Simulator Architecture

Started July 28, 2024
last update August 12, 2024

Contents

Chapter 1

World Evolution

The simulation is done by calculating and storing the state of the world at consecutive instances of time denoted by the sequence t:

$$t = (t_k)_{k=0}^{n_t-1} \tag{1.1}$$

where n_t denotes the number of time instances such that

$$T = t_{n_t} + \Delta t_{n_t} - t_0 \tag{1.2}$$

is the total duration of the simulated world. The sequence t is most likely not uniform in time. At each instance of time t_i, the time increment Δt_i is calculated according to the state of the world at t_i. Clearly,

$$T = \sum_{i=0}^{n_t-1} \Delta t_i. \tag{1.3}$$

The state of the world at t_i is denoted as s_i. The initial state of the world s_0 is a given. Although the nature of the entity s_i is irrelevant to the current discussion and is not mathematically defined here, it must satisfy the following rule.

Rule 1: *Reproducibility* Consider the sequence of states $s_0, s_1, \ldots, s_q$ corresponding to time instances $t_0, t_1, \ldots, t_q$. For any $m < q, m \in \mathbb{N}$, if the initial state is set to s_m, the exact sequence of states must be produced to reach s_q. That is, for a new simulation with states denoted as s'_i, we have $s'_0 = s_m$ and

$$(s'_k)_{k=0}^{q-m} = (s_k)_{k=m}^{q}. \tag{1.4}$$

1.1 examples

1.1.1 ball-throwing robot

1. we're one cycle before the robot reaches 10m distance (when it will throw the ball)
2. freespace motion, moves the robot forward, evolution cycle: from previous cycle intersection pairs there is no collision, object.evolve() does freespace motion, state is updated. Lidar (which is a child) creates a beam, which is the offspring of the lidar. this creation is the result of lidar.evolve(). The world registers the beam. then intersections are calculated.

3. next cycle: from previous cycle intersection pairs, the robot knows its distance. (assuming a lidar, by the end of the previous cycle, the distance is known). Robot does what it has to do in its evolve().

1.2 Evolution cycles

Note that offspring objects and dead objects get handled in the same cycle. So the object gets the intersection information for the new ones in the same cycle.

The `World` does the following loop

1. Collect `Object.get_delta_t()` of all objects and take the minimum.
2. Call `Object.evolve(delta_t)` of all objects and collect the offspring objects and add them to the list.
3. delete the dead objects from the list. How dead objects know that it's time to be dead?
4. Calculate intersections of every pair. Relevant instances of `Intersection` is given to every `Object` at this step TODO[1]. From the `World`'s point of view, there are two possible outcomes:
   (a) A Non-Infinitestimal Intersection (NII) has happened. The `World` handles this as follows.
      i. discard this cycle by calling `Object.StateManager.deleteState(0)` for every `Object`. Does this provide enough flexibility for optimizations?
      ii. rewind the simulation timer.
   (b) No NII has happened. That is, there might be occurrences of infinitesimal intersections, i.e. collision, or no intersection among objects. The `Object` handles these cases. See Section 5. Even in the case of no intersection, the objects could still use a measure of distance to update their delta-t.
   In this step, There is a chance that we discard the whole cycle, Deleting dead objects before this step is not very logical, Because the fact that they are dead depends on some conditions, e.g. intersection.
   take this to state managment chapter

1.3 Calculation of Δt_i

This is about `Object.getDeltaT()`

- what inputs does it need from the world?

[1]in all cases you can pass the intersection object. But that object should not be kept in memory. Object must be careful in handling that. how to enforce?

Chapter 2

Objects

2.1 Object Types

In this categorization, we should avoid the physics viewpoint but rather the architectural viewpoint.

- Tangible objects: which cannot have NII. This objects are likely to be the ones responding to collision, having mass, etc.
- Intangible objects: which can have NII. Examples are a beam created to measure distance and fields such as gravity.

Recall that detection of an NII in the evolution cycle of the `World` triggers a series of action if the object is tangible.

2.2 Parent and Children Objects

A number of objects (of any type) which do not evolve independently of each other. The dependence might be established via

- Natural interconnected-objects: laws of physics and the constraints as discussed in Section **??**.
- Artificial inter-connected objects: exchange of information and an algorithm not necessarily following the laws of physics.

An object which consists of a number of inter-connected objects is called "Parent Object" and the constituting objects are called "child(ren) object(s)". Here are the architectural considerations:

- The children objects evolve only when called by the parent object.
- An `Object` can be a parent, a child or both. This implies that a child object can itself be the parent of a group of inter-connected objects and so on. There is no limitation on the depth of this hierarchy.
- An object can have only one parent.

- An `Object` interacts only with its immediate parent and children.
- Being a child or parent object is captured in `Object`. That is, upon construction of any `Object`, it can be given a parent object.
- The `World` handles all objects equally in its evolution cycle.
- A parent object of a group of natural inter-connected objects is an instance of `NaturalParentObject` which handles the interconnections based on the laws of physics by merely calling their `Object.evolve()`. Constraints (connections, hinges, etc) are handled elsewhere.

Chapter 3

Constraints

Constraints are instances of `Constraint` inherited from `Object`. How does it work?

Chapter 4

State management

We need to be able to have the notion of state at two levels: an object's state and the state of the world. The state of the world at time t is simply the collection of the states of all objects. Since we desire to leave room for optimization, it seems better not to mandate that all objects evolve at the same pace and have states recorded at the same set of time instances.

Optimizations might not be compatible with the reproducibility rule (Rule 1). Recall that the world evolves on a sequence of time instances with adaptively adjusted time step. Once a particular sequence is exercised, the particular time instances become the basis for Rule 1. One evident example of such incompatibility is when the initial state s_n of the world at time t_n does not exist in states sequence of a particular object o_i employing optimizations. The mentioned object has to assume a state obtained from linear interpolation. For the sake of simplicity assume that s_{n+1} at t_{n+1} does exist in the states sequence of o_i. Then it is not clear that the evolution of o_i from s_n to s_{n+1} is the same as evolution from s_{n-1} to s_{n+1}.

4.1 proposal 1

Each `Object` has its `StateManager`. When calling its `evolve()`, a new state is added. If required, the last state can be deleted.

- `State& StateManager.newState()` The owning object should call this in its `evolve()`. Whether it is later discarded or multiple of states are discarded is not a concern in `StateManager`.
- `void StateManager.deleteState(int stateCountFromEnd)` where the parameter is a non-positive value with 0 referring to the last state, -1 referring to the one before last and so on.
- `State& StateManager.getState(double t)` returns the state at time instance `t`. If not available in the simulated states, it is linearly interpolated between the immediately preceding and proceeding states.

4.2 proposal 2

There is a central `StateManager`? Saeed: just crossed my mind but I don't see much value in it. I'm keeping it, just in case.

Chapter 5

Intersection

Each pair of `Objects` get an `Intersection` object which handles their intersection. This implies is $\binom{N}{2} = \frac{n(n-1)}{2}$ instances which has the quadratic (polynomial) $\mathcal{O}(N^2)$ order. That is, no problem with the order in the non-optimized way.

These objects can be constructed for every new pair and kept in memory not to add a significant runtime overhead of constructing and destructing objects. Either by such optimizations or not, the `vector` of `Intersection` instances is managed in `World.handleIntersectionPairs()` and stored in `intersectionPairs`. In the evolution cycle, all elements of the vector are looped over.

The idea is that the intersection instance provides all the necessary information to the objects pair. Here is the interface:

- `Intersection.process(int &status)` which does the calculations and update the internal variables. The immediate return value is the status which could be
  - NON_INFINITESIMAL_INTERSECTION,
  - COLLISION_POINTS,
  - COLLISION_SURFACES,
  - NO_INTERSECTION.

  This function is meant to be called by `World`.

- `Intersection.getContactPoints(std::vector<Point> contactPoints)`: A collision might cause several contact points depending on the geometries. For regular convex shapes, this will not happen so no need to worry now. This function is meant to be called by the objects. The `World` handles the situation only when an NII happens.

- `Intersection.getContactSurfaces(std::vector<Surface> contactSurfaces)` A collision might cause a surface contact. Although for ideally rigid objects this is a possible but zero-probably event, in our simulation we might have to have the option. not now anyway. Similar to `getContactPoints`, this function is meant to be called by the objects. The `World` handles the situation only when an NII happens.

- `Intersection.getDistance(double& distance)`: when no intersection has happened, provides a measure. what measure?!

The threshold and the metric for determining an NII is the job of `Intersection`