Haskell in Space

An Interactive Game as a Functional Programming Exercise

Christoph Lüth

FB 3 — Mathematik und Informatik, Universität Bremen
cxl@informatik.uni-bremen.de

Abstract. This short paper describes the final practical exercise in a functional programming course for second year computer science students at the University of Bremen. The exercise was to implement a small game involving a space ship in an asteroids belt, after the fashion of the classic *Asteroids* arcade game. We suggest that interactive graphics programs of this kind make good, entertaining programming exercises for functional programming courses.

Good programming exercises, which have an appropriate level of difficulty, are easily, yet precisely explained, and motivate students are hard to find. A good source of programming exercises are interactive games [2]. Here, we describe a practical exercise set to second-year students at the end of an introductory course to functional programming, in which a version of the interactive game *Asteroids* had to be implemented, which fits all the above criteria.

In *Asteroids*, the player can manoeuvre a space ship by turning left or right, or accelerating. A number of asteroids move about the screen, which have to be dodged or preferably destroyed with the ship's laser bullets. If an asteroid is hit it breaks into successively smaller parts, which ultimately disperse. 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 students had become used to throughout the course. Thus, the exercise also hopefully served 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. But in a wider context, it can serve as an example of the kind of programs that can be fitted in at the end of a functional programming course, which are not too hard to write or explain. 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.

1 The Practical

Designing and implementing an 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.

Hence, in the lectures prior to the handing out of 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 how to use the Hugs Graphics Library (HGL) [4], 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.

The students were in their second year. The had been taught imperative and object-oriented programming in their first year, and at this point had been taught functional programming for eleven weeks, covering roughly the material in [5], which was used as a course book. The students were already aware of `IO` in Haskell, and had already implemented some of the geometry algorithms in earlier programming exercises, but knew nothing about graphics programming in a functional language.

2 The Implementation

In this section, we explain the implementation of the reference system; thus, the contents of this section are the contents of the lectures preceding the handing out of the exercise sheet.

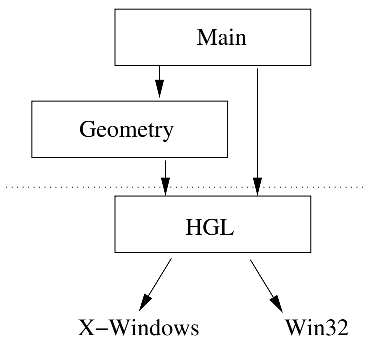

Fig. 1. System Architecture Overview.

Fig. 1 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 first recall some basic physics and linear geometry; this is provided by the `Geometry` module. As can be seen, the Hugs Graphics Library `HGL` completely hides how the actual graphics are implemented by the operating system (such as X Windows for Unix-based systems, or the relevant libraries provided by Windows operating systems).

2.1 A Short Introduction to Space Travel

Our space ship can only turn, and accelerate. Hence, it has a current velocity, and an orientation (the direction it looks in). The velocity is a vector $\boldsymbol{v}$. If the ship flies straight ahead, velocity and orientation have the same direction (Fig. 2 left). However, if the ship turns, the direction of the orientation will differ from the velocity. Since the ship always accelerates in the direction of its orientation, the new velocity $\boldsymbol{v'}$ is given by adding the old velocity $\boldsymbol{v}$ and the thrust $\boldsymbol{t}$ (Fig. 2, right).

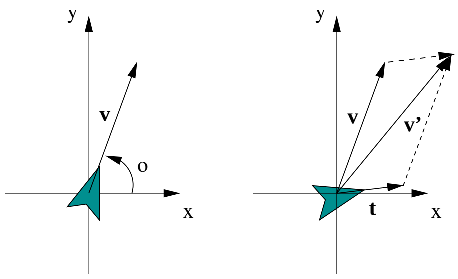

Fig. 2. Determining the ship's movement.

In order to describe this situation in our small program, the `Geometry` module provides utility functions to add points (as vectors), calculate the magnitude (length) of a point considered as a vector, convert polar coordinates to points, and multiply a point with a scalar:

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

The types `Angle` and `Point` (doubles and pairs of doubles respectively) come from HGL. Implementing these functions earlier in the course provides a useful small programming exercise in its own right.

2.2 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 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. 2.1. 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.

2.3 The State

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).

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

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

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

which does a straightforward case distinction on the key press and key 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 =
  do return s1{ship= moveShip (ship s1)} where
         s1= foldl (flip procEv) s evs
```

The function `moveShip` computes the position, velocity and orientation from the current ship state, as described in Sect. 2.1 above. Note that the vector $\boldsymbol{t}$ above is represented in polar form in the ship state, namely its magnitude given by `thrust`, and the direction given by the ship's orientation `o`.

```
moveShip :: Ship-> Ship
moveShip(Ship{pos= pos, vel= vel,
              hAcc= hAcc, thrust= t, ornt= o}) =
  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. [1]

2.4 Drawing the Ship

Drawing the actual ship is easy. The ship's shape is given by a global constant

```
spaceShip :: Figure
spaceShip = Polygon [(15, 0), (-15, 10),
                      (-10, 0), (-15, -10), (15, 0)]
```

which is rotated by current orientation, and moved to the current position. HGL supports drawing polygons, but not rotation and translation. Moreover, when

[1] 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.

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 served as another useful programming exercise.

To this end, the geometry library `Geometry.hs` supports geometric figures in two stages. 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 efficiently be drawn and checked for intersections:

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

Internally, shapes are polygons or circles, but their coordinates are normalised, i.e. the translation, rotation and scaling has been flattened. Since this can be expensive, we only do this once.

The introduction of shapes also means that the `Shape` of the ship should be held in the state of the ship, so we add a field

```
    shape :: Shape
```

to the labelled record `Ship`.

Fig. 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. Further, the full source code for both the reference system, and a reference solution, can all be found at [1].

3 Conclusion

The exercise was very popular with the students. At the end of the course, a questionnaire was handed out. Out of 129 finishing the course, 79 returned the questionnaire, and from these 32 named this exercise as their favourite exercise sheet (out of a total of eight exercise sheets). Since also one of the other exercises had been a small game, the popularity of the exercise cannot be attributed to the fact that it was a game alone, but also that used animated graphics. Some students went well out of their way to embellish the solutions they handed in (the best solutions can be found at [1]). Thus, we surmise that the combination of games and animated graphics makes a popular choice of exercises.

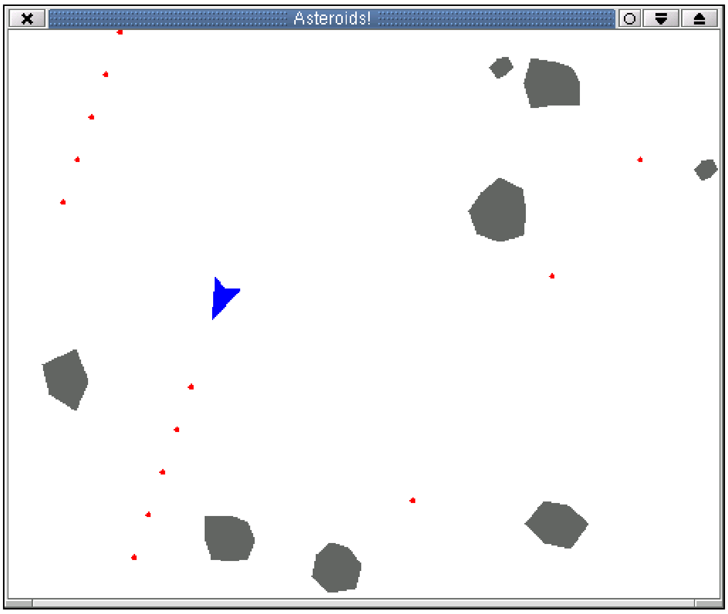

Fig. 3. The finished *Asteroids* game.

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 [2], 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 and such.

The style of implementation in this exercise is fairly straightforward, as opposed to the more abstract style of reactive graphical programming proposed in [3], 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.

To sum up, 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. The assessment was only made possible by the existence of the Hugs Graphics Library, showing the usefulness of a light-weight, portable graphics library for teaching.

References

1. `http://www.informatik.uni-bremen.de/~cxl/haskell-in-space`.
2. Kris Aerts and Karel De Vlamnick. Games provide fun(ctional programming tasks). In *Functional and Declarative Programming in Education, FDPE'99*, 1999.
3. Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.
4. Alastair Reid. The Hugs Graphics Library. `http://www.haskell.org/graphics`.
5. Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, second edition, 1999.