Functional Programming 2014/2015
Assignment 5: Game programming

Ruud Koot

September 30, 2014

In the final assignment you'll be developing a small game in Haskell. This assignment will be a bit more "realistic" than the previous assignments: the program you'll write has to respond in real-time to user input, will consist of multiple modules, and you have more freedom in how and what to implement.

1 Introduction

In this assignment you'll make use of the Gloss graphics library for Haskell, which provides a very high-level interface for drawing graphics on screen and handling user input.

The game you'll be implementing is a variant of the classic 1979 arcade game *Asteroids*.

1.1 Getting started

Either download and extract the starting framework from the course website or clone the assignments repository from Github.

The starting framework contains two folders:

framework This folder contains the modules which you need to modify in order to implement the minimal requirements, as well as the cabal file necessary to build the game.

executables This folder contains executables for Windows, OS X and Linux of a version of the game that already implements all of the minimal requirements.

1.2 The Gloss library

The home page of Gloss can be found at `http://gloss.ouroborus.net/`. It contains some instructions on how to install the Gloss library and solve common problems. On the machines we tested the library with (including those at the university's computer lab) we only needed to run the command:

```
cabal install gloss
```

More interesting is the library's documentation, which can be browsed at `https://hackage.haskell.org/package/gloss-1.8.1.2`. In particular you'll likely be interested in the modules:

Graphics.Gloss (functions that create the main window and handle events);

Graphics.Gloss.Data.Picture (combinators for drawing pictures);

Graphics.Gloss.Data.Color (helper functions for working with colors);

Graphics.Gloss.Geometry.Angle (helper functions for converting between degrees and radians).

Finally, you may want to have a look at `https://hackage.haskell.org/package/gloss-examples`, a package which contains numerous example program written using the Gloss library.

1.3 Compiling and running the starting framework

The compile the starting framework, go into the `framework` folder (not the `src` folder!) and type:

```
cabal install
```

To run the program, type:[1]

```
lambda-wars
```

At this point the program will crash, because not all the required functions have been implemented yet.

2 Overview

The game is a variant of the game *Asteroids*. The player controls a space ship that can move through a region of space, shoot at enemies to gain points, and pickup bonus objects that increase the score multiplier. This is all accompanied by some exciting visual effects.

2.1 Modules

The starting framework follows the model–view–controller pattern. The game is divided into the following modules:

Model This modules contains the data type definitions that are used to represent the game state.

View This module uses the game state to render a picture.

Controller.Event This modules handles keyboard events (by queuing them in the game state). You will not have to change much, if anything, in this module.

Controller.Time This modules specifies what needs to be done on each frame update. It handles the queued input events and updates all state that needs to evolve with time.

[1] Or "`lambda-wars` *`width height`*" to run the game in full screen mode.

3 Requirements

The requirements are separated into *minimal requirements*, those that you have to implement in order to receive a passing grade—assuming your coding style is not too awkward, and *optional requirements* that you can implement in order to receive a higher grade. Grades do not scale linearly with the amount of features implemented: in order to improve your grade further, you will have to do increasingly more work.

Also look at the supplied executables of the game. They may make the requirements clearer than the textual requirements given here.

3.1 Minimal requirements

Player movement The player's space ship should be able to rotate to the left and right, and be able to thrust forward.

Enemy spawning and movement Enemies should spawn randomly in space and move towards the player's ship. If an enemy touches the player's ship, the ship should blow up and the score multiplier be reset to one.

Shooting The player should be able to shoot. If a bullet hits an enemy or bonus object that enemy or object should be destroyed.

Score keeping The score multiplier should increase by one for each bonus object the player picks up, and the score itself should be increased by the score multiplier for each enemy the player shoots.

Particle effects If the player's ship is destroyed it should explode using a nice visual effect. If the player thrusts, an exhaust trail should be left behind.

Background A star field should be drawn in the background. Stars should have depth, which is made visible by *parallax scrolling*.

3.2 Optional requirements

Here are some suggestions for additional features that you can add to the game. You can also come up with your own ideas.

More enemy types Add multiple types of enemies with various kinds of appearance and behavior. Spawn them in an interesting pattern.

High scores Add a high score table to the game that is saved and read from disk. This obviously also requires limiting the number of lives a player has.

Multi-player Add a multi-player mode to the game. You will probably want to have a look at the Network, Network.Socket or one of the other networking libraries on Hackage.

Menu Add a menu to the game that allows you to view the high scores and/or select the number of players, difficulty setting or level. It is probably easiest to model this as some kind of finite automaton, so you can keep a nice separation between model, view and controller.

4 Documentation

Include a PDF file together with your submission, documenting the changes you've made to the starting framework and features you've implemented. Many students underestimate the importance of writing proper documentation, but without proper documentation we might miss some of the features you've implemented during grading and consequently you will miss out on some points. That would be a real pity!

5 Hints

5.1 Record syntax

The game state is represented as a record. To keep your code elegant you should know how to work with records effectively. Haskell has some convenient syntax for working with records, especially if you have enabled the *NamedFieldPuns* and *RecordWildCards* language extensions (as has been done in the starting framework).

Pattern matching on a record To pattern match on a record and bring two of its field into scope, write:

$$foo\ (World\ \{field1, field2\}) = bar\ field1\ field2$$

To pattern match on a record and bring all of its field into scope, write:

$$foo\ (World\ \{..\}) = bar\ field1\ field2$$

To pattern match on a record and also give the whole record a name at the same time, write:

$$foo\ world@(World\ \{field1, field2\}) = bar'\ world$$

"Updating" a record To update some fields in a record, write:

$$\begin{aligned} &foo\ world@(World\ \{field1, field2, field3\}) \\ &\quad = world\ \{field1 = 42, field2 = bar\ field1\} \end{aligned}$$

Note that the argument passed to *bar* is the original value of *field1* in the record *world*, and not the value 42 in the newly constructed world object!

Accessing a field that is not in scope If you need to access a field that is not in scope, write:

$$foo\ world@(World\ \{field1\}) = bar\ field1\ (field2\ world)$$

5.2 Advanced approaches

Instead of using records to keep track of your game state, there are also some other approaches that may work. These are more advanced techniques and not necessary to be able to get the game working. We mention them here, but we'll leave it up to you to figure out how these approaches work for yourself.

Monads Instead of passing the game state explicitly, you can hide it inside an appropriate (state) monad and abstract the state updates with a suitable monadic interface. This has the advantage that some parts of the state (such as the random number generator's seed) can be updated automatically by the monad, while you only have to worry about the important parts. The potential disadvantage is that this will give your program a much more imperative feel.

Lenses Record syntax works reasonably fine when you have flat records. Once you start working with nested records, it can quickly become a pain, though. One of the solutions proposed for this problem are lenses. There are several implementations of lenses: `https://hackage.haskell.org/package/lens` is one of the more popular ones, but has a very steep learning curve; `https://hackage.haskell.org/package/data-lens` and `https://hackage.haskell.org/package/fclabels`[2] are less powerful, but easier to use.

Some tutorials on working with lenses can be found at:

- `http://www.haskellforall.com/2012/01/haskell-for-mainstream-programmers_28.html`
- `https://www.fpcomplete.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial`
- `http://fvisser.nl/post/2013/okt/1/fclabels-2.0.html`

But again, this is all advanced stuff and not necessary for you to know or use in order to be able to finish the assignment.

5.3 Cabal

If you want to add extra modules to the framework, you will also have to list them in the cabal script (`framework/lambda-wars.cabal`). For more information on cabal, see `http://www.haskell.org/cabal/users-guide/developing-packages.html`.

[2] Written by former students from Utrecht University, who now work at a start-up that develops web applications using Haskell.