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

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

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 code wrapped in monads that lacks the expressiveness and apparent

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`.

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 hiccup, 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. Once variables are assigned their value, it cannot 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 the functional paradigm in his paper *Why Functional Programming Matters*.[2] He first explains how modularity of code is essential, 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 does not need to worry about memory errors causing crashes because the type-checker catches everything during compilation. The types also serve as documentation and can help greatly with writing and understanding code. At the same time, the compiler can also infer types, so it is not necessary to explicitly declare them everywhere. 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 task. One of the consequences of functional purity is that the program's state 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 workaround.

Conversely, games are real-time, interactive applications often simulating complex systems and holding a non-trivial state updated many times a second. Such s state generally involves representing 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 decomposed functions. Moreover, because of their established position, there are many libraries and game engines with supporting documentation and tutorials. Besides, a company will most likely have no problem finding skilled C++ programmers with an interest in the game industry, whereas finding their functional counterparts might be much more challenging. Additionally, in the case of C++ the performance also fits the requirements of big 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 significant 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. Haskell also has the edge in the area of parallelism and concurrency, which is becoming more relevant as new hardware keeps increasing in core counts, and complex, intertwined, imperative systems are challenging to run in parallel.

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 is the available frameworks and whether the cost of potential pioneering is worth it.

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 and 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[1], 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 published by Keera Studios, who also stand behind the Yampa game engine[8]. Then there is Chucklefish, indie game developer studio, publisher, and the 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 nowhere close to the rest of the industry, but there *are* more games than just the stated few. They are created by passionate individuals and shared with the community. Dozens of them can be found on the Haskell game development Reddit page[1] 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 thought 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 a great overview and inspiration.

1. Subreddit about game development in Haskell: `http://www.reddit.com/r/haskellgamedev/`

1.2 Rendering and interfacing with the OS

Essential part of a game engine is communicating with the operating system and rendering models or textures. To do this, we can either use a complete engine like the before mentioned Helm or a library built for this purpose like Gloss[**glossrepo**], 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 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, poll input events, create windows, and render scenes. Specifically, we use the `sdl2`, `sdl2-image` and `sdl2-ttf` packages. We chose SDL because there are already examples of its use in games we can learn from, the underlying C library works across multiple platforms, is well documented[2] and is widely used. Moreover, the Haskell library 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 the powerful Foreign Function Interface (FFI), which makes Haskell even more useful in the real world.

Here are the most essential `SDL` functions we use in our two games, all examples of the high-level bindings:

- `SDL.createWindow`
  is used to create a new window.
- `SDL.createRenderer`
  is used to create a rendering context for a window.
- `SDL.Font.load`
  is for loading fonts.
- `SDL.Image.loadTexture`
  is for loading images as textures.

2. The documentation of the C library: `https://wiki.libsdl.org/`

- `SDL.clear`
  clears the rendering target/context,

- `SDL.copy` and `SDL.copyEx`
  copies our loaded textures to the rendering target like stamping a picture on a canvas.

- `SDL.present`
  displays the current state of the target in the window.

- `SDL.pollEvents`
  is called to get input events.

In listing 1.1, we can see how FFI is used. It shows `SDL.copy`, which wraps around the low-level binding, abstracting from the pointers, replacing them with Haskell's `Maybe`, and throwing an error instead of returning a negative value.

Listing 1.1: Example of FFI binding.[**sdlrepo**]

```
-- the C function
↪  (https://wiki.libsdl.org/SDL_RenderCopy):
-- int SDL_RenderCopy(SDL_Renderer * renderer,
--                    SDL_Texture * texture,
--                    const SDL_Rect * srcrect,
--                    const SDL_Rect * dstrect);

-- the binding from SDL.Raw.Video
foreign import ccall "SDL.h SDL_RenderCopy"
    renderCopyFFI :: Renderer
                  -> Texture
                  -> Ptr Rect
                  -> Ptr Rect
                  -> IO CInt

-- the wrapper from SDL.Video.Renderer
copy :: MonadIO m
     => Renderer
     -> Texture
     -> Maybe (Rectangle CInt)
     -> Maybe (Rectangle CInt)
     -> m ()
```

1.3 Asteroids by Atari as an example

Asteroids is an arcade game created by Atari in 1979. To evaluate Haskell as a language for game development in general would be a task far beyond the scope of 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 more giant games. We chose Asteroids as an example because its world comprises only a few object types, yet their relationships make the game quite interesting.

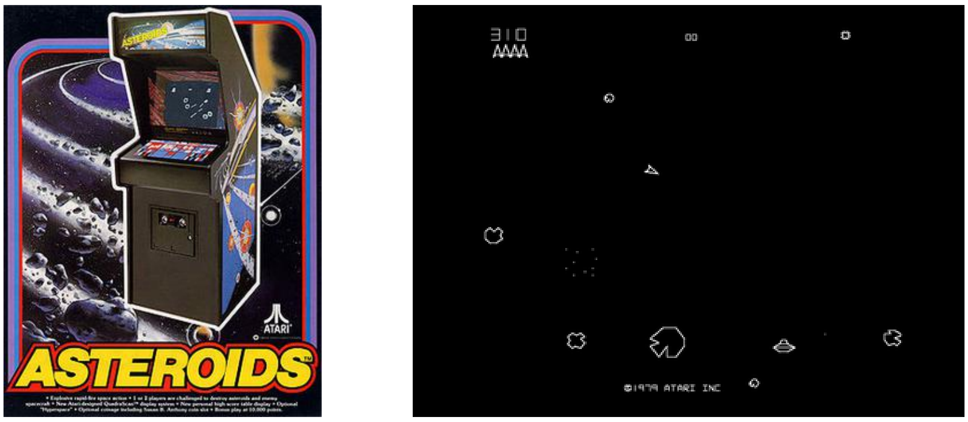

Figure 1.1: Atari Asteroids — Promotional flyer cover[**asteroidsflyer**] and gameplay screenshot.[**asteroidsscreenshot**]

It also does not rely on complex graphics, therefore we can focus on the code implementing the game rules and behaviors.

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 rocks (asteroids, in three sizes), projectiles, flying saucers (large and 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 are a few features that we are omitting, like sound effects, some animations, and several minor gameplay 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.

2 Using the apecs library — hAsteroids

2.1 About apecs

"apecs: a fast, type-driven Entity–Component–System library for game programming"[1] is one of the more recently released libraries. Entity–Component–System (or ECS) is a data-oriented architectural pattern often used in video game engines to represent the game world state. In a true ECS, a game object or an **entity** is only an ID number, and data is attached to it by being stored under that ID. This data is organized into **components**, which are then stored in separate lists with other components of the same type from other entities.[14] This can be represented as a table where every column is its own list or array (see table 2.1). Then we define game logic as **systems** — a set of functions that operate on certain components regardless of the entity as a whole. A typical example of a system is adding an entity's velocity to its position for every entity with both of those components. ECS usually provide better performance than object-oriented designs (OOD) because of their increased data locality — a system needs to load into memory only components that are relevant for it, not the whole *objects* as it would be with OOD.[15]

Table 2.1: A simple example of ECS represented by a table.

Entity	Position	Velocity	Type of Unit	Ammunition
0	$(4,2)$	$(0,0)$	Player	314
1	$(5,1)$	$(1,1)$	Enemy	–
2	$(2,2)$	$(1,0)$	Enemy	–
3	$(2,3)$	–	Obstacle	–

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]

1. The making of the Ecstasy library was actually inspired by author's issues with apecs as she explains on her blog.[16]

Listing 2.1: Defining instance of `Component`.

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

To write a game using apecs we must define **components** and **systems**. Systems also create new **entities**, as creating them means to write some components under a new ID.

First, defining a **component** means to define an instance of the type-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` (and `Cashe`). 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 at the same time. Finally, we call `makeWorld`, which uses Template Haskell to generate `World` product type along with `initWorld` function and instances of the `Has` type-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)
    }
```

System in apecs is anything with the `SystemT w m a` return type, which means that it may alter the world state. One such "micro-system" is the `newEntity` function, which accepts a tuple of components and adds them into their records under a new ID. 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.[17]

```
-- '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. Furthermore, with `cmaM_` there is no direct writing, only side effects.

2.2 Writing of hAsteroids

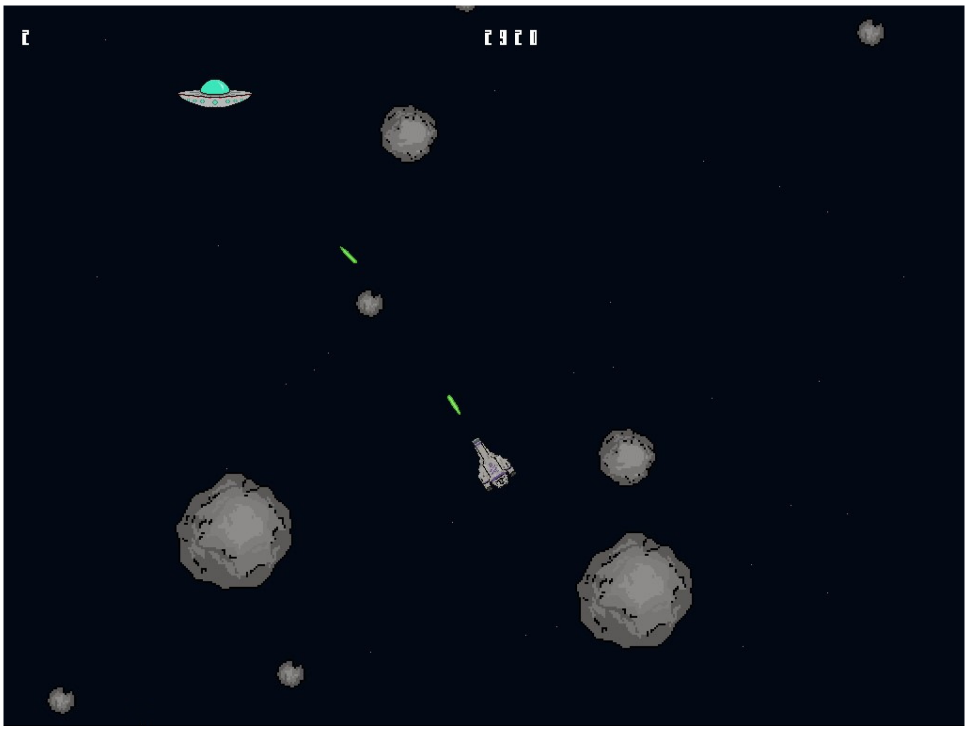

Figure 2.1: A screenshot of hAsteroids gameplay.

In this section, we outline how we used apecs in the hAsteroids game. For the exact implementation, please refer to the attached source files in the hAsteroids directory. Figure 2.2 shows project's modules with loosely indicated dependencies — upper modules may have direct dependencies on the lower modules, arrows show most important ones.

2.2.1 Components in hAsteroids

There are more approaches to designing components but in hAsteroids, we have three categories of components: "marker" compo-

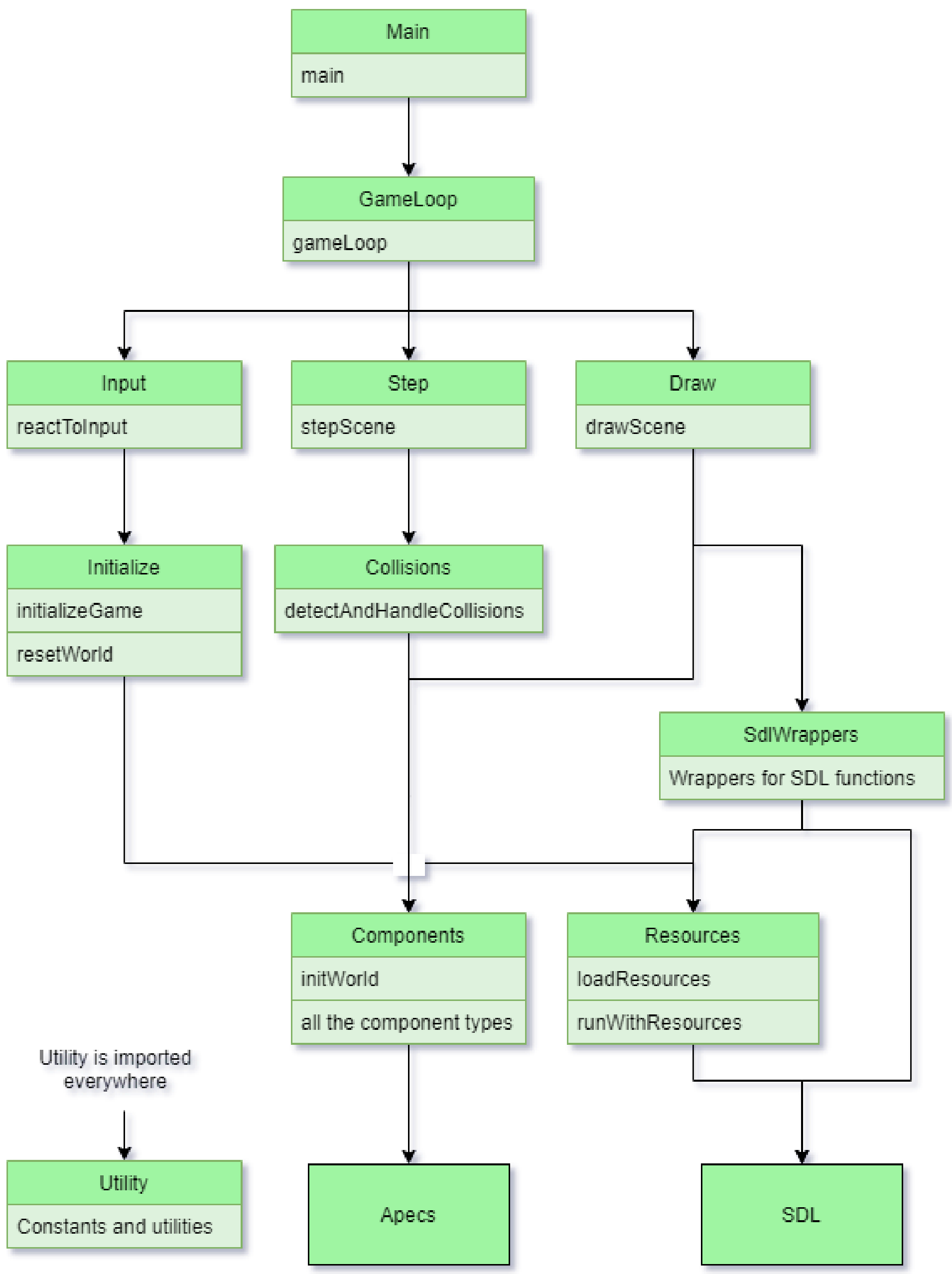

Figure 2.2: hAsteroids module structure.

nents, “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 and the transitions between scenes.

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. Table 2.2 shows which game object types should have which components; however, it cannot be enforced by types due to the nature of ECS.

Table 2.2: Game objects and their components.

Object	Components
Ship	Position, Velocity, Ship
UFO	Position, Velocity, TimeToLive, Ufo
Bullet	Position, Velocity, TimeToLive, Bullet
Asteroid	Position, Velocity, Asteroid

2.2.2 Initialization and looping

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, it is passed to the `gameLoop` through a stack of monad transformers (`SystemWithResources`). When the gameLoop quits, the window is destroyed, and resources are freed.

2.2.3 Systems in hAsteroids

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

Reacting to input

`reactToInput` manages the state of the input, and as its name suggests, 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` that changes ship's angle, increases its velocity, or creates a new bullet entity — all conditioned by the input state.

Stepping the scene

`stepScene` takes care of simulating physics and game rules over time when the loop is in the `Playing` state. This is divided into a series of twelve 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 than the large ones 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 a quarter of π increments towards the ship's current location.

– `awardLifeIf10000`
  increments ship's lives by 1 for reaching every 10000 score points.

– `modify global $ \(WaveTime t) -> WaveTime $ t + dT`
  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.

– `cmap $ \(Ttl t) -> Ttl $ t - dT`
  simply subtracts the frame delta time `dT` from all the `TimeToLive` 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. 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 $\binom{6}{2} = 6$ slighlty different collision detection 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.

Drawing the scene

Once the scene is stepped, it is then **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. Naturally, 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 function 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 that apecs makes game programming in Haskell surprisingly accessible. Once we understand the ECS principle, every world modification is "intuitively imperative" thanks to the `SystemT w m` a monad and the component map functions. Any system side effects can be added to an existing `cmap` call with a simple change of the return types as demonstrated in listing 2.4

```haskell
-- 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
                     )
```

Listing 2.4: We can easily switch between non-effectful and effectful calls.

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

No less important is the fact that the game works. The development experience was smooth, with no significant hiccups. 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 type-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 of an entity's components automatically in apecs. One has to do destroy them explicitly. That way, nothing 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 of purely functional programming.

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

3 Focusing on functional purity — pure-asteroids

3.1 Game engines in purely functional style

In the previous chapter we see how a program in Haskell can still be written in a very imperative style. This is certainly a valid approach and has its benefits as we discuss further in chapter 5 but for the second version of Asteroids (project *pure-asteroids*) we focus on functional purity, clarity and proper compartmentalization.

The design is inspired by a keynote[18] of John Carmack, the co-founder of id Software and co-creator of the Doom and Quake games.At one point in the keynote he talks about how he had been moving towards a functional style of programming with C++ and seeing the long term benefits. He also talked about his experiments with Haskell and his vision for multi-core game logic. "State of where we're at right now with game code, is that we run all the game code in one thread because the idea of using locks to synchronize amongst all of our current game code was just absolutely terrifying," he says and continues to explain how independent parallelism is much more feasible with pure functions and how events can be used for interactions between the isolated groups.

The world-updating function can be split into individual sub-functions that update each group of entities. Each sub-function is passed the game world and the group of entities and it returns the updated group, therefore, in Haskell, such function cannot affect anything outside of the group it returns. For that reason, it could be easily run in parallel, increasing the performance.

With this approach, comes one more advantage — all entities are being updated based on the same image of past state, meaning it does not matter in what order entities are updated, the result will be the same. This can be contrasted with a naive imperative approach, where entities are updated sequentially in-place based on the current state. Such approach may lead to inconsistencies if for example two units attack each other at the exact same time. Or to give an example from Asteroids, if a UFO and the ship shoot at each other with an asteroid in between them, whichever bullet's collision is detected first would destroy the bullet and the asteroid and the other bullet would fly through killing either the UFO or the ship. However, the opposite approach comes with a caveat and that is the issue of two objects moving into a common space when they are not allowed to overlap. John Carmack addresses this in his already mentioned keynote[18] and says it can be solved for instance by some kind of repulsion force. Fortunately, this is a non-issue in Asteroids because all objects may overlap, usually causing a collision and destruction.

3.2 Writing of pure-asteroids

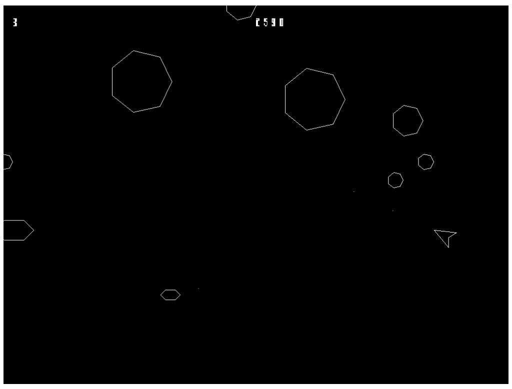

Figure 3.1: A screenshot of pure-asteroids gameplay.

In this section we describe how *pure-asteroids* implements a pure style engine described in the previous section. The program's main function is very similar to the one from hAsteroids with the exception of texture loading because pure-asteroids draws vector graphics using `drawScene`. There is also no reader monad and `gameLoop` is passed everything explicitly as arguments. The looping itself is also very similar to hAsteroids but naturally, the world updating and drawing differs. The world state passes through `processWorldEvents`, which fulfils all the requests from events, through `stepWorld`, which simulates physics and game rules and generates new events, `drawScene` draws it and `resetIfNewGame` returns a reinitialized world only if the loop state is transitioning from `InMenu` to `Playing`. This is illustrated in figure 3.2. Next, we describe the used **data structures** and then explain how the **stepper function** and **event processing** use them.

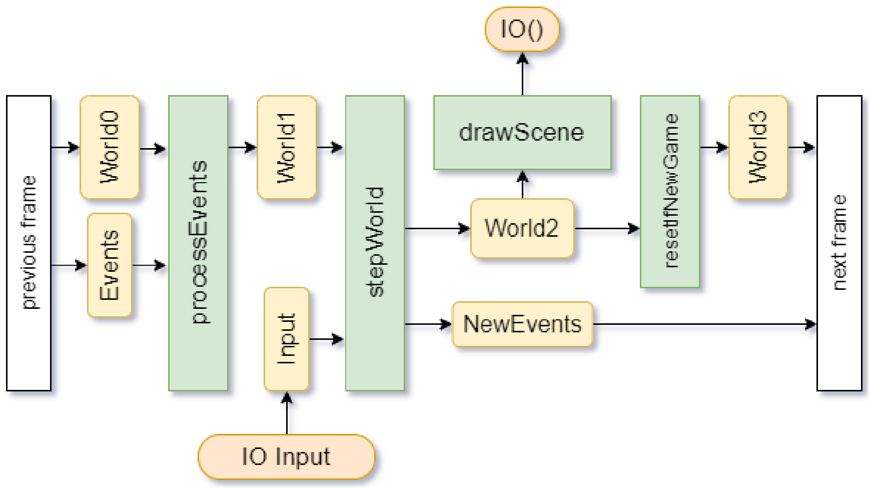

Figure 3.2: Data flow of `gameLoop`.

3.2.1 Data structures

The two main data structures, as already shown in figure 3.2, are the world sate represented by `World` and the events for communication between entity groups represented by `WorldEvents`.

The definition of `World` can be seen in listing 3.1. `World` contains separate groups of entities and some other state variables. `Asteroids`, `Ufos` and `Bullets` are type aliases for hash maps of the respective entities. The entities themselves then contain similar data as components in hAsteroids, only here, the data is all in one place, grouped by the game object. Listing 3.2 shows `Ship` definition as an example. We can notice the preceding underscores in the record names. This is used to generate lenses by using `makeLenses` from the `mtl` package to facilitate easier data manipulation in our nested structures.

```haskell
-- | Game world state structure
data World =
    World
    { _wShip      :: Ship
    , _wAsteroids :: Asteroids
    , _wBullets   :: Bullets
    , _wUfos      :: Ufos
    , _wWaveTime  :: Time
    , _wWavePause :: Time
    , _wWaveNum   :: Int
    , _wScore     :: Score
    }
```

Listing 3.1: World structure in pure-asteroids.

```haskell
-- | Ship state structure
data Ship =
    Ship
    { _sPosition :: Position
    , _sVelocity :: Velocity
    , _sAngle    :: Angle
    , _sLives    :: Int
    , _sState    :: ShipState
    }

data ShipState
    = ShipAlive
    | ShipExploding Time
    | ShipRespawning Time
```

Listing 3.2: The `Ship` representation in pure-asteroids.

Listing 3.3 shows the event package type `WorldEvents`. It also has an instance of `Monoid`, which allows us to collect the packages from each entity group and combine them into one in the **stepper function** and then we distribute the events to their addressed entity groups in **event processing**. The individual event types and the information they carry are described later, with the functions that process them.

```
-- | Structure for event passing between entity groups
data WorldEvents =
    WorldEvents
    { _forAsteroids :: [AsteroidEvent]
    , _forShip      :: [ShipEvent]
    , _forUfos      :: [UfoEvent]
    , _forBullets   :: [BulletEvent]
    , _forScore     :: [ScoreEvent]
    }
```

Listing 3.3: The event package structure.

3.2.2 Stepper function

We can see the combining of event packages happen in the definition of `stepWorld` in listing 3.4. The events are returned by the individual steppers of each entity group together with the stepped versions of those groups. Note how lenses are used to "focus" on the contents of world state and change them with the setter operator `.~` or the function applicator `%~` (or also the lens equivalent of +=). Each "substepper" is compartmentalized and could be made to run in parallel.

```
-- | Update the world
--   simulating physics and reacting to input
stepWorld :: Time -> InputState -> RandomStream Double ->
 ↪  World -> (WorldEvents, World)
stepWorld dT input rand oldW =
    let
        (eventsS, newShip)     = stepShip dT input oldW $
         ↪  oldW ^. wShip
        (eventsB, newBullets) = stepBullets dT input oldW
         ↪  $ oldW ^. wBullets
        (eventsU, newUfos)     = stepUfos dT rand oldW $
         ↪  oldW ^. wUfos
        (eventsScr, newScore) = stepScore dT $ oldW ^.
         ↪  wScore
    in
        (,) (eventsS <> eventsB <> eventsU <> eventsScr) $
        checkWave $
        oldW
          & wShip      .~ newShip
          & wAsteroids %~ stepAsteroids dT
          & wBullets   .~ newBullets
          & wUfos      .~ newUfos
          & wWaveTime  +~ dT
          & wScore     .~ newScore
```

Listing 3.4: The stepper function.

Figure 3.3 shows which events, represented by their data constructors, entities may send to others. The working of the individual "substeppers" can be summarized as follows:

– `stepShip`
  does things # TODO.

– `stepAsteroids`
  does things # TODO.

– `stepBullets`
  does things # TODO.

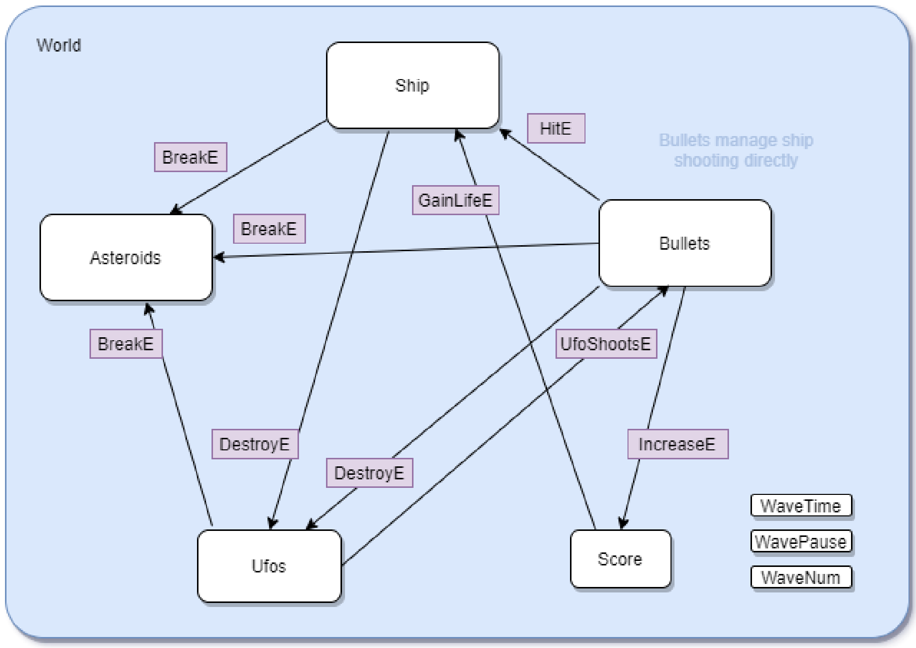

Figure 3.3: Entity groups and their interactions.

– `stepUfos`
  does things # TODO.

– `stepScore`
  does things # TODO.

After that, `checkWave` takes care of spawning new waves of asteroids and then the stepped world is paired with the generated events and returned.

3.2.3 Event processing function

In listing 3.5 we can see the implementation of event processing and again the use of lenses. Similarly to `stepWorld`, the work is compartmentalized and could be made to run in parallel. # TODO maybe {itemize} the description of subfunctions, maybe do not, and describe the individual event types too

```
-- | Process all WorldEvents
processWorldEvents :: WorldEvents -> World -> World
processWorldEvents events world =
    world
      & wAsteroids %~ processAsteroidsEvents
                        (events ^. forAsteroids)
      & wShip      %~ processShipEvents
                        (events ^. forShip)
      & wUfos      %~ processUfosEvents
                        (events ^. forUfos)
      & wBullets   %~ processBulletEvents
                        (events ^. forBullets)
      & wScore     %~ processScoreEvents
                        (events ^. forScore)
```

Listing 3.5: The event processing function.

3.3 Reflection

Overall, pure-asteroids achieves what it sets out to do — we can observe many of the benefits described in section 1.1. Because it uses less abstraction, the code is safer and more expressive. For example, by looking at `shoot` from the `Step.Bullets` module

```
shoot :: InputState -> World -> Bullets -> Bullets
shoot input w =
    if wasPressed input spaceKeycode
        then insertBullet
        else id
```

We immediately see from the name and the type signature that the function presumably somehow alters the collection of `Bullet`s based on the input and that the alteration is probably adding a new bullet, which is confirmed by looking at the next four lines of light code. And this also means that it is *the only thing* the function can do — it clearly cannot change the state of the ship, read from a file or render a white square.

It also incorporates what we may call "functional elegance". Throughout the code base, we use functions like `map`, `filter`, `fold`, `traverse` and many others, that Haskell programmers are familiar with and which make the code brief and save us time.

Most importantly, as we demonstrated, the design could be adapted to use parallelism, which we expect would increase performance. Parallel computation would be especially beneficial if the game was of larger scale and the computation was more demanding — for instance collision detection in pure-asteroids is very simple but it could be improved by using more complicated algorithms.

However, even though we could use parallelism, the scalability of our design is somewhat lacking. It would be appropriate to use more type-classes in general. A class interface for collision detection, stepping, communication using events and for resources abstraction. Pure-asteroids is too explicit, which makes it relatively rigid — adding a random number generator to a bottom-level function, requires passing it from the top level as an argument through the whole function chain. And the long type signatures of the top-level functions make them actually less clear.

Ultimately, we see that explicitness is good but has its boundaries, and that some level of abstraction is needed. It should be also pointed out that designing the complete engine from scratch was a significant amount of effort, despite the development being smooth and the final product working well.

4 Analyzing an existing C++ implementation

4.1

Explain a bit about the choice of the particular implementation and comment on the quality of code

the options turned out to be not so great, like my implementations this version is not flaw-less. we point out its mistakes and present them as a possible outcome of this language choice, although the choice definitely does not need to imply them nor conditions them -> there can be a great program in C++ and a really bad one in Haskell.

4.2

outline its working and architecture

4.3

reflect on it

in a way very intuitive - objects, “imperatives”

good scalability in terms of adding new entities or features, however there should be separate manager classes for entities

some whoopsies - valgrind shows memory leaks and uses of uninitialized values

effects can be everywhere -> high flexibility low safeness

...maybe this chapter could be a section in **Comparing the approaches**

5 Evaluation of the approaches

In this chapter we first compare Haskell to imperative languages in general or specifically to C++, often referencing back to the promised benefits of Haskell from section 1.1. Then we discuss the conflicts observed in the comparisons and their possible solutions and finally summarize our findings.

5.1 Comparison

In this section we compare the three example implementations or more broadly Haskell and imperative languages represented by C++. For this we reference the previous chapters about individual implementations. It is important to note that C++ is by no means meant to represent imperative languages fully, even less so is the chosen imperative implementation of Asteroids (further referred to as imp-asteroids) meant to represent the ideal of a game in C++. Not only it has flaws but it is also an even simpler version of the original game in some ways than our two implementations. We try to address this in especially relevant places, in others, it is left out for the sake of conciseness and readability.

The comparison is organized by qualities important in game development: flexibility and scalability, expressiveness and safeness, development costs, performance. The section is then closed off by additional comparison of state abstraction and with few statistical observations about the code bases.

5.1.1 Flexibility and scalability

Both **hAsteroids** and **imp-asteroids** are very flexible designs. Because in hAsteroids, the monad return type of most functions is not very restrictive at all, we can add in and take away effects or resources with ease. Similarly the imperative version is highly flexible in this sense. When it comes to scalability, both do decently well. Adding new game objects in hAsteroids can be done by defining more components and needed systems. Already existing components and systems can be reused like `Position`, `Velocity` and `stepKinetics`. One issue would be the expansion of collision detection, which is done on an individual basis, therefore the amount of work would grow linearly with increasing number of game objects. imp-asteroids do better in this regard — kinetic properties can be inherited from `FlyingObject` as well as the ability to detect collisions. The issue there might be if we wanted to inherit only parts of `FlyingObject`. Also with large scale, the safeness becomes more of a concern for both games — more on that in the following subsection.

We have already described how **pure-asteroids** are rather rigid because of their lack of abstraction. Scalability is also worse than the other two, since adding new objects would require writing their whole "substepper" and new event paths, and collision detection is handled similarly to hAsteroids.

5.1.2 Expressiveness and safeness

As we have already discussed in chapter 1, functional languages are designed to be clearer (*expressiveness*) and to have better control of side effects (*safeness*) then conventional imperative languages. Now the imperative code can be even more or less unsafe depending on its type system. C++ is mostly strongly and statically typed, which does provide a certain amount of assurance, some other languages are not. This can protect us from trying to add an object to a number for instance.

Many languages also take various steps towards controlling side effects. One such example could be the `const` keyword used to define a variable that cannot be mutated. In C++ this keyword can also be used on methods to prevent them from changing the state of the object, allowing us to call these methods even on `const` objects. Usage of this keyword, would prevent the author of imp-asteroids from writing this drawing function that unexpectedly also changes the state of bullets:

```
void Game::draw(const Interface &ui) {

    /* ... other objects are drawn... */

    for (int i = 0; i < bullets.size(); i++) {
        if (bullets[i]->isAlive()) {
            bullets[i]->draw();
            bullets[i]->setHealth(); // a setter?!
        }
    }
}
```

However, languages like Java or Python do not support immutable objects and we can see how control of side effects in general is a "plan B" in the imperative paradigm.

Conversely, since Haskell originated as an academic language, functional purity was allowed to be plan A and having side effects was actually the plan B in this case. We saw how this together with the type system works out well for expressiveness and clarity in the example of **pure-asteroids**, as showcased on an exemplary function at the beginning of section 3.3. And because all variables are immutable, change is made explicit as a *new* returned value that we pass forward. Moreover, any input/output side effects are clearly separated by the `IO` monad and the rest of the code is completely deterministic, which all together with strong static typing makes for a very safe language — not only there is less potential sources of bugs, but the correctness of a program is *easier to perceive* and argue about.

That is, if used well — **hAsteroids** turned out to be a very interesting game. apecs makes game programming in Haskell approachable and provides very good performance by being *strict and imperative*. This inner imperativness makes purity sort of a plan B again, and the programmer has to be clever about using monad polymorphism if they want to keep "the danger of `IO`" contained. This has occurred to us in the middle of the hAsteroids development process and so we decided to embrace it and see where it leads. If we redesigned the types we could have reached better safeness and even expressiveness, however, with the game world sate abstracted it is still not always clear what functions do when applied through `cmapM` or `cmapM_`.

5.1.3 Development costs

The purpose of any language feature can ultimately be seen as reducing the cost of development.[1] The already discussed Haskell features all bring the costs down in certain scenarios, but there is more that we want to look at through the lens of development speed and comfort. Both **pure-asteroids** and **hAsteroids** more or less benefit from these.

Interpreter

One of such features is **GHCi**, "GHC's interactive environment, in which Haskell expressions can be interactively evaluated and programs can be interpreted."[**ghciwiki**] Haskell is quite unique by being a compiled language that comes with an interpreter as well. An interpreter allows us to test short expressions quickly, either to try something new or to run bits of already existing code. `GHCi` can also print the types of expressions or information about type-classes. This reduces the need to reference actual documentation, although the documentation is also very helpful. "Hoogle is a Haskell API search engine, which allows you to search the Haskell libraries on Stackage by either function name, or by approximate type signature."[**hoogle**] This makes searching for functions easier and can prevent "wheel reinvention".

Functional elegance

Furthermore, there is many predefined high-order functions like `map`, `filter`, `foldl` and so forth. This way a part of the job is already done and we only need to provide the elemental function to fold with for instance. This together with being able to partially apply functions allows us to write very **brief and elegant code**. In listing 5.1 we can see asteroid stepper implementations from imp-asteroids and pure-asteroids.[2] The Haskell version uses `map` with a partially applied function, while too being only partially applied. Granted, other languages

1. By development costs, we mean mainly time and endured frustration. Generally speaking, any piece of software can be written in any language, what differs is the cost.
2. The C++ version also spawns new wave of asteroids if the current has been cleared. That part of code was omitted as it is irrelevant for the example.

have structures to make this sort of task at least slightly more elegant too and even C++ itself has `std::for_each`, which could be used to make this piece of code more concise. However, this characteristic extends to the whole Haskell code base — in both of our games there is only three instances of explicit recursion in total: the game loops and `drawLines` in pure-asteroids. This is rather impressive considering the games have to manipulate collections of events, textures, entities, components and more. More on conciseness in subsection 5.1.6 (Code statistics).

```
// C++
void Game::advanceAsteroids() {
    for (list<Asteroid*>::iterator it = asteroids.begin();
         it != asteroids.end(); it++) {
            (*it)->advance();
    }
}

-- Haskell
stepAsteroids :: Time -> Asteroids -> Asteroids
stepAsteroids dT = HM.map (stepAsteroid dT)
```

Listing 5.1: Asteroid stepping with for-loop and map.

Testing and debugging

Another benefit discussed at the beginning in section 1.1 is **ease of testing and debugging**. Testing a pure function is as simple or as complicated as the function's parameters and its output. Testing an impure function is theoretically nearly impossible to do *exhaustively*, since without looking at its definition we do not know weather it reads from a file or not for instance. Practically, we do not need to be completely exhaustive and can make assumptions, however, those unchecked assumptions are the weak spots that may cause problems.

Due to time constraints, Haskell's safeness and the simple nature of Asteroids, we did not use any automated tests during our development. Still, to substantiate the claim of easier testing we look at the two functions[3]in listing 5.2 and consider how we would go about test-

```
// C++
void Game::advanceBullets() {
    for (int i = 0; i < bullets.size(); i++) {
        if (bullets[i]->isAlive()) {
            bullets[i]->advance();
            if (!isOnScreen(bullets[i]->getPoint())) {
                bullets[i]->kill();
            }
        }
    }
}

-- Haskell
stepBullets :: Time -> World -> Bullets -> (WorldEvents,
 ↪  Bullets)
stepBullets dT w =
    traverse (stepBullet dT w) . filterDeadBullets
```

Listing 5.2: Impure and pure stepping of bullets.

ing them. To test `stepBullets` we need to provide three arguments. First is a simple integer and the other two are much more complex but still could be generated randomly using Haskell QuickCheck or provided "manually". The only effect produced is the returned pair that can be checked for correctness by other functions. On the contrary, to test `advanceBullets` we need to set up a `Game` object with its contents, many of which are `private`. That requires refactoring and the same applies to testing the state after `advanceBullets` has been called. Additionally, we may want to test not only weather the collection of bullets has changed correctly but also weather the rest of the state is correct.

hAsteroids is no better as the test needs to run inside of the `SystemT w m a` monad, where we set up components using `newEntity` function

3. The examples are again taken from imp-asteroids and pure-asteroids. `stepBullets` has been simplified in this case. The real version also inserts new bullets based on the input state, which happens elsewhere in the C++ implementation.

and then after running tested system checking the validity of all the components.

Tools

Lastly, an obvious disadvantage is the fact that writing a game in Haskell is still largely a pioneering process. There is a decent amount of existing general tools, one of them being the already described interpreter, which can also be used to setup breakpoints and step through evaluation. More extensive debuggers like Hat exist as well and package and project managers like Cabal and Stack also speed up development. However, even some general tools such as a robust IDE are easier to find for other languages and in terms of game development specific tools, there is not many besides the several libraries like apecs listed in section 1.1.2. This can be contrasted with the complete, general engines, that come included with tools for the whole development pipeline such as Unity.

This was expected and it turned out not to be a problem as our example game was very simple. Yet, we can imagine that with a larger game we would start to feel the need for more tools.

5.1.4 Performance

Performance in games is very important since there is a real minimum requirement given by desired frame rate and player's expected hardware. A game can offer a decent experience at 60 FPS, which gives us roughly 16 ms to do any computation we need for a single frame. With smaller games this is plenty, but as the entity counts grow and rules get more complex, performance optimization becomes essential. For that reason we want to compare the performance of our implementations and the tools available to optimize it.

We put together simple **benchmarking** versions of all three games and ran them to find how many frames can be rendered in given time in similar situations.[4] The games were altered to spawn various, abnormal quantities of asteroids and run for ten seconds with the

4. The benchmarking versions can be found in their respective GitHub repositories on branch called "bench":
`https://github.com/honzaflash/cpp-asteroids/tree/bench`

Table 5.1: Average FPS in the benchmarks.

Asteroid count	**imp-asteroids**	**hAsteroids**	**pure-asteroids**
10000	41	2	1
500	59	30	16
250	59	46	23
100	59	55	37
50	59	74	92
25	59	117	140
10	59	141	154

ship constantly spinning and shooting one bullet per frame. Note that the results are rather illustrational as the differences between games were not controlled very well: imp-asteroids uses GLUT with the FPS limited to maximum of 59, because of issues with Stack on Windows the Haskell games were run in the interpreter, hAsteroids renders textures instead of drawing vectors like the other two and there are other minor differences. The benchmark was run at least 3 times for every asteroid count and the median is shown in the results in table 5.1.

We ran the benchmark for 100 asteroids also in a virtual machine with Ubuntu, where `stack build` worked normally, and the results were quite different even if we account for the slower "virtual hardware": 46, 21 and 34 FPS respectively for imp-asteroids, hAsteroids and pure-asteroids. From these results with all three games being compiled we can assume, what is said to be true for Haskell in other fields, that its performance is mostly comparable to C++. Unfortunately, we did not have time to diagnose why apecs performed so poorly in this case. However, when comparing the two versions (hAsteroids and pure-asteroids) running in GHCi, hAsteroids with apecs does outperform pure-asteroids for higher entity counts as we expected.

Worthy of noting is also the profiling we did on the runs in Ubuntu. Most importantly it showed that neither of the games had issues with

https://github.com/honzaflash/ba-thesis/tree/bench/hAsteroids
https://github.com/honzaflash/ba-thesis/tree/bench/pure-asteroids

Garbage Collection (GC) as this can become a problem with larger lazy systems. The productivity reported by the `-s` profiling flag was 99.1 % for hAsteroids and 97.6 % for pure-asteroids, meaning GC is not causing any meaningful delays in our games.

This brings us to the **tools** available and ease of **optimization** in general. As mentioned in the previous subsection, there are various tools including profilers for Haskell but C++ will have larger variety. Another advantage for C++ in this discipline is its low-level access to hardware and memory management. Generally the less abstracted code is easier to optimize, since it is closer to what the CPU will actually execute. With more abstraction, more work must be done to find out what the CPU does in the end. This is especially the case for Haskell which has high level of abstraction and during compilation goes through *three* intermediate languages, therefore if one desires to see how the code gets "unabstracted", he must first learn at least the first of the intermediate languages — Core. To make up for this at least partially, as we have also stated before, GHC can do a lot of optimizing on its own and purely functional code is also easier to execute in parallel and concurrently, which Haskell is very good at.

5.1.5 State abstraction

One of the largest differences between the implementations is how they model game's state. Through this prism we can see once again how **hAsteroids** and **imp-asteroids** are very similar. As we have mentioned before, because of the abstraction, the code in hAsteroids has a certain amount of imperative style and that includes the state of the game world kept in a monad. Similarly, despite C++ having a lower-level approach to memory and some other things, the object-oriented paradigm allows us to abstract the state in a relatively elegant way. Object's attributes define its current state, which is implicitly passed to any of its methods.

The monad state is technically pure because in reality, its change is only a computation hidden behind the >>= operator, that passes the change as a *new* value forward without any mutation, and the operator itself then can be further hidden using the `do` block.[5] But

5. Reminder: `do str <- getLine; putStr str` is a shorthadnd for `getLine >>= (\str -> putStr str)`

```
-- gameLoop returns SystemWithResources ()
reactToInput deltaTime
stepScene deltaTime
drawScene

// pGame is a pointer to Game object
pGame->advance();
pGame->handleInput(*pUI);
pGame->draw(*pUI);
```

Listing 5.3: Cores of the main loops in hAsteroids and imp-asteroids.

from a practical standpoint, there is only a little difference between a monad state and an object's state. Just as an object gives a context to a method's side effects, a state monad gives context to the monadic computation's side effects. In the same way that an object stored in a single variable is changed, a monad state can change without any outer signs. Listing 5.3 shows how similar in appearance are the main game loops of hAsteroids and imp-asteroids.

This brings us to the argument of why we want to avoid mutability in the first place. Mutability makes code harder to reason about, easier to make mistakes in and very difficult to run in parallel. Granted, since the state monad is technically pure, the hidden syntax will not allow shared mutability, which is what makes parallel systems complicated. With that in mind, it also makes some things much simpler to program.

As we have already pointed out, **pure-asteroids** steers clear of monads when possible, modeling the state explicitly. This does indeed make things easier to reason about, but it was much more complicated to write. We discuss this need for balance later, in section 5.2 about the observed dilemmas.

5.1.6 Code statistics

As a bonus we include few simple statistics about code length. We counted the lines, words[6] and characters using the `wc` command. Then

6. `wc` defines word as a sequence of any printable characters delimited by white space.

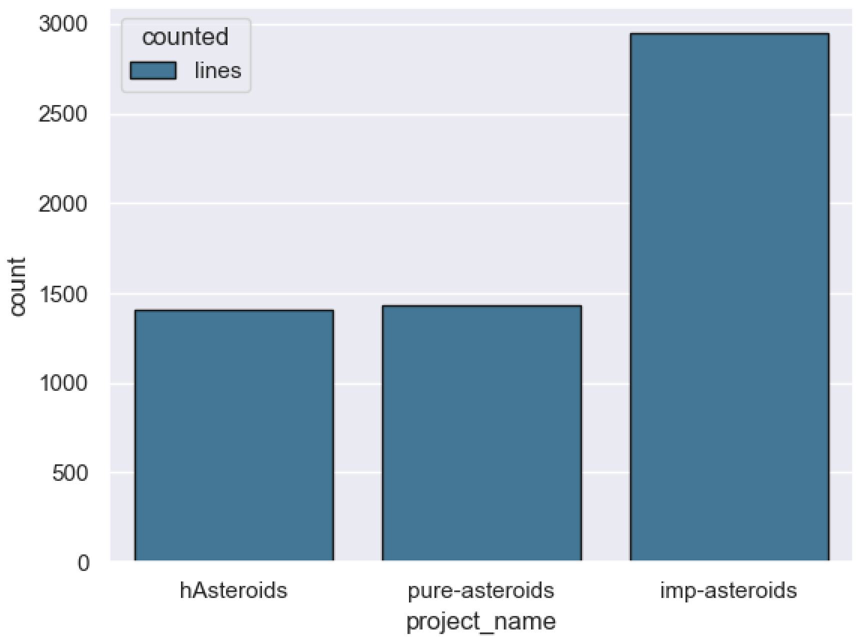

Figure 5.1: Total number of lines by project.

we plotted the total line counts (figure 5.1) and ratios between words, lines and characters in each file (figure 5.2) using Python and `seaborn`. We must take into account the differences between the implementations because imp-asteroids uses a custom drawing library built around GLUT, which includes several lengthy line drawing functions and it is much more heavily commented[7] , but it also does not implement UFOs, score, ship lives or menus.

From the ratios in figure 5.2 we see that the code has similar density, therefore we know it is not shorter lines inflating the total line count, but generally longer code. This further supports the claim for Haskell's ability to be brief and concise. We can also notice the slight, opposite tendencies of words per lines and characters per words. It would appear that in the C++ code, words are longer and there is less of them per line, which can be explained by `wc` understanding strings like

7. Using `grep` we found 850 lines of comments in the imp-asteroids project and only 110 and 50 in hAsteroids and pure-asteroids.

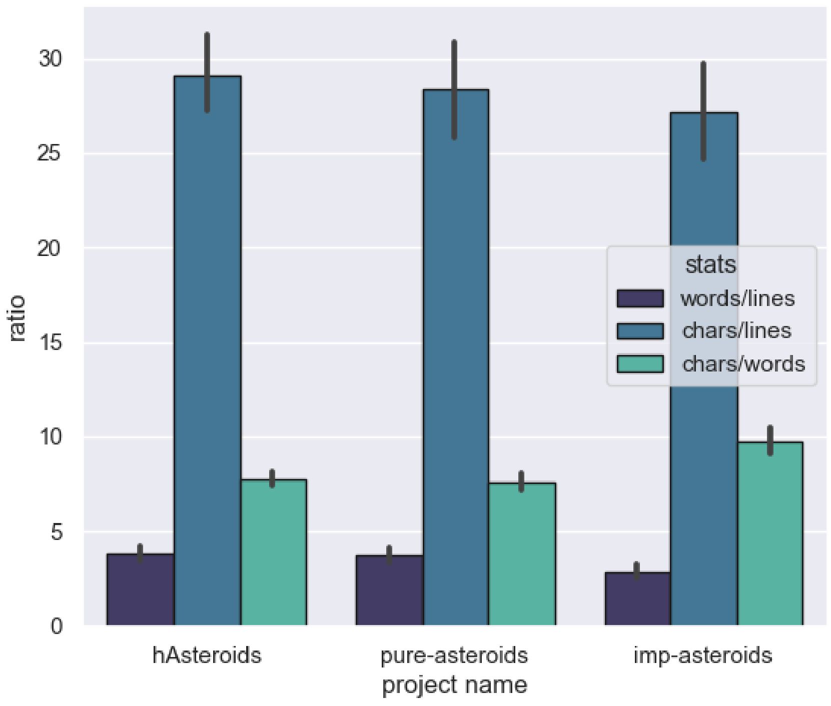

Figure 5.2: Code density by project.

`asteroids.erase(it);` as one word as are understood also operators, more frequent in Haskell.

5.2 Observed dilemmas

Here we discuss further the conflicts observed in the previous section and suggest possible better compromises or solutions. Namely we identify the tension between flexibility and safeness, and between abstraction and explicitness.

5.2.1 Freedom vs. safeness

We saw examples of two extremes — pure-asteroids with high safeness but low flexibility and hAsteroids together with imp-asteroids with high flexibility but low safeness. It may seem as though the two are mutually exclusive. However, we see another characteristic hiding behind the flexibility label: *freedom of side effects* or unbound flexibility. That is what cannot be achieved simultaneously as complete safeness rooted in control and limits. Nevertheless, it definitely is possible to design a system that includes constraints and control implemented in a relatively *flexible* way.

We would argue that complete freedom is rarely even desirable. It may be tolerable for smaller programs, possibly beneficial for simple scripts, but for any more extensive application, programmers are taught "good practices" such as "do not use global variables" and "do not use goto." A safe design can be achieved with good practices in a language like C, but modern languages help us save time and money. For this reason, it is of great use when a language supports robust control of side effects.

Being able to compromise on unlimited side effects then, we believe that a reasonably flexible system can by designed in Haskell, which is also very safe. The addition of type-classes was a great break through for Haskell and it is exactly this feature that is used for good, modular, flexible and safe design of complex systems. Type-classes are used to put constraints on types and can be also compounded, that way the addition or removal of limits is relatively easy. This not only helps controlling the return type, but can also help with polymorphous arguments, making code more useful while staying safe.

We have criticized both of our own implementations for the way they model state and pass along resources or access to `IO`. In listing 5.4 we can see a much better modular design suggestion with a usage example in listing 5.5. The monad that allows functions to access any of its contents and perform I/O is restricted through polymorphism when subfunctions are called. Constraints can be added to a function's type to grant access to more of the state or or removed to limit it. Moreover, the `Resources` type and the type-classes could be expanded to accommodate for more resources.

That way we can achieve decent system flexibility and still clearly compartmentalize functions, limiting their access to what would be otherwise very close to a global, imperative-like state. Included in that is the "evil" I/O access which can be added or removed using the `MonadIO` class, which `ReaderT r IO` already has an instance for.

5.2.2 Abstraction vs. explicitness

In the previous subsection, we have already mentioned abstraction as the means of gaining flexibility — excessive abstraction results in too much flexibility, as seen in hAsteroids. Conversely, too much explicitness leads to a lack of flexibility, as seen in pure-asteroids.

However, despite explicitness being one of the key features that make functional programming safer and easier to argue about, *expressiveness* is the feature more worthy of pursuing. We can see explicit data flow as the way of reaching formal safeness, but it is expressiveness that helps us humans *see it*. Furthermore, explicitness does not imply expressiveness. The flow of data can be described very precisely, but it may be so complicated that it does not make the code any clearer. That is when abstraction steps in place, and when used wisely, it leads to a more understandable code with fewer distractions. All of that, while in the background still being technically pure and formally safe.

Figure 5.3 illustrates the relationships between our four identified foreground attributes and the two main background attributes we should be seeking.

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

import Control.Monad.IO.Class
import Control.Monad.Reader

class MonadReader r m => TexturesReader r m where
    askForTextures :: m Textures

class MonadReader r m => RandomReader r m where
    askForRandom :: m RandGen

type WithResources a = ReaderT Resources IO a

type Textures = String
type RandGen = [Int]
data Resources =
    Res
    { textures :: Textures
    , random   :: RandGen
    }

instance Monad m => TexturesReader Resources (ReaderT
 ↪  Resources m) where
    askForTextures = asks textures

instance Monad m => RandomReader Resources (ReaderT
 ↪  Resources m) where
    askForRandom = asks random
```

Listing 5.4: Modular reader monad design.

```haskell
newtype World = World Int deriving Show

main :: IO ()
main = do
    let world = World 10
    let resources = Res
            { textures = "Current world state: "
            , random = [4, 2]
            }
    runReaderT (game world) resources

game :: World -> WithResources ()
game w = do
    newW <- step w
    draw w
    quit <- liftIO $ getLine
    unless (quit == "q") $
        game newW

step :: (RandomReader r m) => World -> m World
step (World state)= do
    rand <- head <$> askForRandom
    return $ World $ state + rand

draw :: (MonadIO m, TexturesReader r m) => World -> m ()
draw w = do
    t <- askForTextures
    liftIO $ putStrLn $ t ++ show w
```

Listing 5.5: Modular reader monad usage example.

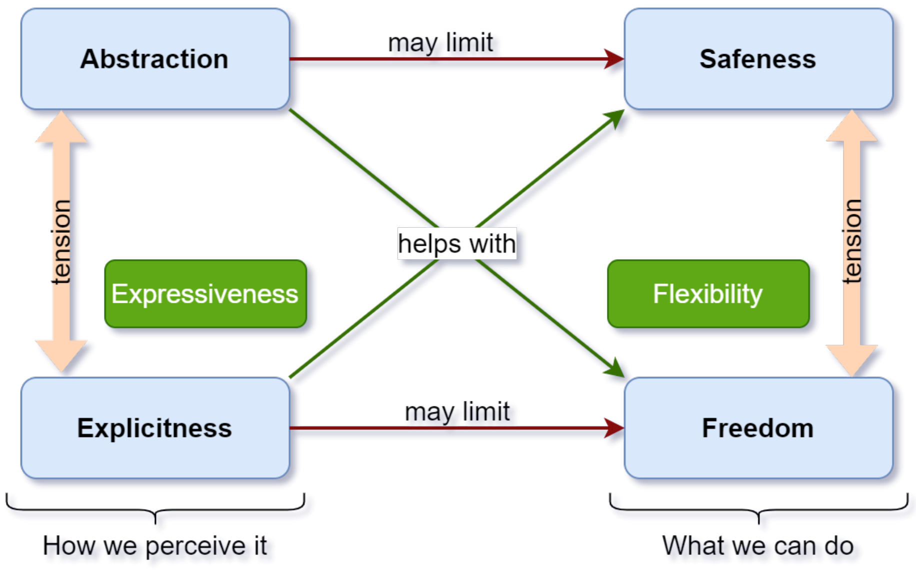

Figure 5.3: The four discussed qualities of design.

As with many things in life, we need to find a balance between abstraction and explicitness so that our programs are still easily provable to be correct while not being painful to write and modify.

When comparing the modeling of state in our three games in subsection 5.1.5, we have already pointed at how monads can emulate certain features of imperative languages, and at the same time, stay technically pure.[8] We see the example in listing 5.5 as a good balance between abstraction and explicitness. We lean towards modeling the game world state as an explicit variable since it is the most important data structure. Mistakes in manipulation with it may be critical, and as the likely most extensive data structure, it could be highly beneficial to process it in parallel.

On the other hand, things like the state of a random number generator or other resources meant only for reading, such as textures,[9] can

8. Philip Wadler speaks more on "combining the indulgences of impurity with the blessings of purity" by using monads in his paper *Comprehending Monads*.[**wadlermonads**]
9. In a larger game, textures may need to load dynamically; still, they are read-mostly data.

be hidden in an appropriately named monad without an issue. That way, we keep expressiveness but allow for flexibility. As an additional supporting example, we consider the following two function types:

```
step :: (RandomReader r m) => World -> m World

step' :: (RandomReader s m, WorldState s m) => m ()
```

One function takes a value of type `World` and returns a new value of the same type under the influence of a reader monad. While the other is obscurer and returns nothing but monadic effects. At last, we admit that this is to an extent a matter of personal taste and that some may prefer the effectful approach, despite what we have argued.[10]

5.3 Summary

– draft/notes

it remains to be true even for game development that the code can be more expressive, safer and briefer — there is a balance between abstraction and explicitness

we have discussed the relationships of design qualities 5.3

the language it self being very high-level and abstract can make it much more difficult to optimize, which is important for games — languages like C++ have the advantage

it is said that Garbage Collection (GC) can be a problem in Haskell with interactive real-time applications, like video games, but it has been proven that it is ok for small to medium sized games — paradox: pure functional design is great for long term and large scale, with Haskell the upfront cost is higher but the system produced is more likely to be of high quality

We have rediscovered that Haskell can be very imperative, as explained by Simon P. Jones and Philip Wadler in their paper *Imperative Functional Programming*.[**imperativefp**] In that paper, they described for the first time, how monads could be used for input/output operations in lazy languages, and confirmed what we see in hAsteroids — with the `IO` monad Haskell can very closely resemble C. Yet, it

10. An example of such an approach can be seen in the Dino Rush game, described in the blog post *A Game in Haskell - Dino Rush*[10] mentioned previously in section 1.1.2.

keeps the benefits of functional style intact by clearly separating the imperative from the functional.

classes can (and should) be used for polymorphism, increasing flexibility scalability and reducing code duplicity

perhaps also quote Carmack again, when he said how over the years of supervising hundreds of programmers working of thousands and thousands of lines of code, he saw that anything that is legal for the compiler will end up in your code base eventually — Haskell's "brutal" functional purity is a win

however it remains to be proven in practice on large scale games...

Temporary notes on comparing

These are other ways of structuring the comparison. Please, excuse the mess.

Common game engine features

here we compare the already described games by how they deal with the common needs of a game

Data modelling

how data is modeled — we have an example of ECS, "struct" collections and object collections (in OOP data is modelled as objects together with related functionality)

ECS - data locality and component design - fast and flexible

struct collections - compartmentalization - safe and could be made to run in parallel

object collections -

Input handling

in pure-asteroids we keep a input state variable and then pass it to the stepper function

in hAsteroids and the imperative version it is an effectful code — *difference between a state monad abstraction and the state of an object*

World stepping

again pure-asteroids sends an explicit world state variable through functions

hAsteroids with ECS a sequence of 12 calls - could be grouped but the changes are still across components not objects

the imperative version iterates over the collections and calls `advance()`, for some reason `draw()` increments the asteroid rotation — *the line between freedom and safeness*

consistency - with apecs the result could differ based on the order in which entities are iterated over (# todo - make sure it is that way +

apecs-stm solves this (?)), pure-asteroids naturally step regardless of order and imp-asteroids use a flag to mark objects for destruction and then do it later

Detecting and handling collisions

Output handling

in pure-asteroids the drawScene is the only impure part of `gameLoop`, besides event polling, time measuring and calling a delay. it cannot alter the world

in apecs `cmapM_` gives us an almost safe almost read only access to world, but other `cmap` can be used inside of it so not very safe

imperative version, unfortunate design choice, `Game.draw()` not only draws but also increments asteroid rotation and decrements bullet health (that is used as time to live) and it even flags bullets as dead if health reaches 0

5.4 Attributes

We evaluate our described implementations based on several desirable attributes for video games, some of which were promised in section 1.1 as benefits of Haskell.

aspects of Modularity:

Flexibility and scalability

Briefness and development speed

Haskell has interpreter but can also be compiled -> fast development *and* fast programs

Haskell code is often said to be much shorter than its imperative equivalent.
˜here are some statistics, take it for what it is worth...˜
number of lines, words, characters
`wc --help`: "A word is a non-zero-length sequence of characters delimited by white space"

hAsteroids

```
~/ba-thesis/hAsteroids$ wc -lwm src/*.hs app/Main.hs
  207   844  6609 src/Collisions.hs
  185   717  5380 src/Components.hs
  135   437  3551 src/Draw.hs
   46   165  1229 src/GameLoop.hs
   64   227  1854 src/Initialize.hs
  132   647  4811 src/Input.hs
  245   803  7054 src/Resources.hs
   57   219  1603 src/SdlWrappers.hs
  175   874  5969 src/Step.hs
   99   356  2583 src/Utility.hs
   67   172  1540 app/Main.hs
 1412  5461 42183 total
```

pure-asteroids

```
~/ba-thesis/pure-asteroids$ wc -lwm src/*.hs src/Step/*.hs
↪  app/Main.hs
  149   635  4327 src/Draw.hs
   95   350  3017 src/EventProcessing.hs
  109   404  3631 src/GameLoop.hs
   58   203  1509 src/Initialize.hs
   85   384  2859 src/Input.hs
  114   381  3334 src/Resources.hs
   56   221  1862 src/Step.hs
  197   578  3975 src/Types.hs
   89   306  2013 src/Utility.hs
   17    41   315 src/Step/Asteroids.hs
  124   571  4467 src/Step/Bullets.hs
   18    87   515 src/Step/Common.hs
   24    63   468 src/Step/Score.hs
  136   667  4754 src/Step/Ship.hs
  109   540  3630 src/Step/Ufos.hs
   50   123  1167 app/Main.hs
 1430  5554 41843 total
```

Asteroids by Jason Halverson

```
~/Asteroids$ wc -lwm *.cpp
  195   376  5416 asteroids.cpp
```

```
   78   182  2133 bullet.cpp
   45   163  1303 driver.cpp
  141   278  3043 flyingObject.cpp
  531  1549 14674 game.cpp
   67   197  1662 point.cpp
   98   228  2218 ship.cpp
  712  2805 22545 uiDraw.cpp
  331  1355 11533 uiInteract.cpp
   38    80   602 velocity.cpp
 2236  7213 65129 total

~/Asteroids$ wc -lwm *.h
  105   196  2327 asteroids.h
   36    81   790 bullet.h
   60   127  1384 flyingObject.h
  103   261  2938 game.h
   48   159  1302 point.h
   63   126  1227 ship.h
  135   580  5979 uiDraw.h
  133   644  5045 uiInteract.h
   29    62   626 velocity.h
  712  2236 21618 total

~/Asteroids$ wc -lwm *.h *.cpp | grep total
 2948  9449 86747 total
```

# TODO: process into charts perhaps and comment on it

Safeness and testability

C++ code needs testing more and it is more difficult to do - example... `const` keyword is a very good mechanism that is not used to its full potential in the imperative example.

Haskell has more than `const`, we have already pointed out how the type of a function in pure-asteroids clearly limits its effects. However, hAsteroids hints that poor design choices can negate this effect.

Program performance

with low-level languages it is easier to optimize, they do not hide what the computer is actually doing. Haskell is *very* abstract, its formality allows the compiler to do a lot of optimizations, but it requires deeper knowledge of `ghc` for the programmer to know how the code will perform. Laziness is an example of a language feature that can cause problems with performance...

Listing that I ended up not using... probably not!

Conclusion

Haskell is great.

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, 2013 [visited on 2021-05-05]. Available from: `https://web.archive.org/web/20131023233640/http://thedoteaters.com/?bitstory=asteroids`.
14. JORDAN, Mark. *Entities, components and systems* [online]. Medium, 2018-11-20 [visited on 2021-05-05]. Available from: `https://medium.com/ingeniouslysimple/entities-components-and-systems-89c31464240d`.
15. CARPAY, Jonas. Apecs: A Type-Driven Entity-Component-System Framework [online]. 2018 [visited on 2021-05-01]. Available from: `https://github.com/jonascarpay/apecs/blob/master/apecs/prepub.pdf`.
16. MAGUIRE, Sandy. *Why Take Ecstasy* [online] [visited on 2021-05-01]. Available from: `https://reasonablypolymorphic.com/blog/why-take-ecstasy/`.
17. CARPAY, Jonas. *Apecs* [online] [visited on 2021-04-22]. Available from: `https://hackage.haskell.org/package/apecs-0.9.2/docs/Apecs.html`.
18. CARMACK, John. *John Carmack's keynote at Quakecon 2013 part 4* [online]. YouTube, 2013 [visited on 2021-05-06]. Available from: `https://youtu.be/1PhArSujR_A?t=125`.

```
// C++
void drawShip(const Point &center, int rotation) {
    const PT pointsShip[] = {
    // top     r.wing   r.engine l.engine  l.wing    top
       {0, 6}, {6, -6}, {2, -3}, {-2, -3}, {-6, -6}, {0, 6}
    };
    glBegin(GL_LINE_STRIP);
    for (int i = 0; i < sizeof(pointsShip)/sizeof(PT); i++)
    {
        Point pt(center.getX() + pointsShip[i].x,
                 center.getY() + pointsShip[i].y);
        rotate(pt, center, rotation);
        glVertex2f(pt.getX(), pt.getY());
    }
    glEnd();
}
```

```
-- Haskell
drawShip :: SDL.Renderer -> Ship -> IO ()
drawShip renderer =
    drawShape renderer . shipPoints

shipPoints :: Ship -> [V2 Double]
shipPoints s =
    map (+ sPos)
        [ -17 *^ (facing - perp facing) -- left
        ,  25 *^  facing                -- front tip
        , -17 *^ (facing + perp facing) -- right
        , -8  *^  facing                -- back
        ]
    where
        sPos = s ^. sPosition . pVect
        facing = angle $ s ^. sAngle
```

Listing 5.6: Comparing ship drawing in imp-asteroids and pure-asteroids.

Index

A More examples of functional elegance

maybe?
compile a bunch of things that are just "nice"

B Electronic attachments

attachments structure and description

C Build instructions

The build process for both hAsteroids and pure-asteroids is very similar. The following requirements must be met first.

Haskell Tool Stack

> Both games are projects managed using the Haskell Tool Stack,[1] therfore `stack` needs to be installed

SDL libraries

> sdl must be installed
>
> instructions for windows and ubuntu

Operating System

> Tested on Ubuntu 20.04 and Windows 10 build 19042

With that, we can run `stack build` in the game's directory ... things happen ... To launch the game run `stack run` or `stack exec <name-of-the-executable>`.

1. About `stack`: `https://docs.haskellstack.org/en/stable/README/`