Design issues for a DSL for particle systems

Marcus Lindblom - 780824-7196
d98macke@dtek.chalmers.se

June 2, 2003

1 Introduction

Particle systems have many properties that makes designing a "good" language for describing such systems an interesting and challenging task. This article aims to explain these properties, their implications and give solutions to some of the problems that arise.

The background is a thesis work on an embedded language and compiler written in Haskell, and some research done afterwards. The thesis report focuses heavily on the problems that arise by using an embedded language. This article focuses more on the fundamental demands for any language (and associated compiler) used for particle systems.

This article will use a mockup-language for describing our particle systems, that is used to show what we would like the syntax to be. Hopefully, it will serve as as a foundation for designing a embedded language that still has much of Haskell's "cuteness" but is better suited to the task.

2 Particle physics

The position of a simple particle depends only on time, and can often be written as a set of integrals. I.e the position of a particle affected only by gravity (and some initial position and velocity) would be defined by the mathematical expression in equation 1. Such an expression is trivially expressed in any language and can be evaluated both directly (after some simple algebraic transformations) and iteratively (using some numerical integration method).

$$p(t) = p_0 + \int_0^t v_0 + \left(\int_0^t mg \, dt \right) dt \tag{1}$$

If we make the relationship between the position and its derivates a more complex (such as the addition of air resistance), we get a differential equation, such as that in equation 2. If the equation is linear (as this one is) it is possible to solve it directly. Non-linear differential equation is a different matter (often needing numerical solvers), but as long as we can express our behaviour in the language while knowing that it can be compiled and solved, we are content.

$$\frac{d^2p}{dp^2} = mg + a\frac{dp}{dt}(\text{with } p_0' = \ldots, p_0 = \ldots) \tag{2}$$

The straightforward way of describing this in a language is as a set of mutually recursive expressions, as that of figure **??**. Evaluating such code in a language with referential transparency (such as Haskell) yields an infinite expression. This problem crops up in many different problem domains and has been solved previously (See Koen Claessen's *Observable Sharing*).

```
particle (p0,v0,m,a) {
  p = p0 + integrate v;
  v = v0 + integrate a;
  a = m * g - a * v;

  point p white 2; -- pos, color, size
}
```

3 Side-effects

So far, we have been able to express our particle systems by mathematics, then transforming them into computer code. If we want to model collisions and similar events, the behaviour gets more complex and it is hard to use mathematics to describe our intended behaviour. Taking the example from above, say that we want the same behaviour of our particles, but with the addition that they will bounce of the floor (for simplicity, we will choose the xz-plane). What we would want is just to add a test and update the velocity, as seen in below. (Here we mix both let-declarations and operations with side-effects. The intention is that we want to reuse the simulation (variables p,v,a) and at the same time be able to update its state.)

```
particle (p0,v0,m,a,c) {
  p = p0 + integrate v;
  v = v0 + integrate a;
  a = m * g - a * v;

  if(p.y == 0) {
    v.y := -v.y * c;
  }

  point p white 2;
}
```

This is what we want to write, but determining *when* the boolean expresson $p_y < 0$ is true is not trivial. It could be possible to calulcate it directly, but probably rather difficult given that it contains a differential equation. We could switch to numerical integration and test at each iteration, or do something more advanced such as the interval analysis used in Fran.

Assuming that the exact times of each collision can be calculated, we should be able to compile the code to a behaviour that changes at each collision time. Unless there are only a few collisions during the lifetime of the particle, this approach needs lazy evaluation in order to keep the space requirements down (which is important, as particle systems usually contain hundreds or thousands of particles).

If we cannot know the exact time, because of issues as randomization (which we will cover later), collision with another moving object (over which we have no control) or simulation inaccuracies, we must check for collision each iteration. This means that we cannot test for equality (it is highly unlikely that we end up exactly *on* the plane). We will defer this to the user (or provide functions to do so in a library), because it is a complex task and we do not know how accurate the test should be.

The test and response can be done in a few different ways, depending on the desired accuracy. The simplest approach is to test if the point has passed the plane, and then mirror the velocity and set the position to be on the plane itself. The most accurate is to test for intersection between the particle's path (assuming either constant acceleration) and the plane during the simulation step. This test will, in case of a collision, also produce the time. Then, we do a partial simulation step until the time of collision, adjust the velocity and simulates the rest of the step. In order to perform these *partial simulations*, we use a primitive called **simulate**, which takes the destination time as an argument and updates all expressions. An example on this is given below.

```
particle (p0,v0,m,a,c) {
  p = p0 + integrate v;
  v = v0 + integrate a;
  a = m * g - a * v;

  tc = collisionTimeWithXZ p v a; -- returns -1 if no collision

  if(time < tc < time+dt) { -- time and dt are defined internally
    simulate tc;
    v.y := -v.y * c;
    simulate (time+dt);
  }

  return p;
}
```

The compiler would have to make sure that if-branches without simulate-calls still simulate properly. Having the user insert these manually (i.e. a `simulate (time+dt)` in the else-branch) is both error-prone and annoying.

4 Nondeterminism

In order to approximate natural phenomena such as smoke, fire, whirlwinds, etc. that have really complex behaviour, we will simply apply randomization to our behaviours. This is a very common technique, but care must be taken when using such expressions in languages with referential transparency, as we often want the random value bound to a variable and used in many places.

There is also an inherent ambiguity in the use random expressions during simulations. Should the value be calculated once or at each update (if we use an iterative simulation) ? An example would be a random initial velocity but continously randomized acceleration. The easiest way to solve this is to define some variables as constant (similar to C), which means that their value are calculated only once (see below). One could also use a *snapshot* construct (the approach taken in the thesis) which takes an expression, evaluates it and returns its value at all subsequent evaluations. This has some problems with parameter passing as we shall see in the next section.

```
particle (p0,v0,m) {
  const v0' = rand + v0;

  p = p0 + integrate v;
  v = v0' + integrate a;
  a = m*g + rand;

  point p white 2;
}
```

5 Several particles

So far, we have only concerned ourselves with singular particles. When creating several particles, one usually wants to pass values to the emitted particles, to affect their behaviur and reuse common particle types.

```
emitring () {
  p = (sin time,0,cos time);
  emit 200 (particle p (0,0,0) 2);
}
```

Here, one must (again) be careful with referential transparency, as we have chosen to have the variable *time* corresponding to each object current age. If the expression assigned to p is simply substituted into the particle's simulation, it will not have the same meaning. In the thesis, objects where represented as functions from time to circumvent this problem. Applying that technique to the given example would mean that each particle would simulate the rotation of the emitter, unless care was taken to move that calculation. Combined with snapshots (explained in the previous section), and especially snapshots of snapshots, it became very hard to keep track of what goes where.

This can be solved by not having referential transparency between objects *and* using different syntax for pass-by-value and pass-by-reference. The most flexibility and reuse would be gained if this was defined by the caller, rather than the callee (as it would be in the case of using snapshot or const variables).

6 Summary

All in all, there are a number of different features we want in our particle language:

- Mutually recursive expressions in order to express differential equations easily.
- Side effects to test and update state in the event of collisions.
- Partial simulation to conduct proper collision handling
- Controlled non-determinism to make randomization behave as desired.
- Observable scope to ensure calculation happens where it is intended.

Most of this can probably be expressed in a Haskell-embedded language if monads where used instead of common expressions, because the latter gives us too little information on the computations. Monads where not used in the previous language, since we wanted recursive lets, but since GHC now supports *mdo*, it might be possible to do properly.