Otto-Friedrich-Universität
Bamberg

Lehrstuhl für Grundlagen der Informatik

Topic:

Haskelloids

Realization of a 2D Arcade Game in Haskell using
Functional Reactive Programming

Submitted by:

Stanislav, Christian

Advisor: Manuel Bärenz

Professor: Dr. Mendler

Bamberg, Monday 26th September, 2016

Contents

Acronyms

2D	two dimensional. 1, 15
AI	artificial intelligence. 1, 4
FP	Functional programming. 1, 13
FRP	functional reactive programming. 1, 2, 15
IMP	imperative programming. 1, 13
IO	input output. 11
UI	user interface. 4, 11

1 Introduction

Functional programming (FP) goes back to the 1980's or even to the 1930's, depending on the definition[Hudak, 1989]. In short, instead of telling a program what to do in what order (imperative programming (IMP)), FP goes about and tells the program what the desired output is and what to do in which cases. It is virtually side effect free, if done right. functional reactive programming (FRP) on the other hand is around since the very end of the 20th century [Wan and Hudak, 2000]. The *reactive* in FRP describes the connection between values and time. Imagine a ball that you drop from a certain height. It's distance to the ground can now easily described in dependence of time as each point in time corresponds to a certain distance and can be described by all moments before it[1].

1.1 Goal

The goal of this project was to implement a simple game with graphical output using functional reactive programming. We chose to aim for the clone of an all time classic: *Asteroids*[2]. In it, the player steers a space ship and tries to destroy asteroids and enemy space ships while trying not to be destroyed by them instead. This choice was based on the facts that firstly, *Asteroids* was a rather simple graphical game, being two dimensional (2D) and therefore seemed achievable and secondly, still offered multiple interesting fields like graphics, physics, artificial intelligence (AI), and general game logic. Still, as we knew from previous experience, there existed a couple of risks (see Section 1.3), which is why the ultimate goal was rather arrive at a proof of concept stage rather than a finished game (although the latter would be even better of cause).

1.2 Requirements

Apart from following time constraints, the finished project should contain the following objects:

Ship: The main game object, the player can control via keyboard input. It can rotate, accelerate, decelerate, and fire **projectiles**.

Asteroid(s): Most common game object that moves through space and is capable of destroying the **ship** by bumping into it. It can be destroyed (or possibly split into smaller pieces) if hit by a **projectile**.

Projectile(s): Can be fired by a **ship's** gun. They destroy what they hit on impact.

Enemy Ship(s): They navigate through space independently with some form of primitive AI. They can harm the player's **ship** on collision or impact of their **projectiles**.

[1] see http://paulstovell.com/blog/reactive-programming (*Paul Stovell*), last visited on 2016–9–23 for a great explanation

[2] see http://www.rawbw.com/~delman/pdf/making_of_Asteroids.pdf (*Atari*), last visited on 2016–9–20

Additionally to the basic functionality described as part of the object definitions, the finished project should provide the following:

Game: The project should have an objective and therefore should be more than a toy. In this case, it is to survive as long as possible. If time allows for it, game logic like levels, points (for shooting **asteroids** or **enemy ships**) should be added.

Graphics: The objects described above should be represented by rather primitive geometric forms. Also, there should be a loading screen and information about the game (time etc.) displayed on the screen. All graphics should be fluent.

Also, we were required to use two frameworks, *Yampa* and *OpenGL*, which are explained in Sections 1.2.1 and 1.2.2 and the programming language *haskell*.

1.2.1 Yampa

Yampa is "*a domainspecific language for [FRP] in the form of a self-optimizing, arrow-based Haskell combinator library*" [Nilsson, 2005]. Its main concepts are best described in the literature in [Courtney et al., 2003][3]. It handles values over time through *signals*. They define a *signal* to be a function from time to value. A function from one *signal* to another, they call *signal function*. They also introduce a semantically fitting analogy to the *maybe* monad, called *event*. These *events* are then used to *switch* from one execution into another and depict triggers of some sort. All of these are put together into a program by the main loop that is called *reactimate*, but is highlighted in more detail in Section 4.

1.2.2 openGL

The *Open Graphics Library*[4] or *openGL* is a library for – who might have guessed it – graphics. It was originally published in *C* in 1992 and made available for *haskell* in 2006[5]. *OpenGL* itself is a state machine, internally representing how it should currently operate in a state. To use it, one must first initialize it, add something that can be drawn onto (a window including a viewport), and draw away[6].

1.3 Risks

Apart from the usual software development risks described in [Hoodat and Rashidi, 2009], we evaluated them to include but are not limited to:

Human resources: As this project is a group effort, it involves multiple (originally three) students. One, if not the biggest, risk factor is always that some student(s)

[3] see https://github.com/rlupton20/yampaTutorial/blob/master/yampa.pdf (*Richard Lupton*), last visited on 2016–9–23 for a more comprehensible explanation

[4] see https://www.opengl.org/ (*Silicon Graphics Inc.*), last visited on 2016–9–23

[5] see https://hackage.haskell.org/package/OpenGL (*Sven Panne*), last visited on 2016–9–23

[6] see http://www.learnopengl.com/ (*Joey de Vries*), last visited on 2016–9–23 for a really detailed explanation

might quit the group and project. If the case, it is always associated with some work overhead.

Communication: The group members did not know each other prior to the project. To reduce risks of miscommunication, the group settled early on to use *gitter*[7] due to its connectedness with *github*[8] and *skype*[9] as a fallback.

Documentation: It was realized early on, that some of the required libraries or frameworks had rather poor if existent documentation. That lead to setting goals rather low and calculating with extra effort.

Expertise: Or rather the lack of it was considered another main risk as none of the team members had more experience in functional programming than a couple dozens of hours.

[7] see https://gitter.im (*Troupe Technology Ltd.*), last visited on 2016–9–23

[8] see https://github.com (*Tom Preston-Werner, Chris Wanstrath, PJ Hyett*), last visited on 2016–9–23

[9] see https://www.skype.com (*Microsoft Corp.*), last visited on 2016–9–23

2 Organization

This section gives an overview over our approach to the overall project and our organization while doing so. While Section 2.1 describes how we proceeded, Section 2.2 shows who was responsible for what. We would like to quickly mention some technologies, we used. For a version control system and code cooperation tool, we used *git*[10] on *github*[11]. Due to its great integration into *github*, we used *gitter*[12] as communication tool. It allows for direct linking of issues or commits and simple code syntax highlighting. This report was written in LaTeX[13] with *Overleaf*[14].

2.1 Work Flow

The first few weeks of the project were used to find a topic and familiarize ourselves with frameworks, libraries and each other. After deciding a goal, we started with our "*regular*" work flow: We met once or twice a week in short meetings, the protocols of which can be found on *github*[15]. There we discussed our progress, made decisions about the further proceedings, created new tasks to work on and planned the next meeting. Sometimes, we did work during the meetings as well, especially if our adviser was present. One could say that our procedure was inspired by *scrum* as presented in [Schwaber, 1997].

The previously mentioned tasks were usually small chunks of work, easily doable in about two weeks or less. We organized our work by creating an *issue* for them on *github* in our meetings. Then, we created a new *branch* of the project for each issue, worked on it (mostly from home), *committed* the changes, *pushed* them, and created a *pull request*. The other person than had a look at the request for the *master branch* and accepted it if it was okay or commented on it if it was not.

2.2 Work Distribution

We tried to evenly distribute the work throughout the semester and between the members of the team. For the sake of completion, we will list the areas of responsibility:

Stanislav: Implementation (most of graphics, AI, user interface (UI), random generation of objects and levels, game logic), report

Christian: Project management, organization, documentation, report, implementation (main frame of the project, animation, integration of *Yampa*, most of physics, most of the signal functions, most of user input)

[10] see https://git-scm.com/ (*Linus Torvalds*), last visited on 2016–9–23
[11] see https://github.com (*Tom Preston-Werner, Chris Wanstrath, PJ Hyett*), last visited on 2016–9–23
[12] see https://gitter.im (*Troupe Technology Ltd.*), last visited on 2016–9–23
[13] see https://www.latex-project.org// (*Leslie Lamport*), last visited on 2016–9–23
[14] see https://www.overleaf.com (*Writelatex Ltd*), last visited on 2016–9–23
[15] https://github.com/turion/haskell-asteroids/wiki/meetings

3 Design and Architecture

Having only little experience in *haskell* programming and especially whole projects, instead of planning everything out in the beginning and then simply follow the plan by implementing, as one could do with sufficient knowledge of the programming language and project, we progressed step by step and rather tried to comply to a set of rather general principles and guidelines.

We took an object oriented approach as described in [Rumbaugh et al., 1991]. Further we followed the design principles of high cohesion and loose coupling as depicted in [Pressman, 2005]. That lead us to a project containing encapsulated files each of which is responsible for a certain part or module of the program. To handle our dependencies (also to third party libraries) and build our project, we used *cabal*[16].

[16] see https://www.haskell.org/cabal/ (*Isaac Jones*), last visited on 2016–9–24

4 Realization

This section gives a more detailed overview of what we did with respect to implementation.

4.1 Main Frame

As the project overall was a small game with only little functionality because it was meant as a proof of concept, putting the main frame of the game together was the biggest issue, especially because both, *Yampa* and *OpenGL* had to be integrated into one program and cooperate nicely instead of hinder each other.

4.1.1 Yampa and Signal Functions

The previously mentioned *reactimate* function consists of different parts[17]:

```haskell
reactimate :: IO a                           -- init
  -> (Bool -> IO (DTime, Maybe a))          -- input/sense
  -> (Bool -> b -> IO Bool)                 -- output/actuate
  -> SF a b                                 -- process/signal function
  -> IO ()
```

Here, **init** represents an initial **IO** action that is performed prior to the **process** and yields in the first **input**. The **input/sense** should return another input sample and the time that passed since its last execution. If no new sample is provided, the last one is used. **output/actuate** handles the output as in it renders game objects or something similar. It also decides whether the program should terminate or continue. The **process/signal function** is the piece of code that is actually animated.

Alternatively to this function, one can also use *react* and *reactinit*[18]:

```haskell
type ReactHandle a b = IORef (ReactState a b)

react :: ReactHandle a b
  -> (DTime, Maybe a)                                  -- (time, input)
  -> IO Bool

reactInit :: IO a                                      -- init
  -> (ReactHandle a b -> Bool -> b -> IO Bool)         -- output/actuate
  -> SF a b                                            -- process
  -> IO (ReactHandle a b)

-- example usage (pseudo code) assuming other functions exist
react (reactInit initialization (actuator output) process) (time, input)
```

Here, the same building blocks are used in a slightly different way to allow for better combination with *OpenGL*.

[17] see https://wiki.haskell.org/Yampa/reactimate (*Unknown author*), last visited on 2016–9–25
[18] see https://hackage.haskell.org/package/Yampa-0.10.5/docs/FRP-Yampa-Simulation.html (*Unknown author*), last visited on 2016–9–25

4.1.2 OpenGL

The important concepts here are the initialization which we took care of in a `main` in **Main.hs** and `initGL` in **Graphics.hs**, and multiple callbacks that handle different parts of the program. The user input is processed in a `keyboardMouseCallback` which we implemented like follows[19]:

```
keyboardMouseCallback $= Just (\key keyState modifiers _ -> handleInput
    window gameState resetTriggered input $ Event $ KeyboardInput key
    keyState modifiers)
```

Here, `handleInput` takes a couple of parameters including the actual `KeyboardInput` which then are interpreted and according actions are taken, like the transformation of a generic input to one known to the system and saved in an **IORef**[20].

The `idelCallback` contains what was described in Section 4.1.1, ergo takes care of the animation, while the `displayCallback` takes some of the output from there and draws it on the screen (see Section 4.5.

Finally, there is the `mainloop` which is part of *OpenGl* and controls the program flow. It makes sure that all different callbacks are called properly and therefore ensures the continuance of the program.

4.2 Animation

The main function of the game is this:

```
game :: GameLevel -> SF UserInput GameLevel
game    iLevel       = gameLoop iLevel `switch` (game . uncurry3
    applyEvents)
```

Here, `gameLoop` is the main *signal function*, that takes care of animation as long as no additional objects need to be added (e.g. by firing a new projectile) or one of the existing objects needs to be removed (e.g. by an explosion). Both of these events would cause an *event* which would cause a *switch* into `applyEvents`, where they would be handled before continuing.

```
gameLoop :: GameLevel -> SF UserInput (GameLevel, Event (GameLevel,
    ExplosionEvents, FireEvent))
gameLoop iLevel = proc (input) -> do
     rec
          (correctionEvents, explosionEvents) <- iPre ((noEvents iLevel),
              []) -< collideAll level
          level  <- animateManyObjects iLevel -< (correctionEvents, input,
               lastLevel)
          lastLevel <- iPre (iLevel) -< level
```

19 The code samples in this section are taken directly from our source code and therefore not formatted ideally for this report, thank you for understanding

20 see https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-IORef.html (*Unknown Author*), last visited on 2016–9–25

```
7         -- definition of level as an Event depending on input, left out due to brevity
8     returnA -< (level, event)
```

Here, collisions are detected and categorized in line 4, while the next line takes care of the actual animation and line 6 is an initialized delay within a recursive block. Following you will find a closer look at the animation process:

```
animateManyObjects :: GameLevel ->                      SF (CorrectionEvents, UserInput, GameLevel) GameLevel
animateManyObjects    (GameLevel [])                    = arr $ const $ GameLevel []
animateManyObjects    (GameLevel (iObject:iObjects)) = proc ((event:events), input, lastLevel) -> do
    object              <- animateGameObject iObject                      -< (event, input, lastLevel)
    (GameLevel objects) <- animateManyObjects (GameLevel (iObjects)) -< (events, input, lastLevel)
    returnA             -< GameLevel (object:objects)


animateGameObject :: GameObject ->                                  SF (Event CollisionCorrection, UserInput, GameLevel) GameObject
-- animation of ships is almost identical and missing due to brevity
animateGameObject (GameObject iLocation iVelocity iOrientation id iGameObjectType) = proc (collisionCorrection, userInput, lastLevel) -> do
    let input = handleType iGameObjectType lastLevel
    orientation      <- (iOrientation+) ^<< integral -< turn input
    let acc = acceleration input *^ Vector (-sin orientation) (cos orientation)
    velocity         <- (iVelocity ^+^) ^<< impulseIntegral -< (acc, deltaVelocity <$> collisionCorrection)
    preTorusLocation <- (iLocation ^+^) ^<< impulseIntegral -< (velocity, deltaLocation <$> collisionCorrection)
    let location = torusfy preTorusLocation
    returnA          -< GameObject location velocity orientation id iGameObjectType where
     handleType objectType level = case objectType of
          EnemyShip      -> aim id level
          Asteroid _ _ _ -> rotateAsteroid objectType
          _              -> UserInput 0.0 0.0 False
```

Many objects are animated by recursive animation of all objects separately within a list of all objects (contained in a `Level`). For the animation of an individual object, orientation, velocity, and location are then calculated via `integral`s or `impulseIntegral`s supplied by *Yampa*.

4.3 Physics

The most interesting part of physics was collision. In it, for any two objects, we check whether they overlap and if so, depending on the type of the objects, they either recoil (if both are asteroids) or explode (otherwise). This process makes use of the previously introduced *events*. Either there is no event (no overlap), or there is one that represents an explosion or collision. For both, there are additional calculations necessary like the exact location of the explosion and its size or the new force input upon the two asteroids necessary to recoil realistically. The latter is done in the following code section:

```haskell
collideAsteroids :: GameObject -> GameObject -> Event CollisionResult
collideAsteroids object other
    | overlap object other = Event (Correction (
      objectCollisionCorrection, otherCollisionCorrection))
    | norm difference < 0.00000001 = NoEvent
    | otherwise               = NoEvent where
        -- calculate collision normal
        difference = location object ^-^ location other
        collisionNormal = FRP.Yampa.VectorSpace.normalize difference

        -- calculate parts of v1 that collide and the remainder
        v1 = velocity object
        v1Dot = dot collisionNormal v1
        v1Colliding = v1Dot *^ collisionNormal
        v1Remaining = v1 ^-^ v1Colliding

        -- calculate parts of v2 that collide and the remainder
        v2 = velocity other
        v2Dot = dot collisionNormal v2
        v2Colliding = v2Dot *^ collisionNormal
        v2Remaining = v2 ^-^ v2Colliding

        -- calculate results of the actually colliding parts via an
           inelastic collision
        v1PostCollision = v2Colliding ^-^ v1
        v2PostCollision = v1Colliding ^-^ v2

        -- add the remaining velocities not involved in the collision
        deltaV1 = v1PostCollision ^+^ v1Remaining
        deltaV2 = v2PostCollision ^+^ v2Remaining

        -- calculate the location correction
        distance = norm difference
        radiusSum = radius (gameObjectType object) + radius (
           gameObjectType other)
        correction = radiusSum - distance
        deltaL1 =    (correction * 4) *^ collisionNormal
        deltaL2 = (-correction * 4) *^ collisionNormal

        objectCollisionCorrection = CollisionCorrection deltaL1 deltaV1
        otherCollisionCorrection = CollisionCorrection deltaL2 deltaV2
```

4.4 User Input

The program can handle the following user input:

Quit: The game can be quit by pressing "*q*".

Rotate: The player's ship can be rotated clockwise by pressing the right arrow key or counterclockwise by pressing the left arrow key.

Accelerate: It can also be accelerated by pressing the arrow up key or decelerated by pressing the arrow down key.

Other: It can also register the input of firing a projectile by pressing "*b*", activate a shield (currently only an additional graphic) by pressing the space bar or resetting the game via "*r*".

This is achieved by the following code called from the `keyboardMouseCallback` described earlier.

```haskell
handleInput :: Window -> IORef GameState -> IORef Bool -> IORef UserInput -> Event KeyboardInput -> IO ()
handleInput window _ _ _ (Event (KeyboardInput (Char 'q') (Down) _)) = destroyWindow window
-- ... resetting and shields are handled here, but left out due to brevity
handleInput _ gameState _ gameInput userInput = do
    gs <- readIORef gameState
    writeIORef gameState $ GameState (level gs) (lifeCount gs) (score gs) (shields gs) False
    oldInput <- readIORef gameInput
    writeIORef gameInput $ UserInput (parseAcceleration oldInput userInput) (parseOrientation oldInput userInput) (parseFire oldInput userInput)
    return ()
```

Acceleration, Orientation and Fire are parsed in separate functions that either produce new values or continue to return the previous values if no user input was detected as displayed in the next example:

```haskell
parseOrientation :: UserInput -> Event KeyboardInput -> Orientation
parseOrientation _ (Event (KeyboardInput (SpecialKey KeyRight) (Down) _)) = (-1.0)
parseOrientation _ (Event (KeyboardInput (SpecialKey KeyLeft) (Down) _)) = 1.0
parseOrientation _ (Event (KeyboardInput (SpecialKey KeyRight) (Up) _)) = 0.0
parseOrientation _ (Event (KeyboardInput (SpecialKey KeyLeft) (Up) _)) = 0.0
parseOrientation oldInput _ = turn oldInput
```

4.5 Graphics

The visualization was handled in the **Graphics.hs** file. It can be structurally divided into 4 parts: viewport settings, drawing primitive objects, drawing game objects and advanced visualization.

The viewport settings consist of the functions that are responsible for the game window. For it's creation and viewport size, which is set to always be squared and be placed in the center of the full screen window.

The drawing primitive objects section contains the functions for the visualization of the circles and polygons by coordinates, without using the actual objects.

The most important section "drawing game objects" has the core function `drawScreen` which divides the visualization into two parts: UI elements and the game objects themselves, i.e. ships and asteroids. For each of those objects the `drawGameObject` function is called that accordingly to the object type draws it on the starting coordinates and then using the preservation matrix it rotates, scales and re-positions it to the desired measure.

The last section advances visualization is responsible for displaying the text, that is used in the loading screen and in UI, as well as for showing the border. Also to give the detailed look to player's ship and enemy ships an interesting technique was used. First the mock-ups were drawn in the paint and then the coordinates of the vertices were written down to be then recalculated using `recalculateCoords` function from the pixels to the coordinates of the reference system that was used in the project. And since all the ships are symmetrical Only one half of the points was needed and the other half was then just mirrored nearby with the help of `copyAndReverse` function.

4.6 Random Generator

In our project we used the random numbers for generating the game level which consists of the known amount of enemy ships, asteroids and a player ship. And for all that only one standard generator was used, that was created at the start of the program in the `main` function and then it was passed along to the next function that used it to make a random number and to create the next standard generator to pass it further. So if for example we would execute the program twice and we will use the exact same generator in the beginning , then every random number from the first run would be equal to the ones from the second, thus making our functions in the Generator.hs file pure, with no side effects, without the use of input output (IO) actions.

When the generation process started at first the player ship was created always in the middle of the screen. Then the first asteroid was created (given that the desired level should have several asteroids and enemy ships) with the random position, rotation and velocity. And also the shape that could be one of the three different sizes and it always had 8 edges randomly placed in such a manner so the asteroid would look like an unformed octagon. After the first asteroid was generated it was checked if it overlapped with the previously created objects, in this case - player's ship. And if it was, then the new asteroid would be generated, until no overlaps occur. Then the second asteroid is generated and so on, when all the asteroids are created, then the same procedure is done for the enemy ships, thus making the entire level with no objects colliding with each other.

4.7 Artificial Intelligence

The enemy ships in our game were equipped with the primitive intelligence that allowed them to do the following actions: slowing down if moving too fast, avoiding other game objects and aiming at the player. Those actions are done in the priority order they are listed in, i.e. the enemy will try to aim only if it moves slowly and there are no approaching objects that can collide with it. All those action are tricky because of the way the acceleration works in the game, the ship can be facing one direction, but moving in another and it can only thrust forwards or backwards.

To check if the ship needs to slow down we first use `speedTooFast` function from AI.hs file. If ship's speed is high, then if it is already facing the direction it is moving, it will slow down, but if it's not, then before slowing down the `fastTurnInTheSpeedDirection` function is used until it faces the correct direction.

The similar technique is used while avoiding other approaching objects. First of all every other object in the game is checked if it meets the three conditions: it is not a player ship, it is not far away from the ship that is trying to avoid collisions and if it does not change its moving direction and speed the collision will occur (that is done using `doObjectsCollide`). If all of those conditions are met then the ship will commence the avoiding procedure: if it is already facing the direction opposite to the closest approaching object it will speed up, trying to get far from the danger, if it is not facing the correct direction, it will turn until it does to only then thrust forward.

Now when there is no danger of collision and the speed is not high, the enemy can continue pursuing the player: it will aim at it and it can also start slowly approaching the player if the positive value of `approachingSpeed` is set. The aiming is simple, it checks where the player is and turns in the player's ship direction. All the turns are done either clockwise or counterclockwise, depending on which way is shorter.

5 Retrospective

Overall, this was a fun project that unfortunately did not go as well as anticipated.

5.1 Group work

We noticed, that great team work is harder when every person works for themselves and remotely instead of coming together and working as a team collectively. Due to that we figure that it would have been better to meet more often and more regularly than we did. Also, integration of actual work phases as part of the meetings would have helped. It probably would have also lead to a more balanced division of work.

5.2 Lessons Learned

What better way to document lessons learned, than a list?

Expertise: Some tools require certain degrees of experience before actually being helpful. That was especially the case with *Yampa*, mostly because it is so poorly documented on the internet with an aim mostly for developers who are already experts in the field. We did not really feel ready for that yet at the beginning of the project. Now, afterwards, we somewhat feel like understanding its concepts to a fuller extend, but we figure we could have gotten further within the project if having used another, simpler and well documented library instead. While we do appreciate the laissez-faire approach of the adviser, we feel like more guidance would have been desirable in this case.

Documentation: We were used to IMP prior to this project and according best practices. For example, if writing good code, no comments explaining the code should be necessary in most cases (see [Martin, 2009]). However, because many functions in FP are of a more general kind than in IMP, the reader of code has to do a lot more mental work, which is why better documentation would have been a good idea. Another observation we made was that a lot of code out there is poorly documented and therefore almost unreadable to beginners in the field.

Refactoring: We noticed that we mostly rewrote existing code instead of adding new code. This was also different from our prior experiences. It made us think more about the code and its effects prior to writing it.

Babysteps: We are very happy with our incremental approach that we persisted throughout the project. It allowed for us to work on multiple problems simultaneously.

Issue tracking: Representative for the whole organizational work, we are very content with our way of issue tracking, as well. It helped to focus on certain problems, prioritize them, divide work, plan further proceedings, and track our progress.

6 Outlook

This section gives a brief outlook on the work that would have followed if more time would have been available or that could follow in another project.

6.1 Improvements

There exists a known bug in collisions that leads to irregular behavior on some systems but was not reproducible on our systems and therefore hard to fix. Due to its rarity and harmless effect on the overall game, we decided to rather focus on other functionality.

However, we would like to refactor the code before any follow up project. Especially our code from the early stage still has a certain potential for improvements. Some naming could be improved to make it more comprehensible and some splitting for separation of concerns is possible.

6.2 Extensions

We think, the following extensions would be a great addition to our project:

Enemy shooting: Currently, only the player's ship can shoot which makes the game unbalanced or really hectic (if the difficulty is increased by just making everything faster). Letting the enemies shoot would fix this problem and allow for some interesting extensions.

Levels: As there so far only really is one level, one can easily beat the game. With the introduction of levels, one could increase the difficulty to make fore a more interesting experience.

Score: With the introduction of an actual score (e.g. for surviving, destroying asteroids and enemies), the game would become more competitive and the introduction of a score board would increase the replay likeliness.

Saving: Adding the possibility of saving and loading games would probably only make sense if there are certain phases in the game that are clearly less hectic than mid-battle. Especially, if there are many stages to the game with different objectives and an increased difficulty, this would be an interesting addition, even though it does not really comply with arcade games.

Variations: Once there is a better infrastructure to the game, one should make it more interesting by adding variations. The most promising of our ideas was to introduce different kinds of asteroids that needed special kinds of weapons to destroy and power-ups that either the player or enemies could collect to access these weapons.

Game Modes: Another interesting addition would be the introduction of other game modes like exploration or harvesting. Instead of being a complete alternative main objective, these could be also integrated into the game as mini-games (e. g., to access new weapons).

7 Conclusion

We wanted to create a simple 2D game in an arcade style with FRP and with the help of *Yampa*, *haskell* and *openGl*, did so to a certain extent. We were surprised how little code was necessary to actually get started and have first results on the screen. However without prior experience, it still took us quite some time and we never really became fond of *Yampa*. Even though we can now see that it offers many great possibilities, it seems rather complicated to begin with.

References

Courtney, A., Nilsson, H., and Peterson, J. (2003). The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM.

Hoodat, H. and Rashidi, H. (2009). Classification and analysis of risks in software engineering. *World Academy of Science, Engineering and Technology*, 56:446–452.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Nilsson, H. (2005). Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP*, volume 5, pages 54–65.

Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. E., et al. (1991). *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ.

Schwaber, K. (1997). Scrum development process. In *Business Object Design and Implementation*, pages 117–134. Springer.

Wan, Z. and Hudak, P. (2000). Functional reactive programming from first principles. In *Acm sigplan notices*, volume 35, pages 242–252. ACM.