BarraCUDA Project: A Homebrew Supercomputer

Matt Redmond

Feb. 4, 2010

Contents

1 Abstract

The BarraCUDA (Beginner's Analytic Research and Rendering Architecture over C Unified Device Architecture) project will ultimately entail a small-scale homebrew supercomputer designed around the NVIDIA CUDA architecture, as well as a set of simulation programs coded to take advantage of this CUDA architecture. The project is meant to serve as an introductory platform for student-researchers to develop parallel-programming skills, and design programs that showcase concepts in modern science research through real-time visualization. The initial seed for this projected was planted by Dr. Kayvon Fatahalian, a graphics researcher at Stanford University. The hardware aspect of the BarraCUDA project will be completed over a different time frame, but before the hardware is finished, I intend to develop and (on a less powerful computer) test a comprehensive and accurate three dimensional simulation of electrodynamics. This project will explore the vast world of scientific computing through the lens of a physicist, and should provide ample opportunities for me to learn new techniques in computational physics as well as numerical analysis.

2 Introduction

Through my Applied Science Research and Computer Science classes, I've gradually become aware of the growing importance of high performance computational power in the fields of physical, chemical, and biological research. A small sample of these problems can be found at http://www.nvidia.com/object/cudahome.html, but three particularly interesting ones (that seems to form a representative sample of the types of problems that one can approach with supercomputing resources) are [1], [2], and [3]. Because of my experiences over the summer, I've become quite interested in learning more about neutron scattering and visual representations of physical modeling problems. Last summer, I got a chance to work in Caltech's materials physics lab on the DANSE project, using distributed computational analysis to model the theoretical results of neutron scattering experiments before actually conducting them. It was here where I was exposed to the elegant concept and power of a Beowulf cluster, and I immediately realized that our high school could make good use of such a powerful tool (on a smaller scale) for instructional purposes, in the demonstration/visualization of scientific concepts as well as the programming process.

I have begun to design the simulation on a CUDA-based supercomputer because CUDA represents the new direction that high performance computing is moving in: instead of waiting for a single fast central processing unit to perform heavy calculations sequentially, these calculations are farmed out in parallel to hundreds of individual graphics processing unit cores for computation. The actual performance increase (across diverse applications) is reported by various sources [4] to be anything from 5x to 2600x faster than a CPU-based system. Furthermore, the CUDA architecture is alive and well-supported by the community, with various development tools available, and many new applications released frequently. CUDA supports the API OpenMP, a well-known parallel processing API and widely supported in the scientific computing community. I'd like to gain further experience (extending my summer research) working with OpenMP on hardware that can unlock all of its power. CUDA is supported on several chipsets, but the two most suitable consumer-grade architectures for this project are the NVIDIA GeForce GTX295 and the NVIDIA Tesla C1060.

The bottom line here is that supercomputing as a field continues to grow very rapidly, and most pure sciences are relying upon supercomputing to provide a basis for carrying out their research. It's particularly interesting to explore the underpinnings of graphics processor based supercomputing, because the onset of GPU computing provides a brand new avenue for massive scalability that simply isn't present in the CPU-based paradigm.

3 Theory

This section will be broken down into the computer science theory behind how the project works, and the physical science theory behind how systems of electrically charged particles interact. Currently (as of January 2010), the simulation is not tuned to run on the CUDA architecture.

3.1 Supercomputing Theory

A GPU-based supercomputer differs from standard CPU-based supercomputers in a number of ways: the most obvious one of these is the actual location a nd type of the numerical processors. A CPU-based cluster (I will interchangeably refer to supercomputers and cluster computers in this paper; they are equivalent for our purposes) uses processors housed directly on the node motherboard, which communicate directly with the system memory through a high-speed bus. This has the advantage of extremely fast memory read/write operations, but it has the disadvantage (currently) of a relatively small number of available cores. The current top-of-the-line Intel i7-920 runs 8 cores at 1600 MHz each [5]. The GPU, in comparison, resides (generally) in an interface bus. In modern computers, this bus is the PCI express bus, which provides an indirect way of communicating with the memory through a controller chip on the motherboard. The advantage of the GPU-based system is that it can run many more cores than a CPU-based system can (480 cores per card in the BarraCUDA system at 1242 MHz each) [6], but the disadvantage is that it has relatively slower memory read/write times. Most GPUs attempt to correct for this by including memory onboard the card (1796 MB in the BarraCUDA system per card), so the read/write times are reduced to those of a CPU-based system. As one can see, there are significant advantages to using a GPU-based setup.

The main advantage (drastically more cores) of the GPU-base system is only harnessable when a program is designed to UTILIZE all of those cores in parallel. Certain programs lend themselves well to this paradigm; others do not. An example of an easily parallelizable problem is one that can be broken down into a number of sub-tasks, each of which can be computed independently of the other tasks. The n-body electrodynamics problem is a perfect example of an easily parallelizable problem: computing the force on each particle at a given timestep only relies on the previous position of all of the other particles. If we store the point charge positions in an array that is globally accessible by all of the cores, then we can update each particle SIMULTANEOUSLY. This is a HUGE speedup over the sequential calculations enacted by a single-core CPU setup, and provides the motivation for a multiple-core setup in general.

3.2 Scientific Theory

The simulation that is currently in place relies upon Coulomb's Law to generate the electric field acting on each individual point charge. This efield generates a force that is similar to the gravitional force in that it is an inverse-square law, but different because the force vector is signed (instead of always attracting like gravity, it can attract or repel). The force vector acting on a point charge is represented by $\vec{F} = q\vec{E}$, where q is the charge of the particle and $\vec{E}$ is the electric field acting on the particle. The explicit way to compute $\vec{E}$ is to sum up the effects of each other particle in the simulation on the particle in question.

Couloumb's Law gives this electric field sum as

$$\vec{E} = \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{n} \frac{q_i(\vec{r} - \vec{r_i})}{|\vec{r} - \vec{r_i}|^3}$$

where $\vec{r}$ is the vector to the point that we are testing the electric field at, and $\vec{r_i}$ is the vector of the point we are testing against. As i ranges from 1 to n, we test the electric field against all other particles. This means that updating ONE particle's electric field requires us to check $n-1$ other particle's states, where n is the total number of particles. Updating ALL of the particle's electric fields requires $n(n-1)$ checks, making the n-body electrodynamic simulation a problem known as an $O(n^2)$ problem (it has time complexity asymptotically equal to some constant multiple of n^2). These problems scale notoriously badly on standard PC hardware, but with a supercomputer, they become approachable again. There are ways to reduce to speed this problem up (by only considering particles within a certain distance of eachother to have an observable effect) which can reduce the runtime complexity, but these methods trade a gain in speed for a reduction in accuracy. I have decided that accuracy is a more desirable characterstic of my simulation than speed is, so my simulation checks all $n(n-1)$ particles. If it becomes an issue in the future, the range-based solution described above can be implemented.

Coulomb's Law is derived under the assumption that the particles are stationary. When they are moving, Coulomb's Law provides only an approximation to the electric field acting on the particles. In motion, these particle actually generate electromagnetic waves that can perturb the electric field of the others. I will attempt to model the effects of these electromagnetic waves later in the project. For now, Coulomb's Law is the only thing that is used to compute the forces on the particles. Furthermore, the scale factor $\frac{1}{4\pi\epsilon_0}$ is roughly equal to 9×10^9, which is too large for accurate computation. Because ϵ_0 is defined as $\frac{1}{\mu_0 c^2}$, where c is the speed of light and μ_0 is the magnetic constant (generally defined as $4\pi \times 10^{-7} \frac{H}{m}$), these values are all system-unit dependent. I've introduced a system of units that is normalized to the electron mass, charge, and radius, so this constant scale factor $\frac{1}{4\pi\epsilon_0}$ is absorbed into the graphics engine.

The integrator that I have written for this project is (essentially) a numeric differential equation solver. This integrator uses the Runge-Kutta 4 method, which works by exploiting Euler's method: RK4 reduces the magnitude of the error term by sampling the derivatives four times over the timestep interval. A general introduction to this technique is given at [7], and indeed, I have adapted the author's technique for my particular implementation. This integrator has average error on the order of $O(n^5)$, which provides a perfect tradeoff between speed and accuracy. It also lets me correctly model situations where two particles are very close together, as the reduced error term (and weighted sum of derivatives) is much more robust under the influence of large outliers than Euler's method is.

4 Accomplishments

Currently, the simulation is able to display two-dimensional interactions between charged particles. The underlying engine is actually processing the vectors as three-dimensional vectors, but I haven't implemented three-dimensional graphics. A fully functional implementation of Coulomb's Law is given in the Physics.java file, and a fully functional Runge-Kutta4 numerical integrator is given in the NumericalIntegration.java file. The graphics are rudimentary at the moment, but they provide a decent visual for what is happening with the particles. You can toggle the momentum and efield vectors on and off inside of the twoDimensionalSim.java file by commenting and uncommenting the code. A lot of the actual explanation for what is being done is provided inside of the code as comments.

5 The Next Steps

Collision detection between particles is partially functioning: it actually works as intended, but the electric field update is so much stronger than the momentum conservation component that it completely overrides the momentum transfer. I need to fix this by temporarily disabling the electric field update during collision processing. Fixing collision detection is a short term goal, but a longer term goal is getting three dimensional graphics set up. I'll probably use the OpenGL standard graphics library, or whatever the CUDA IDE provides, as a test bed. Finally, I still need to find a way to interface the simulation (I am looking at a library called JCUDA for this) and parallelize the numerical calculation so that is it suitable for use with the CUDA architecture. As described in the Scientific Theory section, the algorithm is easily parallelized, but the actual low-leval core operation might not be. I'll spend some time looking through CUDA documentation and working with OpenMP.

I have a CUDA-enabled testbed that I will experiment on, and the next highest priority in this project will become the migration of the simulation code to CUDA code. It's likely that this will be fraught with errors, and will require substantial amounts of debugging, but I hope to have the simulation migrated in about a month. The parallelization will require ongoing effort to optimize, but the initial setup should be fairly straightforward. There are a lot of places that this project can go, from increased accuracy to higher speed, to modelling more complicated approximations, and I look forward to approaching all of these avenues.

6 Citations

[1] CUDA-Based Incompressible Navier-Stokes Solver. Julien Thibault and Inanc Senocak.
<http://coen.boisestate.edu/senocak/files/BSU_CUDA_Res_v5.pdf>
[2] TeraFlop Computing on a Desktop PC with GPUs for 3D CDF. J. Tolke and M. Krafczyk.
<http://www.irmb.bau.tu-bs.de/UPLOADS/toelke/Publication/toelkeD3Q13.pdf>
[3] CUDA Acceleration of Molecular Dynamics. David Kirk and Wen-mei W. Hwu.
<http://www.ks.uiuc.edu/Research/gpu/files/lecture8casestudies.pdf>
[4] NVIDIA CUDA Project Repository. <http://www.nvidia.com/object/cuda_home.html#>
[5] Intel Corei7 Specifications <http://www.intel.com/products/processor/corei7/specifications.htm>
[6] NVIDIA GTX295 Specifications <http://www.nvidia.com/object/product_geforce_gtx_295_us.html>
[7] Fiedler, Glenn. Physics in 3D. <http://gafferongames.com/game-physics/physics-in-3d/>

A Source Code

The source provided in this appendix should provide a valuable resource for understanding the bulk of the simulation. Source files can also be found online at http://github.com/mredmond/BarraCUDA

A.1 Graphics Package

A.1.1 twoDimensionalSim.java

```
/* This class encapsulates the graphics object that is printed to the screen after each physics update.
 * It contains a paint method, and tracks the global array that holds the pointCharges.
 * This is a plug-and-play implementation, designed to be flexibly replaced by an OpenGL implementation.
 *   */

package graphics;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;
import java.util.*;
import javax.swing.*;
import main.BarraCUDA;
import physics.PointCharge;

public class twoDimensionalSim extends JPanel implements ActionListener, ItemListener
{
public twoDimensionalSim()
{
createAndShowGUI();
}
public void setSize(Dimension d)
{
super.setSize(d);
repaint();
}
//This method initializes all of the menus. At the moment, these menus are non-functional.
public void createAndShowGUI()
{
JFrame frame = new JFrame();
frame.setSize(1280, 1024);
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

JMenuBar menuBar = new JMenuBar();
JMenu fileMenu = new JMenu("Controls");

JMenuItem pause = new JMenuItem("Pause ||");
pause.addActionListener(this);
pause.setActionCommand("pause");
fileMenu.add(pause);

JMenuItem play = new JMenuItem("Play  |>");
play.addActionListener(this);
play.setActionCommand("play");
fileMenu.add(play);

JMenuItem step = new JMenuItem("Step Forward  ->");
step.addActionListener(this);
step.setActionCommand("step");
fileMenu.add(step);

menuBar.add(fileMenu);
frame.setJMenuBar(menuBar);
frame.add(this);
frame.setVisible(true);
}

//This method accesses the global arrayList held in the BarraCUDA main class.
//It will paint each charge, as well as its electric field vector (representing acceleration)
//or its momentum vector (chosen by default... if you want the other one, uncomment it)
public void paint(Graphics g)
{
super.paint(g);
ArrayList<PointCharge> charges = BarraCUDA.mainChargeManager;
g.setColor(Color.white);
g.fillRect(0,0,this.getBounds().width,this.getBounds().height);

Iterator<PointCharge> myIter = charges.iterator();
while(myIter.hasNext())
{
PointCharge pc = myIter.next();
drawCharge(g, pc);
//drawEField(g, pc);
```

```
drawMomentum(g, pc);
}
}

//This method draws a charge according to its position and parity.
public void drawCharge(Graphics g, PointCharge pc)
{
if(pc.myState.charge > 0) g.setColor(Color.BLUE);
else if(pc.myState.charge < 0) g.setColor(Color.RED);
else g.setColor(Color.LIGHT_GRAY);

//Actually draw the particle
g.fillOval((int) Math.round(pc.myState.position.x - pc.myState.radius),
(int) Math.round(pc.myState.position.y - pc.myState.radius), (int) Math.round(2*pc.myState.radius), (int) Math.round(2*pc.myState.radius));
}

//Draw the electric field vector on any point charge.
public void drawEField(Graphics g, PointCharge pc)
{
if(pc.myState.charge > 0) g.setColor(Color.BLUE);
else if(pc.myState.charge < 0) g.setColor(Color.RED);
else g.setColor(Color.LIGHT_GRAY);

g.drawLine((int) Math.round(pc.myState.position.x), (int) Math.round(pc.myState.position.y),
(int) Math.round(pc.myState.position.x + pc.myState.efield.x), (int) Math.round(pc.myState.position.y + pc.myState.efield.y));
}

//Draw the momentum on any point charge.
public void drawMomentum(Graphics g, PointCharge pc)
{
g.setColor(Color.black);
g.drawLine((int) Math.round(pc.myState.position.x), (int) Math.round(pc.myState.position.y),
(int) Math.round(pc.myState.position.x + pc.myState.momentum.x), (int) Math.round(pc.myState.position.y + pc.myState.momentum.y));
}

@Override
//Non-functional menus. Might be implemented at some point, might not.
public void actionPerformed(ActionEvent arg0)
{
String command = arg0.getActionCommand();
if(command.equals("play"))
{
//do play event handling
}
else if(command.equals("step"))
{
//do step handling
}
else if(command.equals("pause"))
{
//do pause handling
}
}
@Override
public void itemStateChanged(ItemEvent e)
{
// TODO Auto-generated method stub
}
}
```

A.2 Main Package

A.2.1 BarraCUDA.java

```
/* This class is the main class (and should be compiled as such) that houses all of the functionality of the BarraCUDA project.
 * The general idea is that we maintain the global variable mainChargeManager here that is accessible from all of the other classes.
 * This class also holds the physics engine object in it (which itself subcontains a numerical integrator and force-finder), as well as
 * a two-dimensional graphical simulation. This class is flexible enough to allow a fairly easy overhaul to CUDA specifications.
 * The graphics module is design as a plug and play style class, so it can be swapped out for a three-dimensional one soon.
 */

package main;
import graphics.twoDimensionalSim;
import java.util.ArrayList;
import physics.*;
import util.*;

public class BarraCUDA
{
public static ArrayList<PointCharge> mainChargeManager = new ArrayList<PointCharge>();
public static Physics physicsEngine;
public static final int NUM_PARTICLES = 130;
public static void main(String[] args)
{
//Makes a new graphics object to render into
twoDimensionalSim myGraphicsObj = new twoDimensionalSim();

// Manual charge creation
// PointCharge charge0 = new PointCharge(0, 1.0, 1, 5);
// PointCharge charge1 = new PointCharge(1, 1.0, 1, 5);
// PointCharge charge2 = new PointCharge(2, -1.0, 1, 5);
```

```
//
// mainChargeManager.add(charge0);
// mainChargeManager.add(charge1);
// mainChargeManager.add(charge2);

//Auto Initialization
addRandomCharges(NUM_PARTICLES);

//physics engine object initialization
physicsEngine = new Physics(mainChargeManager);

// Manual state initialization
// physicsEngine.initializeChargePosition(0, new Vector(300, 200, 0));
// physicsEngine.initializeChargeMomentum(0, new Vector(0, 0, 0));
// physicsEngine.initializeEField(0, new Vector(0,0,0));
//
// physicsEngine.initializeChargePosition(1, new Vector(400, 200, 0));
// physicsEngine.initializeChargeMomentum(1, new Vector(0, 0, 0));
// physicsEngine.initializeEField(1, new Vector(0,0,0));
//
// physicsEngine.initializeChargePosition(2, new Vector(350, 150, 0));
// physicsEngine.initializeChargeMomentum(2, new Vector(0, 0, 0));
// physicsEngine.initializeEField(2, new Vector(0,0,0));

//Auto Initialization
initializeCharges(NUM_PARTICLES);


//Main update loop. Does one physics iteration, then renders scene.
double dt = 0.01;
for(double t = 0.0; t <= 30.000; t+=dt)
{
physicsEngine.updateAll(t, dt);
try
{
Thread.sleep(10);
//Thread.sleep(2000/NUM_PARTICLES); //scales wait time to deal with number of particles? not very useful
}
catch (InterruptedException e)
{
e.printStackTrace();
}
myGraphicsObj.repaint();
}
}

//Does what it says on the tin: adds n random charges to the physics engine
public static void addRandomCharges(int n)
{
for(int i = 0; i < n; i++)
{
double charge = Math.random();
double chargeSignModifier = Math.random();
double mass = 1;
double radius = 5;

if(chargeSignModifier <= 0.5)
{
charge = -charge;
}

PointCharge pc = new PointCharge(i,charge,mass,radius);
mainChargeManager.add(pc);
}
}

//Again, name says it all. Sets the charges with random positions, zeroes their efield vector, and gives them a random momentum
public static void initializeCharges(int n)
{
for(int i = 0; i < n; i++)
{
double momx = Math.random();
double momxMod = Math.random();
double momy = Math.random();
double momyMod = Math.random();
if(momxMod <= 0.5)
{
momx = -momx;
}
if(momyMod <= 0.5)
{
momy = -momy;
}

//The magic numbers in the first line here are scaled to deal with the resolution 1280 x 1024, but can be adjusted accordingly.
physicsEngine.initializeChargePosition(i, new Vector(Math.rint(1000*Math.random() + 50), Math.rint(800*Math.random() + 50), 0));
physicsEngine.initializeChargeMomentum(i, new Vector(momx, momy, 0));
physicsEngine.initializeEField(i, new Vector(0,0,0));
}
}
}
```

A.3 Physics Package

A.3.1 Physics.java

```
/* This class gives a higher-level interface access to the numerical integration.
 * It holds a chargeManager identical to the one found in the BarraCUDA.java class, and all of its operations
 * are done on this arrayList. Most of the methods are self-explanatory, but updateElectorFieldApproximation is not.
 * This method basically uses Coulomb's Law to determine the force on each particle (taking into consideration the effects of all of the other particles)
 * and will also do (very very very basic) collision detection. The collision detection is merely there to prevent weird things like interpenetration from happening
 * (and they still do happen pretty frequently, so ... yeah).
 */
package physics;

import java.util.*;
import util.Vector;

public class Physics
{
public final double GRAPHICS_EFIELD_SCALE_FACTOR = 50000;
//this is roughly analogous to the constant value K, except instead of 9 x 10^9, I use a smaller value
public ArrayList<PointCharge> chargeManager;
public NumericalIntegration Integrator;

public Physics(ArrayList<PointCharge> chargeManagerIn)
{
this.chargeManager = chargeManagerIn;
this.Integrator = new NumericalIntegration();
}

public void addCharge(int id, double charge, double mass, double radius)
{
chargeManager.add(id, new PointCharge(id, charge, mass, radius));
}

public void removeCharge(int id)
{
chargeManager.remove(id);
}

public void initializeChargePosition(int id, Vector positionIn)
{
chargeManager.get(id).myState.position = positionIn;
}

public void initializeChargeMomentum(int id, Vector momentumIn)
{
chargeManager.get(id).myState.momentum = momentumIn;
}

public void initializeEField(int id, Vector efieldIn)
{
chargeManager.get(id).myState.efield = efieldIn;
}

public void updateElectrofieldApproximation()
{
//This method updates the efield vector for each point charge in the chargeManager.
//It is used in the NumericalIntegration class for determining forces.
for(PointCharge pc1 : chargeManager)
{
//create the efield acting on one charge
Vector sum = new Vector(0,0,0);
for(PointCharge pc2: chargeManager)
{
if(pc2.idNum == pc1.idNum)
{
//skip this case ... don't want to add a particle to its own e-field
}
else
{
//necessary variables for eField calc
Vector r = pc1.myState.position;
Vector rHat = pc2.myState.position;
Vector rDiff = r.subtract(rHat); //a vector going from r to rHat
double qi = pc2.myState.charge;

if(rDiff.length() < pc2.myState.radius + pc1.myState.radius) //only add if the particles are suitably far apart
{
//they're too close. get ready for collision detection
pc1.myState.touchingOther = true;
pc2.myState.touchingOther = true;
pc1.myState.touching = pc2;
pc2.myState.touching = pc1;
//System.out.println("Particle " + pc1.idNum + " is touching particle " + pc2.idNum + ".");
}
else
{
Vector numerator = rDiff.scale(qi);
double inverseDenominator = Math.pow((rDiff.length()), -3);
sum = sum.add(numerator.scale(inverseDenominator)); //add up the other particles' effects
pc1.myState.efield = sum.scale(GRAPHICS_EFIELD_SCALE_FACTOR); //arbitrary scale factor to make graphics work.
}
}
}
```

```
if(pc1.myState.touchingOther)
{
//2D Collision Response at the moment.
//kind of touchy... would probably think hard about where I am doing this test (shouldn't it be inside the main loop and not the forces update?)
PointCharge pc2 = pc1.myState.touching;
Vector collisionUnitNormal = pc2.myState.position.subtract(pc1.myState.position).normalize();
Vector collisionUnitTangent = new Vector(collisionUnitNormal.y, collisionUnitNormal.x, 0);
Double v1n = collisionUnitNormal.dot(pc1.myState.velocity);
Double v1t = collisionUnitTangent.dot(pc1.myState.velocity);
Double v2n = collisionUnitNormal.dot(pc2.myState.velocity);
Double v2t = collisionUnitTangent.dot(pc2.myState.velocity);
Double v1tprime = v1t;
Double v2tprime = v2t;
Double v1nprime = (v1n*(pc1.myState.mass - pc2.myState.mass) + v2n*(2*pc2.myState.mass))/(pc1.myState.mass + pc2.myState.mass);
Double v2nprime = (v2n*(pc2.myState.mass - pc1.myState.mass) + v1n*(2*pc1.myState.mass))/(pc1.myState.mass + pc2.myState.mass);
Vector vec1nprime = collisionUnitNormal.scale(v1nprime);
Vector vec1tprime = collisionUnitTangent.scale(v1tprime);
Vector vec2nprime = collisionUnitNormal.scale(v2nprime);
Vector vec2tprime = collisionUnitTangent.scale(v2tprime);
Vector vec1final = vec1nprime.add(vec1tprime);
Vector vec2final = vec2nprime.add(vec2tprime);
pc1.myState.velocity = vec1final;
pc2.myState.velocity = vec2final;
pc1.myState.efield.zero();
pc2.myState.efield.zero();
pc1.myState.touchingOther = false;
pc2.myState.touchingOther = false;
pc1.myState.touching = null;
pc1.myState.touching = null;
//pc1.myState.momentum.zero();
}
}
}
public void updateAll(double t, double dt)
{
updateElectrofieldApproximation(); //just do this ONCE per update, otherwise you've got some problems
for(PointCharge pc : chargeManager)
{
Integrator.integrate(pc.myState, t, dt);
}
}
}
```

A.3.2 NumericalIntegration.java

```
/* This class provides a state-based RK4 integrator for the physics engine.
 * The premise is that we integrate objects one STATE at a time, instead of one field at a time.
 * RK4 is based on a fourth-order Taylor Series error estimate, and is pretty well covered in the literature.
 * This implementation relies on the Derivative class as a structure to store the various differentials, and
 * while it's not entirely necessary to use this class, it really saves a lot of confusion in the code.
 */

package physics;
import util.*;

public class NumericalIntegration
{
public Derivative evaluate(State initial, double t)
{
Derivative output = new Derivative();
output.dx = initial.momentum;
output.dv = initial.efield.scale(initial.charge);
return output;
}

//overloading the evaluate() method to account for the other derivatives in RK4
public Derivative evaluate(State initial, double t, double dt, Derivative d)
{

State state = initial;
    state.position = initial.position.add(d.dx.scale(dt));
    state.momentum = initial.momentum.add(d.dv.scale(dt));

    Derivative output = new Derivative();
    output.dx = state.momentum;
    output.dv = state.efield.scale(state.charge);
    return output;
}

public void integrate(State state, double t, double dt)
    {
        Derivative a = evaluate(state, t);
        Derivative b = evaluate(state, t+dt*0.5f, dt*0.5f, a);
        Derivative c = evaluate(state, t+dt*0.5f, dt*0.5f, b);
        Derivative d = evaluate(state, t+dt, dt, c);

        state.position = state.position.add((a.dx.add(b.dx).add(b.dx).add(c.dx).add(c.dx).add(d.dx)).scale(dt/6));
        state.momentum = state.momentum.add((a.dv.add(b.dv).add(b.dv).add(c.dv).add(c.dv).add(d.dv)).scale(dt/6));
        state.recalc();
   }
}
```

A.3.3 PointCharge.java

```
/* A single point charge. Pretty simple class, really. It holds its own state.
 */
package physics;

import util.State;

public class PointCharge
{
public State myState;
public int idNum;

public PointCharge(int idNum, double charge, double mass, double radius)
{
this.idNum = idNum;
myState = new State(charge, mass, radius);
}

public String toString()
{
return "pointCharge" + idNum + "\n charge: " + myState.charge + "\n mass: " + myState.mass
+ "\n position: " + myState.position.toString() + "\n velocity: " + myState.momentum.toString() + "\n efield: " + myState.efield.toString() + "\n";
}
}
```

A.4 Utilities Package

A.4.1 Derivative.java

```
/* The derivative structure used by the RK4 integrator. Very simple.
 */

package util;
public class Derivative
{
public Vector dx; //velocity
public Vector dv; //acceleration

public Derivative(){};

public Derivative(Vector velocity, Vector acceleration)
{
this.dx = velocity;
this.dv = acceleration;
}
}
```

A.4.2 Matrix.java

```
package util;

//a 4x4 matrix class that should do pretty much everything useful
//convention is row-col ordering

public class Matrix
{
public double m11, m12, m13, m14, m21, m22, m23, m24, m31, m32, m33, m34, m41, m42, m43, m44;

public Matrix()
{

}

// construct a matrix from explicit values for the 3x3 sub matrix.
// note: the rest of the matrix (row 4 and column 4 are set to identity)

public Matrix(double m11, double m12, double m13, double m21, double m22, double m23, double m31, double m32, double m33)
{
this.m11 = m11;
this.m12 = m12;
this.m13 = m13;
this.m14 = 0;
this.m21 = m21;
this.m22 = m22;
this.m23 = m23;
this.m24 = 0;
this.m31 = m31;
this.m32 = m32;
this.m33 = m33;
this.m34 = 0;
this.m41 = 0;
this.m42 = 0;
this.m43 = 0;
this.m44 = 1;
}

// construct a matrix from explicit entry values for the whole 4x4 matrix.
```

```
public Matrix(double m11, double m12, double m13, double m14,
double m21, double m22, double m23, double m24,
double m31, double m32, double m33, double m34,
double m41, double m42, double m43, double m44)
{
this.m11 = m11;
this.m12 = m12;
this.m13 = m13;
this.m14 = m14;
this.m21 = m21;
this.m22 = m22;
this.m23 = m23;
this.m24 = m24;
this.m31 = m31;
this.m32 = m32;
this.m33 = m33;
this.m34 = m34;
this.m41 = m41;
this.m42 = m42;
this.m43 = m43;
this.m44 = m44;
}

public void setToZero()
{
m11 = 0;
m12 = 0;
m13 = 0;
m14 = 0;
m21 = 0;
m22 = 0;
m23 = 0;
m24 = 0;
m31 = 0;
m32 = 0;
m33 = 0;
m34 = 0;
m41 = 0;
m42 = 0;
m43 = 0;
m44 = 0;
}

// set matrix to identity.

public void setToIdentity()
{
m11 = 1;
m12 = 0;
m13 = 0;
m14 = 0;
m21 = 0;
m22 = 1;
m23 = 0;
m24 = 0;
m31 = 0;
m32 = 0;
m33 = 1;
m34 = 0;
m41 = 0;
m42 = 0;
m43 = 0;
m44 = 1;
}

// set to a translation matrix.

public void setAsTranslation(Vector v)
{
m11 = 1;   // 1 0 0 x
m12 = 0;   // 0 1 0 y
m13 = 0;   // 0 0 1 z
m14 = v.x;      // 0 0 0 1
m21 = 0;
m22 = 1;
m23 = 0;
m24 = v.y;
m31 = 0;
m32 = 0;
m33 = 1;
m34 = v.z;
m41 = 0;
m42 = 0;
m43 = 0;
m44 = 1;
}

// calculate determinant of 3x3 sub matrix.

public double determinant()
{
return -m13*m22*m31 + m12*m23*m31 + m13*m21*m32 - m11*m23*m32 - m12*m21*m33 + m11*m22*m33;
}

// determine if matrix is invertible.
```

```
// note: currently only checks 3x3 sub matrix determinant.

public boolean invertible()
{
return (this.determinant() != 0);
}

// calculate inverse of matrix

public Matrix inverse()
{
Matrix returnMat = new Matrix();
double determinant = this.determinant();

if(invertible())
{

double k = 1.0f / determinant;

returnMat.m11 = (m22*m33 - m32*m23) * k;
returnMat.m12 = (m32*m13 - m12*m33) * k;
returnMat.m13 = (m12*m23 - m22*m13) * k;
returnMat.m21 = (m23*m31 - m33*m21) * k;
returnMat.m22 = (m33*m11 - m13*m31) * k;
returnMat.m23 = (m13*m21 - m23*m11) * k;
returnMat.m31 = (m21*m32 - m31*m22) * k;
returnMat.m32 = (m31*m12 - m11*m32) * k;
returnMat.m33 = (m11*m22 - m21*m12) * k;

returnMat.m14 = -(returnMat.m11*m14 + returnMat.m12*m24 + returnMat.m13*m34);
returnMat.m24 = -(returnMat.m21*m14 + returnMat.m22*m24 + returnMat.m23*m34);
returnMat.m34 = -(returnMat.m31*m14 + returnMat.m32*m24 + returnMat.m33*m34);

returnMat.m41 = m41;
returnMat.m42 = m42;
returnMat.m43 = m43;
returnMat.m44 = m44;

return returnMat;
}

else
{
returnMat.setToIdentity();
//NOTE!!!
//WARNING WARNING WARNING!!!
//THIS IS BAD!!
//SHOULD THROW AN ERROR BACK TO THE THING THAT CALLS IT
return returnMat;
}

}


// calculate transpose of matrix and write to parameter matrix.

public Matrix transpose()
{
Matrix transpose = new Matrix();
transpose.m11 = m11;
transpose.m12 = m21;
transpose.m13 = m31;
transpose.m14 = m41;
transpose.m21 = m12;
transpose.m22 = m22;
transpose.m23 = m32;
transpose.m24 = m42;
transpose.m31 = m13;
transpose.m32 = m23;
transpose.m33 = m33;
transpose.m34 = m43;
transpose.m41 = m14;
transpose.m42 = m24;
transpose.m43 = m34;
transpose.m44 = m44;
return transpose;
}

// add another matrix to this matrix.

public Matrix add(Matrix o)
{
return new Matrix(m11 + o.m11, m12 + o.m12, m13 + o.m13, m14 + o.m14,
m21 + o.m21, m22 + o.m22, m23 + o.m23, m24 + o.m24,
m31 + o.m31, m32 + o.m32, m33 + o.m33, m34 + o.m34,
m41 + o.m41, m42 + o.m42, m43 + o.m43, m44 + o.m44);

}

// subtract a matrix from this matrix.

public Matrix subtract(Matrix o)
{
return new Matrix(m11 - o.m11, m12 - o.m12, m13 - o.m13, m14 - o.m14,
m21 - o.m21, m22 - o.m22, m23 - o.m23, m24 - o.m24,
```

```
m31 - o.m31, m32 - o.m32, m33 - o.m33, m34 - o.m34,
m41 - o.m41, m42 - o.m42, m43 - o.m43, m44 - o.m44);

}

// multiply this matrix by a scalar.

public Matrix scale(double s)
{
return new Matrix(m11*s, m12*s, m13*s, m14*s,
m21*s, m22*s, m23*s, m24*s,
m31*s, m32*s, m33*s, m34*s,
m41*s, m42*s, m43*s, m44*s);
}

// matrix times matrix

public Matrix mtm(Matrix matrix)
{
Matrix result = new Matrix();
result.m11 = m11*matrix.m11 + m12*matrix.m21 + m13*matrix.m31 + m14*matrix.m41;
result.m12 = m11*matrix.m12 + m12*matrix.m22 + m13*matrix.m32 + m14*matrix.m42;
result.m13 = m11*matrix.m13 + m12*matrix.m23 + m13*matrix.m33 + m14*matrix.m43;
result.m14 = m11*matrix.m14 + m12*matrix.m24 + m13*matrix.m34 + m14*matrix.m44;
result.m21 = m21*matrix.m11 + m22*matrix.m21 + m23*matrix.m31 + m24*matrix.m41;
result.m22 = m21*matrix.m12 + m22*matrix.m22 + m23*matrix.m32 + m24*matrix.m42;
result.m23 = m21*matrix.m13 + m22*matrix.m23 + m23*matrix.m33 + m24*matrix.m43;
result.m24 = m21*matrix.m14 + m22*matrix.m24 + m23*matrix.m34 + m24*matrix.m44;
result.m31 = m31*matrix.m11 + m32*matrix.m21 + m33*matrix.m31 + m34*matrix.m41;
result.m32 = m31*matrix.m12 + m32*matrix.m22 + m33*matrix.m32 + m34*matrix.m42;
result.m33 = m31*matrix.m13 + m32*matrix.m23 + m33*matrix.m33 + m34*matrix.m43;
result.m34 = m31*matrix.m14 + m32*matrix.m24 + m33*matrix.m34 + m34*matrix.m44;
result.m41 = m41*matrix.m11 + m42*matrix.m21 + m43*matrix.m31 + m44*matrix.m41;
result.m42 = m41*matrix.m12 + m42*matrix.m22 + m43*matrix.m32 + m44*matrix.m42;
result.m43 = m41*matrix.m13 + m42*matrix.m23 + m43*matrix.m33 + m44*matrix.m43;
result.m44 = m41*matrix.m14 + m42*matrix.m24 + m43*matrix.m34 + m44*matrix.m44;
return result;
}

//vector transformation (used in the case of inertia-tensor times angular momentum? might be useful at some point?)

public Vector transform(Vector v)
{

double rx = v.x * m11 + v.y * m12 + v.z * m13 + m14;
double ry = v.x * m21 + v.y * m22 + v.z * m23 + m24;
double rz = v.x * m31 + v.y * m32 + v.z * m33 + m34;
Vector returnVec = new Vector(rx, ry, rz);
return returnVec;
}
}
```

A.4.3 State.java

```
/* The data structure for the numerical integrator.
 * Every pointCharge has a state, which contains the variables listed below.
 * Velocity (as a secondary variable) is not dealt with directly, but rather recomputed from momentum.
 * I'm anticipating a need to do rendertime-based interpolation, so I provide a method for alpha-based linear interpolation.
 */

package util;

import physics.PointCharge;

public class State {
// primary
public Vector position;
public Vector momentum;
public Vector efield; //the value of the electric field at this position
public PointCharge touching;
public Boolean touchingOther;

//secondary
public Vector velocity;

// constant
public double radius;
public double charge;
public double mass;
public double inverseMass;


public State() {}

public State(double charge, double mass, double radius)
{
this.position = new Vector();
this.momentum = new Vector();
this.charge = charge;
this.mass = mass;
this.radius = radius;
this.inverseMass = 1 / mass;
this.velocity = momentum.scale(inverseMass);
this.touchingOther = false;
```

```
this.touching = null;
}

// interpolation used for animating inbetween states
public State interpolate(State a, State b, double alpha)
{
State interpolatedState = b;
interpolatedState.position = a.position.scale(1 - alpha).add(b.position.scale(alpha));
interpolatedState.momentum = a.momentum.scale(1 - alpha).add(b.momentum.scale(alpha));
interpolatedState.efield = a.efield.scale(1 - alpha).add(b.efield.scale(alpha));
return interpolatedState;
}

public void recalc()
{
velocity = momentum.scale(inverseMass);
}
}
```

A.4.4 Vector.java

```
//Basic Vector class. Does pretty much everything you'd want a vector to do.

package util;

public class Vector
{
public double x, y, z;

public Vector()
{
}

public Vector(double x, double y, double z)
{
this.x = x;
this.y = y;
this.z = z;
}

public void zero()
{
x = 0;
y = 0;
z = 0;
}

public Vector add(Vector o) {
return new Vector(x + o.x, y + o.y, z + o.z);
}

public Vector cross(Vector o) {
return new Vector(y * o.z - z * o.y, z*o.x - x*o.z, x*o.y - y*o.x);
}

public double dot(Vector o) {
return x * o.x + y * o.y + x * o.z;
}

public double length() {
return (double)Math.sqrt(x * x + y * y + z * z);
}


public Vector normalize() {
double length = this.length();
return new Vector(x / length, y / length, z / length);
}

public Vector scale(double scalar) {
return new Vector(scalar * x, scalar * y, scalar * z);
}

public Vector subtract(Vector o) {
return new Vector(x - o.x, y - o.y, z - o.z);
}

public String toString()
{
return "<" + x + ", " + y + ", " + z + ">";
}
}
```