Leveling Up: Could Functional Programming Be a Game Changer?

Christopher Maraffi and David Seagal
University of California, Santa Cruz
Santa Cruz, CA 95064
{topherm, drseagal}@soe.ucsc.edu

1. INTRODUCTION
1.1 Functional Programming and Games
This survey is intended to examine why the paradigm of Functional Programming (FP), while increasingly popular for academic research and mission critical applications, has had only limited success in the domain of video games. Although there are some recent commercial games that use FP, most notably the F# AI engine for the Xbox 360 game The Path of Go (2011) by Microsoft Research, imperative languages like C++ have remained the favorite of game developers because of real-time efficiency and industry support. This current state of affairs is rather different than what was predicted by FP researchers in the mid-1990s, when work on real-time reactive versions of FP began to be developed.

Most of the attempts to integrate FP into videogame development have used a variation known as Functional Reactive Programming (FRP). By adding reactive elements, most notably the concepts of behaviors/signals and events being varied over time, FRP allows for an updatable game state. Even though FRP can be treated as a subset of FP, for the purposes of this paper we specify which is used for a particular implementation. In addition, although there have been other FRP surveys done (Amsden 2011), they did not have a games focus.

In this paper we examine the following categories for FP use in games from three different points of view:

- Section 2 – A limited number of finished 2D arcade style and 3D console style games that employ FRP.
- Section 3 - Creating simple games for pedagogical purposes, either in teaching game development using FRP, or for teaching FP concepts in a CS curriculum.
- Section 4 - Using FP to create the reactive temporal media tools that can be used for game development, such as reactive animation and sound.

We hope this paper will reveal some insights into why FP has had more success in academia than commercial games, and show how some trajectories of FRP research could possibly impact this domain.

2. FULL GAMES IN FRP
2.1 Yampa Arcade
Although there had been some earlier investigations into using FRP for game development, one of the first breakthroughs came in 2003 with the Yampa Arcade, discussed in detail in (Courtney et al 2003). The Yampa Arcade serves as the jumping point for most subsequent research, in particular (Cheong 2005).

Since Yampa is a FRP language, it is based on signals and events. A signal is a function from time to a value. This time is not discreet, but rather signals use continuous time. These signals can in turn be passed around and applied to each other using signal functions. Events are used to handle aspects that require discreet time, such as a mouse click.

![Figure 2.1. The Yampa Arcade is a Space Invaders-inspired game designed to highlight the functionality of Yampa.](image)

The game created for the Yampa Arcade was inspired by traditional arcade shooter games such as Space Invaders. The player controls a movable weapons platform and must shoot down increasingly difficult waves of alien spacecraft. The game consists of three objects:

- Gun. The object controlled by the player. The mouse is used to control its movement, and clicking the mouse triggers a bullet to be fired.
- Bullet. The object fired by the player against the alien ships. When created, they move towards the top of the screen and are checked for collisions against the aliens. They are removed upon collision with an alien, or after a certain amount of time.
• Alien. The enemy craft trying to be destroyed by the player. They are created at the beginning of an attack wave, and are removed when their shields are depleted by collisions with bullets.

Figure 2.2. The structure of the Yampa Arcade is designed to be easily adjustable for any type of game.

Although the Yampa Arcade demonstrated how FRP can be used for videogame development, as a game it is considerably sparse, consisting of elementary game mechanics and rudimentary graphics. Two years later, Mun Hon Cheong of the University of New South Wales would create a much larger game with (Cheong 2005). The game, FRAG, is a deathmatch style first-person shooter that is clearly inspired by games of that genre created in the late 1990s and early 2000s. The player moves around an environment and shoots at AI-controlled characters.

Figure 2.3. FRAG was designed to showcase how Yampa could be used for commercially-viable games.

Figure 2.4. FRAG uses the same structure as the Yampa Arcade, but expands it to include more sophisticated logic and separate graphical functions.

Cheong divided the code of FRAG into three categories:

• The game facilities called by the main loop.
• The dynamic collection of game objects.
• The graphical representation of the game.

The basic structure demonstrated by the Yampa Arcade is preserved and built upon. The route function still takes a combination of the output from the main loop and the player input, processes them, and determines what inputs to send to the collection of game objects. The game objects still keep track of their own state, and after processing their inputs, send out signals of their own. In addition to being used as the new input for the route function, the output is sent to the killOrSpawn function, which determines when to add or remove game objects to and from the collection.

2.2 FRAG

Due to the more complex nature of FRAG, the operations performed by the route function are expanded on from the Yampa Arcade. Collision detection is still performed, both among game objects and between objects and game world geometry. Each game object exists isolated from direct contact from any other game object. A messaging system is introduced, allowing game objects to communicate with each other using the route function as an intermediary. Since FRAG employs AI agents for the player enemies, the route function also performs the necessary visibility calculations to determine the AI behavior.
Performance for FRAG was measured in terms of framerate given a certain number of AI objects, first with low rendering settings and then with high settings. 60 frames per second was the highest framerate, with 30 being the lowest acceptable framerate. In both cases, the framerate dropped quickly as the number of AI objects increased, falling below 30 frames per second quicker than similar games in the genre. Cheong attributes the reduced performance to a lack of optimization on many aspects of the rendering code and logic code, especially collision detection. Nonetheless, performance continues to be the greatest obstacle that must be overcome before videogame development using FP becomes more widespread.

3. FP/FRP GAMES AND PEDAGOGY

3.1 Rationale

The growth of the videogame industry over the last few decades can in part explain the renewed interest in computer science. This change is attributed in part to the greater feeling of individuality and creativity that comes with developing a videogame instead of a traditional academic program (Morazán 2010). As a result, more and more schools are incorporating videogame development into their curriculums. At the same time, FP continues to gain in popularity. This section examines the intersection of these two trains of thought with attempts to teach FP/FRP by having students create videogames.

Figure 2.5. The framerate of FRAG with varying numbers of game objects.

Figure 3.1. An Asteroids-inspired game designed as the final project for second-year students using the Haskell Graphical Library (Lüth 2003)

Videogame development has traditionally fallen into one of two categories. One paradigm is an extension of traditional Computer Science curriculum, with students still learning the established Object-Oriented Programming languages (C++, Java). On the other end are curriculums that deemphasize programming to the point of triviality, providing a minimum of programming using specially designed development tools or high-level scripting languages. FRP has been suggested as a compromising middle ground between these two paradigms (Peterson and Evans 2008). This desire to strike a balance is particularly emphasized for prospective and introductory students, where the emphasis should be on design and problem solving, while abstracting away less exciting details such as maintaining state, variable mutations, and convoluted I/O (Morazán 2010).

It should be noted that to some degree pedagogy has influenced all the projects discussed in this survey. In particular, the Yampa Arcade was written in part to address difficulties that were emerging when attempting to create a game using the Fruit library for Haskell (Courtney et al 2003).

Figure 3.2. A game inspired by Space Invaders is used by introductory students to learn introductory Computer Science concepts
3.2 Implementations

Figure 3.3. Lüth’s structure for a videogame designed in Haskell using the Haskell Graphical Library.

In 2003 Christoph Lüth released his results on incorporating videogame development into an introductory FP class (Lüth 2003). The final assignment for students was to create a game in Haskell based on the arcade game Asteroids. The Hugs Graphical Library is used to render the game to the screen. The Yampa Arcade, which served as a template for the other pedagogical endeavors, had not yet been developed, so Lüth’s techniques provide an interesting alternative route.

In 2008 John Peterson and Kendric Evans proposed a curriculum meant to use FRP to introduce computer science techniques to high school students (Peterson and Evans 2008). The prior curriculum was created using Panda3D, an open-source game engine built using the Python programming language. Instead of using Yampa, Peterson and Evans instead built a custom FRP system on top of Panda3D.

In 2011 Marco Morazán describes an introductory computer science course that mixes videogame development and FP (Morazán 2010). Unlike with Lüth, here the entire class is structured around creating a game instead of as a final assignment, with each assignment adding a component to the game while teaching the necessary concepts. In addition to keeping students motivated throughout the course, this approach also highlights one of the advantages of FP/FRP design: the ease in which each component can be tested in isolation before incorporating it into the larger code base.

Two observations can be made throughout these pedagogical exercises. First, creating these games using either FP or FRP languages resulted in considerable reductions in program sizes. Peterson and Evans observed that the FRP version of their game required only about a fifth of the lines of code required in the original Panda3D version; about 50 lines instead of 250. The game developed for students by Lüth only required about 220 lines of code. Second, students were consistently more inspired and motivated by creating games. Of the eight programming assignments presented to students by Lüth, the Asteroids game received the most praise, with nearly half the class reporting it as their favorite assignment. These studies show the pedagogical potential of both videogame development and FP/FRP.

4. EVOLUTION OF FUNCTIONAL REACTIVE PROGRAMMING

Several issues are quickly encountered when surveying FP for game development. Academic papers are sparse in this domain, and links to working prototypes are too often out of date or broken. There are, however, a couple of related research branches that continue to be relevant: the FRP body of work by Conal Elliott and Paul Hudak (see Figure 3.1). Though their research only occasionally addresses games, their evolution of Domain Specific Languages (DSLs) is polymorphic to any reactive media, and therefore could ultimately impact game development. Starting with their foundational work on Functional Reactive Animation (FRAN) in the late 1990s, now called by Elliott Classic FRP, this section traces their research as it branches into two modern FRP approaches to building tools for artistic composition.

4.1 Classic FRP: Modeling Behaviors in Continuous Time

Classic FRP began with FRAN, a Haskell embedded DSL for reactive animation (Elliott & Hudak 1997), which was then generalized to FRP (Wan & Hudak 2000). Hudak also featured a version of FRAN, which he called Functional Animation Language (FAL), in his Haskell School of Expression book (Hudak 2000). Classic FRP takes a declarative modeling approach to reactive temporal media. Modeling across the time dimension describes what media is at a high-level, as represented by polymorphic behaviors and events, rather than how to present it. Time is represented as continuous, in the same way that space is mathematically continuous in vector graphics, allowing for more flexible scaling. In contrast, imperative approaches focus on presenting animation in discrete time increments, similar to resolution dependent bitmap graphics. Behaviors are first-class time varying reactive values, while events are sequences or streams of occurrences represented as time-value pairs. Classic FRP provided behavior and event operators for modular program composition, such as in this FRAN example for a bouncing ball (see Figure 3.2).
Figure 3.2 Modular program in FRAN for describing a single bounce that can be composed into a bouncing ball animation (Elliott & Hudak 1997).

Static values can be ‘lifted’ to polymorphic type behavior-value, with lifts and integration being used in the bouncing ball example to calculate gravity acceleration on position and velocity. Here are some common lift operators in Classic FRP (Wan & Hudak 2000):

```haskell
> color :: Behavior Color
> color = red `until` (lbp -> blue)
```

Reactive behavior is in the form of mouse movement, and events are in the form of keyboard or mouse clicks. An example would be changing a color behavior from red to blue with a left mouse button event:

```haskell
> color :: Behavior Color
> color = red `until` (lbp -> blue)
```

Behavior switching is done through the ‘until’ infix event operator:

```haskell
> until :: Behavior a -> Event (Behavior a)
>    -> Behavior a
> fb 'until' fe =
>  \ts -> loop ts (fe ts) (fb ts)
> where loop ts@(e:es) (b:bs) =
>  b : case e of
>     Nothing -> loop ts' es bs
>     Just fb' -> tail (fb' ts)
```

A snapshot operator samples a behavior at discreet moments in time:

```haskell
> snapshot :: Event a -> Behavior b
>    -> Event (a,b)
> snapshot fe fb =
>    \ts -> zipWith aux (fe ts) (fb ts)
> where aux (Just x) y = Just (x,y)
> aux Nothing _ = Nothing
```

Though provably sound, Classic FRP encountered efficiency problems when applied to real-time implementations like games. In the next section we will see how each researcher responded differently to these challenges, evolving two research tracks. Hudak developed Yampa as a hybrid FRP approach at Yale from 2001-2003, while Elliott continued developing their original FRP approach at Microsoft Research. Then Elliott made significant revisions to Classic FRP in 2009, creating the modern version Reactive.

4.2 Modern FRP: Applied Reactive Temporal Media

Even as Classic FRPs first principles were being defined, Hudak suggested that streams of sampled behaviors could be represented as signals, while events could be observations of physical signals outside the FRP system (Wan & Hudak 2000). His signal processing approach was developed over the next few years with Real-Time FRP (Wan et al. 2001), Event-Driven FRP (Wan et al. 2002), and finally Yampa (Hudak et al. 2002). Yampa added a generalization of monads in the form of arrow notation to the signal metaphor, fixing some efficiency problems for real-time applications. Although FRAN and FAL had worked reasonably well for small animation prototypes, larger real-world FRP implementations like interactive games had problems with time and space leaks. Yampa was designed to correct this problem by removing the first-class status of signals, hiding them in a point-free style so values are not manipulated directly. Instead of building signals from first-class values, they provide a primitive set of signal functions and combinators that use the arrow notation to sweeten the syntax (see Figure 3.3).
Elliott continued to develop a pure functional approach to
graphic DSLs using Classic FRP, as demonstrated in Pan (Elliott
2001), where he described ideal images as functions, both
continuous and infinite, that were composable as points of color
and transparency in a 2D plane. More generalized, in Pan image
becomes a type constructor, parameterized by an arbitrary type.
Colors are quadruples of rgb real numbers and an alpha real
number, and transforms are space to space functions:

```haskell
>type Image c = point -> c
>type Color = (Frac, Frac, Frac, Frac) -rgba
>type Transform = Point -> Point
```

Elliott continued to evolve his version of FRP with the concept of
Tangible Values (TVs), implemented in Eros (Elliott 2007), that
visualized image functions in order to provide artists with a
Graphical User Interface (GUI) for image composition. His
motivation was to create a more powerful Photoshop-like tool
where the artist can create their own filters. In Eros, he used
‘deep’ arrow combinators to compose image functions through
GUI gestural ‘fusion’ (see Figure 3.5)

Additional revisions were completed in 2009 to deal with latency
issues that caused real-time behavior to take as long as the
sampling period. The FRP syntax was modernized to include
standard type classes such as monoids and functors, and a hybrid
push-pull symantics was implemented based on Sperber’s Lulu
FRP (Elliott 2009). The key revision concept was to combine
demand and data driven sampling of events, called Reactive
Normal Form, which purportedly made reactivity nearly
instantaneous. This was done by separating behaviors into
continuous-time and discrete-time events. New features were
added in the form of future values, improving values, and
unambiguous choice. The latest implementation was called
Reactive, and has been combined with the Fieldtrip functional
3D library (see Figure 3.6)
Elliott has stated a similar motivation for developing language tools for artists that are both useable and composeable. He has pointed out that it is important not to change the way right-brain artists think by forcing them into a left-brain engineering mentality that would alter their creative process. Instead, he has proposed visualizing FRP as tangible values (TVs) in a GUI that allows more ‘gestural’ function composition. The motivation is to make the powerful medium of computation useable to content creators who have not had access to it through end-user software, even beyond scripting and macros (Elliott 2007). The idea was prompted by composing simple UNIX programs using pipes, only instead of being based on text streams, the program utilizes FRP’s streaming media. His Eros application uses GUIs to compose or fuse graphic programs in a similar manner to UNIX piping, only with scalable and consistent data types. His goal was to move away from command-line authoring by focusing on interfaces and functionality, in an attempt to modernize how FRP is done and who does it.

5. CONCLUSION

This survey has shown that the lack of FP implementations in the domain of games has contrasted with academic interest in using the fun factor of gaming as motivation for coding pedagogy. We have also shown that the artistic media that is inherent in games has motivated the evolution of FRP research. Even though students are learning FP, and FRP researchers have developed an interesting body of work, there are still too few finished games being produced with this programming paradigm to make an impact on industry. It is also clear that for FP to be more widely adopted in games, optimizations for real-time gameplay have to be tested and verified in full contemporary 3D game applications with rich graphic and AI content.

It is a possibility that FP has not taken off for game development because there wasn’t an ideal fit for it. However, the semantics of FRP, with continuous time and behavior/signals, seems particularly well suited for streaming Natural User Interface (NUI) applications that use Kinect or Wii technology. Could the paradigms of FRP and NUI be merged to provide a new kind of artist friendly tool that enhances content generation for games? The imperative method of keyframing a rig’s joints on a discrete timeline is closer to antiquated stop motion technique than to modern motion capture. FRP could fundamentally change the way animations are produced by virtualizing gesture into performance signal functions, which could then be composed directly through a NUI device like Kinect. This could fuse Hudak’s concept of virtual instruments with Elliott’s concept of tangible values in a new FRP approach for gestural performance in games.

One question that comes up is whether Haskell is the best functional language for game applications? Elliott purportedly found several bugs in the Glasgow Haskell Compiler (GHC) when developing Reactive, which slowed the development process. Perhaps a commercial language like F#, with better support and integration into the presentation pipeline (.NET, C#, XNA, and the Kinect SDK), would be a better choice for making FP into a real game changer.

6. REFERENCES


Morazán, Marco. Functional Video Games in the CS1 Classroom. TFP’10. 2010


