Ian Miller - 6.905 19 May 2021

Orbis: A Modular Physics Simulation Engine

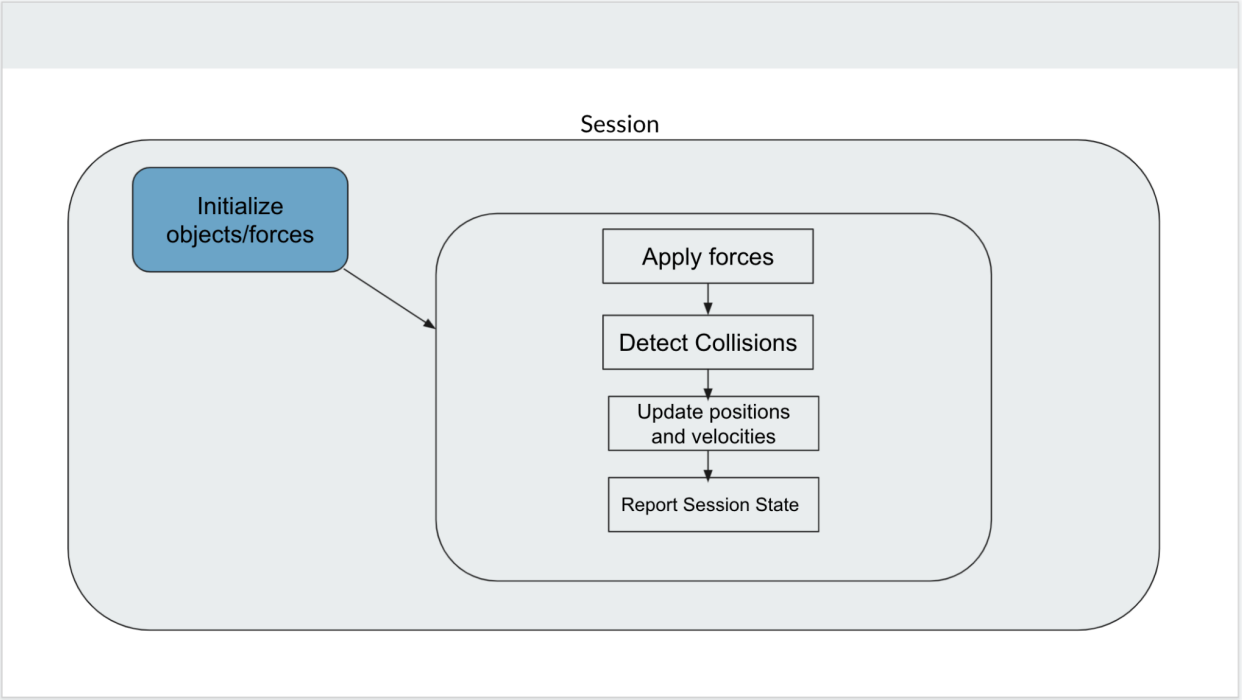
This project attempts to build the foundation of a physics simulation engine. Physics simulation can be properly described as a field within computer science. There is much progress still to be made in that field. But the potential benefits of physics simulation are quite literally endless. Already, modeling the real world on the computer is an integral part of practically every engineering endeavour. My goal is to work out the fundamental skeleton that can be effectively applied to point masses and systems of point masses in order to model motion. This skeleton should be built in such a way that is easily extendable to more complex systems in the future. Because of time and other practical constraints, I limited myself to focusing on rigid bodies. Indeed, the majority of production-grade physics simulation software is focused on rigid body dynamics because it is far simpler and easier to build than modeling software for deformable bodies or fluids, for example.

Why is physical simulation important?

Many-body problems are common in countless scientific fields. In almost every case, it's much cheaper to simulate something than it is to procure and organize the hardware and talent necessary to run an actual experiment. Physical simulation can also help you answer questions probabilistically. Some problems are particularly well-suited to stochastic simulation, like reaction run-times, estimating different properties about biological pathways, and weather modeling to name a few. Mathematically speaking, small, local errors can butterfly into really large errors that might render your model useless or actively harmful. It's also probably impossible to maintain 100% fidelity when discretizing the world, at least according to classical theories. From a software development standpoint, generic procedures can easily become messy and complicated and it is very difficult to "avoid programming yourself into a corner". Indeed, this was my primary focus. I set out to build an easily extendable skeleton for physical simulation.

Building Blocks

The high-level, general procedure Orbis is illustrated by the following diagram:



For the purposes of this project, we will make several simplifying assumptions in regard to the mutability of object properties. It's important to note that while such assumptions do limit the fidelity of the modeled object, they aren't terminal constraints. In other words, these assumptions can easily be removed and extra properties can be added to objects in order to improve model fidelity. Here are the assumptions made for Orbis v0:

1. All objects are singular particles
2. Each object has a fixed mass, initial position, initial velocity, and initial acceleration
3. All collisions between objects are perfectly elastic (we won't account for friction at all in this first iteration of Orbis)
4. Simulation will take place with a fixed timestep (that timestep will default to 1 second unless otherwise specified by user)

The primary task of this project was to build the basic object, session, and time-flow procedures of an extensible physical simulation system. This was the meat of the project.

These procedures use some of the backbone from user-defined-types from *Software Design for Flexibility*, although many new thought patterns were required.

Sessions

The orbis-engine library allows you to start one or more orbis-sessions. A user might want to run several different sessions at once to compare behavior in different environments. A session is an instance of simulation that is registered with the clock upon instantiation. Each session progresses in time independently of other sessions. A particular session has its own set of objects and forces. These things are added to the session by calling `add-thing!` or `add-applied-force!` and removed by calling the corresponding procedure. It also has a user-defined size and currently operates in 2D. For demonstration purposes, you can see the state of objects and forces in a session by making a call to `report-world-state!` although a nice extension would be to build a UI that displays objects in a session and updates in real-time as the user steps through time in a session.

Objects

Objects are the primary subjects in orbis. Each object has its own set of physical properties: position, velocity, acceleration, and mass. In an effort to maintain somewhat of a narrow scope, for this first version of Orbis, I'm only working with point masses. Each object also has a parent session. Because the session is responsible for adding and removing objects and forces, parent session is a necessary inclusion. It's also necessary (from the object's perspective) to know what other objects are in a particular object's session. This list of objects can be retrieved by calling

the session property-getter `get-things`. While some forces act indiscriminately on all objects in a session, forces like friction and normal force require knowledge of collisions. When deciding if a particular object has collided with any other object, I iterate through that list of other objects and compare positions. At instantiation, each object is registered with the clock. Position, velocity, and acceleration are stored as numeric-vectors, utilizing the vector-arithmetic code that probably wasn't the best choice for extendability. It probably would have been more appropriate to use the vector-extender in a generic arithmetic. That way, I could extend my arithmetic over other types, not just vectors. As it currently stands, I can't do any operations on functions of vectors, vectors of functions, or vectors of vectors of functions.

Applied Forces

The other class of objects in a session is applied-forces. At instantiation, an applied-force is added to a session and registered with the clock. An applied-force has a magnitude and direction (vectorized in newtons), a parent session, and a set of concerned-objects. This property represents the set of objects that the force itself acts on. For some forces, like gravity, concerned-objects includes the entire set of objects in a session. For contact forces, like friction and normal force, concerned-objects likely represents a subset of the entire set of objects in a session. Forces are applied to an object with a call to `apply-force!` This procedure calculates a resultant acceleration (with a call to `get-resultant-acceleration`) for each object in that particular force's set of concerned-objects and then updates each object's acceleration accordingly.

Timesteps

A single timestep involves applying forces, updating positions and velocities, detecting collisions, and then reporting or displaying the result. Timesteps are somewhat more complicated than they first appeared to be, at least in my experience. The size of a timestep determines the bounds of the integral that determines change in position, velocity, and acceleration. I have implemented a fixed timestep, but in order for this system to limit numerical integration error, timesteps must be dynamic depending on the state of the session. In the next page or so, I will detail what exactly happens in a single timestep.

Applying Forces

As mentioned above, each force will `apply-force!` to each of its concerned-objects. The result of this call is that each of that force's concerned-objects will potentially get a new acceleration via a call to `set-acceleration!` That object's new acceleration is found by summing (force / mass) and the object's current acceleration.

Detect Collisions

Two sources of error exist with numerical integration: roundoff error and truncation error. I attempted to address both of these sources by building in a buffer for collision detection. The first half of `check-for-collisions-and-move!` takes a single object and iterates through every object in a session and checks if the object of interest occupies a position within 1.41 units of another object. If it does, a collision is flagged and the two incident objects' velocities are

updated with a call to `set-collision-velocities!` I made 3 fundamental assumptions for handling collisions in this iteration of Orbis that will require work to make the system operable without:

1. A collision involves 2 objects
2. Every object has an equal mass
3. Every collision is perfectly elastic

Therefore, in a collision between thing1 and thing2, thing1 will have its velocity updated to thing2's initial velocity and vice versa.

Updating Positions and Velocities

The path that the computer generates for a particular free particle is composed of a bunch of discrete points. In the real world, no such list of discrete points exist. According to classical mechanics theories, that path is effectively continuous in the real world. In a program, you are essentially forcing a point onto a path that might not actually exist. Nevertheless, roundoff error is impossible to avoid completely. After velocities are updated to reflect any potential collisions, a call to `set-position!` will add the current velocity to that object's position. A call to `set-velocity!` will add the current acceleration to that object's velocity. This integration works because timesteps are fixed across a session.

Report Session State

Session state reporting is important for the user to be able to view what is happening in the simulation. Ultimately, this is the product that the simulation exists to produce. Right now, session state just lists all things and forces in a session and the properties of each. In future work,

it would be very cool to build a GUI that displays things in a session and updates in real-time. Reporting session state is the final part of a single timestep.

Next Steps

Because trajectories were strictly linear in this iteration or Orbis, it was considerably easier to calculate each point on the path. I originally imagined a final project that had quite a different goal in mind. Adding more real-life features like gravity and friction quickly complicate the process by which the next point of that object's trajectory is calculated. A solution to this issue of imperfect trajectory prediction can be approximated by dividing the world into classes of objects, such that a suitable Lagrangian might effectively apply to an entire class of real objects. That is what I initially wanted to do for my final project. But I think it's probably more appropriate as a next step for Orbis. In future iterations, I would build out the necessary mathematical procedures that support a Lagrangian view of motion.