PyFRP: Function Reactive Programming in Python

John Peterson
Western State Colorado University
jpeterson@western.edu

Alan Cleary
Montana State University
alan.cleary@cs.montana.edu

Ken Roe
The Johns Hopkins University
roe@cs.jhu.edu

Abstract

Keywords functional reactive programming, declarative languages, Python, educational languages, event-driven programming

1. Introduction

Reactive programming is a general framework for programming hybrid systems in a high-level, declarative manner. It integrates the idea of time flow into a program in a uniform and pervasive manner, giving the program clarity and reliability. It has been used in a variety of contexts, including robotics [13], network control [15], and GUIs [6]. We present PyFRP, an implementation of the reactive programming framework in Python. Although PyFRP is quite robust, we will limit the scope of the examples to its use in a Computer Science summer camp [3]. Not only will this illustrate the versatility of our system, it will show that reactive programming is an appropriate tool for instructing novice programmers in an instructed environment as well.

There are many programming languages that specifically target novice programmers. For example, Alice[5] was designed to teach event driven object oriented programming to children by creating interactive stories. And Scratch[12] is a blocks-based graphical programming language used to create animated stories and games. Though these, and other such languages [8], are effective in their respective domains, they do not provide the full set of features that a more advanced programming language offers, thus limiting their expressive power.

We will demonstrate how the addition of reactive programming to the Python environment significantly increases the ease of programming in an interactive 3D environment without overly constraining the set of programs supported by our system. In particular, this is due to the rich set of operators and high-level abstractions that our system provides the user. Also, the introduction of reactive objects, or *proxies*, encapsulates the conveniences of traditional object oriented programming (OOP) in a reactive environment.

Our paper is structured as follows: in Section 2 we give an overview of functional reactive programming, in Section 3 we introduce reactive proxy objects, in Section 4 we provide the implementation details of our system, in Section 5 we provide examples of our declarative programming style from its in use in a summer camp, and in Section 6 we discuss related work.

CONF '15, October 25–30, 2015, Pittsburgh, Pennsylvania, United States.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Functional Reactive Programming

Functional reactive programming (FRP) is an implementation of the reactive programming model using components from the functional programming paradigm[9]. FRP has the advantage of a well-defined semantic model which prevents semantic ambiguity related to event ordering and provides a framework for optimization. Our goal is to preserve the properties of FRP as much as possible while retaining the object-oriented feel of Python (or any other O-O language). In this section we present the basics of FRP and introduce our Python-based system. The examples in this section are taken from the Reactive Panda system, an integration of the Panda3d game engine[11] into PyFRP.

2.1 Time

FRP is a style in which a program is developed describing a relationship between time and value. In general, reactive programming is a pervasive manifestation of the observer pattern. In FRP, this is further refined to deal with issues such as event simultaneity and psuedo-continuous values. For example, the reactive variable `time` represents the current time as the program proceeds. A "hello world" program in our system would be:

```
text(time)
```

The `text` function is not part of the traditional FRP vocabulary - it is a constructor for an on-screen text label in our game engine binding. This is an example of a *reactive proxy* object, to be described later. This program displays the time since the program started on the screen. The screen display keeps updating itself until the program is terminated.

Lifted math operators allow arithmetic expressions such as `time+1` to be be written. The program `text(time+1)` shows a time on the screen that starts at 1 and continuously updates.

2.2 Behaviors and Events

A *behavior* is a function from a real number representing time (starting from 0.0) to values. An *event* is a sequence of occurrences at specific times carrying values at each time. The first element of each pair is a positive real number representing time and the second is a value for the event. One could, for example, represent all of the key presses that occur during a program execution by a sequence of pairs, such as `[ (1.0,'A'),(2.5,'B') ]`. This sequence represents the fact that the 'A' key was pressed one second into execution and that the 'B' key was pressed at 2.5 seconds into execution. We refer to behaviors and events together as *signals*.

The concepts of behaviors and events is best illustrated with an example. Consider a game in which a spaceship is displayed, initially in the middle of the screen. However, the left and right arrow keys can be used to start the spaceship moving to the left or right.

```
s = spaceship()
s.position = integral(hold(p3(0,0,0),
                          key("leftArrow", p3(-1, 0, 0)) +
                            key("rightArrow", p3(1, 0, 0))),
                            p3(0,0,0))
```

The `spaceship` function is a constructor that places a model in the game engine's virtual world. The second line shows the use of *reactive attributes*. The spaceship object is placed in variable `s` and its `position` attribute is set to a reactive value. This uses a velocity controller - in this coordinate system x decreases to the left and increases to the right so the arrow keys define appropriate motion vectors (the `p3` constructor) in three dimensional space. The second argument to integral is the initial value, the starting location of the spaceship.

The `key` operator represents a sequence of pairs representing the times at which the left arrow is pressed (an *event* in FRP terminology). The value for the left arrow is always `p3(-1,0,0)`, a velocity. If the left arrow key is pressed twice, once two seconds after the game started and a second time five and a half seconds after the start of the game, then the `key` function would denote the event stream `[(1.0,p3(-1,0,0)),(5.5,p3(-1,0,0))]`. The `+` in this case is not addition but instead it represents the FRP event merge operator. Suppose in this same game that the right arrow was pressed at 2.0 and 5.5 seconds after the start of the game. The event sequence returned by the second `key` would be `[(2.0,p3(1,0,0)),(5.5,p3(1,0,0))]`. The `+` (the `.|.` operator in classic FRP) then combines these sequences to create the sequences `[[(1.0,p3(-1,0,0)), (2.0,p3(1,0,0)), (5.5,p3(1,0,0))]`. There is a conflict at time 5.5 as both keys are pressed. The `+` always takes the value on the right when creating the combined sequence. An event is not allowed to have two values for the same time.

The function `hold` converts an event into a behavior–a continuous function from time to values. The first value is what this behavior should return between time 0 and the time of the first key press; here it is the value `p3(0,0,0)`. The second value is the event that updates the result of hold. The first pair in the event is `(1.0,p3(-1,0,0))`. The value of the behavior starting at time 1.0 and up to (but not including) time 2.0 (the time of the next pair in the event) is `p3(-1,0,0)`. The value returned by `hold` can be interpreted as the velocity of the spaceship. It's position can be obtained by taking its integral. The starting point (as defined in the call to `integral`) is `p3(0,0,0)`.

An essential idea in FRP is that all signals advance in synchrony. For example, if two different velocity controllers are using the arrow keys, their velocities will effectively change at the same time - the order in which they change will not be observable.

3. FRP and Object Oriented Programming

The primary innovation in this work is the augmentation of FRP with *reactive proxy* objects. Reactive proxies mediate between the traditional OOP toolkits of interactive systems and the reactive engine. These proxies serve a number of purposes:

- They have reactive attributes which are continuously updated by the reactive engine.
- They can be associated with *reactors* which can change the system configuration.
- Object components are observable through external references to the object.
- Reactive objects can be augmented with new fields, allowing components to be added dynamically.

Reactive proxies are created through an *object lifting* process, similar to the lifting of functions in FRP. This facilitates construction of new reactive libraries and allows the end user to perceive the library as if it was implemented in a reactive fashion.

The set of reactive proxies under control of the reactive engine is termed the *world*. The original FRP implementations were based on a fixed world, usually a graphics screen, keyboard, and mouse. Here, the world is dynamic - proxies come and go from the world as needed.

3.1 Reactive Attributes

Each proxy object has an associated set of attributes which are continuously being synchronized with the contained object. For example, consider creating a new model in a game engine:

```
spaceship(position = p3(time, 0, 0), size = 2*time)
```

This creates a new reactive object, a spaceship, which moves and gets bigger as governed by the `position` and `size` attributes. We support two different styles of object construction: one in which the reactive components are passed as named parameters to a constructor and one in which reactive components are assigned later using the dot operator. This is mostly notational - both accomplish the same thing. The dot syntax is particularly useful for objects with are self-referential, allowing circular references among objects. For example, consider the following code:

```
s = spaceship()
s.position = hold(p3(0,0,0),
   integral(key("leftArrow", p3(-1,0,0)) +
            happen(getX(x.position)<-2, p3(1,0,0))),
       p3(0,0,0)))
```

Here, the equation defining the velocity of the spaceship refers to the position so that when the x component of the position goes below 2 (the `happen` event) it will bounce back right. Here the dot notation is required since Python does not allow forward reference to variables.

Some attributes are used to bring signals into the reactive engine:

```
s = slider(min = 0, max = 10)
text(s)
```

This creates a slider that ranges from 0 to 10 and displays the value as a string in text label. The screen shows both of these objects at default locations.

3.2 Integration of FRP and imperative programming

In traditional FRP, a program's behavior is fully specified by composing reactive functions. No assignments or other side effect operators are ever executed.

In FRP the `switch` function is used to change a signal in response to an event. Our system provides an alternate way to shape the system behavior: proxy object reactions or *reactors* for short. *Reactors* are attached objects in the system and perform an action when ever a particular state is reached. Reactors have the following capabilities:

- Add or remove objects from the world.
- Update object attributes.
- Sample the current value of reactive attributes.
- Add new reactors to a proxy.

Each reactor is associated with an object and a triggering event. These are similar to event handlers in event-driven programming except that the triggering event is defined by an arbitrary FRP event and the response to the event is more constrained.

All reactors are parametrized over the reacting proxy object and the value of the triggering event. This allows reactions to be reused among different reactive objects and provides a communication channel between the event and its handler.

Here is an example of a reactor function:

```
def fire(ship, speed):
   p = now(ship.position)
   missle(position = p + integral(p3(0, speed, 0)))
   ship.position = p

s = spaceship()
react(s, key("upArrow", 1) +
      key("downArrow", -1), fire)
```

The `fire` function is a reactor - it takes two parameters: the object associated with the reaction and the event value. Here, the object is the spaceship firing the missile and the event value determines the y component of the missile velocity, allowing the reactor to fire both up and down.

The `react` function and the spaceship constructor are part of the main program which is an implicit reactor triggered at the start of the program. The addition of objects to the world is implicit in the object constructors, `spaceship` and `missle`. The `now` function returns a static (non-reactive) value, a snapshot of the ship's position when the `fire` reactor is invoked. Attribute update functionality is implicit in the assignment of a new `position` attribute to the ship. This will cause the ship to freeze at the spot where the missile is fired. Also note the use of event merging to trigger the reaction on either the up or down arrow.

3.3 Observation

The attributes of a reactive object are observable by other reactive objects. Since attributes may be updated by reaction functions, it is important to determine whether a reference returns just the current signal or if it always tracks the attribute even when changed. We implement the latter: `o.x` refers to the current value of attribute `x` in object `o`, that is, observation always goes through the object.

For example, in

```
ship1.position = ship2.position + p3(1,0,0)
```

the position of `ship1` tracks the other ship at all times - even if the `position` attribute of the ship is changed by another reactor. So if a reactor teleports `ship2` to another position in space, `ship1` would also be teleported.

One interesting property in our engine is that you can specify undefined behaviors by setting an attribute to a function of itself. Consider this statement:

```
ship1.position = ship1.position+1
```

This has the same sort of meaning as `x=x+1` in Haskell - it is a circular reference that leads to an evaluation loop.

Using the `now` function breaks this recursion. If, for example, we had

```
ship1.position = now(ship2.position + p3(1,0,0))
```

we would not see `ship1` get teleported when the reactor teleports `ship2`. In fact, `ship1` will just stay in the same place until another value is assigned to `ship1.position` by a reactor. If we want `ship1` to be moving with a constant velocity, we could assign

```
ship1.position = now(ship2.position + p3(1,0,0))
                 +integral(velocity)
```

This places `ship1` relative to `ship2` and has it moving at a constant velocity.

In general, we have attempted to provide reasonable error message for such evaluation loops so that the programmer can easily detect such errors.

3.4 Lifting Proxy Objects

One goal of our work is to make the object lifting process simple enough that adding reactivity to an existing library is relatively simple. Lifting objects is more complex than lifting functions since an object may contain an arbitrary set of attributes, some of which may be observations of the underlying object state. Furthermore, the system needs to be able to dispose of the underlying object if it is removed from the world. Thus the proxy lifting code needs to specify the following:

- Object creation code. This may need static parameters in addition to the reactive ones.
- A set of attribute descriptions. Each attribute has a name, a type (used for error messages), a default value, and a reader or writer, depending on whether this attribute takes its value from the underlying object (for example, a slider value) or is used to update this object (for example, the spaceship position).
- Object destruction code.

While this may seem like a lot of work, proxy lifting is done by the library designer, not by the ordinary user. For example, in the panda3d library we lifted about 10 object classes to create the library used by the computer camp students.

3.5 Start Times

This section addresses a somewhat obscure semantic issue and is not essential to later discussion. The meaning of a reactive value that includes stateful operators depends on when it is started. For example, the expression `integral(1, 0)` is meaningless without knowing when the integral starts. In the original FRP system, the start time was determined by signal initialization at the start of execution or in the `switch` operator. In general, we must take care to distinguish an initializing reference to a signal from an observing reference. For example, consider two separate definitions:

```
x = integral(1, 0)
y = x + 1
```

If these two statements are in two different reactors that fire at different times then the start times are different. Should `y` see an integral that starts at its definition or at the time the previous definition? In the original FRP, there is no notion of assignment; both would start at the same time. Other implementations, such as Father Time[4], treat this differently: the first integral starts when `x` is assigned and the second is an *observer*. This situation has been the source of much confusion, especially when FRP is embedded in an imperative language. This issue was first addressed by the Yampa system[7], which uses the arrow abstraction to make initialization explicit.

4. Implementation

In this section we discuss the implementation of our system and some of the semantic issues involved. Much has been written about the implementation of signals and their update strategies in FRP [2, 4, 9]. It is important to note that Python supports higher order functions via its `lambda` construct, but not lazy evaluation. Our implementation is designed to integrate the functional style of traditional FRP with the imperative style of object oriented programming.

The reactive engine, or *reactimate*, implements the overall execution strategy of the system. The primary area of application of our system, PyFRP, has been in the implementation of interactive games. Hence, there is a heartbeat clock that fires updates on a regular interval (such as once every 0.05 seconds for animation). Inputs to the reactive engine are from IO devices such as the mouse and keyboard. These low level events are often converted to higher level of abstraction such as button presses or text field updates. Events from these input devices are generally queued until the next heart beat update fires off. At this point, the hardware events are placed in reactive event signals within the engine's world.

Our update process is pull based. This means that the process is driven by the reactive proxies (which are the top level objects) for which we want to find updated values. A top level loop iterates through each of the proxies to update their values. Signals on which the proxy depends are updated through recursive calls. Updated values for signals are cached so that if a signal is used in two different places, the second call will simply retrieve the cached value.

Our representation / update strategy is perhaps the simplest possible one: we implement signals as continuation functions and use a pull based strategy to update them. A continuation function is a function which updates a signal and returns its current value. Using a pull strategy for update requires that the continuation function evaluate inputs before producing an output. The integral function is the only exception - it produces a value before evaluating the input to allow recursive integral equations. We deal with this situation by providing a mechanism for residual signal update. Our strategy guarantees that no evaluation work is deferred during update, preventing the time leaks that can complicate a lazy implementation of FRP. A time leak occurs when the work to update a signal is delayed until a later time. This can lead to an unacceptable lag if a the current value of the signal is needed requiring the delayed computations to all happen at the same time instead of gradually with each tick of the internal clock. As our update techniques are not novel we will focus on the environment that these signals run in.

This engine mediates between the set of reactive objects and their external presentation. This engine is based on a heartbeat clock which is used to update all signals in the system. The source of the heartbeat is application dependant - a graphics application will typically use the frame rate while numerical simulations may require a faster heartbeat for more precise results.

Each heartbeat requires the following steps:

- Each reactive object in the world undergoes attribute update. This evaluates each reactive attribute and then pushes these changes to the object under the proxy. These reactive values are cached so that an attribute is computed only once even when it has multiple observers.
- The reaction events of each object are evaluated to collect a set of actions to be performed. All reactive values must be evaluated before any reaction function is executed. At this point, any residual updates are performed to make sure that all signals move forward during the heartbeat.
- Reaction functions are executed, possibly updating attributes, creating or removing reactive objects from the world, and possibly adding new reactions.

As we stated above, our system ensures that the changes made by reaction functions are not observable until the next heartbeat. This ensures that all signals are *single valued* - their value cannot change during a heartbeat. This strategy need not be applied to signal level switching, although, most switching is done in reaction functions instead of the FRP `switch` function - this function is still part of the system.

One should note the following important properties of our system:

- Adding and removing reactive objects from the world does not cause problems since the effects are not seen until the next cycle.
- Adding new reaction functions to objects will likewise not cause a problem - these are also represented by sets.
- Observations are always consistent since all observation is done before any object is updated.

One other interesting issue is what happens when a reactive proxy is deleted. Suppose that `b.vx` was assigned a value in terms of some other object (say `y`) and `y` is later deleted. This would normally create a dangling reference - value in the object y are no longer part of the system. We address this by freezing the proxies of deleted objects. Although these no longer change, they can be observed by other reactive values without errors.

Unfortunately there is one aspect to reaction functions that can lead to evaluation order problems. If two different reaction functions update the same attribute of an object only one of these functions can succeed. Update collision is a fundamental problem in FRP - this is a consequence of having multiple event occurrences at the same logical time. While the semantics is well defined, this may not the intention of the programmer if each event is meant to have some sort of effect. Programmer's will have to create code in the reactors to check for and deal with collisions. At present, our system detects write collisions but does not halt the program when such collisions occur.

4.1 Optimizations

There are a number of important optimizations that are needed to ensure adequate performance of the system. When reaction functions initialize signals, these are compared with other signals initialized at the same heartbeat. This is reactive "common subexpression elimination" process that creates explicit sharing in the implementation so that the shared signal is evaluated only once. Furthermore, signals which contain no stateful operators can be shared across heartbeats.

Another important optimization is *chunking* - combining smaller lifted functions into larger ones. In Haskell, this can be expressed as

```
lift f . lift g = lift (f . g)
```

The `once` function creates an event stream that, having delivered one event, goes silent. This is a common idiom when creating reaction events - often only the first occurrence of the event is significant. During the action selection phase of the heartbeat any action connected to a silent event is discarded.

5. Examples

The best way to illustrate the power of our approach is to present examples of reactive programs. These examples show PyFRP in conjunction with the Panda3d game engine (we are currently working on a port to the Blender game engine) and used in the context of our summer camp since 2007. This game engine has been used by kids as young as 12 with great success. While some aspects of the program require assistance (for example, the use of lambda or using for loops), students are able to understand and modify these programs.

An advantage of our system is the expressiveness of the language, particularly evidenced in the artistic touches that students are able to add to these programs. Sound effects, 3-D models, lighting, particle effects, background images, and camera positioning can be used with great effect to individualize and decorate these programs. We believe that by embracing the artistic aspect of these programs we give students more motivation to explore programming and other concepts in our camp.

5.1 Fractals

This program is used to illustrate simple fractal designs. It is entirely event based, using the `delay` function and recursion to make these fractals grow when the program is executed. The program starts with a simple recursive function that creates a line of spheres that slowly shrink. Once this is working, we add branches to the tree of spheres. The branching is probabilistic, creating a variety of different images. We start with two constants: `step` defines the rate of shrinkage in the line of spheres. The value 0.03 results in a line of about 33 spheres. The `tstep` variable defines the rate at which new spheres are added.

```
step = 0.03
tstep = .1
```

This is the basic recursion of this program: given a position, size, and direction create a sphere with the given size and position and then, if growing up, consider branching in the xy plane as well as continuing up. If growing sideways, consider branching left, right, straight, and up. The probabilities of each are given in the call to branch:

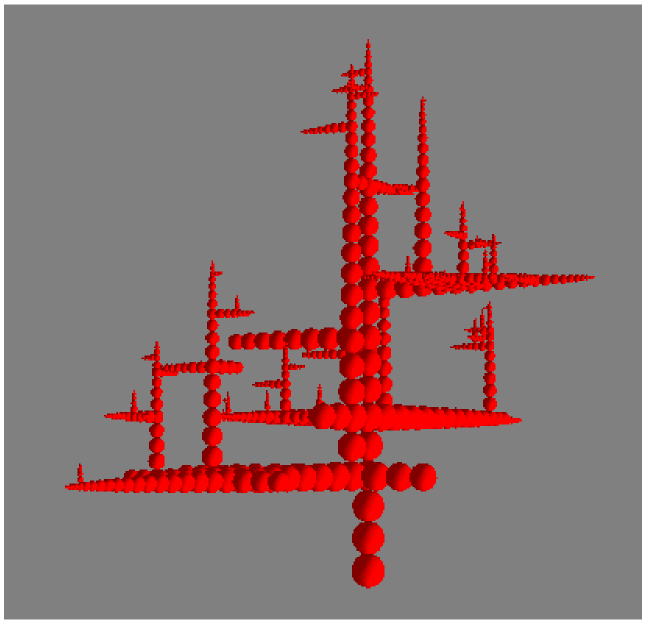
```
def expand(pos, size, dir):
    sphere(position = pos, size = size/2, color = red)
    size = size - step
    if size > 0:
        if getZ(dir) == 1: # Growing up
            branch(.04, pos, size, p3(1,0,0))
            branch(.04, pos, size, p3(0,1,0))
            branch(.04, pos, size, p3(-1,0,0))
            branch(.04, pos, size, p3(0,-1,0))
            branch(.99, pos, size, p3(0,0,1))
        else:
            branch(.95, pos, size, dir)
            branch(.05, pos, size, p3(0,0,1))
            branch(.05, pos, size,  # left
               p3(getY(dir), getX(dir), 0))
            branch(.05, pos, size,  #right
               p3(-getY(dir), -getX(dir), 0))

def branch(prob, pos, size, dir):
    if random01() < prob:
        delay(tstep, lambda: expand(pos + size*dir,
                                    size, dir))
```

Note the use of an ordinary Python if - this is not reactive so it can contain ordinary constructs such as for loops, if-then-else, and calls to the random number generator. These would not be available in behaviors.

Finally, this starts the initial tree:

```
expand(p3(0,0,0), 1, p3(0,0,1))
```

Here is an example of a fractal created by this program. Students are encouraged to work with the probabilities to create different images. Lighting has been added to this image: This program serves as a springboard for discussion about randomness, self-similarity, and other mathematical concepts.



5.2 Physics

This example demonstrates celestial motion based on Newton's law of gravitation. We start with representations of a solar system with a sun and two planets:

```
sun = sphere(size = .6, color = yellow)
planet1 = sphere(size = .1, color = blue)
planet2 = sphere(size = .15, color = green)
sun.mass = 50
planet1.mass = 2
planet2.mass = 3
g = .1   # Gravitational constant
```

Note the use of reactive objects. The sphere are given an attribute named "mass" to use in later equations.

Next, we need a function to set a body in motion. This function needs to know which body, the initial position, and a list of other bodies that interact with it:

```
def pullOn(body, p0, v0, others):
    force = p3(0,0,0)
    for b in others:
        d = abs(body.position - b.position)
        force = force +
                body.mass*b.mass*g/(d*d)*
              (b.position - body.position)
    body.acceleration = force/body.mass
    body.velocity = integral(body.acceleration, v0)
    body.position = integral(body.velocity, p0)
```

This code computes the force vector on a body based the gravitation equation. There is nothing special about any of these attributes except position - the acceleration and velocity attributes are specific to this application. Note the use of the integral function to move the simulated bodies. The reactive engine will update the different equations in a single step - the order of updates does not change the behavior of the program since each creates a new position based only on the positions at the previous time step. The mathematics here is a direct transcription of the underlying equations. Although the use of integral implies that students have been exposed to calculus, we find that understanding the idea of turning velocity into position comes naturally to our them.

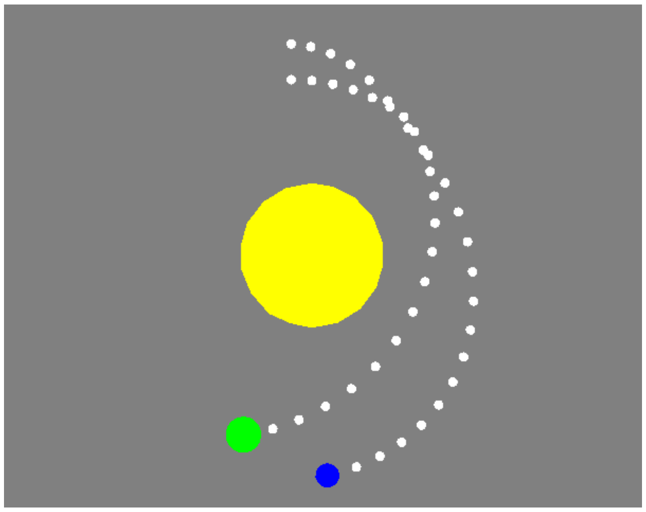
Next, we need to set the bodies in motion:

```
pullOn(sun, p3(0,0,0), p3(-.1,0,0),
       [planet1, planet2])
pullOn(planet1, p3(0,0,2.1), p3(1.7,0,-.1),
       [sun,planet2])
pullOn(planet2, p3(0, 0, 1.8), p3(1.8,0, 0),
       [sun,planet1])
```

To make the resulting motion visible, we leave a "trail" of spheres on the two planets:

```
def trail(m):
    react(clock(.1), lambda m1,v: sphere(
        color = white, size = .04,
        position = now(m.position), duration = 3))
trail(planet1)
trail(planet2)
```

The `trail` function creates a sphere every .1 seconds at the location of the object being trailed. The use of the `now` function freezes each trailing sphere in place. These spheres last for 3 seconds then disappear (the `duration` parameter. Below is an image of the execution of this program.



Students are not expected to write a program such as this from scratch, rather, most of this code is handed out and explained in lecture. Students then can experiment with the underlying constants to gain understanding of planetary motion. Interactive widgets such as sliders can make it possible to rapidly explore the underlying mathematics of motion.

5.3 Games

The following is an example game project completed by one of the camp participants. The basic learning goal here is the use of events to trigger reactions. This student was also interested in the design aspects of the game and added lighting and images to create an aesthetic and enjoyable game.

This dodge game consists of a character controlled by a player that must avoid objects flying across the screen. If the player's character is hit by an object the game ends and the amount of time elapsed becomes the score. This code also demonstrates a number of design elements such as lighting and texturing that are not part of the game play but add considerable interest to the project.

The program begins by creating the objects (models) and variables needed

```
rectangle(P3(-4,0,-3), P3(4,0,-3),
          P3(-4,0,3),  # Background
          texture = "Cloud.jpg")
text(time, size = 1.7, color = white)  # Score
al = ambientLight(color = color(.5,.5,.5)) # Lighting
dl = directionalLight(hpr = hpr(0,-1,0))
p = panda(hpr = hpr(pi*.5,0,0), size = .4,
      color = color(.5,.5,sin(time)))
```

`rectangle`, `text`, and `panda` create reactive proxy objects. `P3`, `hpr`, and `color` are constructors used by the game engine lifted into the reactive vocabulary. `hpr` is an orientation from a heading, pitch, and roll. Note that the color of the panda is time-varying.

Once the panda is created, it is moved the arrow keys in a manner similar to earlier examples:

```
v = hold(p3(0,0,0),
    key("upArrow", p3(0,0,1.5))+
    key("downArrow",p3(0,0,-1.5))+
    key("leftArrow", p3(-1.5,0,0))+
    key("rightArrow", p3(1.5,0,0))+
    happen(getX(p.position)<-3,  P3(1,0, 0)) +
    happen(getX(p.position)>3,   P3(-1, 0, 0)) +
    happen(getZ(p.position)>2.4, P3(0,0,-1)) +
    happen(getZ(p.position)<-2.4,P3(0,0,1)))
p.position = integral(v, p3(0,-.8,0))
```

The only new things here is the `happen` function - our name for the FRP `predicate` function - it signals an event when a boolean behavior becomes true.

Here are two reactors:

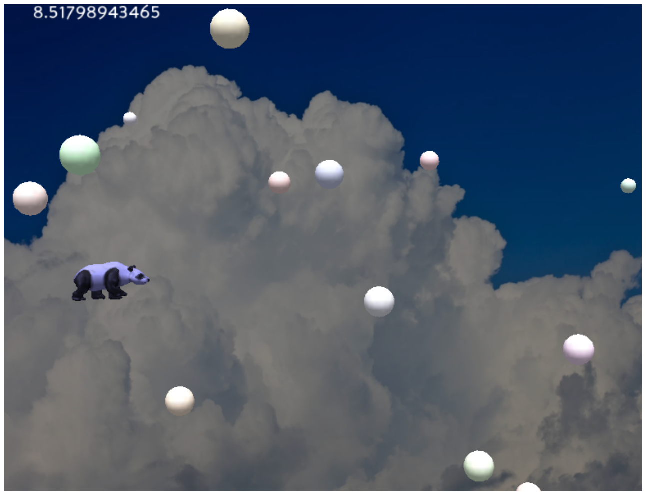
```
def endGame(ball, v):
    resetWorld(lambda:
       text(format("Fly free! Your score: %i seconds",
           now(time)), size = 3, position = p2(0,0),
           color = blue))

def randomBall(m, v):
    s = sphere(position = integral(p3(-1, 0, 0),
        p3(4, -.8, randomRange(-2.5, 2.5))),
        size = randomRange(.05,.2), duration = 8,
        color =
        color(randomRange(.8,1),randomRange(.8,1),
              randomRange(.8,1)))
    hit(s,p, endGame)
```

The `react` function triggers the `endGame` reactor when the sphere hits the panda. The `hit` event is computed in the game engine.

Finally, a recurring clock event goes off every 1/2 second to trigger the generation of a random ball in the `randomBall` reactor.

```
react(clock(0.5), randomBall)
```

A screen shot of this game is as follows:


8.51798943465

Games such as this have been used in a math education context with measurable effects on learning outcomes[14]. While our system is less structured, we believe that it can be used in a similar way to produce educational outcomes.

6. Related Work

Our system is a synthesis of ideas which have been percolating in the FRP world since its introduction. While much of this system has been anticipated in past efforts, we have managed to find a "sweet spot" in the FRP implementation space that is useful in a wide range of applications.

6.1 Comparison With Classic FRP

We have adopted a style similar to classic FRP and share the same signal-level semantics. Our reactive proxies lead to a slightly different programming style with respect to switching. Users of our system tend to place switches at the object level rather than the signal level, a more familiar programming style. The `snapshot` function is mostly replaced by the use of `now` in reaction functions. The use of observation semantics for object attributes makes it easier to mix signals that are started at different times. The treatment of reactive expressions is similar - signals are only initiated by switching. Unlike classic FRP, our system has undefined behavior when an attribute is updated simultanously by multiple reaction functions but, conversely, we seldom need the sort of event merging that can lead to unanticipated event loss.

6.2 Father Time

Much of this work was presaged in the Scheme community by Greg Cooper's Father Time system (frTime)[4]. In frTime, as in our work, FRP coexists with an underlying imperative language. FrTime provides a fixed set of reactive proxy objects as part of a small GUI kit but does not have a user-level object lifter. It also uses the original FRP style switching instead of reaction functions which makes some styles of programming more difficult.

FrTime implements all signals in an observational manner, starting them when first evaluated. This makes it harder to abstract over signal definitions. For example, in our system the definition

```
localTime = integral(1)
```

would create an uninitialized signal that is initialized in the referring context while in frTime, this integral would start at the time of definition.

The basic differences between our work and frTime are exemplified by the Tetris game example in their distribution. In it, reactivity is used to manipulate as single big state value which is then used to draw the game and is transformed by the game logic. In our system, the state can be broken down into small pieces (individual reactive blocks) which are programmed individually rather than as a group.

6.3 Modern FRP Implementations

The Haskell-based implementation of FRP has gone in a number of directions. Conal Elliott has continued to refine Fran style FRP, with the reactive library and a foray into tangible values[10], which addresses the connection between used-visible artifacts and reactive programming. These efforts have an admirable semantic purity and are deeply embedded in the Haskell type system. Bringing this style into an untyped language would be daunting at the very least. It also requires a level of sophistication that is probably beyond the users of our system.

Yampa[7] remains under active development. The use of arrow notation address many of the same issues that we have (dynamic contexts, observation, and initialization) in a pleasing way. However, we believe that this style doesn't move easily to languages without type systems or arrow notation and lacks the simplicity of our approach.

Finally, the *reactive banana* library[1] has become the library of choice for the casual Haskell developer. It has managed to preserve the basic feel of classic FRP while allowing reactive programming to exist in a dynamic context. This system uses explicit calls to `reactimate` to integrate the OO and reactive components of the program. Our work makes the boundaries between reactive code and the rest of the application more implicit and encourages a more functional connection between these worlds.

6.4 Other Reactive Systems

There are a number of other systems which simplify interactive programming. In Python, the Trellis system improves callback management and simplifies event-driven programming but does not have a high level object abstraction or deal with continuous behaviors. Yoopf is a simple reactive system but without stateful signals, making it similar to spreadsheets.

7. Conclusions

We have built a system which comfortably integrates Functional Reactive Programming into general interactive toolkits. Using reactive proxy objects, entities defined by the underlying system become first-class citizens of the reactive world, allowing their attributes to implicitly change and propagate over time. Our system also demonstrates that a declarative framework such as FRP can fit comfortably into conventional programming languages.

We have extended FRP in a number of ways: we are able to lift object classes into a reactive context and have simplified the use of the reactive language by introducing reactors and attribute observation. Furthermore, these reactors support dynamic reconfiguration of the application and support a programming style that is intuitive and well suited for novice programmers. This system has been implemented and shown to be practical in multiple contexts: game engine programming, GUI construction, and embedded control.

The source code, documentation, and camp material are available at our website, `www.reactive-engine.org`.

Acknowledgments

This work is supported by NSF Grant 1302327.

References

[1] The reactive-banana package. `https://hackage.haskell.org/package/reactive-banana`, 2013. Accessed: 2015-06-30.

[2] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. In *ACM Computing Surveys*. Citeseer, 2012.

[3] A. Cleary, L. Vandenbergh, and J. Peterson. Reactive game engine programming for stem outreach. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 628–632. ACM, 2015.

[4] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*, pages 294–308. Springer, 2006.

[5] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.

[6] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell workshop*, pages 41–69, 2001.

[7] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM, 2003.

[8] S. Crawford and E. Boese. Actionscript: a gentle introduction to programming. *Journal of Computing Sciences in Colleges*, 21(3):156–168, 2006.

[9] C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[10] C. M. Elliott. Tangible functional programming. In *ACM SIGPLAN Notices*, volume 42, pages 59–70. ACM, 2007.

[11] M. Goslin and M. R. Mine. The panda3d graphics engine. *Computer*, 37(10):112–114, 2004.

[12] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[13] J. Peterson, G. D. Hager, and P. Hudak. A language for declarative robotic programming. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1144–1151. IEEE, 1999.

[14] E. Schanzer, K. Fisler, and S. Krishnamurthi. Bootstrap: Going beyond programming in after-school computer science. In *SPLASH Education Symposium*, 2013.

[15] A. Voellmy, A. Agarwal, and P. Hudak. Nettle: Functional reactive programming for openflow networks. Technical report, DTIC Document, 2010.