

Iterative Functional Reactive Programming with the Nu Game Engine

An Informal Experience Report

NOTE: This is an entirely informal 'paper', originally targeted for a simple post on a forum of <http://lambda-the-ultimate.org>, but due to my inability to achieve a readable formatting with the tools of said forum, is made available via this PDF.

So, I have this purely functional game engine in F# - <https://github.com/bryanedds/FPWorks>

For readers who wish to build some familiarity with the system before reading this admittedly terse description, please look here for Nu's high-level documentation - <https://github.com/bryanedds/FPWorks/blob/master/Nu/Documentation/Nu%20Game%20Engine.pdf?raw=true>

The Nu Game Engine certainly qualifies as 'functional reactive' in that -

- 1) it is reactive in that it uses user-defined events to derive successive game states.
- 2) it is functional in that it uses pure functions everywhere, even in the event system.

However, I cannot describe it as the typical first-order or higher-order FRP system since it uses neither continuous nor discrete functions explicitly parameterized with time. Instead it uses a classically-iterative approach to advancing game states that is akin to the imperative tick-based style, but implemented with pure functions.

Thus, I gave it the name of 'Iterative FRP'. It's a purely-functional half- step between classic imperative game programming and first / higher-order FRP systems. There are plusses and minuses to using this 'Iterative' FRP style -

In the plus column -

- 1) It's perfectly straight-forward to implement most any arbitrary game behavior without worrying about things like space or time leaks, or having to invoke higher-ordered forms of expression.
- 2) End-users don't need to understand new forms of expression beyond primitive functional expressions to encode their game logic. No lifting is required (though it is available).
- 3) It has a level of pluggability, data-drivenness, and serializability that are extremely difficult to pull off in FO/HOFRP systems.
- 4) None of its design is the subject of open research questions, and is therefore simple enough for non-PhD's to understand as well as complete in its implementation.

In the minus column -

- 1) Debuggability suffers from the declarative nature of the API. For example, in Visual Studio while stalled on a breakpoint, we cannot inspect the value of a simulant's property or field simply by mouse-hovering over its reference (or in this engine's equivalent, its proxy). In order to inspect a simulant, we must enter something akin to the following into the Watch window -

```
Debug.Entity.peek(entity, world), ac
```

The first thing to notice here is that the peek function is called with a C#-style syntax, which beyond just having to type this out rather than hover, is inconvenient. Also, we may optionally suffix the expression with ,ac in order to have the results of the debug query update as we step through the code.

It would be nice if I could contrive of some sort of debugging plugin to make this experience easier, and perhaps even somehow find a way to re-enable the evaluation and visualization of the simulant's properties / fields via a mouse hover. However, I presume both, and especially the latter, to be quite involved if not infeasible.

- 2) Raw performance, while not in a show-stopping state, is also less than ideal due to the indirect nature of simulants. In order to grab a single Boolean from a single simulant, multiple map lookups must be invoked to locate the value. However, there are some low-hanging optimization fruits that could speed up these accesses considerably, but have not yet been implemented due to time constraints. Fortunately this should not take more than a day or two of work.
- 3) Verbosity is an issue also due to the indirect nature of accessing simulant properties / fields. As mentioned above, one cannot access simulant fields with a simple property access expression, but must be accessed by calling a function that takes a world value as a parameter. Dually, outside of the Chain monad (which will be discussed later), the world value must be manually thread through most engine operations.

Neither of these is a big deal, but they might be off-putting people accustomed to the expedience of empirical programming.

Despite these minuses, I currently conclude IFRP is appropriate for such a general-purpose game engine because -

- 1) from my experience, games, and game technology in general, refuse to conform to any single system of thought, often demanding certain features be implemented with escape-hatch approaches like encapsulated mutation in order to be efficient, or with lower-level forms of functional expression for flexibility.
- 2) just because the default means of expression is not as abstract as that of FO/HOFRP systems, it does not mean there are no available ways to 'climb into' more abstract forms of expression.

To expand on point 2, here are some of the higher forms of expression provided by the API -

Here we have an expression form called an 'observation' that allows combinations and transformations of events like so -

```
let observation =
  observe (ClickEventAddress ->>- hudHalt.EntityAddress) address |>
  filter isSelected |>
  sum TickEventAddress |>
  until (DeselectEventAddress ->>- gameplay.ScreenAddress)
```

This affords us the ability to treat events like first-class collections.

Here we have an expression form called a 'chain'. A chain is a monad that allows a procedural-style expression to span 0 or more events while also taking the world as an implicit state -

```
let chain = chain {
  do! update ^ character.SetActivityState ^ Action newActionDescriptor
  do! during (fun world -> ActivityState.isActing <| character.GetActivityState world) ^ chain {
    do! update ^ fun world ->
      let actionDescriptor =
        match character.GetActivityState world with
        | Action actionDescriptor -> actionDescriptor
        | _ -> failwithmf ()
      let world = updateCharacterByAction actionDescriptor character world
      runCharacterReaction actionDescriptor character gameplay world
    do! pass }}
```

We also have the ability to compose chains overs observations like so -

```
let observation = observe TickEventAddress character |> until (DeselectEventAddress ->>- gameplay.ScreenAddress)
snd <| runAssumingCascade chain observation world
```

Finally, there is a means to declaratively forward the changing value of a simulant's field to that of another -

```
let world =
  world |>
    (bob, bob.GetVisible) *-> (jim, jim.SetVisible) |>
    (jim, jim.GetVisible) /-> (bob, not >> bob.SetVisible)
```

Here, *-> denotes the forwarding of the value of Bob's Visible field to Jim's Visible field. To throw a monkey-wrench into the declaration, /-> is used to in turn forward the value of Jim's Visible field back to Bob's Visible field, albeit **not**'d and with cycle-breaking so that the circularity of this expression is broken appropriately. This is a contrived example, but illustrative of the API's expressiveness.

So, depending on the nature of the game behavior you want to implement, the API provides enough surface area to do things at different points on a spectrum of flexibility. But there is a down-side to an API with a large surface area... With multiple expression forms and levels of abstractions at which to operate, the learning curve can be steepened. Additionally, imperative programmers may be bothered by the performance compromises implied by pure functional programming compared to doing everything in-place, even if performance is satisfactory for most games.

Still, even with functional programming as employed here, and even assuming my approach is optimal for the given point in my targeted design space, game engine design remains fundamentally difficult and peppered with compromise.