addison-Wesley.