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/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 is uses pure functions everywhere, even in the event system.

However, I cannot describe it as the typical 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' (or IFRP for short). It's a purely-functional half- step between classic imperative game programming and first-order FRP systems. There are pluses 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 requires some manual intervention when there are multiple World values to choose from. This is because only the last globally-constructed World value is used in the debugger for simulant state inspection, and that value may have been discarded due to handling an exception (or some other future-disambiguating code such as with a Left constructor).

   For example, in Visual Studio, while stalled on a breakpoint, we can inspect the value of a simulant's properties by mouse-hovering over it. To make this possible when multiple World values can exist simultaneously, the chosen World value used to look up these values is a cached reference to the last globally-constructed World value (yes, *that* type of global). If the value of that last construction has been thrown away due to, say, an exception, then the shown entity's values may incorrectly reflect the value of the discarded World - *unless* the previous World value has been chosen with **World.choose**. That's where the manual intervention comes in.

   Always calling **World.choose** in the face of exception handling (and in other code paths where Worlds may diverge) is required to keep the debugging experience consistent.

If the user suspects that the World value used by the debugger is inconsistent, they can inspect a simulant based on the world in the inspected stack frame with the following into the Watch window expression –

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

If the simulant's values are inconsistent, either the stack frame contains a World older than the one on the lowest stack frame, or the correct world has not been chosen. If the latter, this must be corrected either in the engine or the user's code by adding a **World.choose** invocation in the place where the Worlds diverged.

2) Raw performance is good, but is not, in will never likely be, best in class. Let's face the fact that many game programmers are speed addicts, and while their endless thirst may never be satisfied, the engine has low-level imperative optimizations that make it more than fast enough when used with care.

3) Verbosity is an issue also due to the indirect nature of accessing simulant properties. As mentioned above, one cannot access simulant properties 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 imperative 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 a 'stream' that allows combinations and transformations of events like so -

```
let stream =
    Stream.until
        (Stream.make Gameplay.DeselectEvent)
        (Stream.sum
            (Stream.make HudHalt.ClickEvent)
            (Stream.make Player.UpdateEvent))
```

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! Chain.update (character.SetCharacterActivityState (Action newActionDescriptor))
    do! Chain.during (character.CharacterActivityState.GetBy state.IsActing) $ chain {
        do! Chain.update $ fun world ->
            let actionDescriptor =
                match character.GetCharacterActivityState world with
                | Action actionDescriptor -> actionDescriptor
                | _ -> failwithumf ()
            let world = updateCharacterByAction actionDescriptor character world
            runCharacterReaction actionDescriptor character world
        do! Chain.pass }}
```

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

```
let stream =
    Stream.until
        (Stream.make Gameplay.DeselectEvent)
        (Stream.make character.UpdateEvent)
Chain.runAssumingCascade chain stream world |> snd
```

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

```
let world = !-- Bob.Visible --- map not -|> Jim.Visible $ world
```

Here, !-- is an operator that turns a property tag into a stream, and --- is for sequencing stream operations (here, **not**'ing the value of Bob's Visible property). --> is an operator that takes a stream and sets a property tag to its current value.

We can also have cycles in these forwarders so long as the cycle is broken with the -/> operator -

```
let world = !-- Bob.Visible -|> Jim.Visible $ world
let world = !-- Jim.Visible -/> Bob.Visible $ world
```

Note that even without cycle-breakers, cycles will be detected and broken in Debug mode, logging an info message. In Release mode, however, it won't be detected without the cycle breaker, and it will loop until the stack overflows. So make sure to get it right while in Debug mode!

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 and abstraction. But there is a down-side to an API with such a larger surface area... With multiple expression forms and levels of abstractions at which to operate, the learning curve is 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 more than satisfactory for most games.

I believe that this 'iterative' approach of FRP is inherently more complete, understandable, scalable, and generally than the more declarative FRP models (First-Order, Higher-Order, Arrowized), and that these properties of completeness, understandability, et al, are the minimum necessary for developing commercial games with sufficient integrity. And as to how this approach compares to imperative / OOP style of game development – I can only hope that this approach, or at least some other functional

approach, will eventually come to replace the current methods. Game development has, in my mind, grown far too complex to make current approaches satisfactory.