University of Wisconsin Milwaukee

UWM Digital Commons

Theses and Dissertations

May 2023

Future of Functional Reactive Programming in Real-Time Systems

Anisha Tasnim
University of Wisconsin-Milwaukee

Follow this and additional works at: https://dc.uwm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Tasnim, Anisha, "Future of Functional Reactive Programming in Real-Time Systems" (2023). *Theses and Dissertations*. 3219.
https://dc.uwm.edu/etd/3219

This Thesis is brought to you for free and open access by UWM Digital Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of UWM Digital Commons. For more information, please contact scholarlycommunicationteam-group@uwm.edu.

FUTURE OF FUNCTIONAL REACTIVE PROGRAMMING IN REAL-TIME SYSTEMS

by

Anisha Tasnim

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Computer Science

at

The University of Wisconsin-Milwaukee

May 2023

ABSTRACT

FUTURE OF FUNCTIONAL REACTIVE PROGRAMMING
IN REAL-TIME SYSTEMS

by
Anisha Tasnim

The University of Wisconsin-Milwaukee, 2023
Under the Supervision of Professor Tian Zhao

The evolution of programming paradigms and the development of new programming languages are driven by the needs of problem domains. Functional reactive programming (FRP) combines functional programming (FP) and reactive programming (RP) concepts that leverage asynchronous dataflow from reactive programming and higher-level abstractions building blocks from functional programming to enable developers to define data flows and transformations declaratively. Declarative programming allows developers to concentrate more on the problem to be solved rather than the implementation details, resulting in efficient and concise code. Over the years, various FRP designs have been proposed in real-time application areas. Still, it remains unclear how FRP-based solutions compare with traditional methods for implementing these applications.

In this survey, we studied the usefulness of FRP in some real-time applications, such as game development, animation, graphical user interface(GUI), and embedded system. We

conducted a qualitative comparison for game development and studied various applications in animation, GUI, and embedded systems. We found that using FRP in these applications is quite difficult because of insufficient libraries and tools. Additionally, due to high learning curves and a need for experienced developers, the development process in FRP takes time and effort. Our examination of two well-known games: Asteroid and Pong, in three programming paradigms: imperative programming using the Unity game engine, FP in Haskell, and FRP in the Yampa library, showed that imperative programming is effective in terms of performance and usability. The other two paradigms for developing games from scratch are inefficient and challenging. Despite the fact that FRP was designed for animation, the majority of its applications are underperforming. FRP is more successful for GUI applications, where libraries like RxJS have been used in many web interfaces. FRP is also applied in developing embedded system applications for its effective memory management, maintainability, and predictability.

Developing efficient solutions from scratch is not suitable in FRP due to several factors, such as poor performance compared to other programming paradigms, programming complexity, and a steep learning curve. Instead, developers can be benefited from utilizing FRP-supported modular platforms to build robust and scalable real-time applications.

To

my parents,

and brothers

TABLE OF CONTENTS

1 Introduction **1**

2 Background Studies **4**

2.1 Imperative and Functional Programming 4

2.2 Reactive Programming 6

2.3 Functional Reactive Programming 7

3 Comparison Methodology **10**

3.1 Resource Collections 10

3.2 Exampled-based Comparison for Game Development 12

3.3 FRP in Animation, GUI, and Embedded System 13

4 Game Development **14**

4.1 Game Architecture 14

4.1.1 ECS in Unity Engine 15

4.1.2 Game in FP and FRP 17

4.2 Installation Process 19

4.2.1 Unity 19

4.2.2 Haskell with Gloss 20

4.2.3 Haskell with Yampa 20

4.3 Asteroid Game 21
4.3.1 Asteroid Game Elements 21
4.3.2 Unity 21
4.3.3 Functional Programming 25
4.3.4 Functional Reactive programming-Yampa 29
4.4 Pong Game 46
4.4.1 Pong Game Elements 46
4.4.2 Unity 47
4.4.3 Functional Programming 50
4.4.4 Functional Reactive programming-Yampa 56

5 Animations, GUIs, and Embedded Systems 69
5.1 Animation 69
5.1.1 Types of Animation 70
5.1.2 How to create animation 72
5.1.3 FRP and Animations 73
5.2 Graphical User Interface 74
5.2.1 FRP and GUIs 75
5.3 Embedded Systems 76
5.3.1 FRP and Embedded Systems 77

6 Discussion and Conclusion 79
6.1 Game Development 79
6.1.1 Similarities 79
6.1.2 Comparisons 80
6.2 Animations, GUIs, and Embedded Systems 85
6.3 Conclusion 85

LIST OF FIGURES

4.1 Entity Component System Game Architecture[1] 16

4.2 Game Architecture in FP and FRP [1] 18

4.3 The main loop execution for Asteroid in FP 25

4.4 The main loop execution for Asteroid in FRP 30

4.5 The *endlessLoop* function 43

4.6 The main loop execution for Pong in FP 51

4.7 The main loop execution for Pong in FRP 56

LIST OF TABLES

6.1 Comparison among Unity, Functional Programming with Haskell, and Functional Reactive Programming with Yampa 81

6.2 The game features comparison among Unity, Functional Programming with Haskell, and Functional Reactive Programming with Yampa 83

ACKNOWLEDGMENTS

First and foremost, I express my deepest gratitude and appreciation to my advisor and thesis supervisor, Dr. Tian Zhao. His guidance, expertise, and support throughout the research journey have been invaluable. Without his insightful feedback and constant encouragement, I could not think of completing this research.

I would also like to thank my classmates and friends for their valuable input, discussions, and suggestions.

Lastly, I must extend my gratitude to my parents and brothers for their unconditional love, support, and encouragement. Their belief in me has been a constant source of motivation. Their unwavering support made this accomplishment possible.

Chapter 1

Introduction

Information and communications technology (ICT) has grown significantly in recent years, providing more robust, flexible applications that meet all modern demands, ensuring instantaneous response and uninterrupted service. Traditional programming is no longer sufficient to meet all the needs of large-scale data alone, and scientists are discovering new approaches for building irrepressible modern applications. Scientists introduced reactive systems that are effective for real-time applications to create more scalable, efficient, and responsive software. RP was initially developed to solve complex event-driven systems. However, with the complex and vast number of events, systems become difficult to manage.

Fran [2] was introduced as the first FRP language to simplify the development of reactive systems, particularly for animation utilizing FP concepts. Functional reactive programming provides a declarative and composable approach well-suited to solving real-time interactive applications. However, Fran is implemented in Haskell, which inherits memory management characteristics of the language, including lazy and pull-based evaluation leading to *space-time* leaks. A time leak in a real-time system is when a time-dependent computation falls behind the current time because it requires time to compute and respond [3]. Scientists solved the problems in NewFran [4] employing a new evaluation model named the push-pull

or hybrid model and solved *space-leak* in Yampa [5] by limiting the expressiveness using the signal function [6].

Game development is a complex and interactive process of creating video games. Without user interaction, a game would be an animated movie. An excellent game will consider high-user interaction, high-resolution graphics support, and amusing animation, which makes it more realistic and appealing to gamers, especially children. Unity and Unreal engines are widely adopted in game development due to the engines' flexibility and performance. Unity is popular among indie developers because of its low cost. These game engines also provide animation facilities. However, animators find some animation tools efficient, including Blender, Animate, and Maya, because of their versatile supports. Animation engineers may provide animation to the game development team to integrate those into the game or marketing team for an advertisement. Animations work as a storyteller in the game. Performance and feature flexibility are the fundamental requirements for animations and games.

Graphical user interfaces (GUIs) are essential to application development. It deals with several key features, including real-time response. A user's main focus in using a GUI application will be the real-time response and performance. Among several GUI libraries widely used are React and RxJS. Embedded systems highly depend on real-time response, including satisfying hard real-time constraints. Engineers often choose responsiveness over performance if any trade-off is necessary. Developers prefer C or C++ languages for embedded systems due to their low-level abstractions and performance.

These real-time applications require frequent user interactions with a timely response. Real-time applications are designed to process events or user inputs immediately as they occur. FRP is designed to handle real-time applications focusing on data flow and time-varying relationships among data and behaviors. FRP uses declarative programming concepts, event streams, and reactive values to describe the behavior of a system over time. It allows developers to model the dynamic nature of real-time systems clearly and concisely. FRP concepts

can improve the responsiveness and interactivity of real-time applications and simplify the development process in every real-time application. Because of this consideration, developers have found it effective in real-time systems and have been utilizing it in development. However, several complexities, such as lack of resources, steep learning curve, and lack of development tools, make the paradigm challenging for developing applications from scratch.

Our primary focus in the survey is to investigate the effectiveness of FRP in various domains, such as game development, animations, GUIs, and embedded systems. To achieve this, we conducted a thorough study by implementing two demo games in Unity, collecting existing implementations in FP and FRP, and reviewing related research papers in FRP for animations, GUIs, and embedded systems. Our survey results provide valuable insights into the potential benefits and limitations of using FRP in these domains, providing future research and development direction.

The report is organized as follows. The background information on programming paradigms is in Section 2. Section 3 represents how we gathered resources to perform the survey and based on what criteria, we conducted the comparison. Section 4 presents the implementation of Asteroid and Pong games in Unity, FP, and FRP. We provide the usefulness of FRP in animations, GUIs, and embedded systems, including adopted languages and libraries in Section 5. We discuss the results in Section 6 and conclude in Section 6.3 with future directions.

Chapter 2

Background Studies

This section describes the fundamental concepts of imperative programming, functional programming, reactive programming, functional reactive programming, and their application areas.

2.1 Imperative and Functional Programming

Imperative programming focuses on solving the program by composing explicit commands in a specific order. The computer executes these instructions in the defined order to produce the desired output and side effects using the control flow statements, including iterations. It may involve mutable states (variables) that refer to the values stored in the memory. A function or method may produce side effects by updating mutable variables. Imperative programming may be divided into procedural programming and object-oriented programming (OOP). For example, C and Fortran are procedural languages, while Java, C++, and C# are OOP languages.

Because of their flexibility, low-level programming constructs, and efficiency, imperative languages, especially C and C++, are widely used in high-performance and real-time

domains, including system programming, game development, and scientific computing. Imperative languages enable the efficient use of memory and CPU resources that ensure performance. Computing languages such as Fortran also offer high-performance scientific computing libraries. Despite the performance, imperative languages are only suitable in some application domains due to their limitations, such as heavy reliance on mutable states, which can lead to intricate programming errors and code that is difficult to understand, refactor, reuse, integrate, or extend.

Functional programming (FP) is a declarative-style programming paradigm where programs are constructed by applying and composing functions, and functions are treated as first-class citizens. The main focus of FP is what to solve rather than how to solve, and it is suitable for solving complex mathematical problems without extensive side effects. FP offers clean and efficient code with abstractions, such as pure functions that depend only on input parameters. A single-threaded functional program is deterministic in always returning the same output for the same input parameters. Its immutable states help control the changes in an application's states. It promotes referential transparency, which is the ability to modify an expression in a program and its result without affecting its result or behavior such that the program always delivers the same result for a given input parameter. The first functional programming language is Lisp (list processing), designed to manipulate data strings easily. Haskell, Erlang, Elixir, etc., are widely used languages in FP.

FP has been gaining popularity in recent years because of its productivity in particular application domains. For example, *Whatsapp*, which is a widely used messaging app, utilizes Erlang in its implementation so that it only needs 50 engineers to support 500 million users [7]. FP languages such as Erlang are superior in concurrency functionalities partly due to their use of immutable states. FP ensures comprehensibility through its pure functionalities, and testing and debugging are easy in functional programming with an immutable state and no hidden output. However, since FP mostly relies on garbage collection, which can add

additional overhead compared to imperative programming, it can lead to poor performance. Beginners may find coding in a recursive style challenging.

Imperative and functional paradigms are different in approaches to problem-solving, and both are successful in specific areas. Many programming languages, like Python and Kotlin, use both paradigms' concepts and are great at managing complex computing problems. Rather than selecting a language based on its popularity, it is essential to consider what a program needs to accomplish when choosing the correct language and approach. It ensures that the solution best fits the application and makes it more effective and efficient.

2.2 Reactive Programming

Reactive programming (RP) is based on asynchronous data streams and the propagation of changes through streams. The fundamental concept of RP is the stream of events. It is particularly well-suited for developing applications involving real-time data processing and event-driven user interfaces. Reactive systems are highly responsive, providing interactive feedback. As stated in the *Reactive Manifesto* [8], reactive systems are more flexible, loosely-coupled (which refers to the independence of components that provides flexibility), scalable, and fault tolerant.

Recently, RP has been getting popular in handling event-driven code efficiently, especially in web, mobile, and IoT applications. Reactive libraries, such as ReactiveX [9], ObservableComputations [10], Svelte [11], ReactiveCocoa [12] are being used largely in creating responsive applications effectively. Despite the benefits of reactive programming, drawbacks, such as the steep learning curve, make the system challenging to adopt and implement. Additionally, the lack of well-performed tooling sometimes demotivates the developers.

2.3 Functional Reactive Programming

Functional Reactive Programming (FRP) is a programming paradigm that emphasizes the use of FP concepts for reactive systems. FRP highlights higher-order functions to manipulate data streams efficiently, such as *map* and *filter*, to solve reactive programs. These functions may be used to modify and combine data streams, making it easier to express complex activities in a concise and composable manner. It was initially used in Fran [2] for animation. Despite its apparent flaws, it has inspired many scientists. FRP languages and libraries are designed for specific problem domains. But still, just a few are successful and well-adopted.

FRP inherits features from FP and RP, which makes it a powerful programming paradigm. FRP adopts the declarative style from FP, where the programmer specifies what the system should do rather than how it should do it. The declarative style makes the program easy to understand. Immutability is another key feature adopted from FP. The use of immutable data structures ensures consistency and predictability. Immutable data structures also make it easier and avoid concurrency problems. FRP utilizes the event-driven system, which reacts to changes in data streams as they occur. This feature makes FRP well-suited to applications with a high volume of real-time events, such as user interfaces. FRP uses higher-order functions to compose simple behaviors into more complex ones, which makes it easier to build reusable and maintainable code.

Fran introduced two abstractions: *Events* and *Behavior*. *Events* are streams of values that occur at a particular time. *Behaviors* are time-varying values that continuously change over time. Behavior is treated as a first-class value.

$$Event\,\alpha = Time \times \alpha \tag{2.1}$$

$$Behavior\,\alpha = Time \rightarrow \alpha \tag{2.2}$$

In *Arrowwized-FRP* [6], two different abstractions were introduced: *Signal* and *Signal Function (SF)*. *Signal*s are continuous data streams that change over time, similar to *Behavior*. The only difference is *Signal*s are not a first-class value here. *SF*s transform a time-varying input into a time-varying output and are treated as first-class citizens.

$$Signal\,\alpha = Time \rightarrow \alpha \tag{2.3}$$

$$SF\,\alpha\,\beta = Signal\,\alpha \rightarrow Signal\,\beta \tag{2.4}$$

The evaluation model concerns how the changes are propagated across a dependency graph of values and computations. The evaluation models of FRP must ensure that change propagation works correctly and efficiently. With pros and cons, three types of models are pull-based, push-based, and push-pull (Hybrid) models. A pull-based or demand-driven evaluation model requires a value to pull itfrom the source, which leads to a *time-leak*. Additionally, when there are too many devices to check for the value, the time required to pull them can exceed the time available to service the IO device. This model is easy to achieve in Haskell-based programs due to Haskell's lazy evaluation. Push-based model is called the data-driven model, where the data is pushed to its dependent computation when the source has newly available data. The push-based model does not have the time leak problem though it introduces glitches. The problem with the push-based model is the wasteful recomputation of everything there is available data. However, languages implementing a push-based model need an efficient solution to handle the recomputation. The hybrid model solves the problem by utilizing both models.

Fran was introduced for interactive multimedia animation that supports 2D animation. The fundamental abstractions are *Event* and *Behavior*. These notations are captured as datatypes, not as values. Fran is embedded in Haskell and uses a pull-based model. It has a *space-time* leak. As an extension to Fran, in 1997 and 1998, the author presented [13] and

[14] on 3D animation containing *space-time* leaks because of the notations and evaluation model. Yampa [5, 3, 15] was introduced as an *Arrowwized-FRP* that uses *Signal* and *SF* as an abstraction. New notation solves the *space-leak* but uses the pull-based model that leads to *time-leak*. Several *Arrowwized-FRP*s are proposed [16, 17, 18, 19, 20]. FrTime [21, 22] was introduced in 2006, which uses *Events* and *Behavior* and a push-based model. To prevent recomputation, the language ensures that computations dependent on unchanged values are not scheduled for execution. Flapjax [23, 24] is a language for web programming embedded in javascript inspired by FrTime. It uses the *Event* and *Behavior* and pure push-based model. Processing the dependency graph in topological order avoids recomputation and enhances efficiency. Newfran [4] is an extension of the original Fran developed in 2009 to solve the *space-time leak*. It is a Domain Specific Language (DSL) for animation and uses a hybrid model embedded in Haskell.

FRP is well-suited for building user interfaces, as it allows developers to create reactive, dynamic interfaces that respond to user input and changes in the underlying data with successful libraries like RxJS and Elm. FRP allows developers to develop reactive game mechanics that respond to user input and events in real-time. Some FRP libraries and game development tools include Frag [25], Haskanoid [26], Bogre-Banana [27], FunGEn [28], Gloss [29, 30], and Helm [31]. FRP also fits in robotics applications, allowing developers to build reactive control systems that respond to sensor input. Tools and libraries include YFrob [32] and swarm [33]. Additionally, FRP has been used in other application domains, including animation [2, 13, 14, 4], graphical user interface [34, 35, 36, 37, 38], embedded systems [39, 40, 41, 42], robotics [43, 44, 45], vision [46, 47], security [48], time-domain applications: synchronous system [49, 50, 51], asynchronous system [52, 53, 54] and priority-based task scheduling [55, 56, 57, 58, 59, 60, 61], fault tolerance [62, 63, 64, 65, 66], refactoring [67], and testing and debugging [68, 69, 70].

Chapter 3

Comparison Methodology

This section describes how we collected the review materials, how we evaluated the application of FRP in game development using examples, and how we surveyed the use of FRP for animation, GUI, and embedded systems.

3.1 Resource Collections

This survey focused on game development, animation, GUI, and embedded systems. We used two types of resources to perform the study: examples for game development applications and research papers. Additionally, we used numerous online resources. Due to the paradigm characteristics and lack of published works, we used two demo implementations of age-old games (**Asteroid**[1] and **Pong**[2]) to compare the results among three paradigms (Imperative, FP, and FRP).

We implemented a simple **Asteroid** game in imperative programming using Unity and C# [71] and collected source codes implemented in FP [72] and FRP [73]. We used two reference implementations collected from *GitHub* for **Pong**, one in FP [74] and the other in

[1]asteroid, https://en.wikipedia.org/wiki/Asteroids_(video_game)
[2]https://en.wikipedia.org/wiki/Pong

FRP [75], and implemented a simple version of it in imperative programming in Unity [76] with the help of [77].

The research papers are divided into game development, animation, GUIs, and embedded systems. We collected publications from various sources, including IEEE Xplore[3], ACM Digital Library[4], ResearchGate[5], and Google Scholar[6], and found numerous published works on FRP languages, libraries, frameworks, and applications. Our search criteria included functional reactive programming, game development, animation, GUIs, and embedded system. We collected papers in functional reactive programming from the beginning (1997) to 2022.

A Survey [78] was focused on the effectiveness of functional reactive programming language, especially on Elm in Game development. To conduct the survey, they implemented a shooting game named aa[7]. They found several improvements are necessary for Elm, including flexible features like Haskell and testing and debugging facilities.

In 2013, a survey was performed on reactive programming [79], where the authors focused on programming languages. They surveyed two types of languages: the FRP languages as siblings of reactive programming and the cousins of reactive programming that do not provide the primitive abstraction for time-varying value but provide automatic propagation of state changes and offer other reactive programming features such as glitch avoidance. Siblings of RP include Fran, Yampa, Frappe, FrTime, NewFran, Flapjax, Scala.React, AmbientTalk/R, and cousins are Cells, Lamport Cells, Trellis, SuperGlue, Radul/Sussman Propagators, Coherence, and .NET Rx. They performed a comprehensive study on the languages using six criteria, including basic abstractions, evaluation model, lifting, multi-directionality, glitch avoidance, and support for distribution. They found that multi-directionality is not sup-

[3]https://ieeexplore.ieee.org/
[4]https://dl.acm.org/
[5]https://www.researchgate.net/
[6]https://scholar.google.com/
[7]https://www.generaladaptive.com/

ported by any FRP languages they examined, whereas a few cousins of RP support multi-directionality features. There is still an open area for research in the interactive distributed application for FRP and cousins of RP. They also stated that reactive and distributed programming could raise glitches that need to be investigated. In comparison, our work is not a comprehensive survey of FRP-related languages but focuses on FRP's usefulness in application domains, especially in game development.

3.2 Exampled-based Comparison for Game Development

We examined two games: Asteroid and Pong, and compared the results among three programming paradigms: imperative programming (the Entity Component System or ECS design), FP, and FRP, in two ways, including the internal functionalities of the paradigms and the development process flexibilities.

The internal differences among the three paradigms include performance, workflow, live coding, unit testing, feature extension, debugging single component or logic, understanding code environment, environment setup, scalability, and learning curve. Table 6.1 represents the comparison.

The comparison among the paradigms in terms of game features and development flexibilities include game object creation, importing an image, destroying a game object, collision detection, reaction after a collision, detecting game objects, re-positioning, re-scaling, and game loop. The development process flexibility for the game is in Table 6.2.

3.3 FRP in Animation, GUI, and Embedded System

We examined the effectiveness of FRP languages, libraries, and frameworks in animation, GUIs, and embedded systems. We included adopted applications, libraries, and languages.

Chapter 4

Game Development

This section represents the game development process in different programming paradigms, which is the process of transforming real-life or fictional activities into a game. The process is complex and involves a variety of computer science disciplines, including artificial intelligence, graphics design, and algorithms. The procedure entails comprehending the game's fundamental ideas and its design and development, testing, and release. Every stage is significant, especially game design that picks the right programming paradigm such as imperative, functional, or reactive functional programming) and the host language.

4.1 Game Architecture

Software game production is expanding and becoming more and more well-liked every day. One of the problems with game creation is to create high-caliber games, where developers must adhere to the best practices and processes from the software engineering field. Games can be classified into many genres, which include action, shooters, combat, racing, adventure, sports, role-playing, strategy, simulations, puzzles, dancing, and music. Game development is a complicated process with diverse collaborative team activities and pro-

cesses, including sound, game-play, art, artificial intelligence, control systems, and human factors. Fundamentally, game development is a subset of the software development process that includes a few extra steps for artistic design, creative presentation, and visual appeal. Visualization or attracting audiences is a crucial job for game developers.

4.1.1 ECS in Unity Engine

Entity Component System (ECS) is a new design pattern that plays an important role in modern real-time interactive systems. ECS aims to decouple objects into separated data and logic. Objects are divided into three elements: entities, components, and systems. This pattern is similar to data-oriented programming (DOD), which favors composition over inheritance. The three elements of ECS are composed of a context manager, which helps to keep track of which component belongs to which entities and which system should be used. It has the following characteristics:

- **Entities** that are unique identifiers.
- **Components** are logical representations of an entity's properties and data.
- **Systems** are functions associated with entities that have a certain set of components.
- Entities can contain zero or more components.
- Entities can change components dynamically.

ECS game architecture is shown in Figure 4.1. ECS allows highly optimized memory management, parallelization, and flat, easy-to-manage class hierarchies [80, 81]. The system does not return any value. It changes the state of different components by performing data transformation instead. ECS ensures modularity by using DOD composition, which makes the development process easier to manage by keeping the functions small and independent

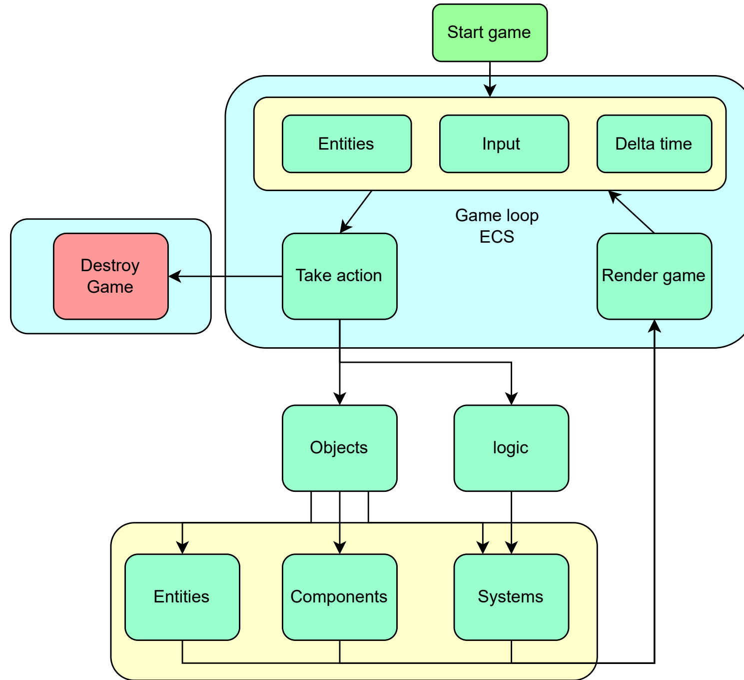

Figure 4.1: Entity Component System Game Architecture[1]

by avoiding dependencies between different parts. It also improves readability. Focusing on transforming the data from the input makes testing and debugging easy [82]. Generally, game developers prefer Entity Component System (ECS) architecture to develop games [83] because of the platform availability, tooling flexibility, and resources.

The top most popular game engines that use ECS are Unity [84] and Unreal Engine (UE) [85]. UE is used to develop 3D first-person shooting games and First-person role (FPR) games. Unity is a widely used cross-platform game engine popular among indie game developers because of its detailed documentation, flexibility, portability, and rich resources. To use Unity efficiently, developers need to know the ECS architecture, C# language, and sufficient knowledge of game logic.

Unity has a computation loop that performs a number of iterative tasks, including reading user inputs, updating animation graphics, and calculating the attributes of the game objects. Unity uses a polling evaluation model that checks the game state during each animation

frame. To mitigate the effects of the variable speed of frame update, Unity has a fixed update method that allows certain computations, such as physics be updated in regular intervals regardless of the frame speed. To improve performance, Unity also allows interval events such as collision to be handled using callbacks to detect these events centrally, while user logic only needs to react to these events if they occur. Thus, while Unity has a pull-based model, it achieves good performance by avoiding excessive checking of internal events. In comparison, push-based models, which only react to events when they occur, have to deal with potential glitches [86]. Unity has ScriptableObject, a serializable unity class that allows storing a large quantity of shared data independently. It makes it easier to make changes, debug, and test.

Unity has a large developer community, a well-documented tutorial, and a user-friendly engine. Developers can develop toy games by dragging and dropping and creating entities and components the engine provides. Unity also allows the runtime ordering of an entity. For games with a large number of entities, the game performance will deteriorate. To address this, Unity employs the Data-Oriented Tech Stack (DOTS) [87], a data-oriented design approach to game development in Unity. DOTS is a combination of the ECS framework, C# Job system for multithreading, and Burst compiler for optimized native code. However, DOTS has a steep learning curve, and it is suitable only if the performance requirement justifies it.

4.1.2 Game in FP and FRP

Functional programming is constructed by applying and composing functions. Functional programming is popular for its immutable state and clean and elegant code base. FRP combines functional and reactive programming and is used in real-time applications.

Animation can be handled using libraries. There are numerous libraries in Haskell for game development. Game development in FP and FRP uses similar game architecture. The

game development process can be divided into three sections: initial state, game loop, and destroy (or end) state. Three sections have individual tasks. The first step is creating the initial state or initial world. It will be used when the game starts for the first time. The second and most crucial step is the game loop, which includes several tasks. Lastly, the end or destroy state is responsible for the end game state. Figure 4.2 shows the game architecture for FP and FRP.

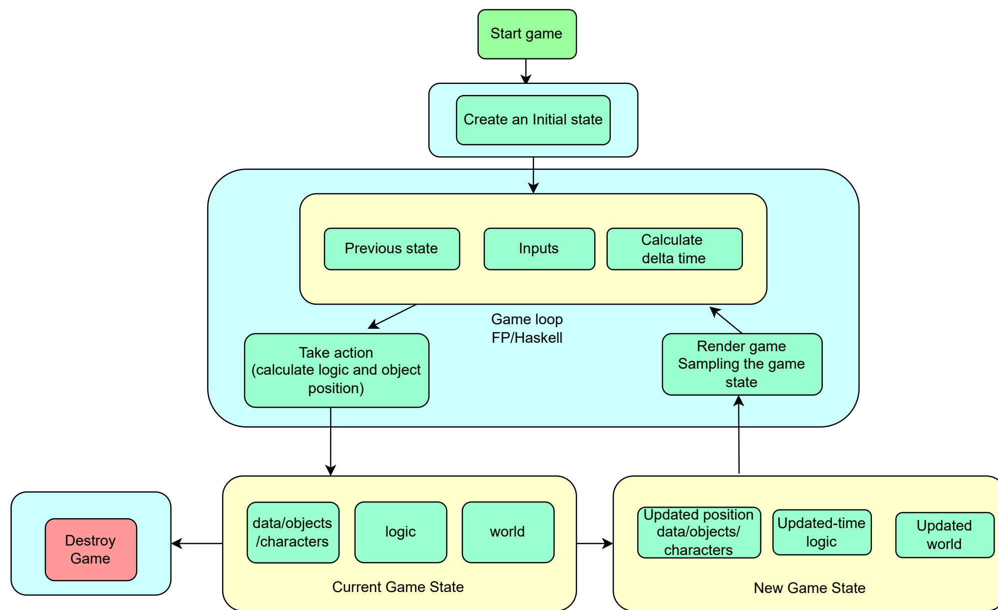

Figure 4.2: Game Architecture in FP and FRP [1]

The game loop includes three steps. The first step is determining delta time, user input, and previous state. The second step is to take action based on the first step. There will be two types of actions: exit the game state or calculate the current and the new game states using the previous game state, time, and logic. The last step is to render the game by sampling the game state. In the next loop, the new game state will become the previous state.

Haskell is rich in libraries to develop games [88], including FP and FRP libraries and game engines. Among several libraries, Gloss [89] is popular and widely used. Both (FP

and FRP) paradigms have the Gloss library. Gloss is primarily used for graphics, animation, and simulation-related works and can also handle game loops, game states, and I/O in Haskell [90]. Each language has a different gloss play library. For instance, Yampa [16] has a gloss interface [29] where the update state uses a signal function from the Yampa library, and Reactive-banana has a gloss interface [30]. To handle the game loop, Yampa has the *reactimate* [91] function, which is an input-process-output loop. It interfaces between pure Yampa *SF* and the potentially impure external world.

Gloss is popular in the graphical application field, including animation and game development, and Haskell has Gloss packages for 2D vector graphics, animations, and simulations containing many useful modules. Gloss uses OpenGL[1]. Haskell also has a library named apecs [92], which is a fast and type-driven Entity-Component-System library. A few game engines [93, 94] are in FP and some FRP game engines [95, 96, 31] using OpenGL. However, these engines need common functionalities such as audio and 3D.

4.2 Installation Process

This section highlights the installment process for Windows OS to develop Asteroid and Pong games in Unity, Haskell in the FP, and Haskell with the Yampa library in FRP.

4.2.1 Unity

Unity is a well-developed tool for game development with paid and free tools. From the Unity website, download Unity Hub for the appropriate OS. After downloading, install it with a specific Unity version where the latest is preferable. Unity has some device dependencies that can be found on the download page [97].

[1]openGL, https://www.opengl.org/

4.2.2 Haskell with Gloss

The installation process for Haskell is provided in [98]. It is straightforward in Windows using GHCup. Users can download all the necessary tools to manage the Haskell project using the *Cabal or Stack* at the same time.

The next step is to install the Gloss library [89]. Gloss uses OpenGL, which requires an OpenGL toolkit GLUT [99]. Users should get the .dll file, rename it to *glut32.dll*, and move it to the System32 directory in Windows.

4.2.3 Haskell with Yampa

The installation process for Yampa is similar. After installing Haskell, Gloss, and *GLUT*, we need to install Yampa. After a successful installation of Haskell, Yampa should be installed.

```
$ cabal update
$ cabal install --lib Yampa
```

Additional libraries need to be installed. For example, the Asteroid game uses the Haskell Graphics Library (HGL)[2] instead of the Gloss library. To install HGL, the following command should be executed.

```
$ cabal install HGL-version
```

All the installation processes can be done with either Cabal or Stack. The HGL for Windows has some base or basic library dependencies. Developers should be aware of the appropriate version of the base and HGL and dependencies of tools, and devices.

[2]HGL, https://www.haskell.org/hugs/pages/libraries/HGL/Graphics-HGL.html

4.3 Asteroid Game

Asteroid [100] is a space-themed multi-directional shooter arcade game designed in 1979 by Lyle Raines. It uses high-resolution and black-and-white vector-scan screens, where Delman originally wrote the program code in the machine language of the MOS 6502 CPU.

4.3.1 Asteroid Game Elements

The Asteroid game contains three main elements: players, asteroids, and bullets. A player has several functionalities, including position changing and bullet shooting. When a player shoots and destroys asteroids, they get points. Asteroids are spawned randomly and fall downwards after spawning. If a player collides with an asteroid, the player dies, and the game ends.

4.3.2 Unity

We developed the asteroid game in Unity 2021.3.12f1 editor on Windows 10 [71]. A sample scene has been created with three entities (*Player, Asteroid*, and *Bullet*), custom scripts (components), and built-in components. *Asteroid*s and *Bullet*s will be spawned over time. We positioned the asteroid in six random positions from which they will be spawned.

Transform, *SpriteRenderer*, and *RigidBody2D* are built-in components. By default, every object has a *Transform* component along with the other properties to manipulate its position and size. *SpriteRenderer* renders 2D graphics, and *RigidBody2D* is a physics component for 2D sprites, which provides functionalities that affect the sprite by gravity and can be controlled from a script using the appropriate collider component. Additionally, we have a canvas component named *score*, which shows the player's score that increments by one when the player destroys an asteroid. This functionality is handled in Score.cs script.

Several scripts are created and attached to the entities, including PlayerMovement, Play-

erShooting, Score, Bullet, Asteroid, Spawner, and SpawnManager. Each script contains customized methods. The *Player* entity uses the PlayerMovement and PlayerShooting scripts, the *Spawner* entity uses the Spawner script, the *SpawnManager* entity uses the SpawnManager script, and the *Score* entity uses the Score script. The Spawner and SpawnManager scripts handle the asteroid spawning feature.

The input devices for a game can be a keyboard, mouse, touchpad, or custom input devices. *EventSystem* processes the input and sends events to objects. The input device for this game is the keyboard, where input keys are 'W', 'S', 'A', and 'D' to change the player's position to up, down, left, and right, respectively. The input functionalities are in the PlayerMovement script shown in Listing 4.1.

```
if (Input.GetKey(KeyCode.W)){
    transform.position += transform.up * Time.deltaTime * _speed;
    transform.eulerAngles = new Vector3(0f, 0f, 0f);
}
else if (Input.GetKey(KeyCode.S)){
    transform.position += transform.up * Time.deltaTime * _speed;
    transform.eulerAngles = new Vector3(180f, 0f, 0f);
}
else if (Input.GetKey(KeyCode.A)){
    transform.position += transform.up * Time.deltaTime * _speed;
    transform.eulerAngles = new Vector3(0f,0f, 90f);
}
else if (Input.GetKey(KeyCode.D)){
    transform.position += transform.up * Time.deltaTime * _speed;
    transform.eulerAngles = new Vector3(0f, 0f, -90f);
}
```

Listing 4.1: Player movement I/O in Asteroid

```
if (Input.GetKeyDown(KeyCode.Space)){
    Instantiate(_bullet,transform.position,transform.rotation);
}
```

Listing 4.2: Shooting bullets I/O in Asteroid

A player shoots with the space key from the keyboard, implemented in PlayerShooting.cs file and shown in Listing 4.2. We handled two types of collisions: an asteroid colliding with a player and a bullet hitting an asteroid. Every entity, such as *Player, Asteroid*, and *Bullet*, has a *Collider2D* component attached, and the *OnTriggerEnter2D* method is used to verify if the game object collides with another *Colloder2D* object. The *CompareTag* method makes the process easier to find the appropriate object.

The game is over if an asteroid collides with the player. The player object and collided asteroid object must be destroyed in the state. The relevant code is in Listing 4.3.

```
private void OnTriggerEnter2D(Collider2D collision){
    if (collision.CompareTag("Player")){
        Destroy(collision.gameObject);
        Time.timeScale = 0;
        Destroy(gameObject);
    }
}
```

Listing 4.3: Collision between player and asteroid

The bullet-asteroid collision is handled in Bullet.cs, and Listing 4.4 contains the code. The score needs to be incremented when the bullet hits an asteroid. Score.cs script includes the functionality, and Listing 4.5 shows the relevant code.

```
private void OnTriggerEnter2D(Collider2D collision){
    if (collision.CompareTag("Asteroid")){
```

```
        OnAsteroidHit?.Invoke();
        Destroy(collision.gameObject);
        Destroy(gameObject);
    }
}
```

Listing 4.4: Destroy the asteroid

```
void UpdateScore(){
    _currentScore++;
    _text.SetText($"Score: {_currentScore}");
}
```

Listing 4.5: Updating Score when a bullet hits a asteroid

A static event *OnAsteroidHit* is created and registered to the Bullet script.

```
public static event Action OnAsteroidHit;
```

Listing 4.6: Static Action event

The event callbacks are in Listing 4.7 from two source files (Bullet.cs and Score.cs). These callbacks are related to the polling stage. Deregistering the static event after destroying the bullet is not associated with polling.

```
OnAsteroidHit?.Invoke();
Bullet.OnAsteroidHit += UpdateScore;

Bullet.OnAsteroidHit -= UpdateScore;
```

Listing 4.7: Callback methods

4.3.3 Functional Programming

This game was developed in 2015 [72] using Haskell and Gloss. As shown in Figure 4.3, the architecture of the game is a state machine that includes the initial game state, a game-over state, and a main loop driven by Gloss. The main loop uses the *simulateWorld* function to check for collisions between game objects and update game states, and it also uses the *handleEvents* function to react to user inputs.

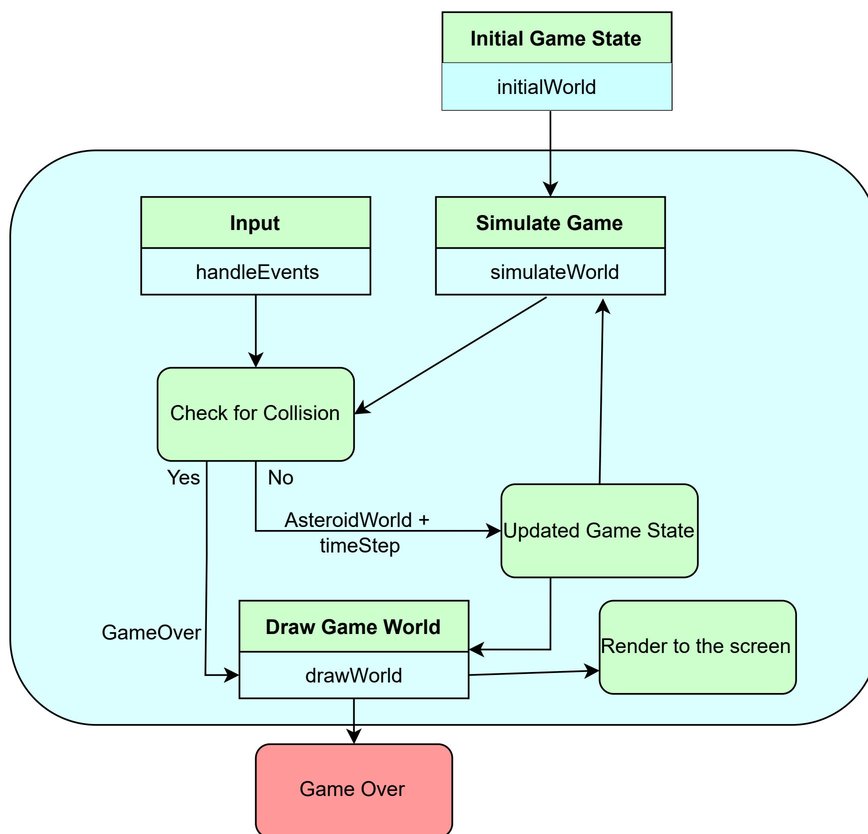

Figure 4.3: The main loop execution for Asteroid in FP

The game objects are defined using type constructors shown in Listing 4.8, which includes *Ship, Bullets, Rocks*, and *UFO*.

```
data Ship   = Ship PointInSpace Velocity       deriving (Eq,Show)
data Bullet = Bullet PointInSpace Velocity Age deriving (Eq,Show)
```

```haskell
data Rock    = Rock PointInSpace Size Velocity  deriving (Eq,Show)
data UFO     = UFO PointInSpace Velocity        deriving (Eq,Show)
```

Listing 4.8: Type Constructors for *Ship, Bullets, Rock*, and *UFO*

The game state is defined in the *AsteroidWorld* type constructor shown in Listing 4.9.

```haskell
data AsteroidWorld = Play [Rock] Ship UFO [Bullet]
                   | GameOver           deriving (Eq,Show)
```

Listing 4.9: *AsteroidWorld* type constructor

Some related type variables are defined in Listing 4.10.

```haskell
type Velocity     = (Float, Float)
type Size         = Float
type Age          = Float
```

Listing 4.10: Necessary Type Aliases

The *play* function from *Pure.Game* module of Gloss handles the game loop and is shown in Listing 4.11, where the initial world type is *AsteroidWorld*. The game starts with spawning five rocks from different positions, a ship in the middle of the screen, a UFO, and an empty list of bullets. Listing 4.12 shows the *initalWorld* game state.

```haskell
main = play
       (InWindow "Asteroids!" (550,550) (20,20)) black 24
       initialWorld drawWorld handleEvents simulateWorld
```

Listing 4.11: Main game loop for Asteroid in FP

```haskell
initialWorld :: AsteroidWorld
initialWorld = Play
```

```
[Rock (150,150)  45 (2,6)
,Rock (-45,201)  45 (13,-8)
,Rock (45,22)    25 (-2,8)
,Rock (-210,-15) 30 (-2,-8)
,Rock (-45,-201) 25 (8,2)
] -- The default rocks
(Ship (0,0) (0,0)) -- The initial ship
(UFO  (75, 75) (2, 5)) -- The initial UFO
[] -- The initial bullets (none)
```

Listing 4.12: Initial state/ Initial game world for Asteroid in FP

The *drawWorld* function converts a game world (*AsteroidWorld*) to a *Picture*. When *drawWorld* takes a gameplay state, it returns a picture with the transformed ship, asteroids, UFO, and bullets. It produces a picture with the text "Game Over" and "Click right mouse button to restart" when the parameter is *GameOver* state. The relevant code is in Listing 4.13 and 4.14.

```
drawWorld (Play rocks (Ship (x,y)(vx,vy))(UFO (ux,uy)(uvx, uvy))bullets)
    = pictures [ship, asteroids, ufo, shots]
      where
        ship      = color red (pictures [translate x y (circle 10)])
        asteroids = pictures[(color orange(polygon(asteroidShape x y s)))
                    | Rock   (x,y) s _ <- rocks]
        ufo       = color green (pictures [translate ux uy (circle 10)])
        shots     = pictures [translate x y (color red (circle 2))
                    | Bullet (x,y) _ _ <- bullets]
```

Listing 4.13: The *drawWorld*: Game play world for Asteroid in FP

```
drawWorld GameOver  = pictures [
    scale 0.3 0.3 . translate (-400) 0 . color red . text $ "Game Over!",
    scale 0.1 0.1 . translate (-1150) (-550) . color white
    . text $ "Click right mousebutton to restart"
]
```

Listing 4.14: The *drawWorld*: game over for Asteroid in FP

The *handleEvents* function takes an *Event* (user input), an *AsteroidWorld*, and generates an *AsteroidWorld*. The Game interface has an efficient data constructor named *EventKey*, which takes *Key, KeyState, Modifiers*, and *(Float, Float)* as input. When a player clicks the right key of the mouse and the *AsteroidWorld* is *GameOver* state, it returns to its initial state. If the player clicks the left key and the world is in some gameplay state, it will produce a new *AsteroidWorld* with a released bullet towards the mouse direction. Otherwise, it will remain in the same game world. The source code is provided in Listing 4.15.

```
handleEvents :: Event -> AsteroidWorld -> AsteroidWorld
handleEvents (EventKey(MouseButton RightButton)Down _ _)GameOver =
initialWorld

handleEvents (EventKey (MouseButton LeftButton) Down _ clickPos)
     (Play rocks (Ship shipPos shipVel) ufo bullets)
     = Play rocks (Ship shipPos newVel) ufo (newBullet : bullets)
    where
        newBullet = Bullet shipPos (negate 150 .* norm (shipPos .-
clickPos)) 0
        newVel    = shipVel .+ (50 .* norm (shipPos .- clickPos))
handleEvents _ w = w
```

Listing 4.15: IO Events in Haskell for Asteroid

The *simulateWorld* handles the game state transformation, which converts *AsteroidWorld* to *AsteroidWorld* based on the timestep and input *AsteroidWorld*. When the *AsteroidWorld* is the *GameOver*, it will return to the *GameOver* world. Otherwise, it will process time steps and a game world and detect collision. If a collision happens with a ship and rock, it will be *GameOver*. Otherwise, it will return to another gameplay world containing updated rocks, a newly positioned ship, UFO, and bullets. This method has several internal functions, and calculations are not included in Listing 4.16. A bullet-rock collision generates two scenarios: both will destroy, or a rock can be divided into some pieces according to its size. The minimum size of the rock is 7, and the maximum age of a bullet is 5 seconds. If the bullet hits the rock, it will destroy it or break it based on its size, and the bullet will be destroyed. If the bullet misses a rock and the age of the shot is 5 seconds, the bullet will be destroyed automatically. Listing 4.16 shows the relevant code.

```
 simulateWorld :: Float -> AsteroidWorld -> AsteroidWorld
 simulateWorld _         GameOver           = GameOver
 simulateWorld timeStep (Play rocks(Ship shipPos shipV)(UFO ufoPos ufoV
)bullets)
        | any (collidesWith shipPos) rocks = GameOver
        | otherwise = Play (concatMap updateRock rocks) (Ship newShipPos
 shipV)
                          (UFO newUFOPos ufoV) (concat (map updateBullet
bullets))
```

Listing 4.16: Update/Simulate the game world in Haskell Asteroid

4.3.4 Functional Reactive programming-Yampa

Haskelloids [73] is a recreation of the original Asteroid game in Yampa. It uses the HGL Graphics library, FRP *Event*s, and HGL key module for IO. Unlike the FP version of the

Asteroid game, the main loop of Haskelloids is implicit, while the game elements are more modular and combined through the FRP operators. The main abstractions in Yampa FRP are signal functions, which are instances of the *Arrow* type class. Each SF transforms one or more time-varying signals. As shown in Figure 4.4, Haskelloids runs the *endlessLoop* SF, which takes inputs through the *sense* function and makes game state decisions with the *actuate* function. The *endlessLoop* SF is composed of several SFs responsible for tasks such as generating random asteroids, updating the attributes of ships and bullets, checking for collisions, and producing dust effects.

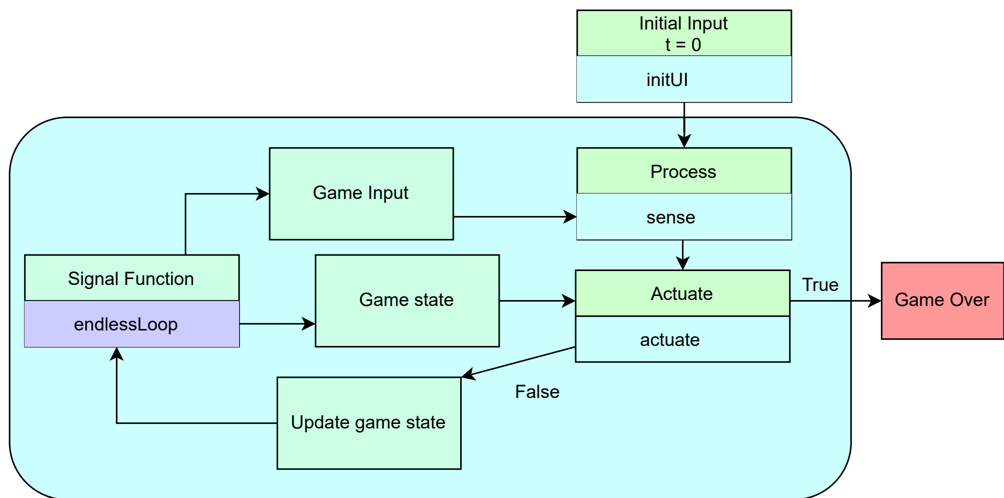

Figure 4.4: The main loop execution for Asteroid in FRP

Haskelloids defines user inputs as FRP events, where the *procWindowEvs* function collects all user inputs and folds them into the *UserInput* record, which together with interval events (i.e., asteroid hits) form the *ObjectInput* record. The *ObjectOutput* record includes information such as the game object positions, collision results, and spawned objects. The primary data type *Object* is a SF that maps *ObjectInput* to *ObjectOutput*.

The data type for *UserInput* defines the input types. Listing 4.17 shows the record type data structure. Yampa has two events to model discrete events: *Event ()* when an event occurs and *NoEvent*.

```haskell
data UserInput = UserInput { -- events that correspond to user input
    uiLeftClick  :: FRP.Event (),
    uiFire       :: FRP.Event (),
    uiHyperSpace :: FRP.Event (),
    uiTurnLeft   :: FRP.Event Bool,
    uiTurnRight  :: FRP.Event Bool,
    uiThrust     :: FRP.Event Bool
}
```

Listing 4.17: User Input Data Structure

Listing 4.18 includes the source code for *inputInit*, where the initial user input type is *UserInput*, and the field values are *NoEvent* because, at time 0, nothing happens.

```haskell
inputInit :: UserInput
inputInit = UserInput {
    uiLeftClick  = FRP.NoEvent,
    uiFire       = FRP.NoEvent,
    uiHyperSpace = FRP.NoEvent,
    uiTurnLeft   = FRP.NoEvent,
    uiTurnRight  = FRP.NoEvent,
    uiThrust     = FRP.NoEvent
}
```

Listing 4.18: Initial User input at t=0 for Asteroid in FRP

The *getWindowEvs* function collects all the events since the last window tick based on any pending events on the window using *maybeGetWindowEvent* from the HGL Window module. If there is any pending event, it will return a list of IO events; otherwise an empty list. The *procWindowEvs* function processes the window events and applies to the game input using another function named *procWindowEv*. The *procWindowEv* works on a single

event, which handles two discrete events: mouse clicks and keyboard presses. The source code is in Listing 4.19, and Input.hs contains functions related to input events.

```
procWindowEvs :: UserInput -> [Event] -> UserInput
procWindowEvs gi evs = foldl procWindowEv gi evs
procWindowEv :: UserInput -> Event -> UserInput
procWindowEv gi Button{ isLeft = left, isDown = down }
  | left && down = gi{ uiLeftClick  = FRP.Event ()   }
procWindowEv gi Key{ keysym = k, isDown = down }
  | isLeftKey   k          = gi{ uiTurnLeft   = FRP.Event down }
  | isRightKey  k          = gi{ uiTurnRight  = FRP.Event down }
  | isUpKey     k          = gi{ uiThrust     = FRP.Event down }
  | isShiftLKey k && down  = gi{ uiHyperSpace = FRP.Event ()   }
  | isCharKey   k && keyToChar k == ' ' && down = gi{uiFire = FRP.Event()}
procWindowEv gi _ = gi
```

Listing 4.19: Handling User input

Every data constructor, type, and function related to geometry is in Geometry.hs. It uses several HGL modules (*Draw, Units*), Data (*Ratio, List*), and many more. The type of *Angle* is *Double*, *Point2* is (*Double, Double*), a *Segment* is (*Point2, Point2*). Type constructor for *Shape* including polygon, circle, and the line defined in Listing 4.20.

```
data Shape = Poly [Point2]
           | Circ Point2 Double
           | Ln [Point2]        deriving (Show)
```

Listing 4.20: Geometric data constructor for *Shape*

```
data Figure = Polygon [Point2]
            | Circle Point2 Double
            | Line [Point2]
```

```
| Translate Point2 Figure
| Scale Double Figure
| Rotate Angle Figure     deriving (Show)
```

Listing 4.21: Geometric data constructor for *Figure*

A purely functional representation for linear mapping is shown in Listing 4.21. The type constructor named *Figure* handles polygons, circles, lines, translate, scale, and rotate.

Graphics.hs uses Haskell *Control Arrow* library, HGL *Draw* module with Graphics constructor, which is a special type of *Draw* monad, *Picture* constructor, and *Geometry* module. This file contains two functions: one is to draw a figure, and the other is to draw a shape. The source code is in Listing 4.22.

```
drawFigure :: Figure -> Graphic
drawFigure f = drawShape . shape $ f
drawShape :: Shape -> Graphic
drawShape (Ln ps)         = polyline . map (round *** round) $ ps
drawShape (Poly ps)       = polyline . map (round *** round) $ ps
drawShape (Circ (x,y) r) = ellipse (x'-r', y'-r') (x'+r', y'+r')
                            where
                                x' = round x
                                y' = round y
                                r' = round r
```

Listing 4.22: Transform *Figure* or *Shape* to *Graphic* for Asteroid in FRP

Asteroids, a ship, bullets, and dust are this game's four types of objects or characters. The data constructor of the *ObjectClass* contains these. *ObjectInput* is a record type constructor with *oiUserInput* and *oiHit*. Listing 4.23 contains the source code of *ObjectClass* and *ObjectInput*.

```haskell
data ObjectInput = ObjectInput {
    oiUserInput :: !UserInput,
    oiHit       :: Event ()
}
data ObjectClass = Asteroid
                 | Bullet
                 | Ship
                 | Dust     deriving (Eq)
```

Listing 4.23: The *ObjectClass* and *ObjectInput* type constructor for Asteroid in FRP

The data constructor for the output object type is in Listing 4.24. It uses an *Object* type as a SF of *ObjectInput* and *ObjectOutput*. The data constructor has a position, collision, graphics, spawn, object class, and kill.

```haskell
data ObjectOutput = ObjectOutput {
    ooPos      :: Point2,
    ooCllsnBox :: !Shape,
    ooGraphic  :: !Graphic,
    ooSpawnReq :: Event [Object],
    ooObjClass :: ObjectClass,
    ooKillReq  :: Event ()
}
type Object = SF ObjectInput ObjectOutput
```

Listing 4.24: Constructors for Output Object (*ObjectOutput*) and the *Object* SF

The *teleport* and *reload* are the other helper SFs shown in Listing 4.25. The *teleport* wraps coordinates around a one-dimensional coordinate system with a fixed buffer size. The *reload* yields an *Event* on the first *Event* to arrive and then waits for the specified interval until yielding another *Event* again.

```
teleport :: Int -> Int -> Double -> SF Double Double
teleport sz buf x0 = switch(init &&& (init >>> evt))
                      (\(f, x) -> teleport sz buf . f $ x)
  where
    init :: SF Double Double
    init = (integral >>^ (x0+))
    evt :: SF Double (Event (Double -> Double, Double))
    evt = proc x -> do
    let sz' = fromIntegral sz
      buf' = fromIntegral buf
      lt = (\d -> if d then Event(2*buf' + sz' +)
                      else NoEvent).(< 0 - buf')$x
      gt = (\d -> if d then Event((2*buf' + sz') `subtract`) else NoEvent)
                      .(> sz' + buf') $ x
    returnA -< flip attach x . merge lt $ gt

reload :: DTime -> SF (Event ()) (Event ())
reload intvl = proc e -> do switch (constant NoEvent &&& identity)
                            (\_ -> pause) -< e
               where
                 pause :: SF (Event ()) (Event ())
                 pause = switch(once &&& after intvl())(\_ ->reload intvl)
```

Listing 4.25: Auxiliary SFs for Asteroid in FRP

This file contains some additional functions to check which object collides with another object and return that is defined in the *hits* function. The *hits* function uses another helping function named *hits'*. Listing 4.26 contains the source code. This method uses another new type (*ILKey*), defined in the Data/IdentityList.hs file. The type of *ILKey* is *Word32* and represents the maximum number of *Object*s that are 2^{32} after initialization. The actual

collision function is in Listing 4.27.

```
hits :: [(ILKey, ObjectOutput)] -> [ILKey]
hits objs = hits' objs []

hits' :: [(ILKey, ObjectOutput)] -> [(ILKey, ObjectOutput)] -> [ILKey]
hits' []              _    = []
hits' ((k, oo):rest) seen =
  let cllsn = any (\x -> (collideObj (ooObjClass oo)
                 . ooObjClass . snd $ x) &&
                 (intersect (ooCllsnBox oo).ooCllsnBox.snd $ x))
                 (seen ++ rest)
  in if cllsn then k : hits' rest ((k,oo):seen)
              else hits' rest ((k,oo):seen)
```

Listing 4.26: Object collision functionality For Asteroid in FRP

```
collideObj :: ObjectClass -> ObjectClass -> Bool
collideObj Dust     _        = False
collideObj _        Dust     = False
collideObj Asteroid Asteroid = False
collideObj Asteroid _        = True
collideObj Ship     Bullet   = False
collideObj Ship     _        = True
collideObj Bullet   Ship     = False
collideObj Bullet   _        = True
```

Listing 4.27: Collision detection For asteroid in FRP

Mandatory objects or characters and their properties are mentioned in the src/Haskelloids/Object folder. *Asteroid, Ship, Bullet*, and *Dust* properties are defined in the Asteroid.hs, Ship.hs, Bullet.hs, and Dust.hs, respectively.

The Asteroid.hs contains the constructor for the asteroid, SFs, and other necessary functions. Three different sizes of asteroids: small, medium, and large, are named *RdSmall, RdMedium*, and *RdLarge*, respectively. These have some specific *Double* type values handled by the *Scale* function. The size of *RdLarge* is 1, *RdMedium* is 0.5, and *RdSmall* is 0.25. The speed range of an asteroid is (30, 100). The *asteroidSF* SF randomly generates asteroids in different places with different shapes depending on the asteroid's size. Listing 4.28 contains the *asteroidSF*.

```
asteroidSF :: RandomGen g => g -> Point -> Point2 -> Double
                              -> Angle -> RoidSz -> Double -> Object
asteroidSF g (w, h) (x0, y0) s o sz fig =
    let s'   = minSpeed + ((maxSpeed - minSpeed) * s)
        !vx  = s' * cos o
        !vy  = s' * sin o
        !buf = round $ fromIntegral buffer * scale sz
        (fg',cllsnFig) = case fig of
                r | r <= 0.25 -> (roidFigure1, roidCllsnFigure1)
                  | r <= 0.5  -> (roidFigure2, roidCllsnFigure2)
                  | r <= 0.75 -> (roidFigure3, roidCllsnFigure3)
                  | otherwise -> (roidFigure4, roidCllsnFigure4)
    in proc oi -> do
        x <- teleport w buf x0 -< vx
        y <- teleport h buf y0 -< vy
        let roidShape = shape . Translate (x,y) . Scale (scale sz) $ fg'
        hit <- arr oiHit -< oi
        returnA -< ObjectOutput {
            ooPos      = (x,y),
            ooCllsnBox = roidShape,
            ooGraphic  = drawShape roidShape,
            ooSpawnReq = hit `tag` (flip evalRand g $ do {
                          frag <- replicateM 2.makeFragment(w,h)(x,y)s $ sz
```

```
                    ;dust <- replicateM 8.makeDust$(x,y)
                    ;return(dust++ if sz/=RdSmall then frag else [])
                    }),
      ooObjClass = Asteroid,
      ooKillReq  = hit
    }
```

Listing 4.28: Asteroid SF (*asteroidSF*)

Ship.hs contains the properties, including the SFs. The initial ship object (*shipFigure*), thrusting ship object (*thrustersFig*), and the collided ship (*cllsnFigure*) source codes are in Listing 4.29.

```
shipFigure :: Figure
shipFigure = Polygon[(15,0),(-15, 10),(-9, 8),(-9, -8),(-15, -10),(15, 0)]

thrustersFig :: Figure
thrustersFig = Polygon [(-9, 8), (-16, 0), (-9, -8)]

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

Listing 4.29: Definition of three different types of ships

Listing 4.30 contains the SF (*shipSF*), which uses several functions, including *teleport, translate, shipFigure*, and *oiHit* importing from the other files. The parameter for the *teleport* function is 30 as the buffer. To make the ship's movement more realistic, it uses *acceleration* and *friction* loss due to the *velocity, turning rate*, and *thruster* flicker period. The default values of these variables are 350, 0.6, 4, and 0.05, respectively. The *reload* time is 0.1 to check whether the player is firing or needs to reload the gun. Additionally, this function has a bullet SF *bulletSF*, which helps spawn the bullet with the ship object. The *bulletSF* uses

a constant value that is the size of the bullet named *bulletBox*. The default value of the *bulletBox* is 15.

```
shipSF :: Point -> Point2 -> Object
shipSF (w, h) (x0, y0) = proc oi -> do
    let ui = oiUserInput oi
    l <- (\d -> if d then -turnRate else 0.0) ^<< hold False -< uiTurnLeft
     ui
    r <- (\d -> if d then  turnRate else 0.0) ^<< hold False -<
   uiTurnRight ui
    -- calculate orientation...
    o <- ((-pi/2)+) ^<< integral -< l + r
    -- ...velocity and acceleration...
    t <- hold False -< uiThrust ui
    th <- arr (\d -> if d then accel else 0.0) -< t
    let tx = th * cos o
        ty = th * sin o
    rec
      ax <- uncurry(-)^<<(returnA ***((* frictionLoss)^<< integral))-< (tx
   , ax)
      ay <- uncurry(-)^<<(returnA ***((* frictionLoss)^<< integral))-< (ty
   , ay)
    vx <- integral -< ax
    vy <- integral -< ay
    -- ...position
    x <- teleport w buffer x0 -< vx
    y <- teleport h buffer y0 -< vy
    -- is the user firing? have we reloaded our guns?
    f <- reload reloadTime -< uiFire ui
    -- are we drawing the thrusters?
    dt <- reload thrusterFlickerPeriod <<^ gate (Event ()) -< t
    -- check for crash
```

```
    die <- arr oiHit -< oi
    -- ...return observable state
    returnA -<
      ObjectOutput {
        ooPos      = (x,y),
        ooCllsnBox = shape . Translate (x,y) . Rotate o $ cllsnFigure,
        ooGraphic  = do { drawFigure . Translate (x,y) . Rotate o $
shipFigure
                        ; if isEvent dt then drawFigure . Translate (x,y)
                        . Rotate o $ thrustersFig else return () },
        ooSpawnReq = f `tag` [blltSpwn (x,y) (vx,vy) o],
        ooObjClass = Ship,
        ooKillReq  = die
      }
    where
     -- blltSpwn - create a new bullet signal function
        blltSpwn :: Point2 -> Point2 -> Angle -> Object
        blltSpwn (x0,y0) (vx, vy) o =
         let (x, y) = (x0 + (bulletBox * cos o), y0 + (bulletBox * sin o))
         in bulletSF (w, h) (x,y) (vx, vy) o
```

Listing 4.30: SF for the ship to come from the opposite direction when it goes out of the window

Bullet.hs contains the SFs for the bullet and other constants. The SF uses the *teleport* function that uses two as buffer size. The speed of the bullet is 500 pixels per second. The maximum age of a bullet is 0.75 seconds. The shape of the bullet is the *Figure* type, which is a *Circle* with a radius of 2. The source code is in Listing 4.31.

```
bulletSF :: Point -> Point2 -> Point2 -> Angle -> Object
bulletSF (w, h) (x0, y0) (vx0, vy0) o =
```

```
    let !vx = vx0 + bulletSpeed * cos o
        !vy = vy0 + bulletSpeed * sin o
    in proc oi -> do
         x <- teleport w buffer x0 -< vx
         y <- teleport h buffer y0 -< vy
         let bulShape = shape . Translate (x,y) $ bulletFigure
             die <- edge <<< (> bulletMaxAge) ^<< time -< ()
             hit <- arr oiHit -< oi
             returnA -<
                ObjectOutput {
                  ooPos      = (x,y),
                  ooCllsnBox = bulShape,
                  ooGraphic  = drawShape bulShape,
                  ooSpawnReq = NoEvent,
                  ooObjClass = Bullet,
                  ooKillReq  = merge die hit
             }
```

Listing 4.31: SF for the *Bullet*

The developers used the dust to provide a realistic view defined in Dust.hs. This file contains two functions. The first function (*makeDust*) is randomly generating dust, and the second and most important is the dust SF (*dustSF*). Additionally, it has other necessary constants. For example, creating dust requires a minimum and maximum dust age of 0.1 and 0.5, respectively. The dust will move automatically without any other particular functionalities. The speed of dust is 200 pixels per second. The source code of the *makeDust* is in Listing 4.32, and Listing 4.33 contains the *dustSF*. Dust does not collide with other objects or request a spawn.

```
makeDust :: RandomGen g => Point2 -> Rand g Object
makeDust (x, y) = do
```

```
    age <- getRandomR (minAge, maxAge)
    x0  <- (x+) `liftM` getRandomR (-10.0,10)
    y0  <- (y+) `liftM` getRandomR (-10.0,10)
    o   <- getRandomR (0, 2*pi)
    return (dustSF (x0,y0) o age)
```

Listing 4.32: Dust generation (*makeDust*)

```
  dustSF :: Point2 -> Angle -> Double -> Object
  dustSF (x0,y0) o age = proc _ -> do
    let vx = speed * cos o
        vy = speed * sin o
    x <- (x0+) ^<< integral -< vx
    y <- (y0+) ^<< integral -< vy
    die <- after age () -< ()
    returnA -< ObjectOutput {
             ooPos      = (x,y),
             ooCllsnBox = shape figure, -- doesn't collide with anything
             ooGraphic  = drawFigure . Translate (x,y) $ figure,
             ooSpawnReq = NoEvent,
             ooObjClass = Dust,
             ooKillReq  = die
           }
```

Listing 4.33: SF for the Dust (*dustSF*)

```
main = do
  g <- newStdGen
  runGraphics $ do
    w <- openWindowEx ("Haskelloids v" ++ verNo) Nothing (wWidth, wHeight)
           DoubleBuffered (Just refreshPeriod')
```

```
reactimate initUI (sense w) (actuate w) (endlessLoop g)
```

Listing 4.34: Haskelloids game loop

The main game loop is in Haskelloids.hs. The *reactimate* function handles the main loop. It takes an initial input, game input and delta time, signal functions related to the window events, and game state. Listing 4.34 shows the main loop for the game.

The main loop executes as Figure 4.4. The loop execution requires several functions, including *sense* (collects window events and returns game inputs and delta time) shown in Listing 4.36, *actuate* (renders the collections of SFs in the window until it gets a *True*) included in Listing 4.37, and *endlessLoop* (game state) SF. The *endlessLoop* comprises several SFs and functions to validate and update the game state by checking collision and updating the score to the screen. The functionality of *endlessLoop* is in Figure 4.5. The source code is in Haskelloids.hs. The initial user input function is in Listing 4.35.

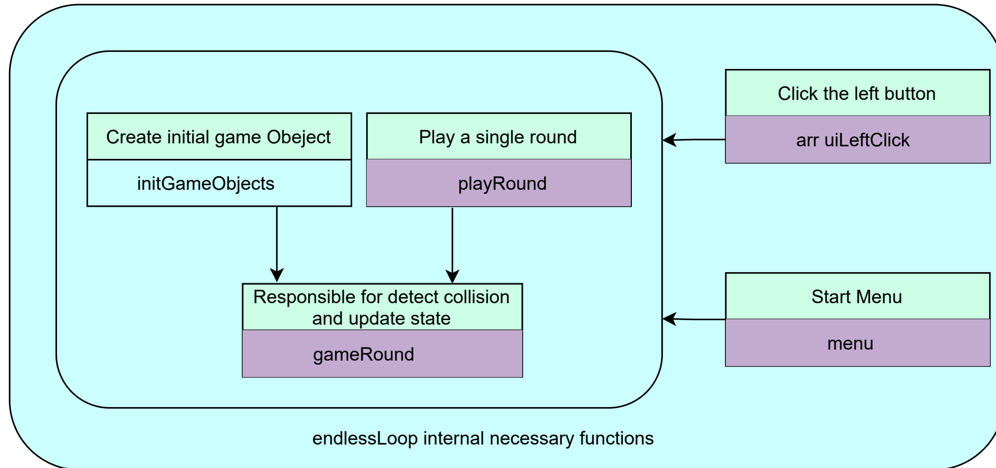

Figure 4.5: The *endlessLoop* function

```haskell
initUI :: IO UserInput
initUI = return inputInit
```

Listing 4.35: Initial user input for Haskelloids game loop

```haskell
sense :: Window -> Bool -> IO (DTime, Maybe UserInput)
sense w _ = do
  evs <- getWindowEvs w
  let gi = procWindowEvs inputInit evs -- fold window events over game
   input
  getWindowTick w
  return (refreshPeriod, Just gi)
```

Listing 4.36: Haskelloids *sense* function

```haskell
actuate :: Window -> Bool -> Graphic -> IO Bool
actuate w _ out = do
  setGraphic w out
  return False
```

Listing 4.37: Haskelloids *actuate* function

```haskell
endlessLoop :: RandomGen g => g -> SF UserInput Graphic
endlessLoop g = let (initObjs, g') = flip runRand g initGameObjects
      (g'', _)       = split g
  in switch (menu g &&& arr uiLeftClick)
       (\_ -> switch (playRound g' initObjs)
       (\sc -> switch(gameOver sc &&& arr uiLeftClick)
                (\_ -> endlessLoop g'')))
```

Listing 4.38: Haskelloids game loop SF (*endlessLoop*)

Initially, game objects are generated randomly through the *initGameObjects* shown in Listing 4.39, which uses the *shipSF* SF to generate ship and *initAsteroids* to create initial asteroid objects where the *initAsteroidsCount* is 5.

```
initGameObjects :: RandomGen g => Rand g (IList Object)
initGameObjects = do
    let ship = shipSF (wWidth, wHeight) centre
    rds <- initAsteroids initAsteroidsCount centre
    return (IL.fromList $ ship : rds)
```

Listing 4.39: Initial game objects creation

Listing 4.40 shows the routing function that routes the object input to the appropriate SF.

```
route :: forall sf. (GameInput, IList ObjectOutput) -> IList sf
                    -> IList (ObjectInput, sf)
route (gi,oos) sfs = IL.mapWithKey routeAux $ sfs
  where
    routeAux :: ILKey -> sf -> (ObjectInput, sf)
    routeAux k obj = (ObjectInput {
                    oiHit = if k `elem` hs then Event () else NoEvent,
                    oiUserInput = giUserInput gi }, obj)
    hs = hits . IL.assocs $ oos
```

Listing 4.40: Routing for the SF

The *gameCore* SF uses the *route* function mentioned in Listing 4.41.

```
gameCore :: RandomGen g => g -> IList Object
               -> SF(GameInput,IList ObjectOutput)(IList ObjectOutput)
gameCore g objs = let (g',_) = split g
                  in dpSwitch route objs(arr(killOrSpawn g)>>> notYet)
```

```
                    (\sfs f -> gameCore g' (f sfs))
```

Listing 4.41: SF that collects the visible SF objects

The *playRound* function handles a single-round game that takes a random number and a list of game objects, and *gameRound* SF is responsible for a single-round game, checks for collisions, and update the game state with the score.

```
playRound :: RandomGen g => g -> IList Object
                              -> SF UserInput(Graphic, Event Score)
playRound g init = proc ui -> do
    go <- gameRound g init -<  ui
    returnA -< (goGraphic go, goGameOver go)
```

Listing 4.42: Play a single round game

Haskelloids.hs contains several SFs to maintain the game state and update the score.

4.4 Pong Game

Pong [101] is one of the earliest table tennis-themed 2D video games created by Allan Alcorn in 1972 and manufactured and published by Atari[3]. It was the first commercially successful video game.

4.4.1 Pong Game Elements

The original Pong has three main elements: two paddles and a small ball. The game can be multiplayer by providing paddles access to two users or a single-player game by playing with the computer. A player can move the paddles using keyboard keys.

[3]Atari, https://atari.com/

4.4.2 Unity

We developed a 2D pong game in Unity 2021.3.12f1 editor on Windows 10 machine [76] with the help of [77].

Pong has four entities a ball, a left racket, a right racket, and walls requiring pictures of a ball, wall, and racket. We named four walls *LeftWall, RightWall, TopWall*, and *BottomWall*, representing the left, right, top, and bottom walls, respectively. Two different entities for rackets named *LeftRacket*, and *RightRacket*, and a ball entity named *Ball*. Canvas elements are also added to represent the separate left and right player scores.

The built-in components are the same as we used in the Asteroid game: the *Transform, SpriteRenderer*, and *RigidBody2D*. The left racket is on the left, the right racket is on the right side of the game scene, and the ball is in the center at the initial stage of the game. The left player controls the racket using the keyboard button 'W' and 'S', and the right player uses the 'up' and 'down' keys to move the racket up and down. Unity's input manager handles the user input functionality.

The six scripts for the game are Ball.cs, Racket.cs, HumanRacket.cs, GameManager.cs, GameOverPanel.cs, and TagManager.cs. Each entity uses one or multiple scripts according to its functionalities and requirements. For instance, the *Ball* entity uses the Ball.cs script, *LeftRacket*, and the *RightRacket* entities use the HumanRacket.cs and TagManager.cs scripts, and so on.

TagManager.cs script has a variable named *wallTag* typed *Tag*. Tag is an *enum* type variable containing four tag values: *leftWall, rightWall, leftRacket*, and *rightRacket* for left wall, right wall, left racket, and right racket, respectively.

The racket game object uses the HumanRacket.cs script, which extends the Racket.cs abstract class. The Racket.cs script contains one event named *OnGameOver*, one abstract method named *Movement*, and several necessary variables and methods.

The source code of Movement implementation is given in Listing 4.43. *AxesName* is a

String variable that takes the direction in which the racket will move.

```
public class HumanRacket : Racket{
    protected override void Movement(){
        float moveAxesValue = Input.GetAxis(AxesName) * moveSpeed;
        rb.velocity = new Vector2(0, moveAxesValue);
    }
}
```

Listing 4.43: Racket movement

To update the game score on the game scene is shown in Listing 4.44. The game is over when one of the players scores 5 points. The *_maxScore* is 5 and is declared in the GameManager.cs.

```
public void GetScore(){
    Score++;
    scoreText.text = Score.ToString();
    if (Score >= GameManager.Instance._maxScore){
        OnGameOver?.Invoke();
        Time.timeScale = 0f;
    }
}
```

Listing 4.44: Score on the game scene

The *GetScore* method is called from the *OnCollisionEnter2D* implemented in the Ball.cs script. When a ball hits the left wall, the right player gets one point, and when the Ball hits the right wall, the left player scores one point. However, when the ball hits the left or right rackets, it calls another method named *wayBall* that defines the direction the ball should move. This script also uses TagManager.cs to get the correct game object. Listing 4.45 is the source code for hitting the ball with other game objects (walls or rackets).

```
 private void OnCollisionEnter2D(Collision2D collision){
   TagManager tagManager = collision.gameObject.GetComponent<TagManager>()
   ;
   if (tagManager == null){
       return;
   }
   Tag tagName = tagManager.wallTag;
   if (tagName.Equals(Tag.leftWall)){
       RightRacket.GetScore();
   }
   if (tagName.Equals(Tag.rightWall)){
       LeftRacket.GetScore();
   }
   if (tagName.Equals(Tag.leftRacket)){
       wayBall(collision, 1);
   }
   if (tagName.Equals(Tag.rightRacket)){
       wayBall(collision, -1);
   }
}
```

Listing 4.45: Functionalities when ball hits with rackets or walls

The *wayBall* function is shown in Listing 4.46, which handles the ball movement after a collision with one of the rackets. When the ball hits the left racket, the second parameter will be 1, and it will be -1 if it hits the right racket. The direction will be calculated according to the angle it makes.

```
private void wayBall(Collision2D collision, int x){
    float a = transform.position.y - collision.gameObject.transform.
   position.y;
```

```
    float b = collision.collider.bounds.size.y;
    float y = a / b;
    ballRb.velocity = new Vector2(x, y)*moveSpeed;
}
```

Listing 4.46: Ball bounces after hitting (*wayBall*)

4.4.3 Functional Programming

This Pong game was redeveloped in 2018 [74], inspired by the original Pong using Haskell and Gloss. It also used the *Exit* and *Random* libraries from the System library to destroy the game state and generate a random position for the initial state, respectively.

Pong in FP uses the *PlayIO* function from Gloss to handle the game loop. The loop execution is in Figure 4.6. In the initial state, the ball generates in a random position. The *handleKeys* function takes user input, processes, and responds to the game world. The *render* converts the game world into a *Picture*. The *update* is responsible for updating the game world using multiple functions, including two types of collisions: *wallCollision* and *paddleCollsion*, response functions for both: *wallBounce* and *paddleBounce*, and update the ball position using the *moveBall*.

The type constructor for the game world is presented in Listing 4.47, and the required types are mentioned in Listing 4.48. *PongGame* is a record type constructor with four fields named *ballLoc, ballVel, player1*, and *player2* with the appropriate type. The *Position* type is a pair of *Float*s, and the radius of the ball is 10.

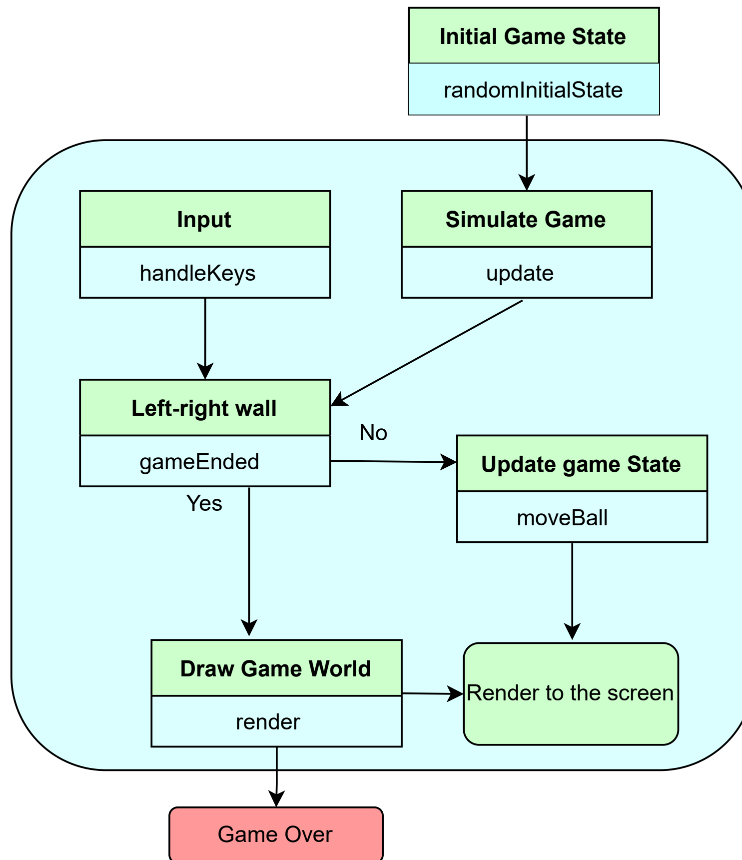

Figure 4.6: The main loop execution for Pong in FP

```
data PongGame = Game
      { ballLoc :: Position          -- ^ Pong ball (x, y) location.
      , ballVel :: (Float, Float)    -- ^ Pong ball (x, y) velocity.
      , player1 :: Float             -- ^ Left player paddle height.
                                     -- Zero is the middle of the screen
.
      , player2 :: Float             -- ^ Right player paddle height.
      }     deriving Show
```

Listing 4.47: Type Constructor PongGame in Haskell

```
type Radius = Float
type Position = (Float, Float)
```

Listing 4.48: Types in Haskell Pong game

The initial state uses the *stdGen* instance and *getStdGen* function to generate numbers to position the paddles and ball randomly and returns a *PongGame* state. The initial state is given in Listing 4.49.

```
randomInitialState :: StdGen -> PongGame
randomInitialState gen = Game
      { ballLoc = (a, b)
      , ballVel = (c', d')
      , player1 = 0
      , player2 = 0
      }
    where
        a:b:c:d:_ = randomRs (-50, 50) gen
        c' = c * mag
        d' = d * mag
        mag = 300 / sqrt (c^2 + d^2)
```

Listing 4.49: Initial State Generator in Haskell Pong game

The *width, height*, and *offset* are 1480, 780, and 100, respectively, with a *Fullscreen* window and *black* background. The *playIO* function from IO.Game module handles the game loop, and the execution of the loop is shown in Figure 4.6. Listing 4.50 shows the game loop, where a random value is generated from the *getStdGen* function and passed as an argument to the *randomInitialState* function that generates the initial state. The frame per second (*fps*) is 60.

```haskell
main :: IO ()
main = do
    gen <- getStdGen
    let initState = randomInitialState gen
    playIO window background fps initState render handleKeys update
```

Listing 4.50: Pong Game loop in Haskell

The *render* converts the game world to an *IO Picture* is presented in Listing 4.51. It takes the game world (*PongGame*) and returns a *Picture* with two white-bordered blue paddles, a white ball, and two top-down white-bordered walls. The ball can only bounce between two walls. Listing 4.52 shows the ball bounce functionality.

```haskell
render :: PongGame -> IO Picture
render game = return $
  pictures [ball, walls,
            mkPaddle white 700 $ player1 game,
            mkPaddle white (-700) $ player2 game]
```

Listing 4.51: Pong render function to convert Game world to an IO picture

```haskell
wallBounce :: PongGame -> PongGame
wallBounce game = game { ballVel = (vx, vy') }
  where
    -- The old velocities.
    (vx, vy) = ballVel game
    vy' = if wallCollision (ballLoc game) ballRadius then -vy else vy
```

Listing 4.52: Pong ball bounces when it touches top-down walls

Listing 4.53 shows the game over state. If a paddle misses the ball and the ball touches one of the walls (left and right), the game will be over.

```
gameEnded :: PongGame -> Bool
gameEnded game = farLeft || farRight
  where
    (x, _) = ballLoc game
    farLeft = x < -fromIntegral width / 2 + 2 * ballRadius
    farRight = x > fromIntegral width / 2 - 2 * ballRadius
```

Listing 4.53: Collision with left or right wall

The collision of the ball and paddle is represented in Listing 4.54. When a ball collides with the paddle, it changes its velocity to the opposite direction. The *paddleBounce* handles the functionality and uses *paddleCollision* to verify the collision of the ball and paddle.

```
paddleCollision :: Position -> PongGame -> Bool
paddleCollision (x, y) game =
  (x + ballRadius > 700 && abs (y - player1 game) < 40) ||
  (x - ballRadius < -700 && abs (y - player2 game) < 40)

paddleBounce :: PongGame -> PongGame
paddleBounce game = game { ballVel = (vx', vy) }
  where
    -- The old velocities.
    (vx, vy) = ballVel game
    vx' = if paddleCollision (ballLoc game) game then -vx else vx
```

Listing 4.54: Collision of Ball and Paddle

The user input is handled by the *handleKeys* function, which takes an *Event* (keyboard event), *PongGame* world, and returns *PongGame* world. Player1 presses ‘E’ and ‘D,’ and

Player2 presses the 'Up Key' and 'Down Key' to move the paddle up and down. Input events are shown in Listing 4.55.

```
handleKeys :: Event -> PongGame -> IO PongGame
handleKeys event game = case event of
  EventKey (Char 'q') _ _ _ -> exitSuccess
  EventKey (Char 'e') _ _ _ -> return $
    game { player2 = player2 game + 15 }
  EventKey (Char 'd') _ _ _ ->  return $
    game { player2 = player2 game - 15 }
  EventKey (SpecialKey KeyUp) _ _ _ ->  return $
    game { player1 = player1 game + 15 }
  EventKey (SpecialKey KeyDown) _ _ _ ->  return $
    game { player1 = player1 game - 15 }
  _ -> return game
```

Listing 4.55: IO Event Handler in Haskell Pong game

Lastly, the *update* is shown in Listing 4.56, where the function returns a string "Game ended!" if the game ends. This termination is performed using the *exitSuccess* from System.exist. Otherwise, it updates the game state with the updated ball position.

```
update :: Float -> PongGame -> IO PongGame
update seconds game =
  if gameEnded game' then do
    putStrLn "Game ended!"
    exitSuccess
  else return game'
  where
      game' = paddleBounce . wallBounce . moveBall seconds $ game
```

Listing 4.56: Update game state in Haskell Pong

4.4.4 Functional Reactive programming-Yampa

Vladimir Lopatin reproduces the classic pong game in Haskell using Yampa and SDL2 libraries [75]. The game structure is not the same as the prior discussed versions. It only covers a few functionalities of the game. It uses the SDL library to handle user input. In addition to FRP Yampa, and SDL2, the developer uses several Haskell libraries, such as Data, Linear, Control Monadonad, and Control Exception.

Figure 4.7 illustrates the custom game loop function named *animate* that uses *reactimate* to handle the actual game loop, calculates sense (*senseInput*) and actuate (*renderOutput*) values, and feeds to the *reactimate* function. The animate takes *Text, CInt, CInt*, and SF, where *Text, CInt*, and *CInt* are used to define the window. The initial input for *reactimate* is *return NoEvent*.

The *parseWinInput* takes user input and maps it with the (*mainGame, handleExit*) using the Yampa sequence (⋙) operator. The *mainGame* and *handleExit* are both SFs that produce the (*Game, Bool*) using the Yampa operator (&&&). *mainGame* SF is responsible for processing the user input, handling the collision, and updating the game states using several functions.

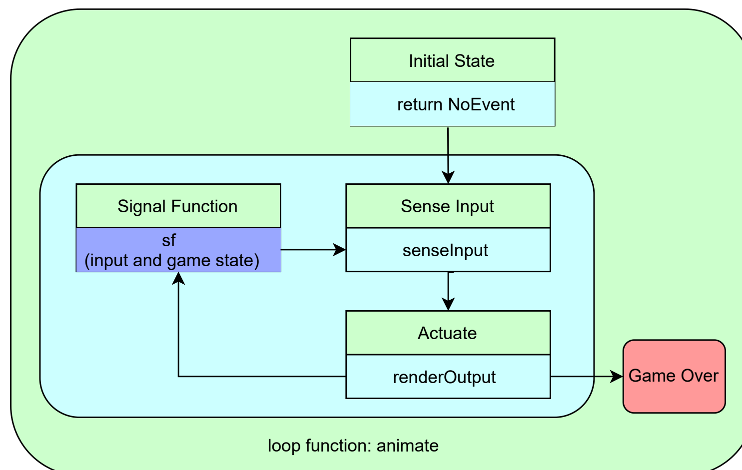

Figure 4.7: The main loop execution for Pong in FRP

User input has been defined in /src/input.hs. This game has two types of input events: mouse event and keyboard event. The type constructor for input is *AppInput*, which has six fields shown in Listing 4.57.

```
data AppInput =AppInput{
   inpMousePos   :: (Double, Double)        -- ^ Current mouse position
 , inpMouseLeft  :: Maybe (Double,Double) -- ^ Down button currently down
 , inpMouseRight :: Maybe (Double,Double) -- ^ Right button currently down
 , inpQuit        :: Bool                    -- ^ SDL's QuitEvent
 , inpKeyPressed  :: Maybe SDL.Scancode
 , inpKeyReleased :: Maybe SDL.Scancode
}
```

Listing 4.57: Data constructor for user input

The input file contains several supporting SFs, such as the current mouse position (*mousePos*), the left button is pressed (*lbp*), the left button is clicked and tagged with the mouse positions (*lbpPos*), and the left button is down (*lbDown*). Same functionalities for the right button (*rbp, rbpPos, rbDown*), a key function (*key*) that takes *Scancode* and *String* values and returns a SF of *AppInput*, where *Scancode* handles *ScancodeLeft, ScancodeRight*, and *ScancodeSpace* values and *String* values are pressed or anything else. Listing 4.58 shows the signal functions for left and right mouse button clicks. Listing 4.59 contains function composition based on the event and returns a SF, and Listing 4.60 represents the signal function for a quit event (*quitEvent*).

```
mousePos :: SF AppInput (Double,Double)
mousePos = arr inpMousePos

lbp :: SF AppInput (Event ())
lbp = lbpPos >>^ tagWith ()
```

```haskell
lbpPos :: SF AppInput (Event (Double,Double))
lbpPos = inpMouseLeft ^>> edgeJust

lbDown :: SF AppInput Bool
lbDown = arr (isJust . inpMouseLeft)

rbp :: SF AppInput (Event ())
rbp = rbpPos >>^ tagWith ()

rbpPos :: SF AppInput (Event (Double,Double))
rbpPos = inpMouseRight ^>> edgeJust

rbDown :: SF AppInput Bool
rbDown = arr (isJust . inpMouseRight)
```

Listing 4.58: SF for left and right mouse event

```haskell
key :: SDL.Scancode -> String -> SF AppInput (Event ())
key code mode
  | code == SDL.ScancodeLeft  ||
    code == SDL.ScancodeRight ||
    code == SDL.ScancodeSpace
    = (inpKeyMode ^>> edgeJust) >>^ filterE (code ==) >>^ tagWith ()
      where
       inpKeyMode
         = if | mode == "Pressed"
                -> inpKeyPressed
              | otherwise
                -> inpKeyReleased
```

Listing 4.59: SF for key pressed

```
quitEvent :: SF AppInput (Event ())
quitEvent = arr inpQuit >>> edge
```

Listing 4.60: SF for quit event

Input.hs contains several additional functions to filter and transform SDL events into relevant game events. The current mouse position is initially at (0,0), and the default value for the other fields is *Nothing*. The *Quit* event is *False*. Scanning app input and based on the *EventPayload* function, *nextAppInput* returns another *AppInput* value. There are five types of *EventPayLoad* events: *QuitEvent, MouseMotionEvent, KeyboardEvent, MouseButtonEvent*, and anything. Based on the *EventPayload* event and processing the event, *nextAppInput* mutate the app input field. Listing 4.61 contains the source code.

```
nextAppInput :: AppInput -> SDL.EventPayload -> AppInput
nextAppInput inp SDL.QuitEvent = inp { inpQuit = True }
nextAppInput inp (SDL.MouseMotionEvent ev) =
    inp { inpMousePos = (fromIntegral x, fromIntegral y) }
    where P (V2 x y) = SDL.mouseMotionEventPos ev
nextAppInput inp (SDL.KeyboardEvent ev)
    | scancode ev == SDL.ScancodeEscape = inp { inpQuit = True }
    | scancode ev == SDL.ScancodeLeft  || scancode ev == SDL.ScancodeRight
    ||
      scancode ev == SDL.ScancodeSpace =
        if | SDL.keyboardEventKeyMotion ev == SDL.Pressed
            -> inp { inpKeyPressed  = Just $ SDL.keysymScancode
                $ SDL.keyboardEventKeysym ev, inpKeyReleased = Nothing }
           | otherwise
           -> inp { inpKeyPressed  = Nothing
                  , inpKeyReleased = Just $ SDL.keysymScancode
                  $ SDL.keyboardEventKeysym ev }
```

```haskell
nextAppInput inp (SDL.MouseButtonEvent ev) =
                inp { inpMouseLeft  = lmb, inpMouseRight = rmb }
    where motion = SDL.mouseButtonEventMotion ev
          button = SDL.mouseButtonEventButton ev
          pos    = inpMousePos inp
          inpMod = case (motion,button) of
              (SDL.Released, SDL.ButtonLeft)  -> first (const Nothing)
              (SDL.Pressed, SDL.ButtonLeft)   -> first (const (Just pos))
              (SDL.Released, SDL.ButtonRight) -> second (const Nothing)
              (SDL.Pressed, SDL.ButtonRight)  -> second (const (Just pos))
              _                                    -> id
          (lmb,rmb) = inpMod $ (inpMouseLeft &&& inpMouseRight) inp
nextAppInput inp _ = inp
```

Listing 4.61: Input event processing

Shaders and rendering are in ../src/Shaders.hs and ../src/Rendering.hs. A shader [102] is a computer program that calculates the appropriate levels of light, darkness, and color while rendering a scene. The developer used fragment shader (*frag*) and vertex shader (*vert*) among different types of shaders. OpenGL uses *frag* and *frag* shaders to render 2D and 3D graphics. These files are written in C-like code Graphics Library Shading Language (GLSL). The initial game scene and gameplay scene are the two different *frag* and *vert* files of the game.

The../app/Main.hs contains the game functionalities. The type constructor for the game is named *Game*, which includes three properties: player position (*pPos*), ball position (*bPos*), and game state (*gstg*). There are four game states, including game menu (*GameMenu*), game introduction (*GameIntro*), game playing mode (*GamePlaying*), and exit state (*GameFinished*). Listing 4.63 and Listing 4.62 contain the type constructor and game state source code, respectively.

```haskell
data GameStage = GameIntro
               | GamePlaying
               | GameFinished
               | GameMenu        deriving Show
```

Listing 4.62: Data constructor for game

```haskell
data Game = Game{
    pPos :: Double    -- Player Position
    , bPos :: Pos        -- Ball   Position
    , gStg :: GameStage -- Game   Stage
    } deriving Show
```

Listing 4.63: Data constructor for Game state

The data type for physics (*PhysicsContext*) is defined in Listing 4.64, and the (*defPhysics*) is defined in Listing 4.65. The window constructor (*Bounds*) and initial window (*bounds*) are defined in Listing 4.66 and 4.67, respectively. The type of acceleration(*Acc*), velocity(*Vel*), Direction(Dir), and coefficient of restitution(COR) are *(Double, Double), (Double, Double), (Double, Double)*, and *Double*, respectively.

```haskell
data PhysicsContext = PhysC{
    gee :: Acc    -- A unit of acceleration due to gravity
    , cor :: Double -- coefficient of restitution
    } deriving Show
```

Listing 4.64: Data constructor for Physics for Pong in FRP

```
defPhysics = PhysC{
    gee = (0.0,-4.9)
    , cor = 1.01
    }
```

Listing 4.65: Physics Definition

```
data Bounds = Bounds{
    xMin  :: Double
    , xMax  :: Double
    , yMin  :: Double
    , yMax  :: Double
}
```

Listing 4.66: Data constructor for Window Bound for Pong in FRP

```
bounds :: Bounds
bounds = bounds' xMin xMax yMin yMax
    where
      xMin = -400 -- minX
      xMax =  400 -- maxX
      yMin =  0   -- minY
      yMax =  600 -- maxY
```

Listing 4.67: Window Bounds for Pong in FRP

The *playerPos* determines the player position, which takes a *Double* value of the player position and returns a SF. The *playerPos* uses another helper function named *movePlayer*, which takes the current position and velocity and produces a SF. If a left mouse button is clicked, the velocity will be -0.5; otherwise, 0.5. Listing 4.68 and Listing 4.69 contain the source code for both functions.

```
playerPos :: Double -> SF AppInput Double
playerPos pp0 = switch sf cont
    where
      sf = proc input -> do
        keyLeft  <- key SDL.ScancodeLeft  "Pressed" -< input
        keyRight <- key SDL.ScancodeRight "Pressed" -< input
        let res :: (  Double, Event (), Event ())
            res =  ( pp0, keyLeft, keyRight)
        returnA -< (pp0, mergeEvents [ keyLeft, keyRight ] `tag` res)
      cont (x, keyLeft, keyRight) =
        if | isEvent keyLeft -> movePlayer x (-0.5)
           | otherwise       -> movePlayer x   0.5
```

Listing 4.68: To determine player position for Pong in FRP

```
movePlayer :: Double -> Double -> SF AppInput Double
movePlayer pp0 v0 = switch sf cont
  where
    sf = proc input -> do
          p        <- (pp0 +) ^<< integral -< v0
          keyLeft <- key SDL.ScancodeLeft  "Released" -< input
          keyRight<- key SDL.ScancodeRight "Released" -< input
            returnA -< (p, mergeEvents [ keyLeft, keyRight ] `tag` p)
                        :: (Double, Event Double)
          cont = playerPos
```

Listing 4.69: To move player function (*movePlayer)*) for Pong in FRP

To determine the ball position, a method named *ballPos* takes the current position and velocity of the ball and returns a SF with a pair of position and velocity. It uses a helper function *ballPos'* that uses other functions, such as the collision function (*collidingBall'*), to

determine the collision. Listing 4.70 and Listing 4.71 are the source codes for *ballPos* and *ballPos'* functions. The value of *resX* and *resY* are 800 and 600 pixels.

```
ballPos :: Vel -> Pos -> SF () (Pos,Vel)
ballPos v0 p0 = ballPos' cor' rad p0 v0
    where
        cor'   = cor defPhysics
        rad    = 10 / (fromIntegral resY)
```

Listing 4.70: To determine ball's position

```
ballPos' :: COR -> Radius -> Pos -> Vel -> SF () (Pos,Vel)
ballPos' cor rad p0 v0 = bouncingBall' p0 v0
    where
      bouncingBall' p0 v0 = switch sf cont
        where
          sf   = proc () -> do
            ((p,v), col) <- collidingBall' rad p0 v0 -< ()
            returnA -< ((p, v), col `tag` fromEvent col )
                        :: ((Pos, Vel), Event (Dir,(Pos,Vel)))
          cont (dir,(p,v)) = bouncingBall' p (reflect dir ((-cor) *^ v))
      reflect l v = (2*(v `dot` l)/(l `dot` l)) *^ l ^-^ v
```

Listing 4.71: Helper function for *ballPos* for Pong in FRP

The collision function (*collidingBall'*) calculates the collision of the ball with the four walls or edges of the screen. When the ball hits the bottom-left wall, it will update the *XMin* and *YMin* by adding the radius value. When the ball hits the top-left edge of the screen, it will add the radius value with *XMin* and subtract the radius from *YMax*. When the bottom-right hit occurs, it will mutate the *XMax* by subtracting the radius from *XMax* and adding the radius value with *YMin*, and lastly, top-right hits will be handled by subtracting

the radius from *XMax* and *YMax*. This function uses *fallingBall*, which returns a SF with a pair of positions and ball velocity based on the *gee* force and physics definition. Listing 4.72 and Listing 4.73 contain the source code.

```
collidingBall' :: Radius -> Pos -> Vel ->
                SF () ((Pos, Vel), Event (Dir,(Pos,Vel)))
collidingBall' rad p0 v0  = proc () -> do
  pv@(p,v) <- fallingBall p0 v0 -< ()
  hitXMin  <- edgeTag ( 1, 0)   -< fst p <= xMin bounds + rad
  hitYMin  <- edgeTag ( 0, 1)   -< snd p <= yMin bounds + rad
  hitXMax  <- edgeTag (-1, 0)   -< fst p >= xMax bounds - rad
  hitYMax  <- edgeTag ( 0,-1)   -< snd p >= yMax bounds - rad
  let hitInfo = foldr1 (mergeBy mergeHits) [hitXMin,hitYMin,hitXMax,
   hitYMax]
  returnA -< (pv, hitInfo `attach` pv)
  where
    mergeHits = (^+^) -- simply add the two collision directions together.
```

Listing 4.72: Collision function for pong in FRP

```
fallingBall :: Pos -> Vel -> SF () (Pos, Vel)
fallingBall bp0 bv0 = proc () -> do
    v <- (bv0 ^+^) ^<< integral -< gee defPhysics
    p <- (bp0 ^+^) ^<< integral -< v
    returnA -< (p,v)
```

Listing 4.73: SF for ball when it is falling

The default game state is shown in Listing 4.74. It takes *GameInro* as a game state, and the player and ball positions are 0 and (0.0,0.4), respectively. The *GameIntro* is defined in Listing 4.75.

```haskell
defaultGame :: Game
defaultGame = Game pp0 bp0 GameIntro
  where
    pp0 = 0          :: Double
    bp0 = (0.0,0.4) :: (Double, Double)
```

Listing 4.74: Default game scene for Pong

```haskell
gameIntro :: SF (AppInput, Game) Game
gameIntro = switch sf cont
  where sf = proc (input, gameState) -> do
          introState <- returnA -< gameState
          playState  <- returnA -< gameState { gStg =  GamePlaying}
          skipE      <- key SDL.ScancodeSpace "Pressed" -< input
          waitE      <- after loadDelay () -< ()
          returnA   -< (introState,(skipE `lMerge` waitE)`tag` playState)
        cont game  = proc input -> do returnA  -< game
```

Listing 4.75: Definition of initial game state SF for Pong

Listing 4.76 contains the code for gameplay that uses two other functions: *mainGame* shown in Listing 4.77 and *gameSession* included in Listing 4.78.

```haskell
gamePlay :: SF AppInput Game
gamePlay = switch sf (const mainGame)
     where
       sf = proc input -> do
             gameState <- gameSession -< input
             reset     <- key SDL.ScancodeSpace "Pressed" -< input
             returnA   -< (gameState, reset)
```

Listing 4.76: SF forGame play state

```
mainGame :: SF AppInput Game
mainGame = loopPre defaultGame $ proc (input, gameState) -> do
            gs <- case gStg gameState of
                    GameIntro   -> gameIntro -< (input, gameState)
                    GamePlaying -> gamePlay  -< input
            returnA -< (gs, gs)
```

Listing 4.77: The *mainGame* SF definition

The *mainGame* uses *loopPre* function from Yampa, which is a loop with an initial value for the signal being feedback. The value for *loadDelay* is 5.0. The *gameSession* returns a SF.

```
gameSession :: SF AppInput Game
gameSession = proc input -> do
                ppos         <- playerPos   $ pPos defaultGame -< input
                (bpos, bvel) <- ballPos bv0 $ bPos defaultGame -< ()
                returnA      -< Game ppos bpos GamePlaying
                  where bv0 = (0.5,0.5) :: (Double, Double)
```

Listing 4.78: Game Session

The main loop is in Listing 4.79, where the loop uses Yampa's library function *reactimate* with the help of another custom function *animate* showed in Listing 4.80. Additionally, the definition of *handleExit* is in Listing 4.81.

```
main :: IO ()
main =  do
  window    <- openWindow "e1337" (resX, resY)
  animate "Pong" resX resY (parseWinInput >>> (mainGame &&& handleExit))
```

Listing 4.79: Pong game loop in Yampa

```haskell
animate :: Text                          -- ^ window title
        -> CInt                          -- ^ window width in pixels
        -> CInt                          -- ^ window height in pixels
        -> SF WinInput (Game, Bool) -- ^ signal function to animate
        -> IO ()
animate title winWidth winHeight sf = do
    window <- openWindow title (winWidth, winHeight)
    lastInteraction <- newMVar =<< SDL.time
    -- Input Logic
   ---------------------------------------------------------
    let senseInput _ = do
            currentTime <- SDL.time
            dt <- (currentTime -) <$> swapMVar lastInteraction currentTime
            mEvent <- SDL.pollEvent
            return (dt, Event . SDL.eventPayload <$> mEvent)
    -- Output Logic
   --------------------------------------------------------
        renderOutput _ ((gameState), shouldExit) = do
            draw window gameState
            return shouldExit
    -- Reactimate -----------------------------------------------------
    reactimate (return NoEvent) senseInput renderOutput sf
    closeWindow window
```

Listing 4.80: Game loop helper function (*animate*)

```haskell
handleExit :: SF AppInput Bool
handleExit = quitEvent >>^ isEvent
```

Listing 4.81: Definition of *handleExit*

Chapter 5

Animations, GUIs, and Embedded Systems

Real-time applications are designed to process and respond to data to events in real time without significant delay. Real-time application domains such as animations, GUIs, and embedded systems require a deep understanding of the problems to provide more accurate and instant responses. This section represents these applications in functional reactive programming.

5.1 Animation

Animation is a collection of images that express a sense of motion when displayed. In today's world, animation is one of the most pervasive and permeating visual communication. It can be found everywhere, from the countless TV channels entirely dedicated to cartoons to the opening titles of our favorite films to the responsive graphic interfaces on our smartphones. **Émile Cohl** [103] is called **The Father of the Animated Cartoon**, and **Fantasmagorie**, the first entirely finished animated film produced by him and known

as **traditional (hand-drawn) animation**.

5.1.1 Types of Animation

The animation used to be made by creating pictures of the characters for each action frame. Storyboards, which are sketches describing the sequence of key acts and the characters' facial expressions, are initially provided to the animator. The animator also uses a completed soundtrack that sets the timing for the composition. Characters in earlier animations were painted on *cels*, which are sheets of transparent celluloid that can be stacked on top of the background landscape but are frequently stationary in older films. A lead animator makes the key, or most crucial frames, and a second animator creates the in-between frames in *keyframing*, which is the method used to generate most hand animation. Regardless of the platform, the animator's task is to produce images that give the character emotion and life.

Computer animation developed in the early 1940s and 1950s when people began experimenting with computer graphics. The most fundamental computer animation technologies help with traditional animation by automatically producing some of the animation's frames. Similar to how layers of *cels* are used in hand animation. The tools have been designed to composite several layers of scenes together. Another significant and effective strategy is utilizing algorithms to create images from a geometric scene description. Through computer technologies, it is now possible to specify how images should evolve rather than having to design sequential images. Among several animations, animation includes the following five categories.

1. **Traditional Animation:** A traditional animator must sketch each frame of an animated scene by hand in cell animation, which is done on a light table so the artists can see the preceding drawing through the top layer of paper. Traditional animation can be

done today on computers with specialized tablets and is used by the most well-known companies, including Disney.

2. **2D Animation:** A 2D vector-based animation, 2D animation is becoming increasingly popular due to technology accessibility. Instead of constantly redrawing the characters, it allows the artist to build *rigs* for them and move specific body parts at a time. Because 2D animation technology is flexible and beginners do not have to rely heavily on drawing skills, it is popular with them.

3. **3D Animation:** The most widely utilized type of animation today is 3D or computer animation. Unlike 2D and traditional animation, 3D animation has a different production method. Both, nonetheless, adhere to the same movement and compositional basis. It has more to do with the animated character and less with the drawing. The National Science Foundation highlights how strongly 3D animators must use physics to produce realistic animations. The animator makes *keyframes* or specified movements, and the computer fills in the remainder.

4. **Motion Graphics:** Motion graphics focuses on the ability to move graphic elements, shapes, and text instead of characters or storylines. This procedure is frequently applied to animated logos, instructional films, and television promotions. Motion graphics don't require the same skill set as other types of animation because it's unnecessary to emulate facial expressions or body language.

5. **Stop Motion:** Stop-motion animation utilizes a succession of slightly altered still images to create the illusion of movement, much like traditional animation. When creating stop motion, the artists first photograph a scene or an object, then slightly move the subject before snapping another picture. The artist repeats this process until the scenario is finished.

Though it is authentic and naturally appealing, stop motion animation requires a lot of work, can take a long time to complete, and requires special skills. Traditional animation can be a fantastic choice for artists who wish to exercise their full artistic license. But, it also demands extensive skills and can be supplanted by 3D animation. Each kind of animation has benefits and drawbacks, depending on the preferences of the animator. It is impossible to say that one type of animation fits all animation projects the best.

5.1.2 How to create animation

In addition to animation tools' new capabilities that are auto-generating frames and the use of algorithms, it increases the range of applications. Real-time applications like games and interactive media can take advantage of computer animation. The production of computer actors that may be seamlessly merged with real-world film is made possible by realistic rendering and animation techniques. A complicated computer animation like *Disney* and *Pixar*'s Toy Story uses various techniques. Two basic kinds of these strategies can be distinguished: two-dimensional (2D) and three-dimensional (3D). Although there are considerable overlaps between the two classifications, 3D techniques typically create virtual worlds with moving characters and objects, while 2D approaches typically concentrate on image modification.

An animator must model, animate, and render the scene in order to produce an animation. When modeling, the components of a scene are described and placed appropriately. Rendering turns the description of the objects and their motion into visuals, while animation explains how the object should move in the world. Except for a few necessary adjustments, modeling and rendering are mostly independent of their involvement in the animation process.

Animations usually display 24 or 30 frames per second. Motion blurring plays a vital role in making it realistic. However, providing motion to animation is surprisingly tricky and

complex because humans are skillful at observing the motion and quickly detect unnatural and implausible motion. Therefore, the animator must be able to specify subtle details of the motion to convey a character's personality or the mood of an animation in a compelling fashion. Numerous techniques have been developed; however, everything has some trade-offs between automation and control. According to [104], if one can merge **keyframing, procedural methods, and motion capture**, these three techniques to generate motion can take advantage of each technique which solves the trade-off between automation and control.

5.1.3 FRP and Animations

Animation can be implemented in different paradigms using several languages. Developers create animations based on tasks, scalability, and performance in several paradigms, including imperative programming, FP, and FRP.

Animation tools allow developers to create and manipulate visual content. Animation tools, including Maya [105], Animate [106], Blender [107], and Unity [108] are widely used. Some common features are a timeline-based interface [109], which provides control to the developers over the timing and sequencing of the animations, key-frame animation, 2D and 3D animation, painting and drawing, etc. In addition to these core features, many animation tools also provide specialized features for specific types of animation, such as *character rigging* [110] that allows creators to create and manipulate virtual skeletons that control the movement of the characters or objects in the animation through the network of movements, particle effects, and physics simulation that allows importing motion captured data to make the content realistic. Moreover, tools are flexible to integrate with other software applications to enable more complex workflows.

Animation tools are increasingly sophisticated and powerful in recent years, enabling animators to create highly complex and realistic animations. However, to be an expert on a

tool to utilize its features efficiently requires significant time and effort.

JavaFX [111, 112] library offers smooth graphics that render quickly through *Prism*, a high-performance hardware-accelerated graphical pipeline. It is used with a supported graphics card or graphics processing unit (GPU). *Prism* utilizes the software render path to process the graphics when the hardware systems do not have sufficient graphics support. *Prism* can also render 2-D and 3-D graphics. Python is often used for scripting in animation software, and a few Python libraries, for example, *pygame* and *OpenCV*, can be used to create complex animations.

FRP was the first DSL for interactive animation in FRP. The extensions of *Fran* are [4, 13, 14], but not widely adopted. ReactiveX [9], a collection of libraries, provides a set of operators for manipulating and combining streams of synchronous and asynchronous data. The libraries of ReactiveX are implemented in multiple languages, such as Java, JavaScript, and Python. One of the animation languages developed using FRP is Flapjax [23]. which is for web programming built on top of JavaScript and uses FRP to enable highly interactive and reactive web applications. Flapjax provides a wide range of features for developing animated and interactive web content. It supports reactive data streams, event handling, and animation timelines. Additionally, it provides a range of built-in animation and visualization tools, such as canvas drawing, SVG graphics, and charting libraries. Flapjax has a library for animation named Flan [113].

Though Flapjax is a powerful and flexible tool for creating an animated and interactive web application using FRP, it may only be suitable for some types of animation projects.

5.2 Graphical User Interface

A Graphical User Interface (GUI) is a type of user interface (UI) that allows users to interact with software applications or Human-computer interaction (HCI) applications us-

ing visual elements. GUI was first introduced as a solution to challenging Command-line Interfaces (CLIs) and gained popularity in every aspect of modern application domains.

5.2.1 FRP and GUIs

The traditional approach to a graphical user interface manages the state of the system using mutable variables. However, it can make the system complex, hard to interpret, and error-prone. The states of the system are the stream of events, user input, timers, mutating data sources, etc. The events can easily be handled using FRP's higher-order functions, such as map, filter, and fold in the appropriate output.

The fundamental benefit of FRP in GUI is FRP allows developers to express the behavior of a system in a declarative way. Declarative programming can be contrasted with imperative programming, which involves writing detailed instructions on how to complete a specific task. In the imperative paradigm, the developers usually follow a procedural approach that includes step-by-step instructions and will be executed in sequences. Declarative programming keeps the code base concise, readable, and more scalable to handle large and complex programs though it requires experienced programmers. Rather than defining a set of mutable states in imperative programming, developers can design a set of composable functions that transform and combine streams of events and keep a clean and understandable code base.

Some of the widely used libraries and frameworks for GUI in FRP are RxJS [34], Bacon.js [37], and Cycle.js [38]. Libraries provide a set of abstractions and higher-order functions to make the system easy to manage streams and function compositions, provides modularity, easy testing and debugging features, and better user experiences. React [114, 115] is popular for GUI applications. While React does not strictly require RxJS, there are a few reasons to use RxJs in React. React uses a unidirectional data flow model, which follows a top-down tree approach via *props*. Props are used to configure the child components. It

works well for small-scale applications. However, it is complex and challenging to manage a large-scale project. RxJS provides a simple way to manage the states in a React application. RxJS provides a declarative way to express complex data flows to manage a large project. Moreover, RxJS allows a flexible way to handle asynchronous data. Providing higher-order functions, it offers a flexible environment making the system simple and understandable. Bacon.js and Cycle.js can be used with React to create reactive user interfaces, and the framework also supports other front-end libraries and frameworks and use RxJS in the background to provide reactive features.

Elm is a functional programming language especially well-suited for web applications, originally designed as FRP [35, 36]. Elm provides higher-order features and abstractions to build interactive, responsive, and scalable interfaces. Elm offers a declarative approach with numerous built-in features for creating elements and updating properties. Developers can achieve everything without the signal function. Therefore, the Author of Elm declared it a functional language [116] in 2016. Elm is popular for a robust and safe environment for building web applications, which detects errors during compilation time rather than run time, provides good performance and scalability characteristics, is easy to learn and code [117], and has a strong and active community.

5.3 Embedded Systems

Embedded systems are designed to perform specific tasks to operate real-time responses, typically consisting of microcontrollers or microprocessors responsible for executing functions with various input-output (IO). Developing embedded systems requires understanding the hardware and software designs and problem specifications to provide effective applications.

5.3.1 FRP and Embedded Systems

Emfrp [39] is a statically typed pure FRP language for reactive programs running on small-scale embedded systems. Programs written in Emfrp intends to run on environments such as microcontrollers with low-power CPUs and a few kilobytes of memory or bare metal system without an OS. Emfrp is designed to solve the space-time leak problem. Program written in Emfrp compiled into a pair of C source files that are platform independent, one containing a single loop implementing continuous reactive process and the other containing function skeletons for input and output. Filling the second file from the external environment, the user can complete an executable Emfrp module. It is an open-source GitHub project [118].

Emfrp has several restrictions to ensure continuous reactive behavior safely, such as (1) the authors do not follow the lambda-calculus tradition to make the language suitable for resource-constrained environments. It lacks suitable capabilities to modularize adaptive behaviors, (2) termination of each node process after updating and determining the memory required during runtime. Because of this, it imposes an unnatural representation of data structures like lists or trees. These restrictions urge us to write poorly maintainable redundant code or stop us from writing certain types of programs. The challenge in Emfrp is that it is difficult to naturally express data structures where the number of elements changes at runtime.

To solve (1), extended Emfrp [40], a mechanism with an implicit activation layer that improves the modularity using Context-oriented Programming(COP) by separating the descriptions of context-dependent behaviors, and hence eliminating various cross-cutting code fragments that often appear in plain Emfrp programs. This extension does not affect the semantics and execution model of the Emfrp [39] language. So, the programs written using the extension are still purely functional and glitch-free. However, fake cyclic node dependencies may arise and reject correct modules. It is an open issue and left for future work.

The author claims that no systems/languages employ context-oriented programming and functional reactive programming to enjoy modularity, adaptability, and purely declarative programming.

Emfrp$^{\text{BCT}}$ [42], an extended Emfrp with size annotated recursive data types [119], to overcome problem (2). However, it still does have the aforementioned static properties. Since the data size is checked statically as a part of the type checking, unlimited memory-used programs will not be compiled and executed. The termination of a reactive process is also guaranteed. The programs such as list and heap trees must be written in an unnatural form using tuples in Emfrp. On the other hand, programs can be written in a concise and maintainable manner in Emfrp$^{\text{BCT}}$.

[42] has been divided into several tasks. The primary part is to design and develop an FRP language containing a recursive data types property containing information on the amount of memory to be used, along with an algorithm for estimating the memory usage of a program written in the language. It formally defines the syntax, operational semantics, and type system of Emfrp$^{\text{BCT}}$, presents an algorithm for estimating the amount of memory required to evaluate expressions, and proves its correctness and soundness. It also implements a compiler from Emfrp$^{\text{BCT}}$ to C. It measures the needed time for type checking and estimates the amount of memory and the time and space overhead of using recursive data types.

Authors want to improve it by introducing type polymorphism and size polymorphism, mutually recursive definitions of functions and types, flexible measure functions other than size for defining recursive functions, and a code optimizer that incorporates existing optimization techniques and is left for future work.

Chapter 6

Discussion and Conclusion

This section represents the similarity and comparison in game development among Unity (Imperative programming), FP, and FRP, and the success of FRP in other application domains.

6.1 Game Development

We explored two games (Asteroid and Pong) in three different programming paradigms (Imperative, FP and FRP) and listed a number of similarities and differences.

6.1.1 Similarities

Based on the programming language and game design, similarities are

- The game design is similar. The primary element consists of Entity/object, component/logic, and system/game world.
- All use the same evaluation model: polling and callback. Unity also has a registered callback.

- Every paradigm helps to divide the monolithic class and separate data from functions.
- Provides a clean design using the decoupling, encapsulation, modularization, and reusability methods.
- Offers a friendly method for parallel processing and multi-threading.
- Difficult to apply correctly, easy to misuse. Appropriate components require more thinking about the design.
- All provide event-driven functionalities.
- Priority-based ordering.

6.1.2 Comparisons

The main difference is performance. Unity offers better performance among the three. Unity is a game engine, and most of the workflow is hidden. In contrast, FP with Haskell and FRP with Yampa is direct. Most of the game task is defined by the developer though there are some advantages of using Gloss, HGL, and SDL2 libraries that handle game loop, Graphics and provide some flexibility.

We conducted two comparisons: The internal differences among the three paradigms and the development process flexibility among the paradigms in terms of game features and development flexibilities. The differences among paradigms include performance, workflow, live coding, unit testing, feature extension, debugging a single component or logic, understanding code environment, environment setup, scalability and learning curve are presented in Table 6.1.

The collision detection in Unity is handled by the engine using *Collider2D* and *RigidBody* properties. The reaction of the object after the collision is handled by the developers with

Table 6.1: Comparison among Unity, Functional Programming with Haskell, and Functional Reactive Programming with Yampa

Criteria	Unity	Functional Programming	Functional Reactive Programming
Performance	Excellent	Less efficient	Less efficient
Collision detection	Implicit	Explicit	Explicit
Workflow	Implicit	Explicit	Explicit
Live coding	Limited	Challenging	Challenging
Unit Testing	Easy	Comparatively Difficult	Difficult
Extending features	Easy to add	Comparatively difficult	difficult
Debugging single component or logic	Difficult	Easy	Comparatively Difficult
Understanding code environment	Complex and time consuming for beginners	Easy to understand who understand Haskell	Comparatively difficult
Setting up environment	Very easy. It is all in a package.	Requires many steps and libraries to be installed.	Requires many steps and libraries to be installed.
Scalability	Suitable for a large project	Difficult to manage	Difficult to manage
Learning curve	Less steep	Steep	Steeper

the help of built-in and custom functions. In Haskell and Yampa, the developers handle collision detection and the reaction after a collision.

Unity is well-documented, and developers find the documentation helpful. The workflow in Unity is hidden. Documentation is enough to develop a game without knowing how it works. However, developers can tell how the program works except for the Gloss package and some graphics libraries' workflow in FP and FRP.

With some restrictions, live coding is available in Unity. Modifications made to the game scene during play mode are visible; however, they are not stored after leaving play mode. Developers can make changes to the project hierarchy, which will be stored after exiting the play mode. The changes must be reloaded into the game scene. Visual Scripting additionally offers live editing. In play mode, developers can change values and add and remove nodes and edges. Everything may be done in both a conventional edit and live editing mode. However, changes made to embeds are undone once the play mode is exited. Live coding in FP and FRP is challenging but not impossible [120]. Haskell applies mostly to soft real-time operations. Since Haskell is a lazy language, it is uncertain that all operations will consume a predictable amount of time. Therefore, the user may find some sudden latency in a hard

real-time scenario. Additionally, it is easy to create a small number of game objects and run the game scene to see the results. However, a large game project has hundred to thousands of game objects. It would be a nightmare for developers to create all the game objects by giving them position, scaling, and lasting the composition feature. Developers can utilize the live coding feature efficiently to solve the problem in Unity. Some third-party tools are available on Unity Asset Store, such as Livity.

Feature extension is a complex process in FP and FRP. A feature could be a single function or a combination of multiple functions related to some previously defined functions and some new functions. Function composition is a critical task in the FP. Additionally, in the FRP, it is comparatively more complicated than the FP because developers have to find a way to compose two functions and utilize the behavior of the signal function. But this task is straightforward in Unity. Unity is a flexible tool that allows extending features smartly.

The debugging feature is available in Unity. However, debugging a single component is problematic because it can be attached to many game objects or other components. Therefore, it is hard to debug without knowing the proper behaviors of a component. On the other hand, debugging is comparatively easy in FP and FRP. The challenging task would be to learn the correct input for the function.

It is laborious for a beginner Unity developer to understand a game source code developed by others without having enough documentation of the code base. Many works are performed in the Unity Editor. Drag and drop features and placing values to Unity's built-in components and methods make the understanding process obscure while it makes the development process effortless. Unlike Unity, with a fair amount of effort, understanding the game control flow is relatively straightforward for Haskell developers or who understand Haskell. Some research is mandatory to understand the libraries.

The installation process is effortless in Unity. Unity is a complete package of everything. Installing other necessary libraries when required is easy to manage from the UnityHub and

Table 6.2: The game features comparison among Unity, Functional Programming with Haskell, and Functional Reactive Programming with Yampa

Game Features	Unity	Haskell	Yampa
Object Creation	Implicit with Sprite	Explicit	Expliit
Image Insertion	Implicit with Sprite	Implicit with library	Implicit with library
Destroy game object	Implicit	Explicit	Explicit
Collision detection	Implicit	Explicit	Explicit
Reaction after collision	Partially Implicit	Explicit	Explicit
Object detection	Implicit	Explicit	Explicit
Re-positioning	Partially Implicit	Explicit	Explicit
Re-scaling	Partially Implicit	Explicit	Explicit
Game loop	Implicit	Implicit with library	Partially implicit

Unity editor. Changing the Unity version is a simple process from UnityHub. However, in Haskell and Yampa, the installation process consists of several steps, including some research required on device and version dependencies.

The learning curve for Unity is less steep because developers are required to be familiar with the Unity editor and C# language to develop a small-scale game. Haskell developers need to know Haskell and its numerous libraries. Yampa developers have to be an expert in Yampa to utilize it correctly, especially the signal function along with other libraries.

The development process flexibility among the paradigms in terms of game features and development flexibilities include game object creation, importing an image, destroying a game object, collision detection, reaction after a collision, detecting game objects, re-positioning, re-scaling, and game loop. The comparison table of the features among the three paradigms is in Table 6.2.

Asteroid and Pong are 2D games. Unity provides a built-in mechanism to create an object, insert pictures with the 2D *Sprite* package, and position and scale an object with the Unity Transform component. In Haskell and Yampa, developers define the data constructors for the game objects. Image insertion is handled by the gloss or sdl2 libraries. Developers must be aware of the dependencies and versions of the library.

The Unity engine handles the destruction of a game object after a collision or at some particular time. In Haskell and Yampa, developers will handle the task using different custom functions, data constructors in Haskell, and additional signal functions for Yampa.

To find the correct game object, Unity has an efficient built-in property named *FindWith-Tag*. This function is associated with an object and takes a String parameter. Therefore, it is implicit in Unity. Finding a game object in Haskell and Yampa is explicit. Developers compare two objects with their type constructors and values.

Rescaling a game object means creating an object with a different size. In Unity, it is easy to accomplish rescaling and repositioning using the *Transform* component, which gives the flexibility to get the position and size of the object. Developers can change the position and size with *Transform*. In Haskell and Yampa, the developers handle rescaling and repositioning. Sometimes, developers create several game objects with different sizes and positions in a small-scale game.

Unity handles the game loop. However, the asteroid and pong game in Haskell uses the gloss library, Yampa uses the HGL library for the asteroid, and pong uses a custom game loop with several functions from many libraries, including Yampa's *reactimate* function.

Overall, game architecture in the imperative paradigm (ECS in Unity) is simple and elegant. It is a framework where everything is hidden from the developer. Following each step for a large project in FP or FRP is complex and challenging. A few game engines [93, 94, 95, 96] developed in FP and FRP paradigms, that are less popular and have many limitations based on functionality and performance. Many libraries are not being maintained nowadays. A simple toy game is easier to develop for a beginner; however, it is sometimes difficult for an experienced programmer to develop a game in FP and FRP for a large project. Debugging and testing are easy in FP for a small-scale project, a nightmare for a real-life AAA[1] game.

[1]AAA, https://en.wikipedia.org/wiki/AAA_(video_game_industry)

6.2 Animations, GUIs, and Embedded Systems

Several initiatives have been taken to integrate FRP in these application areas (Animations, GUIs, and Embedded Systems). Very few have succeeded, and most applications, including platforms, languages, and libraries, are under-performed, not documented, and need an active developing community. However, FRP is gaining popularity and success in GUIs, especially web development.

FRP-based animation offers several benefits over traditional imperative programming approaches, including better modularity, and composability, more readable code. It is difficult and time-consuming to learn and utilize FRP effectively since the learning curve is very steep, especially for developers who are beginners and more comfortable with imperative programming. Moreover, performance is a crucial problem in FRP. FRP in embedded systems is still in its initial state. Researchers are exploring, and it is difficult to meet hard real-time applications with Haskell-based FRP languages.

Developers prefer an imperative paradigm for animation and embedded systems due to the flexibility of having platforms and performance.

6.3 Conclusion

Functional reactive programming is well-suited for real-time event-driven applications. In the survey, we studied four real-time application domains, including game development, animations, graphical user interfaces, and embedded systems. We examined two demo games (asteroid and pong) in three paradigms (imperative, functional, and functional reactive programming), conducted an example-based comparison in game development, and described the success of FRP in the rest of the real-time domains.

Developers are comfortable using platform-based applications because of the flexibility of providing many features and reducing the workload. Due to FRP's steep learning curve, most

application domains do not have efficient platforms and an active community. Therefore, researchers can focus on developing modular platforms instead of programming from scratch and providing documentation to help beginners learn and use it in applications. Therefore, a trade-off must be made between complexity, performance, versus modularity.

REFERENCES

[1] A. Smith, "Of boxes and threads: Game development in haskell." `https://aas.sh/blog/of-boxes-and-threads/`.

[2] C. Elliott and P. Hudak, "Functional reactive animation," *SIGPLAN Not.*, vol. 32, p. 263–273, aug 1997.

[3] A. Courtney, H. Nilsson, and J. Peterson, "The yampa arcade," *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop*, 08 2003.

[4] C. Elliott, "Push-pull functional reactive programming," in *Haskell Symposium*, 2009.

[5] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," *Defense*, vol. 2638, 08 2002.

[6] J. Hughes, "Generalising monads to arrows," *Science of Computer Programming*, vol. 37, no. 1, pp. 67–111, 2000.

[7] C. Metz, "Why whatsapp only needs 50 engineers for its 900m users — wired." `https://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/`.

[8] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, "The reactive manifesto." `https://www.reactivemanifesto.org/`.

[9] "Reactivex." https://github.com/ReactiveX.

[10] I. Buchelnikov, "Igorbuchelnikov/observablecomputations." https://github.com/IgorBuchelnikov/ObservableComputations.

[11] R. Harris, "Svelte • cybernetically enhanced web apps." https://svelte.dev/.

[12] "Reactivecocoa." https://github.com/ReactiveCocoa.

[13] C. Elliott, "Modeling interactive 3d and multimedia animation with an embedded language," in *Conference on Domain-Specific Languages (DSL 97)*, (Santa Barbara, CA), USENIX Association, Oct. 1997.

[14] C. Elliott, "Functional implementations of continuous modeled animation," in *Proceedings of the 10th International Symposium on Principles of Declarative Programming*, PLILP '98/ALP '98, (Berlin, Heidelberg), p. 284–299, Springer-Verlag, 1998.

[15] "ivanperez-keera/yampa: Functional reactive programming domain-specific language for efficient hybrid systems." https://github.com/ivanperez-keera/Yampa/.

[16] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, (New York, NY, USA), p. 51–64, Association for Computing Machinery, 2002.

[17] H. Liu and P. Hudak, "Plugging a space leak with an arrow," *Electr. Notes Theor. Comput. Sci.*, vol. 193, pp. 29–45, 11 2007.

[18] D. Winograd-Cort, H. Liu, and P. Hudak, "Virtualizing real-world objects in frp," in *Practical Aspects of Declarative Languages* (C. Russo and N.-F. Zhou, eds.), (Berlin, Heidelberg), pp. 227–241, Springer Berlin Heidelberg, 2012.

[19] D. Winograd-Cort and P. Hudak, "Wormholes: Introducing effects to frp," in *Proceedings of the 2012 Haskell Symposium*, Haskell '12, (New York, NY, USA), p. 91–104, Association for Computing Machinery, 2012.

[20] I. Perez, M. Bärenz, and H. Nilsson, "Functional reactive programming, refactored," *SIGPLAN Not.*, vol. 51, p. 33–44, sep 2016.

[21] G. H. Cooper and S. Krishnamurthi, "Embedding dynamic dataflow in a call-by-value language," in *Programming Languages and Systems* (P. Sestoft, ed.), (Berlin, Heidelberg), pp. 294–308, Springer Berlin Heidelberg, 2006.

[22] G. H. Cooper and S. Krishnamurthi, "racket/frtime." `https://github.com/racket/frtime`.

[23] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: A programming language for ajax applications," *SIGPLAN Not.*, vol. 44, p. 1–20, oct 2009.

[24] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "brownplt/flapjax: functional reactive programming for javascript." `https://github.com/brownplt/flapjax`.

[25] M. H. Cheong, "Functional programming and 3d games," Master's thesis, University of New South Wales, Sydney, Australia, November 2005.

[26] "ivanperez-keera/haskanoid: A free and open source breakout clone in haskell using sdl and frp, with wiimote and kinect support.." `https://github.com/ivanperez-keera/haskanoid`.

[27] D. Eichmann, "bogre-banana." `https://hackage.haskell.org/package/bogre-banana`.

[28] A. Furtado, "Fungen: A lightweight, cross-platform, opengl-based game engine.." https://hackage.haskell.org/package/FunGEn.

[29] I. P. Konstantin Saveljev, "Graphics.gloss.interface.frp.yampa." https://hackage.haskell.org/package/yampa-gloss-0.2/docs/Graphics-Gloss-Interface-FRP-Yampa.html.

[30] J. T. Kay, "gloss-banana: An interface for gloss in terms of a reactive-banana behavior.." https://hackage.haskell.org/package/gloss-banana.

[31] Z. Corr, "helm: A functionally reactive game engine.." https://hackage.haskell.org/package/helm.

[32] H. Nilsson, "Yfrob: Yampa-based library for programming robots." https://hackage.haskell.org/package/YFrob.

[33] B. Yorgey, "swarm: 2d resource gathering game with programmable robots." https://hackage.haskell.org/package/swarm.

[34] "Reactivex/rxjs: A reactive programming library for javascript." https://github.com/ReactiveX/rxjs.

[35] E. Czaplicki, "Elm : Concurrent frp for functional guis," 2012.

[36] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for GUIs," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (New York, NY, USA), pp. 411–422, ACM Press, June 2013.

[37] "baconjs/bacon.js: Functional reactive programming library for typescript and javascript." https://github.com/baconjs/bacon.js/.

[38] "cyclejs/cyclejs: A functional and reactive javascript framework for predictable code." `https://github.com/cyclejs/cyclejs`.

[39] K. Sawada and T. Watanabe, "Emfrp: A functional reactive programming language for small-scale embedded systems," in *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, (New York, NY, USA), p. 36–44, Association for Computing Machinery, 2016.

[40] T. Watanabe, "A simple context-oriented programming extension to an frp language for small-scale embedded systems," in *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition*, COP '18, (New York, NY, USA), p. 23–30, Association for Computing Machinery, 2018.

[41] Y. Sakurai and T. Watanabe, "Towards a statically scheduled parallel execution of an frp language for embedded systems," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2019, (New York, NY, USA), p. 11–20, Association for Computing Machinery, 2019.

[42] A. Yokoyama, S. Moriguchi, and T. Watanabe, "A functional reactive programming language for small-scale embedded systems with recursive data types," *Journal of Information Processing*, vol. 29, pp. 685–706, 10 2021.

[43] J. Peterson and G. Hager, "Monadic robotics," *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, vol. 35, 11 1999.

[44] J. Peterson, G. Hager, and P. Hudak, "A language for declarative robotic programming," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 2, pp. 1144–1151 vol.2, 1999.

[45] J. Peterson, P. Hudak, and C. Elliott, "Lambda in motion: Controlling robots with haskell," 07 1999.

[46] A. Reid, J. Peterson, G. Hager, and P. Hudak, "Prototyping real-time vision systems: an experiment in dsl design," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pp. 484–493, 1999.

[47] J. Peterson, P. Hudak, A. Reid, and G. Hager, "Fvision: A declarative language for visual tracking," in *Practical Aspects of Declarative Languages* (I. V. Ramakrishnan, ed.), (Berlin, Heidelberg), pp. 304–321, Springer Berlin Heidelberg, 2001.

[48] A. Almeida Matos, G. Boudol, and I. Castellani, "Typing noninterference for reactive programs," *The Journal of Logic and Algebraic Programming*, vol. 72, no. 2, pp. 124–156, 2007. Programming Language Interference and Dependence.

[49] D. N. Xu and S.-C. Khoo, "Compiling real time functional reactive programming," in *Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ASIA-PEPM '02, (New York, NY, USA), p. 83–93, Association for Computing Machinery, 2002.

[50] L. Mandel, C. Pasteur, and M. Pouzet, "Time refinement in a functional synchronous language," in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, (New York, NY, USA), p. 169–180, Association for Computing Machinery, 2013.

[51] G. Berry and M. Serrano, "Hiphop.js: (a)synchronous reactive web programming," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, (New York, NY, USA), p. 533–545, Association for Computing Machinery, 2020.

[52] Z. Wan, W. Taha, and P. Hudak, "Real-time frp," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, vol. 36, 08 2001.

[53] I. Perez, "Back to the future: Time travel in frp," *SIGPLAN Not.*, vol. 52, p. 105–116, sep 2017.

[54] M. Bärenz and I. Perez, "Rhine: Frp with type-level clocks," *SIGPLAN Not.*, vol. 53, p. 145–157, sep 2018.

[55] R. Kaiabachev, W. Taha, and A. Zhu, "E-frp with priorities," in *Proceedings of the 7th ACM; IEEE International Conference on Embedded Software*, EMSOFT '07, (New York, NY, USA), p. 221–230, Association for Computing Machinery, 2007.

[56] C. Belwal, A. M. Cheng, W. Taha, and A. Zhu, "Timing analysis of the priority based frp system," in *Proceedings Work-In-Progress Session of the 14th Real-Time and Embedded Technology and Applications Symposium : 22-24 April, 2008: St. Louis, USA*, no. TR-UNL-CSE-2008-0003 in Technical Report, pp. 89–92, University of Nebraska–Lincoln, Computer Science and Engineering, 2008. This work is supported in part by the U.S. National Science Foundation under Award Nos. 0720856 and 0720857.

[57] J. Ras and A. M. K. Cheng, "Response time analysis for the abort-and-restart event handlers of the priority-based functional reactive programming (p-frp) paradigm," in *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 305–314, 2009.

[58] H. C. Wong and A. Burns, "Schedulability analysis for the abort-and-restart (ar) model," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, (New York, NY, USA), p. 119–128, Association for Computing Machinery, 2014.

[59] H. Wong and A. Burns, "Priority-based functional reactive programming (p-frp) using deferred abort," in *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 227–236, 2015.

[60] Z. Kazemi and A. M. Cheng, "A scratchpad memory-based execution platform for functional reactive systems and its static timing analysis," in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 176–181, 2016.

[61] J. D. Lin and A. M. K. Cheng, "P-frp task scheduling with preemption threshold," *SIGBED Rev.*, vol. 15, p. 15–19, aug 2018.

[62] I. Perez, "Fault tolerant functional reactive programming (functional pearl)," *Proc. ACM Program. Lang.*, vol. 2, jul 2018.

[63] I. Perez, A. Goodloe, and W. Edmonson, "Fault-tolerant swarms," in *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pp. 47–54, 2019.

[64] I. PEREZ and A. E. GOODLOE, "Fault-tolerant functional reactive programming (extended version)," *Journal of Functional Programming*, vol. 30, p. e12, 2020.

[65] R. Mogk, L. Baumgärtner, G. Salvaneschi, B. Freisleben, and M. Mezini, "Fault-tolerant Distributed Reactive Programming," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)* (T. Millstein, ed.), vol. 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 1:1–1:26, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[66] R. Mogk, J. Drechsler, G. Salvaneschi, and M. Mezini, "A fault-tolerant programming model for distributed interactive applications," *Proc. ACM Program. Lang.*, vol. 3, oct 2019.

[67] M. Köhler and G. Salvaneschi, "Automated refactoring to reactive programming," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, p. 835–846, IEEE Press, 2019.

[68] G. Salvaneschi and M. Mezini, "Debugging for reactive programming," ICSE '16, (New York, NY, USA), p. 796–807, Association for Computing Machinery, 2016.

[69] G. Salvaneschi and M. Mezini, "Debugging reactive programming with reactive inspector," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 728–730, 2016.

[70] I. Perez and H. Nilsson, "Testing and debugging functional reactive programming," *Proc. ACM Program. Lang.*, vol. 1, aug 2017.

[71] A. Tasnim, "anisha150213/asteroid: A toy unity game: Asteroid." `https://github.com/anisha150213/Asteroid`.

[72] N. Heikkilä, K. Passi, and A. Väisänen, "asteroids/asteroids.hs at master · nikoheikkila/asteroids." `https://github.com/nikoheikkila/asteroids/blob/master/Asteroids.hs`.

[73] M. Hayden, "Haskelloids/stack.yaml at master · keera-studios/haskelloids." `https://github.com/keera-studios/Haskelloids/`.

[74] J. Rios and J. Burtet, "juan-burtet/haskellpong: The game pong make in haskell." `https://github.com/juan-burtet/HaskellPong`.

[75] V. Lopatin, "madjestic/pong: A pong game in haskell, using yampa frp, sdl2." `https://github.com/madjestic/Pong`.

[76] A. Tasnim, "anisha150213/pong_game_unity." `https://github.com/anisha150213/Pong_Game_Unity`.

[77] Hooson, "How to make a glowing pong game for beginners - easy unity tutorial - youtube." https://www.youtube.com/watch?v=JZvNFrS7wTM.

[78] P. A. Salu, "Functional reactive programming for games," *Electronic Theses and Dissertations*, vol. 442, 2015.

[79] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, aug 2013.

[80] A.Martin, "Data structures for entity systems: Multi-threading and networking – t-machine.org." https://t-machine.org/index.php/2015/05/02/data-structures-for-entity-systems-multi-threading-and-networking/.

[81] A. Papari, "Introduction to entity systems · junkdog/artemis-odb wiki." https://github.com/junkdog/artemis-odb/wiki/Introduction-to-Entity-Systems.

[82] T. Härkönen, "Advantages and implementation of entity-component-systems - trepo." https://trepo.tuni.fi/handle/123456789/27593.

[83] "Entity component system - wikipedia." https://en.wikipedia.org/wiki/Entity_component_system.

[84] "Get your unity pro subscription today — unity." https://unity.com/pages/unity-pro-buy-now?

[85] "The most powerful real-time 3d creation tool - unreal engine." https://www.unrealengine.com/en-US.

[86] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, aug 2013.

[87] "Dots - unity's data-oriented technology stack." https://unity.com/dots.

[88] "Game development - haskellwiki." https://wiki.haskell.org/Game_Development.

[89] "gloss." http://gloss.ouroborus.net/.

[90] B. Lippmeier, "gloss: Painless 2d vector graphics, animations and simulations.." https://hackage.haskell.org/package/gloss.

[91] "Yampa/game engine - haskellwiki." https://wiki.haskell.org/Yampa/game_engine.

[92] J. Carpay, "apecs: Fast entity-component-system library for game programming." https://hackage.haskell.org/package/apecs.

[93] A. Furtado, "Fungen: A lightweight, cross-platform, opengl-based game engine.." https://hackage.haskell.org/package/FunGEn.

[94] L. z. Prezzavento, "fwgl-glfw: Fwgl glfw backend." https://hackage.haskell.org/package/fwgl-glfw.

[95] A. Gushcha and L. Oganyan, "gore-and-ash: Core of frp game engine called gore&ash." https://hackage.haskell.org/package/gore-and-ash.

[96] A. Gushcha, "gore-and-ash-actor: Gore&ash engine extension that implements actor style of programming." https://hackage.haskell.org/package/gore-and-ash-actor.

[97] "Start your creative projects and download the unity hub — unity." https://unity.com/download.

[98] "Downloads." https://www.haskell.org/downloads/.

[99] "The freeglut project :: About." https://freeglut.sourceforge.net/.

[100] L. Raines, "History of asteroids the classic arcade game." http://www.classicgaming.cc/classics/asteroids/history.

[101] "Pong game." https://www.ponggame.org/.

[102] "Shader - wikipedia." https://en.wikipedia.org/wiki/Shader.

[103] "Émile cohl - wikipedia." https://en.wikipedia.org/wiki/\%C3\%89mile_Cohl.

[104] J. O'Brien, J. Hodgins, and R. Bodenheimer, "Computer animation," 04 2001.

[105] "Maya software — get prices and buy maya 2023 — autodesk." https://www.autodesk.com/products/maya/overview?term=1-YEAR&tab=subscription.

[106] "2d animation software, flash animation — adobe animate." https://www.adobe.com/products/animate.html.

[107] "Animation & rigging — blender.org." https://www.blender.org/features/animation/.

[108] "Unity - manual: Animation." https://docs.unity3d.com/Manual/AnimationSection.html.

[109] "Timeline — blender manual." https://docs.blender.org/manual/en/latest/editors/timeline.html.

[110] "How to animate characters with animation rigging — unity." https://unity.com/solutions/rigging-in-animation.

[111] M. Pawlan, "What is javafx? — javafx 2 tutorials and documentation." https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm#:~:text=JavaFX\%20graphics\%20are\%20based\%20on,graphics\%20processing\%20unit\%20(GPU).

[112] J. Juneau, C. Dea, F. Guime, and J. O'Conner, "openjdk/jfx: Javafx mainline development." `https://github.com/openjdk/jfx`.

[113] N. Welsh, "noelwelsh/flan: An animation library for the javascript functional reactive library flapjax." `https://github.com/noelwelsh/flan`.

[114] "React – a javascript library for building user interfaces." `https://reactjs.org/`.

[115] "facebook/react: A declarative, efficient, and flexible javascript library for building user interfaces.." `https://github.com/facebook/react`.

[116] E. Czaplicki, "A farewell to frp." `https://elm-lang.org/news/farewell-to-frp`.

[117] E. Czaplicki, """code is the easy part" by evan czaplicki - youtube." `https://www.youtube.com/watch?v=DSjbTC-hvqQ`.

[118] "psg-titech/emfrp: Pure functional reactive programming language for small-scale embedded systems." `https://github.com/psg-titech/emfrp`.

[119] J. Hughes, L. Pareto, and A. Sabry, "Proving the correctness of reactive systems using sized types," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, (New York, NY, USA), p. 410–423, Association for Computing Machinery, 1996.

[120] M. Bärenz, "essence-of-live-coding: General purpose live coding framework." `https://hackage.haskell.org/package/essence-of-live-coding`.