FOSSSim Online

Writeup and documentation

by Cameron Erdogan

Supervised by Eitan Grinspun

May 15, 2014

INTRODUCTION

FOSSSim (Four Oh Six Seven Simulation) Online is an attempt to port the FOSSSim simulator, used by Professor Grinspun's COMS 4167 Physically-Based Computer Animation class, to javascript.

Motivation

The simulator developed in the class is a powerful learning tool, but the command line interface is clunky and is temperamental depending on your machine's environment. A web-build on top of a javascript implementation of the simulator would allow for platform-independence, rapid stub-writing, and immediate visualization. An application allowing for this would be a valuable tool for the computer scientist looking to delve into the coding behind computer animations.

Academic and Preprofessional Goals

Building out this application allows for engineering experience using both web technology and physical simulation theory. As a soon-to-be graduate looking for deeper experience in both of these things, I took on this challenge. Doing so would allow me to demonstrate a deeper understanding of the physical systems that drive computer animations as well as the front end web technologies that allow users to experience them.

OVERVIEW

Components of the Application

The application has three main components:

1. A coding window that dynamically incorporates the code you've written into the simulator (basically a poor man's JSFiddle).

2. An animation window, displaying the results of the animation as it is stepped through.

3. An interface for defining the initial state of a scene. A scene keeps track of particle positions, velocities, masses, radii, and the forces that act on them.

This being said, the intended function of the application isn't really to use all three of these at once. To me at least, it makes the most sense for a user to either:

A. Work on a code stub for a pre-defined scene. The user can try out different code and immediately see whether it had the intended effect. Or,

B. Build a scene using an already functional simulation. The user has an animation sandbox to play in that doesn't require them knowing what XML is or what it stands for.

Of course, nothing is stopping anyone from having all three up at once (in the demo, in fact, it's this way) I just can't think of a particularly smoother user experience that would go along with it.

Running the Application

If you're simply trying to try out the application, you need not go further than this section. There is a tabbed interface here, one tab for inputting a scene, one for inputting code, and one to display the animation window. Scenes are input by adding particles and forces (accomplished by pressing the relevant buttons) and filling in the form values as they appear. There isn't much input handling here so if you leave blanks horrible, unknown things will probably happen.

For instance, if you press "add particle", and fill into the blanks -4 for px, 0 for py, 1 for vx, 1 for vy, 1 for m, and 1 for r, a particle will be created at position (-4, 0) with initial velocity (1, 1) (which is, up and to the right), with mass 1 and radius 1. You can run the scene with no forces to just see particles move with unimpeded velocity. If you then add a force, accomplished by selecting a force type from the drop down and filling in the relevant blanks, that force will be applied to the relevant particles. For instance, if you create another particle (use the same info as above, except with position (4, 0)), select "spring force" from the drop down, click "add force", and fill in the relevant values for spring force interaction, the two particles will animate according to a spring force acting on the two particles. An example input for spring force input values are 0 and 1 for index i and j* (assuming we have the particles we defined before), 1 for k, 4 for L0, and 0 for b. What these values mean is explained later, but to see en example of what the program can do, input those values, press “change code”, go to the “display” tab, and press play to see FOSSSim in action.

* NOTE: for forces that act on two particles (namely spring and gravitational force), the i index and j index attributes are the the indices of the two particles the force is applied to. The index of a particle is denoted by the order in which it appears in the “Particle:” section of that form. For instance, if you create two particles, the first has index 0 and the second has index 1.

The Code

The code, fittingly, is essentially divided into a few fairly distinct chunks:

1. The FOSSSim source. This is the meat of the physical simulation, and, predictably, it’s heavily derived from the C++ FOSSSim code. What’s built here is relatively bare compared to the whole C++ FOSSSim. If you’ve taken COMS 4167, the code implemented here only implements forces and integrators up to Theme 1 Milestone 2 (For the unfamiliar, I’ll go over specifically what that means in the next section). However, building the framework and learning how to hack together pseudo-object-oriented javascript was no small task on my end. There was also the matter of rendering the particles, which is handled pretty concisely through the Three.js library. This code is contained in the “src” folder in the project directory.

2. An iframe, which is where the actual animation is run. I chose to run the animation in a canvas element inside of a an iframe instead of of one directly on the page for the simple reason that it allows you to reload the animation without reloading the rest of the page. The code for this is in a file called “fiddle.txt”. That name and the fact that it’s a txt file will make sense very shortly. Basically though, it’s just a webpage that has an html5 canvas element, which is used by Three.js to display WebGL graphics and animations. The actual iframe is an element inside of fiddler.html.

3. The Imitation-JSFiddle part. This is a code-highlighted text area whose contents are injected into the iframe’s javascript. The code for this is in the “fiddler.html” and “playfiddle.js” files. fiddler.html is actually the homepage, and thus contains the outer webpage’s markup. Playfiddle contains javascript that loads the contents of fiddle.txt, which is actually the webpage that runs WebGL and the FOSSSim code, into the iframe.

4. The Scene input part. The javascript code for this is all contained within “input-stuff.js”, while the html markup is fiddler.html. This code allows for dynamically and asynchronously generated forms for inputting particles and forces into the scene.

FOSSSIM’S MATH

(NOTE: everything is in 2D here. Not much is stopping you from extending it to 3D, as the current setup implementation simply only allows particles to move along the z = 0 plane in 3D space)

Forces

Four (and a half-ish) forces are implemented here:

- Simple Gravity Force. Simulates a near-earth gravitational field, in which a constant force is applied in a direction [gx, gy] = **g** (bold is a vector quantity here). If m_i is the mass of a particular particle in the scene, the force applied is:

  Force exerted on each particles in scene: m_i***g**.

- Gravitational Force. A force simulating the effect of two particles interacting via gravity. Basically there is a force exerted by each particle pulling the other towards it. For two particles with indices i and j, where **x**_i and **x**_j denote each’s position, m_i and m_j denote each’s masses, L denotes the distance between them, G is a gravitational constant, and **nhat** is a unit-length vector from particle j to particle i given by (**x_i** - **x_j**) / sqrt((**x_i** - **x_j**)^2), the equation for these two forces is:

  Force exerted on particle i by particle j: ((G * m_i * m_j) / L^2) * **nhat**

  Force exerted on particle j by particle i: - ((G * m_i * m_j) / L^2) * **nhat**

- Linear Damping Force (aka Drag). This exerts a force that linearly resists the motion of all particles in the scene, according to some constant, b. If **v**_i is the vector representing a particular particle’s velocity, the equation for this force is:

  Force exerted on each particle in scene: - b * **v**_i

- Spring Force. This simulates the action of a harmonic oscillator (a spring) attached to two particles, i and j. If we have k as a spring constant, L0 as the spring’s resting length, L as the current distance between i and j, and **nhat**, the equation for these two forces is:

  Force exerted on particle i by particle j: k * (L - L0) * **nhat**

  Force exerted on particle j by particle i: -k * (L - L0) * **nhat**

- Spring Damping Force. Simulates an internal force that resists motion of a spring. This is actually build into the SpringForce class in the source code, so this is the “half” force I mentioned earlier. If we have a damping constant b, the velocities of each particle **v**_i and **v**_j, and **nhat**, the equation for the force on each endpoint is:

  Force exerted on particle i by particle j: - b * **nhat** dot_product (**v**_i - **v**_j) * **nhat**

  Force exerted on particle j by particle i: b * **nhat** dot_product (**v**_i - **v**_j) * **nhat**

Integrators

There are two time-steppers/integrators implemented here. For both, we will use the previous and current time step’s position and velocities, respectively denoted by **x**_n-1, **x**_n**,** **v**_n-1**,** **v**_n, as well as the time step, dt, the mass **m**, and the Force F(**x**, **v**) (since the force depends on both position and velocity).

- Explicit Euler. Basically calculates everything for this frame based on everything last frame:

  x_n = **x**_n-1 + dt * **v**_n-1
  v_n = **v**_n-1 + dt * F(**x**_n-1, **v**_n-1) / **m**

- Symplectic Euler. Very similar to explicit, except you calculate velocity first using information from the previous time step, and then use this newly-calculated velocity to calculate this time step’s position:

  v_n = **v**_n-1 + dt * F(**x**_n-1, **v**_n-1) / **m**
  x_n = **x**_n-1 + dt * **v**_n

DOCUMENTATION

This won't be a complete rundown of every function in the code, but rather a description of the general logic used and descriptions of some of the more important functions. The purpose of this section is to give others who wish to further the project some information to orient themselves with. If you have a good knowledge of html, javascript, and jQuery, you should be able to follow what's going on. Some basic familiarity with other technologies used like THREE.js (I'm talking beginner tutorial level), then you're even better off.

Note on FOSSSim

One thing to note: my implementation of FOSSSim is heavily inspired by the object-oriented structure of Three.js. I was new to this aspect of javascript when I started this project, so this was a good starting off point for me. The structure here is as follows: There is a parent FOSSSim object, which is extended via prototypes with FOSSSim.Scene, FOSSSim.Stepper, and FOSSSim.Force objects. FOSSSim.Force is more of an abstract class, so you'd actually create specific, implemented forces, like FOSSSim.SimpleGravityForce.

FOSSSim

The FOSSSim code is entirely contained within the "src" folder. Within src, there is another folder, Force, and three individual files, main.js, Scene.js, and Stepper.js.

Scene.js

The FOSSSim.Scene object acts as a container for all of the particle vertex positions, velocities, radii, and masses, all of the forces in the scene, and all of the THREE.js lights and Spheres. It also keeps a running total of how many particles are currently in the scene.

2D Vector quantities are stored in a single dimensional array, where the values for particle i is accessed via index 2*i for the x component and 2*i + 1 for the y component. In the file, the attributes with this quality are x (the position of every particle in the scene), v (the velocities), m (the masses*), and spheres (the THREE.js spheres).

* of course, a single particle doesn't have an x and y component for mass. Instead, the same mass value is used for in both positions (as in, particle i's mass is stored at both m[2*i] and m[2* + 1]. The fact that the mass array is the same length as the position and velocity arrays allows for you to divide one by the other (pointwise division), which makes some of the math easier.

There are a bunch of functions with the form "initSomething", where Something is Vectors, Spheres, Lights. These are called elsewhere depending on where that information is coming from. For example, initVectors(particles, forces) is called in setScene(particles, forces, integrator) in main.js, which is in turn called in input-stuff.js, since that file takes the particle and force information from the html page and puts it into the Scene. initSpheres() must be called after initVectors (it depends on x and r already containing particle information), and initLights simply has THREE.js lights hardcoded in (this can obviously be extended to allow for greater scene control).

The one other useful method here is the “accumulateForces(F)” function. This goes through every force in forces (the object attribute) and calls the “addForceTotal(F)” function on it. F here is a reference to an accumulator array. This function is called by the Stepper object, and is used by the integrator method to calculate the forces used by the scene .

Force.js (and inheriting classes)

The Force.js base class and the classes that implement it are contained in the Force directory. Force.js contains one function, addForceToTotal(F). As mentioned earlier, F is an accumulator array, that is passed around between all of the forces in the Scene, eventually calculating the net force acting on each particle. Each force that extends Force.js implements this function. Once you know the math (as described in the previous section) the implementations are easy. Each derived Force has a constructor for it’s force-specific constants (whatever values you’d need to calculate the force. The velocities and positions are taken from the global fosssim_scene variable. There is also a class containing helper functions called **ForceHelpers.js**, though it currently only has one function in it. That function, “add2SegIntoF(F, in_seg, index)”, basically functions as a += function for a vector of length 2. It takes in_seg, adds it to F and index, and stores that value back in F.

Stepper.js

This is a fairly straightforward class, and essentially contains all of the functions necessary for stepping through a scene, using the last frame’s positions and velocities to somehow calculate this frame’s. The step(integrator_type) function chooses which integrator to use based on the string argument “integrator_type”. explicitEulerStep and implicitEulerStep are implemented as described in the previous section, and modify the values of the global fosssim_scene object, which is an instance of FOSSSim.Scene. Finally, updateSpherePos updates the values of the THREE.js Sphere objects according to Scene particle positions (conceptually they represent the same thing, but the Scene particle positions are just values while Spheres are rendered by THREE.js via WebGL.

main.js

Main.js is the parent file that contains al of the global variables used for the simulation. There are a few global variables to keep track of things like the time step (dt), and number of frames so far, some THREE.js-specific variables, and, most importantly an instance of FOSSSim.Scene and FOSSSim.Stepper than can be accessed globally.

The important functions are setScene(particles, forces, integrator), initScene(canvas), and animate(). setScene is called in input-scene.js, and basically creates the relevant FOSSSim objects when you add them via the html page forms. initScene does most of the THREE.js initializing, creating a THREE.js instance inside of an HTML canvas element specified through an argument. This is called in the main() method in main.js, which is in turn called in the actual iframe’s html, fiddle.txt. animate() used the requestAnimationFrame(fn) function, which calls a function fn repeatedly, for use in animation applications (there is plenty of documentation for requestAnimationFrame elsewhere).

A note on program flow: due to the nature of web applications/javascript, we don't always have control over when functions are called. This needs to be taken into account. In this particular case, setScene ends up getting called before initScene, but the initSpheres and initLights functions can only be called once a THREE.js Scene (different than FOSSSim Scene…confusing I know) has been created. Thus, why those functions are called within initScene and not setScene. If the two functions executed in the reverse order than this wouldn't be necessary.

Fiddler

Fiddler.html is what I named the main html page of the application, since the main feature (running code written in a text area immediately within an iframe on the same page) is heavily inspired by how JSFiddle works. Other relevant files to this are playfiddle.js, fiddle.txt, and initial.txt. Most of the magic happens in playfiddle.js.

fiddler.html

The main page of the application. It has three tabs, one for inputting scenes, one for inputting code, and the animation window, which is an iframe. When the iframe is loaded, you'll see two buttons in addition to the black animation window, play and step. Play starts the animation and continually steps through until you press play again (the text changes to say "pause" instead of play when it's playing—it starts out paused), while step allows you to step through a frame at a time.

fiddle.txt

Though this is a txt file, it actually contains the html code of the simulator (you could hypothetically copy and paste this into another html file and run it on it's own, you'd just have to hardcode the particles in). This is loaded into the iframe by playfiddle.js.

playfiddle.js

The program flow starts when a user presses the "change-code" button. When this happens, the contents of "fiddle.txt" are loaded into the iframe via a jQuery get request (this occurs in the "load_page_from_txt(filename, user_input)" function. The contents of the coding window are also loaded into a variable at this point, and passed as an argument in load_page_from_txt and play_fiddle. When the get request succeeds, the play_fiddle(page_str, user_input) is called, which loads the contents of the coding into into the template given by fiddle.txt. In this case, the "@@@" string in fiddle.txt is replaced by whatever javascript was in the coding window. There isn't that much error handling here to errors will appear in the console. This string (fiddle.txt with the coding window contents loaded into it) is written to the iframe.contentDocument, thereby reloading the iframe with the simulator code.

initial.txt

I lied a bit when I said the program flow starts when the user presses change-code. When the page loads, whatever is in initial.txt is loaded into the coding window. This is useful for putting in a code stub that a student should fill out to extend the simulator.

Input Stuff

The eloquently titled input-stuff.js file handles inputting a new scene and loading that into the iframe. The change-code button is still here (they're different html buttons but both trigger the same event, so when you click it BOTH the scene and the javascript code are reloaded into the scene). The interesting stuff here happens almost entirely within input-stuff.js, though the relevant markup is still in fiddler.html.

input-stuff.js

When any of the "add" buttons are pressed (add particle or add force), the javascript adds the relevant markup via jQuery to fiddler.html. This code is fairly straightforward, and is contained in the $.on('click', function(){}) bindings. When change-code is clicked, the program goes through the markup and parses through the relevant forms for added particles, forces, and integrator type. In doing so, it creates an array of particles and forces in a format expected by the initVectors method of the FOSSSim Scene. It then passes these arrays, as well as a string denoting integrator type, into the iframe via the setScene method. It does this by delving into the iframe and accessing the global setScene function there. It happens with this line:

```
window.frames['fiddle'].window.setScene(fosssim_particles, fosssim_forces, integrator);
```

Other Technologies

Of course, all of this would've been way more difficult if I didn't have some other technologies at my disposal for rendering, math, and code highlighting. All of these external technologies are contained in the "outside-tools" folder. Here's a quick rundown of what's there and for what:

- **Three.js** Used as a wrapper for WebGL functionality. Provides a much cleaner interface for rendering WebGL graphics.
- **Numeric.js** Used for vector math and linear algebra.
- **CodeMirror** Used for the javascript code-highlighting in the Code tab.
- **Bootstrap** Used for the UI elements, most importantly the tabbed interface.

The websites for these are:

http://threejs.org/

http://www.numericjs.com/

http://codemirror.net/

http://getbootstrap.com/

also check out http://jsfiddle.net/. I didn't use any of it's code, but I borrowed heavily from how it was implemented.

CONTACT

written by Cameron Erdogan

Questions? Bored? Contact me at cameron.the.Erdogan@gmail.com or cde2113@columbia.edu