Proposed Event System for the Force-Based Physics Simulator

Zachary Kieda (zkieda@andrew.cmu.edu)

November 9, 2015

1 Introduction – Event System

One of the cornerstone aspects of our physics simulator is the fact that it is event based. It follows that we will want to have the event system extensible to allow for modifications or extensions to take place.

One might consider adding events into the `WorldObject` class to allow an object-oriented method to take place. Using this method, we have two options – having each object case out the different types of objects on the board (not very extensible). Alternatively, we could attempt to expose an interface for event types possible on the objects themselves. However, this becomes very complicated just from the case of collisions. Suppose we defined an arbitrary object as generic constraints, where the constraints of this rigid body is *self.constraints*. We can define the transformation of another body over time relative to this body as a transformation function $f : t \to T$ where T is a translation matrix. A collision between this object and another is satisfied when we solve for some future time t' in the problem

$$\mathit{self.constraints} \cap f(t') \cdot \mathit{other.constraints} \neq \emptyset$$

Obviously, this is not simple to solve even if we knew the integral for f, just because the constraints are too generic. (Actually, if there is some way to determine this exactly I would really like to know)

Now that we have eliminated some design patterns that could give us a bad time, let's look at a proposed design pattern that could provide useful.

2 Proposed Event System

Define Γ as a context, or as something that can plausibly produce a single future event given a subset of the world (the context). For example, a context may be a dynamic particle and a face. The context *may* produce an event, or it may miss. However, we can give a time for which it could hit if the plane spanned infinitely. So, we can define $\Gamma = (\mathit{getTime} : () \to \mathbb{R}, \mathit{getEvent} : () \to \mathit{Event})$. A client that is generating contexts should generate events lazily, in case if they are thrown out.

Now, we define σ as a context generator such that $\sigma(t) = \Gamma_t \mathit{iterator}$. A client can implement generators that find events for a particular type of event. An example generator $\sigma_{pointMeshCollision}$ generates a context that accepts a point and a mesh and reports a possible collision between them. Again, the iterator allows each Γ to be generated lazily.

For this simulator, we define a function $\mathit{makeIntegrator} : \texttt{Integrator} \to \sigma_{base} \to \sigma'$. The integrator is a common implementation that the generator will use to calculate events. Of course, we will just provide an abstract class for σ and require the client to define an integrator in the parent's constructor.

Finally, we propose a meta-generator as part of the base library which will allow clients to have multiple generators to detect many different event types. This is defined by the signature $\mathit{agg} : \sigma\,\mathit{list} \to \sigma_{agg}$.

Now, we have the full client side setup. The client only needs to define the interface for σ, but it's recommended that the client uses the library σ_{agg} and the subclasses that require an integrator.

3 Use/Dynamic Semantics

We can find the next event using this system by the following function,

$$\mathit{nextEvent}(t) = [\Gamma_t \in \mathsf{sort}(\sigma(t), \Gamma_t \mapsto \Gamma_t.\mathit{expectedTime}()) \mid \Gamma_t.\mathit{getEvent()}\ \texttt{!= None}].\mathit{first()}$$

In the above algorithm, we sort each context by the time that it could occur, then retrieve the first context's event that is non-None. After, we process all of the other objects in the scene to bring them up to date with the time that this event occurred. We may skip this step if our $\Delta t = 0$.