How does FRP work?

By Stephen Blackheath and Anthony Jones, *Functional Reactive Programming*

This article introduces functional reactive programming and explains what problems it solves, why those problems need to be solved and how it solves them.

We're going to give you a conceptual overview of FRP with a game of lunar lander. In your lander, you control the rate of descent by intermittently turning a thruster on and off. The goal is to land by touching the ground at a low velocity before you run out of fuel. If the velocity is too high at impact, then you crash. See figure 1.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/blackheath/](http://www.manning.com/blackheath/)
Figure 1: Lunar lander: Using the thruster to control your descent, try to touch the ground at a low velocity.

We can represent the logic of lunar lander with a diagram. There's only one input:

- **thrust**—either on or off. We will keep things simple by sending the thrust once for each animation frame, so it doubles as an animation clock.

There are three outputs:

- **position**—this is just a height above ground. The lander can't move left or right. Position is used to draw the scene.
- **crashed**—this means you hit the ground too hard and you lost the game.
- **fuel**—the amount of fuel left in the tank.

To read the diagram in figure 2, start in the top left where we feed this frame's value of thrust into the logic.

The first thing we do (to the right) is reduce the fuel if the thruster is on, by accumulating a value of -1. We haven't specified the initial fuel tank level here.

To calculate the acceleration, we either accelerate upwards (+2) if the thruster is on and there is fuel left, or we let gravity take us downwards (-1). We then integrate accelerations (by accumulating them) to give velocity, and then once again to give position – the height above ground.

touched is true if position is less than or equal to 0, our ground level. When this happens, crashed is set if the impact velocity is too high, relative to some constant threshold.
We may or may not think of diagrams when we write FRP. Either way we are working at the same conceptual level as the diagram. At any place in the code you only see only a small part of the structure, but the lines and boxes in the diagrams are guaranteed to work as you would expect.

This diagram is a directed graph that represents a data flow from inputs to outputs. In fact, FRP is often described as a form of data flow programming. We drew the diagram this way because FRP code is written as program statements that directly mirror this structure.

The direction of the arrows gives the direction of data flow. If you reversed the arrows, then they would represent a “depends on” relationship. For instance, by removing extraneous detail and reversing the arrows we can find the dependencies for acceleration in figure 3.
Figure 3: Reverse the data flow arrows to get dependency relationships, done here for acceleration

Figure 4 shows how FRP code is executed. In most FRP systems this all happens at application runtime. There are two phases:

- **phase 1**—FRP code statements are converted into a directed graph in memory. This typically happens once at program initialization.
- **phase 2**—For the rest of the program execution we feed input values in, turn the crank handle, and the FRP engine produces outputs.

NOTE In practice the program spends most of its time executing the graph, but the graph can be easily updated when needed.

A major task of the FRP engine is to ensure that things are processed in the order specified by the dependencies in the graph. In spreadsheets this is referred to as natural order recalculation. It could be better described as the “correct order” to distinguish it from any other order, which could give the wrong result.

For source code, sample chapters, the Online Author Forum, and other resources, go to http:// http://www.manning.com/blackheath/
Figure 4: How FRP code is executed

To give you a flavor of FRP code, we've compared object-oriented programming (OOP) vs. FRP implementations of the lunar lander logic in figure 5. Note the FRP code is really giving you something equivalent to a listener interface for the outputs, but the OOP for that would be too long for our diagram; the FRP code is using Java 8 lambda syntax.

```java
private double fuel = 100;
private double accel = 0;
private double vel = 0;
private double pos = 100;
private boolean crashed = false;
private final double threshold = ...;

public void update(boolean thrust) {
    fuel += thrust ? -1 : 0;
    accel += thrust && fuel >= 0 ? 2 : -1;
    vel += accel;
    pos += vel;
    boolean touched = pos <= 0;
    crashed = touched &&
            Math.abs(vel) >= threshold;
}

public double getFuel() {
    return fuel;
}

public boolean getCrashed() {
    return crashed;
}

public double getPosition() {
    return pos;
}

public final Cell<Double> fuel;
public final Stream<Unit> sCrashed;
public final Cell<Double> pos;
private final double threshold = ...;

public LunarLander(Stream<Boolean> sThrust) {
    fuel = sThrust.accumulate(0, (t, f) -> (t ? -1 : 0) + f);
    Cell<Double> accel = sThrust
            .snapshot(fuel, (t, f) -> t && f >= 0 ? 2 : -1)
            .accumulate(0, (da, a) -> da + a);
    Cell<Double> vel = accel.accumulate()
            .accumulate(0, (a, v) -> a + v);
    pos = vel.accumulate().accumulate(100, (v, p) -> v + p);
    Stream<Unit> sTouched = Stream.filterOptional(
            sThrust.snapshot(pos, (t, p) -> pos <= 0
                    ? Optional.of(Unit.UNIT)
                    : Optional.empty()));
    sCrashed = Stream.filterOptional(
            sTouched.snapshot(vel,
                    (t, v) -> Math.abs(v) >= threshold
                    ? Optional.of(Unit.UNIT)
                    : Optional.empty()));
}
```

Figure 5: Lunar lander: OOP vs. FRP code comparison

For source code, sample chapters, the Online Author Forum, and other resources, go to http://http://www.manning.com/blackheath/