Masaryk University
Faculty of Informatics

Game development in Haskell

Bachelor's Thesis

Jan Rychlý

Brno, Spring 2021

Masaryk University
Faculty of Informatics

Game development in Haskell

Bachelor's Thesis

Jan Rychlý

Brno, Spring 2021

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jan Rychlý

Advisor: doc. Mgr. Jan Obdržálek, PhD.

Acknowledgements

These are the acknowledgements for my thesis, which can span multiple paragraphs.

Abstract

This is the abstract of my thesis, which can span multiple paragraphs.

Keywords

Haskell, functional paradigm, game development, Apecs

Contents

List of Tables

List of Figures

Introduction

// introduction draft written for the VB000 assignment

Video games are a special kind of application that many consider an art form and rewarding to develop. However, they generally involve a complex system with a non-trivial state, a certain amount of pseudo-randomness, and user/player input handling. This makes for non-deterministic programs that are usually incredibly difficult to test efficiently.

Conversely, functional programming strives to eliminate mutable state and make code more deterministic, which allows for programs to be safer and easier to test. These and other benefits have naturally led to people trying out game development in functional languages, but it remains mostly a matter of passion projects. That said, even though the vast majority of the video game industry still uses imperative languages like C++, the communities *are* very active, and there are hundreds of games, blog posts, and libraries that help with game programming in functional languages.

The focus of this thesis narrows down to exploring game development in Haskell in the context of small-scale 2D games. The goal is to give an overview of the process, then compare this approach to a more conventional and imperative one and ultimately highlight the features of Haskell that are beneficial and those that become hurdles in the context of programming a video game.

This is done through reimplementing a single game with an already existing imperative implementation in Haskell, first using the Apecs[1] library and for a second time without it. After a further discussion about chosen technologies in the following chapter, said three implementations are described and analyzed in chapters 2, 3, and 4. Then they are more closely compared and the pros and cons of Haskell in game development are evaluated and demonstrated in chapter 5.

We find that the Apecs library makes developing games in Haskell much more approachable. On the other hand it goes against the functional philosophy, and using it will generally result in very imperative

1. CARPAY, Jonas. *apecs: Fast Entity-Component-System library for game programming* [online] [visited on 2021-04-22]. Available from: `https://hackage.haskell.org/package/apecs`.

code wrapped in monads that lacks the expressiveness and apparent safeness of regular Haskell. Yet, from the second reimplementation, we learn that some use of monads is beneficial, and it makes the code cleaner and more elegant. In both cases, the development was mostly a smooth experience without a single major hick-up, unlike what often happens when dealing with a C++ compiler.

1 Motivation and used methods

1.1 Why functional programming matters

Functional languages are a subset of declarative languages, where the programmer states "what" instead of "how". Unlike in imperative languages, we *declare* what we want a program to return by combining functions (other declarations) instead of giving the computer serialized instructions (*imperatives*). That is manifested in the lack of assignment statements and lesser control structures. What sounds like a bug is actually a feature — once variables are assigned their value, it can't be changed, and the burden of prescribing the flow of control is removed[2]. Moreover, a pure function has no side effects and its return value depends solely on its arguments. This makes for deterministic programs that are easier to test, debug and argue about their correctness.

John Hughes describes the key benefits of functional paradigm in his paper *Why Functional Programming Matters*[2]. He first explains how modularity of code is clearly very important, since separate modules are easier to write and test and then proceeds to show how functional programming increases modularity through higher-order functions and lazy evaluation, demonstrating their importance on several examples.

Additionally, Haskell is a purely functional programming language that is also *strongly typed*. This means that one doesn't need to worry about memory errors causing crashes because everything is caught by the type-checker during compilation. The types also serve as documentation and can help greatly with writing and understanding code. On the other hand types can also be inferred by the compiler so it is not necessary to explicitly declare the type of everything. Furthermore the type system allows extensive user-defined data types, which makes the code even more expressive. All of this potentially increases productivity of a functional programmer even further.

1.1.1 Game development specifics

Functional languages are great tools but we know that not every tool is fit for every job. One of the consequences of the functional purity is that the state of the program has to be modeled explicitly as an argument and is therefore immutable. There are monads that help us abstract from this but the monads themselves are in a way still an explicit work around.

Conversely, games are real-time, interactive applications simulating often very complex systems and hold a non-trivial state that is updated many times a second. Such state generally involves a representation of the game world with all the objects existing in it, their properties and flags, the current state of the input devices and many other variables. Since its beginning, the video game industry has been dominated by imperative languages, that make it easy to model a game world and alter it globally through references and side effects inside of decomposed functions. And because of their established position, there is a plethora of libraries and game engines with supporting documentation and tutorials. Besides, a company will most likely have no problem finding skilled C++ programmers with interest in the game industry, where as finding their functional counterparts might be much harder. Additionally, in the case of C++ the performance also fits the requirements of large games.

However, it does come with a price — modules of such programs may be more dependent and entangled, which makes the whole less flexible and with implicit state more prone to bugs and harder to test and debug. That is, while testing is already a large issue due to the nature of video games. Automated testing is not sufficient and companies have to hire teams of game testers to test games manually. And in terms of high performance, which is generally connected to lower-level languages, developers must wrestle with the lower-level nature, producing problems as well. Not to mention that Haskell is well regarded in the area of parallelism and concurrency, which is becoming more relevant as new hardware keeps increasing in core counts, and could make it comparable to C++ as Haskell can already compile to very fast programs.

We can see that video games, like any other software, could benefit from purely functional design, provided that we are able to model the game state efficiently enough. Another consideration in the real world are the available frameworks and whether the cost of potential pioneering is worth to us.

1.1.2 Existing work

Indeed, people have tried developing games in Haskell and a decent progress has been made over the years. There are libraries/engines like Yampa[3] and Helm[4] for functionally reactive programming (FRP) of video games and other general FRP libraries that have been used to make games like Elerea[5] or Netwire[6]. From non-FRP libraries there is FunGEn[7], the self-proclaimed oldest Haskell game engine, Apecs, which we use to program a game and describe the process in chapter 2, and many others. It is important to note that most of these libraries or engines provide only limited capabilities compared to "real" industry engines like Unity or Unreal Engine and depending on the type and scale of the game, there is still a lot of work left for the developer.

Regarding existing games themselves, there are two — Magic Cookies and Enpuzzled — that have been commercially pulished by Keera Studios, who also stand behind the Yampa game engine[8]. Then there is Chucklefish, indie game developer studio, publisher and creator of popular Starbound, which announced to be working on their next game Wayward Tide in Haskell back in 2014[9]. However, there has not been an announcement of the release date as of 2021 and the studio is focusing on other projects at the moment.

So it remains to be a pioneering process and the games made are no where close to the rest of the industry but there *is* more games than just the stated few. They are created by passionate individuals and shared with the community. Dozes of them can be found on the Haskell game development Reddit page (`www.reddit.com/r/haskellgamedev/`) for instance. Many have also written blog posts or tutorials alongside with their games like Joe Vargas and his *A Game in Haskell - Dino Rush*[10], which goes in-depth and explains his well though out architecture, or Ashley Smith and her *Of Boxes and Threads: Games development in Haskell*[11] and *An Introduction to Developing games in Haskell with Apecs*[12], that provide great overview and inspiration.

1.2 Rendering and interfacing with the OS

Essential part of a game engine is communicating with the operating system and rendering of models or textures. To do this we can either use a complete engine like the before mentioned Helm or a library like Gloss[**?**], which aims to provide an easy to use interface for managing input and rendering. Other libraries provide only Haskell bindings to existing media frameworks like GLUT, GLFW and SDL (Gloss actually uses GLUT or GLFW for its backend). The main goal of these libraries is to abstract from a specific window system and graphics hardware, providing cross-platform APIs for rendering, managing windows and receiving input and events.

In both experiments described in this thesis we use the SDL bindings, to load textures and fonts, to poll input events, create windows and render scenes. Specifically, we use the `sdl2`, `sdl2-image` and `sdl2-ttf` packages. We choose SDL because there already are examples of its use in games we can learn from, the underlying C library works across multiple platforms, is well documented and is widely used. Moreover, the Haskell libraries include both high-level and low-level bindings, meaning we can enjoy a comfortable interface, yet at the same time the lower-level bindings serve as an example of Haskell's powerful Foregin Function Interface (FFI), which makes Haskell even more useful in the real world.

To showcase this we can look at the two most used functions in our two games: `SDL.copy` and `SDL.copyEx`. We use them to copy our loaded textures to the rendering target like stamping a picture on a canvas. Both of these functions are examples of the high-level bindings which wrap around the low-level ones: `SDL.Raw.renderCopy` and `SDL.Raw.renderCopyEx`, abstracting from the pointers, replacing them with Haskell's `Maybe`, and throwing an error if the return value is negative. Those low-level functions are bound to the C functions[1] using the mentioned FFI.[2] Listing 1.1 shows the binding of `SDL.Raw.renderCopy`.

1. The documentation of the C library: `https://wiki.libsdl.org/`
2. `SDL.Raw.renderCopy` is not the direct binding itself, it is a wrapper replacing `IO` in the actual binding `renderCopyFFI` with a `MonadIO` constraint, otherwise identical.

Listing 1.1: Example of FFI binding

```
foreign import ccall "SDL.h SDL_RenderCopy" renderCopyFFI
    :: Renderer -> Texture -> Ptr Rect -> Ptr Rect -> IO CInt
```

Other `SDL` functions used are for instance `SDL.clear`, which clears the rendering target and `SDL.present`, which displays the current state of the target in the window. We also use `SDL.createWindow` and `SDL.createRenderer` to create a rendering context for a new window at the start of the program, after which we can load fonts and images as textures using `SDL.Font.load` and `SDL.Image.loadTexture`. No less important is `SDL.pollEvents` and `SDL.ticks`, called every frame of the game to get input events, and time in milliseconds.

1.3 Asteroids by Atari as an example

We conduct an experiment by recreating Asteroids, an arcade game created by Atari in 1979, once using SDL2 and Apecs and second time using SDL2 and a more pure-style architecture without Apecs, to evaluate the strengths and weaknesses of Haskell and Apecs. To evaluate Haskell as a language for game development in general would be a task far beyond the scope a bachelor's thesis. For that reason we narrow down our focus to smaller two-dimensional games and at the end only speculate how our findings may scale to larger games. We chose Asteroids as an example because its world comprises only of few object types, yet their relationships make the game quite interesting. It also does not rely on complex graphics, therefore we can focus on the code implementing the game rules and behaviors. Finally, because it is a famous, classical and simple game, others have made their recreations in the past and we use this to compare our two functional implementations to an imperative version picked from GitHub.

The game can be described as followed: "A perfect synergy between simplicity and intense gameplay, the game has players using buttons to thrust a spaceship around an asteroid field. When one rock is shot, it breaks into smaller ones, often flying off in different directions at different speeds... Every so often flying saucers enter the screen, intent on the player’s destruction."[13] Its world comprises of rocks (asteroids), projectiles, flying saucers (large or small, trying to shoot the player) and the ship, controlled by the player, trying to survive and gain score points by shooting down rocks and flying saucers. There is a few features that we are omitting like sound effects, some animations and several minor game-play details. But we still need to handle input, simulate simple physics, detect collisions, spawn entities, keep score, transition between the game and its menus and then render everything, which proves to be enough to gain some experience and valuable insight into game development in Haskell.

2 Using the Apecs library — hAsteroids

2.1 About Apecs

Many libraries for game programming in Haskell have come out over the years. One of the more recent ones is Apecs — "a fast, type-driven Entity–Component–System library for game programming."[1] Entity–Component–System (or ECS) is a data-oriented architectural pattern often used in video game engines. It provides better performance by increasing data locality. Instead of the object-oriented pattern where data is grouped by their related **entity**/object, in ECS , we group pieces of data by their character and call them **components**. Simply put, we exchange an array of structures for a structure of arrays. That way, when running functions — which define a **system** — on entities every frame of the game, we iterate only over the arrays of components we need for the system rather than iterating over a much larger data set of whole objects as it would be in the object-oriented paradigm. Typical example of a system would be updating positions of all entities based on their velocity regardless of whether that entity is a player, projectile or a falling anvil.

[https://medium.com/ingeniouslysimple/entities-components-and-systems-89c31464240d]

And since both Unity and Unreal engine use Entity-Component design, we chose Apecs as the current state-of-the-art Haskell library for the traction it has received in the community despite it not being the only ECS library in existence[1].

Listing 2.1: Defining instance of `Component`

```
newtype Position = Position (V2 Double)
instance Component Position where
    type Storage Position = Map Position
```

1. The making of Ecstasy was actually inspired by author's issues with Apecs. https://reasonablypolymorphic.com/blog/why-take-ecstasy/

To define a component in Apecs means to define an instance of the class `Component`, as we see in the listing 2.1. The `Component` class requires us to state how we want to store the given component by assigning a type alias to the specific storage type. We can define our `Stores` or use one of those provided with the library: `Map`, `Unique`, `Global`. With `Map`, there can be multiple components of that type, each belonging to a particular entity. With `Unique`, at most one component may exist belonging to a particular entity. Furthermore, with `Global`, at most one component instance can exist, and it belongs to the special *global* entity together with every other entity. Finally, we call `makeWorld`, which uses Template Haskell to generate `World` product type along with `initWorld` function and instances of the `Has` class needed for altering contents of `World` through the other functions in Apecs. The resulting `World` may look close to something as shown in listing 2.2.

Listing 2.2: Simplified world state type example

```
data World =
    World
    { record1 :: !(Unique Player)
    , record2 :: !(Map Enemy)
    , record3 :: !(Map Bullet)
    , record4 :: !(Map Position)
    , record5 :: !(Global Time)
    }
```

Having components, we need a way to run systems. In Apecs, the `SystemT` monad transformer is where changes to the world happen. Therefore any "system" in the ECS sense must be a function returning `SystemT w m a`. One such "micro-system" is the `newEntity` function. It accepts a tuple of components and adds them into their records — the new entity itself is defined by the components stored under its ID.

```
newEntity ( Ship $ pi / 2 * 3
          , Position $ V2 0 0
          , Velocity $ V2 0 0
          )
```

More noteworthy functions to build systems are the component map functions shown in the listing 2.3. They are the means of altering the world state.

Listing 2.3: Component maps documentation[14]

```
-- 'w' is the world type, 'm' is a monad,
-- 'cx','cy' and 'c' are tuples of components

-- | Maps a function over all entities
--   with a cx, and writes their cy.
cmap :: forall w m cx cy.
    (Get w m cx, Members w m cx, Set w m cy) =>
    (cx -> cy) -> SystemT w m ()

-- | Monadically iterates over all entities
--   with a cx, and writes their cy.
cmapM :: forall w m cx cy.
    (Get w m cx, Set w m cy, Members w m cx) =>
    (cx -> SystemT w m cy) -> SystemT w m ()

-- | Monadically iterates over all entities with a cx
cmapM_ :: forall w m c.
    (Get w m c, Members w m c) =>
    (c -> SystemT w m ()) -> SystemT w m ()
```

`cmap` accepts a function that takes a tuple of components and returns some other tuple of components. It internally iterates over entities with at least those components matching the mapped function's input tuple and writes the output tuple components to those entities. `cmapM` works similarly only as its name suggests the mapped function returns the component tuple wrapped in the system monad, which allows it to execute side effects. And with `cmaM_` there is no direct writing, only side effects.

2.2 Writing of hAsteroids

To familiarize our selves further with the library and with the process of using it for its main purpose — creating games in Haskell — we did exactly so.

Module structure of the project looks roughly as shown in [Fig. X] `![img](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/45929532-7c55-4c74-ba20-41c18c5eb50b/modules.png)` Fig. X - Upper modules may have direct dependencies on the lower modules, arrows show most of the important ones.

The core of the game's design is the components, as they form the primary data structure acting as the state. There are more approaches to designing them, but in hAsteroids, we have three categories of components: marker components, shared components, and control components. Marker components serve two purposes: they contain information that is unique for a given type of game object, and that way, they also mark an entity as that object. Shared components include characteristics that are shared by more types of game objects like position. Lastly, control components are all global and are used in one way or another to control the run of the game. hAsteroids has one `Unique` marker component called `Ship`, which marks an entity representing the player's ship and stores the angle of the direction the ship is facing. The three other marker components are `Map` stored. They are `Asteroid` — holding asteroid size — `Ufo` — holding saucer size and a countdown to the next UFO's shot being fired — and `Bullet` — storing whether the player or a UFO shot it. Next, there are the shared components `Position`, `Velocity` and `TimeToLive` and several `Global` control ones like `ShipLives`, `ShipState`, `GameLoopState`, `WaveTime` and few others. []Fig. X] shows which entity types have which components.

Ship - Ship, Position, Velocity

UFO - Ufo, TimeToLive, Position, Velocity

Bullet - Bullet, TimeToLive, Position, Velocity

Asteroid - Asteroid, Position, Velocity

Now, the `main` function is the entry point of the program. It first initializes the SDL libraries and then creates a window and a renderer. Next, using `loadResources`, it loads object textures into a hash map, prerenders fonts saving them into another hash map and wraps it all together with the renderer and a few stateful random generators into one product type called `Resources`. Then the game world with our components is created by calling `initWorld`, and together with resources, they are passed to `gameLoop` through a stack of Reader monad transformers (`SystemWithResources`).

`gameLoop` is one compound system responsible for updating and drawing the world and the menus, and it loops until the player quits the game. It also measures the time every frame and calls `SDL.Delay` if it was updated and drawn too quickly for the targeted 60 FPS. World updating is split into two functions: `reactToInput` and `stepScene`. The scene (a menu or the world) is then drawn by `drawScene`.

`reactToInput` manages the state of the input, and as its name suggests, it reacts to it. Depending on the global `GameLoopState` component, it either transitions between the states (`InMenu`, `Playing`, `Paused`, `GameOver`, `Quit`) or when in the `Playing` state, it also allows the player to control the ship. That is done by a `cmapM` call with a lambda that changes the angle of the ship, increases its velocity or creates a new bullet entity — all conditioned by the input state.

`stepScene` takes care of simulating physics and game rules over time when the loop is in the `Playing` state. This is divided into multiple function calls:

- `cmap $ stepKinetics dT`

  iterates over all entities and adds their velocity vector multiplied by time `dT` to their position vector and also takes care of wrapping the space — if an entity flies out of the screen on one side, it comes back in from the other side.

- `cmap $ decelerateShip dT`

  simply applies deceleration to the ship by scaling down its velocity vector slightly.

- `cmapM $ stepShipState dT`

  is responsible for transitioning between ship states (`Alive`, `Exploding Int`, `Respawning Int` where the integers serve as countdown timers for the state transition) and the "explosion animation".

- `cmapM_ $ ufosShoot dT`

iterates over all `Ufo` components decrementing the time to shoot and when it reaches 0, it creates a new `Bullet`. The algorithm for finding a shooting direction is different for the two UFO sizes. Small UFOs are more accurate because the algorithm uses the law of sines to calculate the bullet trajectory based on the ship's current position and velocity. Large UFOs shoot in quarter of π increments towards the ship's current location.

- `awardLifeIf10000`

  increments ship's lives by 1 for reaching every 10000 score points.

- A lambda incrementing `WaveTime`

  simply adds the frame delta time `dT` to the wave time counter.

- `spawnUfos`

  randomly creates new UFO entities on the left side of the screen, with the chances increasing as the time spent in one wave (`WaveTime`) passes.

- `spawnNewAsteroidWaveIfCleared`

  uses `cfold` from Apecs to count all asteroids and when there are none it starts counting up using the `WavePauseTimer`. When the timer reaches 1500 it calls `spawnNewAsteroidWave` from the `Initialize` module, which uses the random stateful generators from the `WithResources` reader monad to create new asteroids.

- A lambda decrementing all `TimeToLive` components

  simply subtracts the frame delta time `dT` from all the time to live components.

- `destroyDeadBullets` and `destroyDeadUfos`

  use `cmapM_` to destroy all the components for entities that run out of "time to live".

- `detectAndHandleCollisions`

  is defined in its own module `Collisions`. There the collisions are detected and handled individually between each entity group. The general idea is that we use `cmapM_` inside of `cmapM_` as an equivalent of nested for loops. This way, for every asteroid, we iterate (or map) over all the other entities, checking for collision. We do the similar for the rest of the combination pairs, using in total $\binom{6}{2} = 6$ algorithms. All the collisions are detected simply as a question of "is a point or any of the points inside of a rectangle, an ellipse or a circle." A detected collision always results in some effect — the colliding entities are removed, except an asteroid may break into two smaller ones if it is not already the smallest size, and in the case of the ship, one life is subtracted.

Once the scene is stepped, it is drawn by the already mentioned `drawScene` function from the `Draw` module. If the loop state is `InMenu` or `GameOver` only text is drawn on a cleared black screen. That is done by `drawCenteredTexts`, which only calls a monadic `zipWith` with `drawCenteredText` on a list of y coordinates and a list of text keys. `drawCenteredText` itself then looks up the text texture in the hash map that is part of the `WithResources` environment, queries its width and finally calls `drawText` with the coordinates for the texture to be drawn centered.

When the loop is in the `Paused` state, there is text being drawn as well as the world. Moreover, when the state is `Playing`, only the world is drawn. This is taken care of by `drawWorld`, which calls functions to draw the background, entities and the UI (number of lives and the score). Entities are drawn using `cmapM_` with a lambda and a wrapper function around the `SDL.copy` and `SDL.copyEx`, which renders the corresponding texture for every entity.

2.3 Reflection

From this example alone, we learn several valuable lessons right away. First, Apecs makes game programming in Haskell relatively accessible. Once we understand the ECS principle, every world modification is "intuitively imperative" thanks to the `System` monad and the component map functions. Any system side effects can be added to an existing `cmap f` call with a simple change of the return types.

```
-- steer and thrust
handleInput input =
```

```
    cmap \$ \(Ship a, Velocity vel) ->
               ( Ship \$ a + steering input
               , Velocity \$ vel + thrust input a
               )
-- we realized that we also want to be able to pause the game
handleInput' input =
    cmapM \$ \(Ship a, Velocity vel) -> do
                when (wasPressed input escapeKeycode) (set global Paused)
                pure ( Ship \$ a + steering input
                     , Velocity \$ vel + thrust input a
                     )
```

Moreover, the `WithResources` reader monad provides easy access to resources without having to pass them along everywhere as a function argument.

Not less important is the fact that the game works. The development experience was smooth, with no major hick-ups. Because Haskell is a statically-typed high-level language, there is no reason to worry about random invalid memory access making our game crash, and everything is type-safe.

However, this approach is far from perfect. Handling collisions on an individual basis would scale very poorly, so some universal interface would be better. This could be achieved by defining a class and its instances for the marker components. More polymorphism could also differentiate functions that require the `System` monad, the `WithResources` monad or both. In the current state, there are many functions, such as `reactToInput`, which do not use the resources but still have access to them since everything returns the transformed `SystemWithResources` monad that has `WithResources` inside of it. Furthermore, said monad transformer stack[2] includes `IO`, so any function with this type can perform input or output effects, which goes against functional purity and nullifies many of the reasons why one would choose Haskell as a language in the first place.

Another issue we observe is partially tied to the nature of ECS — there is no way to destroy all entity's components automatically in Apecs. One has to do destroy them explicitly. That way, nothing

2. `type SystemWithResources = SystemT World (ReaderT Resources IO)` where `SystemT` is only a `newtype` around `ReaderT` defined in Apecs

protects us from accidentally adding a component to an entity that will never be destroyed. This could be mitigated by creating helper functions for entity creation and destruction.

Overall, because Apecs is so powerful, it makes it easy for the programmer to rely on it too much and produce code that does not reach all the potential benefits purely functional programming has to offer.

3 Focusing on functional purity - pure-asteroids

3.1 Game engines in purely functional style

// refferencing John Carmack's keynote at Quakecon 2013, discussing its benefits, maybe mentioning `https://indigoengine.io/`

3.2 Writing of pure-asteroids

3.3 Reflection

// sort of a immediate reflection

4 Analyzing an existing C++ implementation

// explaining a bit about the choice of the particular implementation and analyzing it, again with a "reflection" part

5 Comparing the approaches

Conclusion

Bibliography

1. CARPAY, Jonas. *apecs: Fast Entity-Component-System library for game programming* [online] [visited on 2021-04-22]. Available from: `https://hackage.haskell.org/package/apecs`.
2. HUGHES, John. Why Functional Programming Matters. *The Computer Journal*. 1989, vol. 32, no. 2, pp. 98–107. ISSN 0010-4620. Available from DOI: `10.1093/comjnl/32.2.98`.
3. NILSSON, Henrik; COURTNEY, Anotny. *Yampa: Elegant Functional Reactive Programming Language for Hybrid Systems* [online] [visited on 2021-05-01]. Available from: `http://hackage.haskell.org/package/Yampa`.
4. CORR, Zack. *Helm: A functionally reactive game engine.* [Online] [visited on 2021-05-01]. Available from: `http://hackage.haskell.org/package/helm`.
5. GERGELY, Patai. *elerea: A minimalistic FRP library* [online] [visited on 2021-05-01]. Available from: `http://hackage.haskell.org/package/elerea`.
6. SÖYLEMEZ, Ertugrul. *netwire: Functional reactive programming library* [online] [visited on 2021-05-01]. Available from: `http://hackage.haskell.org/package/netwire`.
7. FURTADO, Andre. *FunGEn: A lightweight, cross-platform, OpenGL-based game engine.* [Online] [visited on 2021-05-01]. Available from: `http://hackage.haskell.org/package/FunGEn`.
8. *Products and Services* [online]. Nottingham: Keera Studios [visited on 2021-05-04]. Available from: `https://keera.co.uk/products-and-services/`.
9. PALFREY, Jack. *Wayward Tide: Technical Info* [online]. Chucklefish [visited on 2021-05-04]. Available from: `https://chucklefish.org/blog/wayward-tide-technical-details/`.
10. VARGAS, Joe. *A Game in Haskell - Dino Rush* [online]. 2018-02-28 [visited on 2021-05-04]. Available from: `http://jxv.io/blog/2018-02-28-A-Game-in-Haskell`.

11. SMITH, Ashley. *Of Boxes and Threads: Games development in Haskell* [online]. 2018-06-09 [visited on 2021-05-04]. Available from: https://aas.sh/blog/of-boxes-and-threads/.

12. SMITH, Ashley. *An Introduction to Developing games in Haskell with Apecs* [online]. 2018-09-10 [visited on 2021-05-04]. Available from: https://aas.sh/blog/making-a-game-with-haskell-and-apecs/.

13. *Asteroids - Rockin' the Arcades* [online]. The Dot Eaters [visited on 2013]. Available from: https://web.archive.org/web/20131023233640/http://thedoteaters.com/?bitstory=asteroids.

14. CARPAY, Jonas. *Apecs* [online] [visited on 2021-04-22]. Available from: https://hackage.haskell.org/package/apecs-0.9.2/docs/Apecs.html.

Index

A An appendix

Here you can insert the appendices of your thesis.