Modeling Solar Systems

AiS Challenge
Final Report
4-3-2002

Team 009
Albuquerque Academy

Team Members:

Shaun Ballou
Paul Boyle
Joshua Langsfeld
Ryan McGowan
Matt Strange

Sponsoring Teacher:

Jim Mims

Mentors:

Jim Mims

Table of Contents:

Executive Summary:

This year, our super-computing team chose to do a project involving the simulation of solar systems. More specifically, we wanted to make a program that could take the positions and masses and velocities of some planets, and by calculating their gravitational influences, find out where they would move. In the end, we wanted to simulate an entire solar system, and find out how long it took to fall apart. To turn this idea into a reality, we needed to use basic physics equations, since none of our team members were familiar with Calculus. Our final solution was to write a program in C++ that utilizes Newton's Law of Gravity and a few simple equations of motion.

The program uses the abstraction that the solar system is a large, three-dimensional, Cartesian grid. We used simple vector addition to find the changes in acceleration between planets, and from acceleration followed the change in velocity and then change in position. We were able to successfully implement this method in our program and ended up with a powerful application that could accurately simulate planetary trajectories in a more or less curved manner.

To display our data, we began by using the GnuPlot program to create three-dimensional graphs. These graphs provided a view of the planets' trajectories in linear format, and showed how volatile or stable the solar system were. Our team, however, was aiming to have a graphical representation of the solar system, which would also account for the factor of time. Our solution to this problem (also achieving the title goal of our project) was to create a 3D video that would show the simulation as it could actually be witnessed in space. Using a free 3D modeler and a

script written in the Python language, we were able to go from a linear graph to a three-dimensional video.

Problem Statement:

There are nine planets in our solar system, each orbiting the sun in an elliptical fashion. This may not seem like a lot to think about, but as far as science goes, this raises several important questions. Why don't the planets all shoot off into space or crash into the sun? Gravity pulls them towards the sun; inertia pushes them out of the solar system. The two forces, combined, result in an elliptical orbit typical of planets. But how balanced do the forces have to be? Must they exist in exact proportions to create such an orbit, or is there a broad range of relative values that will result in a stable orbit? That's more difficult to answer without a lot of math. Do the planets really have much effect on each other at all? Definitely, but exactly how much is hard to tell exactly. Do satellites affect the orbits of the planets they belong to? Maybe, but probably not very much. In order to answer these and other questions completely, much experimentation would be needed. Experiments using actual bodies is impractical, to say the least. However, by creating a computer simulation of a solar system, we could easily test the effects of various things on a solar system's stability by changing a few variables.

However, the problem is not simply that of determining the interactions of multiple bodies in space. Programming the simulation itself presents some problems. First, we need equations. How do we mathematically determine the gravitational forces between bodies? How can we combine a body's velocity and the gravitational effects of other bodies to determine which direction the body will go and at what speed?

Next, we need programming skills. How can we implement the equations and calculations in a computer program? Which implementation is the fastest or the most accurate? What is the best way to convey the results to the user of the program?

Description of Method:

Three and a half centuries ago, Isaac Newton developed a way to describe the motion of planets in our solar system. To accurately describe the movement of these bodies, he created a form of math, called Calculus, which could integrate planetary motions with no error. Since no one on our team knows Calculus, we were forced to come up with another method of implementing our project. We chose, instead, to do it the way Newton did in his first masterpiece, before Calculus. He made use of vectors to describe the instantaneous accelerations and velocities. This is exactly what we did.

We made a vector class to serve as a medium for measuring the directions and the magnitudes of the velocities of the planets. We utilized this object in what was probably the most important function of our entire code, *calculateAforce* (see Appendix A). In this function, we turned Newton's Law of Universal Gravitation into C++. Given another *spacebody* object, the function returns a vector from the body the function was called on, that represents the acceleration force the planet will feel towards the planet that was passed as a parameter. Therefore, there were many vectors for each planet, each pointing to the other planets in the simulation, representing where the planet wanted to go.

Of course, a planet cannot move in five directions at once, so all the vectors pointing to other planets had to summed to make the *real* vector that represented the true path of the planet. When we had found this vector, we could change it from acceleration to a velocity vector, and move the planet along that path for the certain time period allotted.

To record the data that this program creates, we make one file for each planet. Each line of the file contains 3 things: the x, y, and z coordinates of the planet after each time step. We could use these files later to make graphical representations.

We had 2 methods of representing our systems graphically. The first was the graphics program recommended to use by the CTG, Gnuplot. We were able to plot the individual positions of the planets as dots and when they were connected by lines, it formed a rough sketch of the planets path. The more data coordinates that we had in the file, the "smoother" the path of the planet looked.

The other method we used was another free 3d modeling program called Blender. Blender, in conjunction with a scripting language called Python, could take the data files created by the driver program, and directly export the coordinates into frames. When the frames were compiled, it produced a complete .avi movie of whatever simulation had been run.

Results:

After designing and debugging our simulation program and graphical script, we were able to view complex simulations of planets as they would actually be seen if we were to view them in space for an extended period of time. This gave us the extraordinary possibility to witness interplanetary gravitational reactions and therefore make some conclusions as to what factors need to be present in a stable and life-sustaining solar system. We found that above all other factors, the differences in mass caused the most important and violent changes in the solar system. Observe the two dimensional graph *Figure 1*:

Notice the level of each planet's reactivity. The least reactive is the Red planet, with its great mass of 2×10^{17} kilograms. While slightly influenced by the other two planets, the Red planet still maintains a rather simple parabolic trajectory, and eventually moves out and away from the other two planets. The Green planet has a

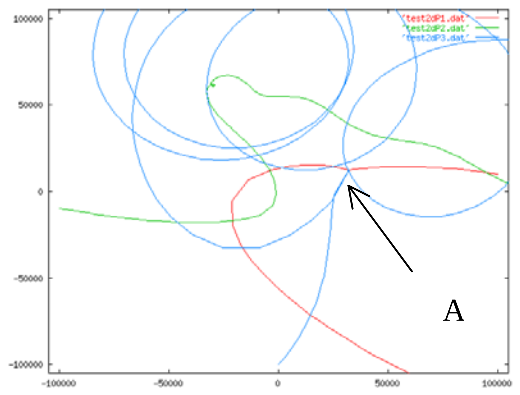
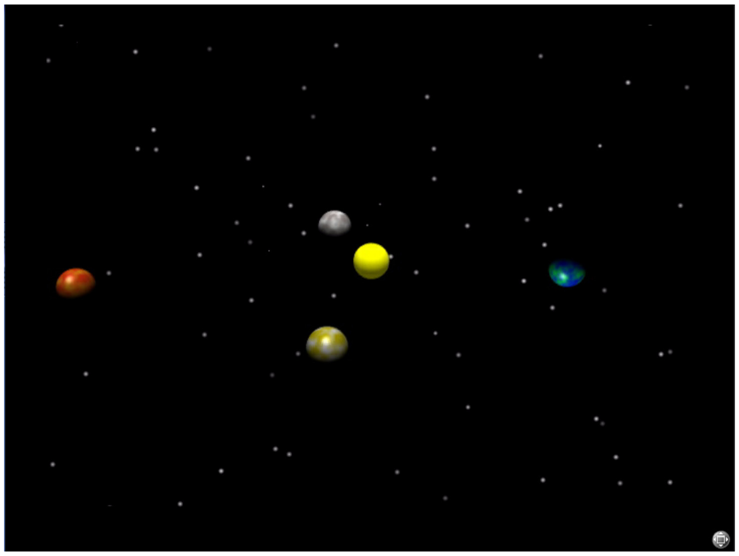


Fig 1. Simulation 3 – A 2D model of three planets with different masses starting in the left, bottom, and right borders

mass of 1 x 10^{17} kilograms. This planet has the second most stable trajectory, drawn towards the Red planet the majority of the time, but also assuming a few minor bumps in its path due to the gravitational pull of the Blue planet. The Blue planet has the smallest mass, 1 x 10^{16} kilograms, and consequently, the most active trajectory. Observe point A. Here one can see the 'slingshot' motion of the Blue planet around the Red one. Due to the extreme difference in mass, the Blue planet makes a very tight turn around the Red planet; so tight, in fact, that it appears to be a straight ricocheted line. After this, (as better demonstrated by the video) the Blue planet has a moving orbit around the moving Green planet until the simulation ends.

After experimenting with simple gravitational interactions, we moved on to the move complex ones, including an orbit. An orbit has to be set up nearly perfectly to avoid either falling inwards to the central star, or flying away altogether. Using some well-known formulas for orbital velocities and positions, we made a quick sim of a single planet around a star. To proceed farther in this direction, we had to look in our own backyard. After a quick search on the Internet, we were able to find reasonably accurate data on our own solar system. After a number of trials, we were finally able to simulate the entire thing (see Appendix B).

We also ended up with some graphical representations of the solar systems. There was Gnuplot pictures (see Appendix B) as well as some Blender movies. A screenshot of one of these movies is displayed here:



Here is an example of what our Blender movies look like. This is a model of our solar system out to Mars. As you can see, the sizes of the planets are grossly distorted in order to see everything better. The .avi movies

Through the many tests we conducted, we were able to make some important conclusions regarding the nature of solar systems.

Conclusions:

In our latest test we were able to simulate our entire solar system for 20 million years. This told us a lot about solar systems. Our program had shown very little deviation in the orbits of the nine planets, even after such an extended period of time. It really showed to us the importance of distance as well as mass in these examples. For example, the planets in the example in the Results above were interacting so much because they were so close together (no more than 100 km). At that distance, even the (relatively) small masses of the planets were enough to be overwhelming. In our solar system, however, it is not a matter of hundreds of kilometers, but of millions of kilometers. When the planets are that far away from each other, they practically only feel the pull of the sun. Therefore, the secondary motions that planets feel from their neighbors are practically negligible. We also have another argument for solar systems being stable and static very often. When we first created a simulation involving an orbit, we made a rough estimate as to where the planet should go, and what speed it should have, based on a few equations. But when we ran the simulation, it formed an obit almost instantly. Either we got really lucky, or making an orbit is not so difficult after all.

This data from our program combined with what we had already known gave us the ability to make a generalization of all solar systems. They are very stable when the planets are far enough apart, and have well-defined orbits. This means that it will not break apart until very far in the future, no sooner than the time when stellar evolution is able to take over.

Our Best Achievement:

The most original accomplishment of our project would have to be using graphics. Our computer science course during school does not cover graphics; thus we were forced to find ways of doing it on our own. Graphics were practically necessary for our success of the project. We wanted to accurately display the planets and other celestial bodies at the various time frames through the demonstration. In order to do this, we somehow needed to create a movie file with in the program. This was accomplished by using a series of programs: C++, Blender, and Python. We used C++ to start off the Python script and to display the variables of the model. Blender is a program designed for any type of movie graphics. It has many lighting and angle capabilities, as well as a three-dimensional coordinate system. Python is a scripting language that can export variables to Blender. We used our Python script for editing the coordinates of the objects (in this case, planets) in-between every frame of the Blender movie file. The script would take coordinates of the planets and other variables from Blender, send them through our math model, and send them back to Blender. Blender would then make a new frame with the updated coordinates and other variables of the planets. Finally, Blender would export the compiled movie file back to C++. Most of our math modeling took place in the Python script. As previously stated, we had no instruction on graphics previous to the project. Since this complex system to obtain a movie file of the model solar system was of our own creation, we can proudly say the our graphics are our greatest achievement of this project.

References and Acknowledgements:

Delaney, Wesley. Physics Information. Accessed: 23 Oct 2001
http://www.geocities.com/wesleydelaney/physics.html

Formulas for Rotational Motion. Accessed: 23 Oct 2001
http://cougar.slvhs.slv.k12.ca.us/~pboomer/physicslectures/ch5formulas.html

Formulas for Acceleration & Newton's Laws. Accessed: 23 Oct 2001
http://cougar.slvhs.slv.k12.ca.us/~pboomer/physicslectures/ch3formulas.html

Geodetic Reference System 1980. Accessed: 23 Oct 2001
http://www.gfy.ku.dk/~iag/HB2000/part4/grs80_corr.htm

Case Van Horsen case@ironwater.com via Usenet comp.lang.python
http://www.blenderbuch.de/tutor/python1/Python1.pdf

Software used:

Gnuplot
http://www.challenge.nm.org/ctg/graphics/gnuplot.shtml

Blender
http://www.blender.nl

Special Thanks to:

Zach Labry

Appendix A
(Our Code)

```
//Team 009 - 3d Driver Program

//File inputs are like so:
//(# of planets) (time-step) (# of seconds to run sim) (increment to record to file)
//Remainder is individual planet data: 1 line per planet
//(planet mass) (radius) (velocity X) (Y) (Z) (location X) (Y) (Z)

//Program will make a file for each planet in the same directory this code is in
//under 'planetA.dat', 'planetB.dat', etc...

#include <iostream.h>
#include <fstream.h>
#include "apstring.h"
#include "apvector.h"
#include "spacebody3d.h"
#include "vector3d.h"
#include "location3d.h"

int main()
{
        int i, dt, ftime;
        double mass, rad, X, Y, Z, time;
        apstring file;
        ifstream readData;
        cout << "Enter file to read data from: ";
        getline(cin, file);
        readData.open(file.c_str());
        readData >> i >> dt >> time >> ftime;
        apvector <spacebody> solarSystem(i);
        vector tempVel;
        location tempLoc;

        for (int j=0, k=0; j < i; j++)
        {
                readData >> mass >> rad;
                readData >> X >> Y >> Z;
                tempVel.setvector(X,Y,Z);
                readData >> X >> Y >> Z;
                tempLoc.setlocation(X,Y,Z);
                solarSystem[j].setObject(mass, rad, tempVel, tempLoc);
        }

        //vector testacc = solarSystem[1].calculateAforce(solarSystem[0]);
        //cout << testacc.x() << " " << testacc.y() << " " << testacc.z() << endl;

        apvector <ofstream> dataFiles(i);
        apstring str1("planet"), str2(".dat");
        for (j=0; j < i; j++)
                dataFiles[j].open((str1 + char(j + 65) + str2).c_str());

        for (j=0; j < i; j++)
                dataFiles[j] << solarSystem[j].position.x() << '\t' << solarSystem[j].position.y() << '\t' <<
solarSystem[j].position.z() << endl;

        int etime = 0;
        while (etime < time)
```

```
	{
		for (k=1; k < i; k++)
		{
			for (j=0; j < i; j++)
			{
				if (j == k)
					continue;
				else
					solarSystem[k].velocity +=
(solarSystem[k].calculateAforce(solarSystem[j]) * dt);
			}
		}
		for (j=0; j < i; j++)
		{
			solarSystem[j].position.setlocation(solarSystem[j].velocity.x() * dt +
solarSystem[j].position.x(),
			solarSystem[j].velocity.y() * dt + solarSystem[j].position.y(),
			solarSystem[j].velocity.z() * dt + solarSystem[j].position.z());
		}

		if (etime % ftime == 0)
		{
			for (j=0; j < i; j++)
				dataFiles[j] << solarSystem[j].position.x() << '\t' <<
solarSystem[j].position.y() << '\t' << solarSystem[j].position.z() << endl;
		}
		etime += 1;
	}
	return 0;
}


//location.h

#ifndef LOCATION_H

#define LOCATION_H

class location
{
public:
      location();
      location(double, double, double);
      ~location();

      double x() const;
		double y() const;
		double z() const;

		void setlocation(double, double, double);
      const location& operator = (const location &);
private:
      double itsX;
		double itsY;
		double itsZ;
};
```

```
location::location(void)
{
	itsX = 0;
			itsY = 0;
			itsZ = 0;
}

location::location(double x, double y, double z)
{
	itsX = x;
			itsY = y;
			itsZ = z;
}

location::~location()
{
}

double location::x(void) const
{
	return itsX;
}

double location::y(void) const
{
		return itsY;
}

double location::z(void) const
{
		return itsZ;
}

void location::setlocation(double x, double y, double z)
{
	itsX = x;
			itsY = y;
			itsZ = z;
}

const location& location::operator = (const location &a)
{
	itsX = a.x();
			itsY = a.y();
			itsZ = a.z();
			return *this;
}

#endif


//spacebody.h

#ifndef SPACEBODY_H
```

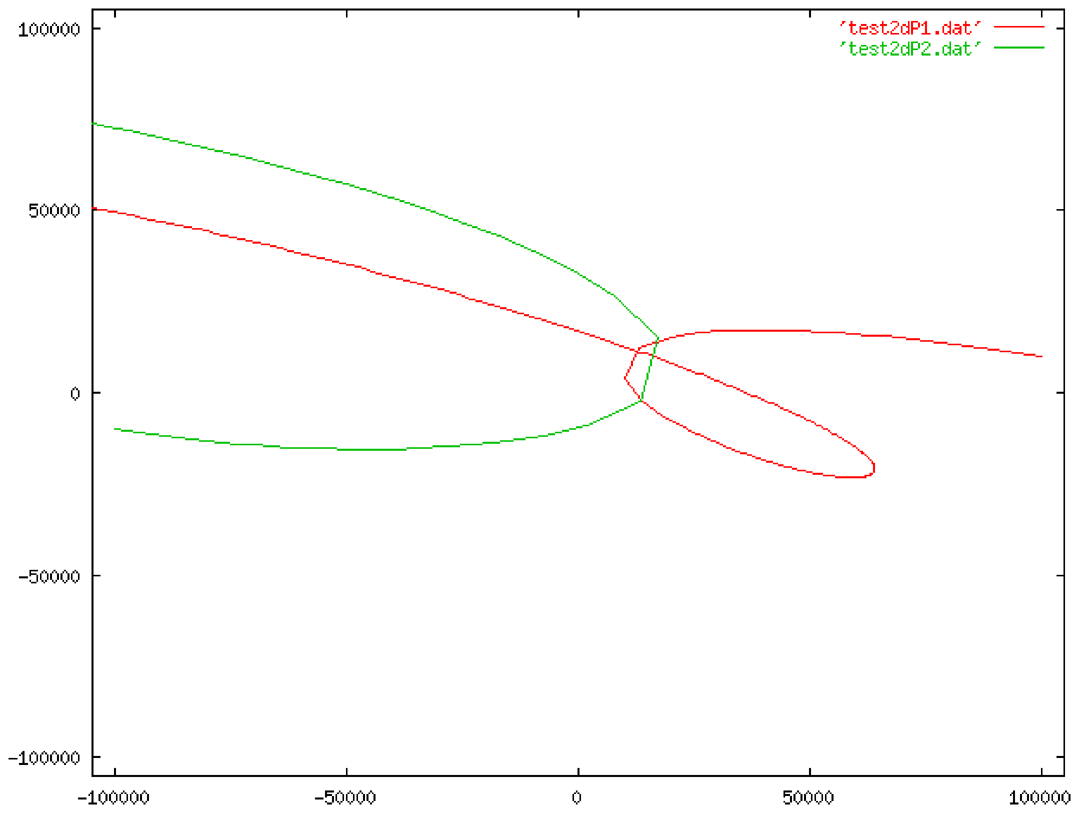
```
#define SPACEBODY_H


#include "vector3d.h"
#include "location3d.h"
#include <math.h>

class spacebody
{
public:
        spacebody();
        spacebody(double, double, vector, location);
        ~spacebody();

        double mass();
        double radius();
        vector velocity;
        location position;

                        void setObject(double, double, vector, location);

        vector calculateAforce(spacebody);
private:
        double itsMass;
        double itsRadius;
};

spacebody::spacebody(void)
{
        itsMass = 0;
        itsRadius = 0;
        velocity.setvector(0,0,0);
        position.setlocation(0,0,0);
}

spacebody::spacebody(double mass, double rad, vector a, location b)
{
        itsMass = mass;
        itsRadius = rad;
        velocity = a;
        position = b;
}

spacebody::~spacebody()
{
}

void spacebody::setObject(double mass, double rad, vector a, location b)
{
        itsMass = mass;
        itsRadius = rad;
        velocity = a;
        position = b;
}

double spacebody::mass(void)
```

```
{
        return itsMass;
}

double spacebody::radius(void)
{
        return itsRadius;
}

vector spacebody::calculateAforce(spacebody a)
{
             double G = 6.6726E-11;
             double x = (a.position).x() - position.x();
             double y = (a.position).y() - position.y();
             double z = (a.position).z() - position.z();
             double d = sqrt(x*x + y*y + z*z);
             double A = G * a.mass() / (d*d);
             return vector((A/d)*x, (A/d)*y, (A/d)*z);
}

#endif

//vector.h

#ifndef VECTOR_H

#define VECTOR_H

#include <math.h>

class vector
{
public:
        vector();
        vector(double, double, double);
        ~vector();

        double x() const;
                          double y() const;
                          double z() const;
        double magnitude() const;

        void setvector(double, double, double);
        const vector& operator = (const vector &);
        const vector& operator += (const vector &);
        vector operator + (const vector &);
                          const vector operator * (double);
private:
        double itsX;
                          double itsY;
                          double itsZ;
        double itsMagnitude;
};

vector::vector(void)
{
        itsX = 0;
                        itsY = 0;
                        itsZ = 0;
        itsMagnitude = 0;
}

vector::vector(double x, double y, double z)
{
        itsX = x;
                        itsY = y;
                        itsZ = z;
        itsMagnitude = sqrt((x*x)+(y*y)+(z*z));
}

vector::~vector()
{
}

double vector::x(void) const
{
        return itsX;
}

double vector::y(void) const
{
             return itsY;
}

double vector::z(void) const
{
             return itsZ;
}

double vector::magnitude(void) const
{
        return itsMagnitude;
}

void vector::setvector(double x, double y, double z)
{
        itsX = x;
                        itsY = y;
                        itsZ = z;
        itsMagnitude = sqrt((x*x)+(y*y)+(z*z));
}

const vector& vector::operator = (const vector &a)
{
        itsX = a.x();
                        itsY = a.y();
                        itsZ = a.z();
                        itsMagnitude = a.magnitude();
                        return (*this);
}

const vector& vector::operator += (const vector &a)
```

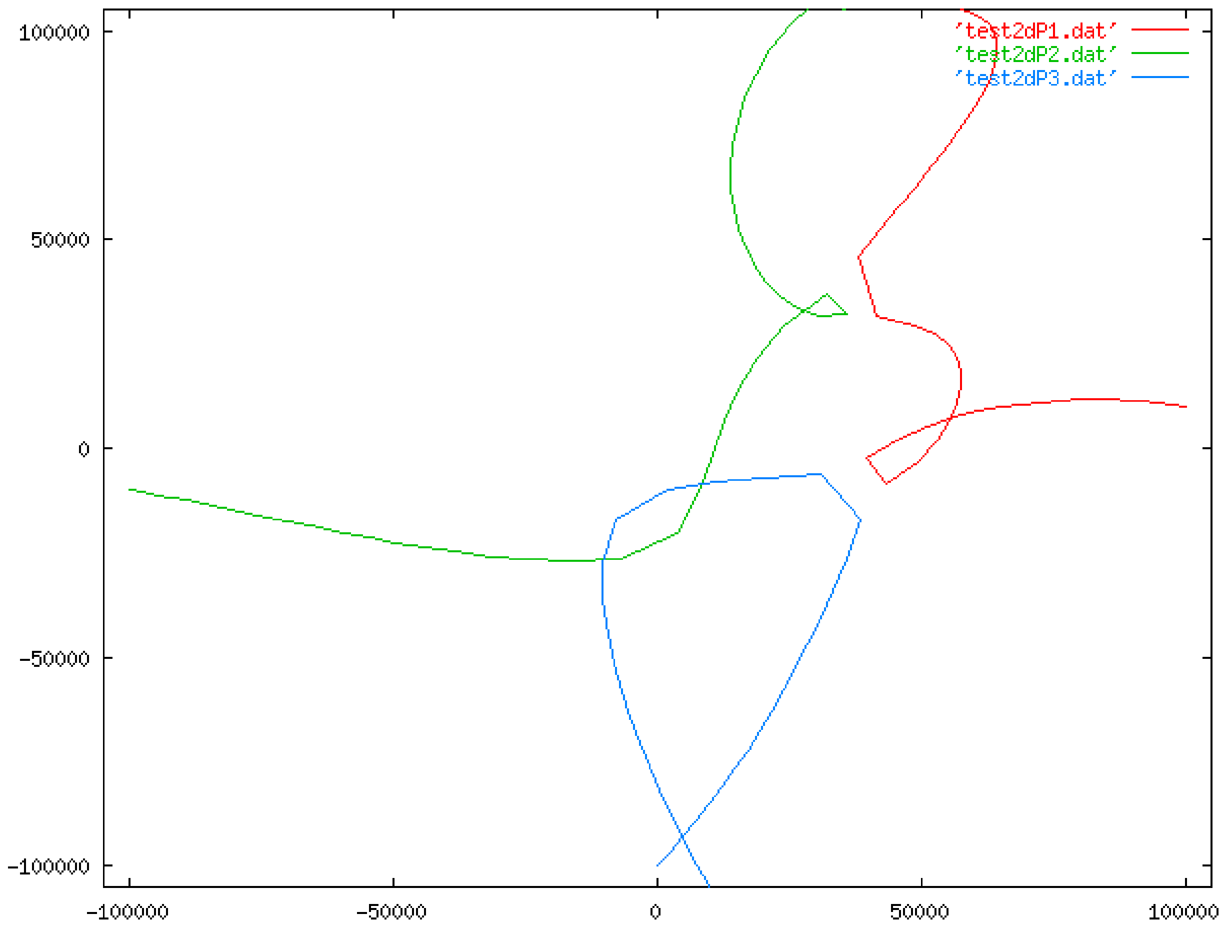
```
{
   setvector(itsX + a.x(), itsY + a.y(), itsZ + a.z());
   return *this;
}

vector vector::operator + (const vector &rhs)
{
      vector temp(itsX + rhs.x(), itsY + rhs.y(), itsZ + rhs.z());
                    return temp;
}

const vector vector::operator * (double factor)
{
          vector temp(itsX * factor, itsY * factor, itsZ * factor);
          return temp;
}

#endif


//3dpscript.py

# 3dpscript.py by Matt Strange
# Version 2.4
# Now Radii come from the PlanetData.txt file and Coords are loaded dynamically
# ASCII data loaded into Lists

# Initialize #
import sys
import Blender
from array import array

# number of planets from planetdata.txt#
if Blender.Get("curframe") == 1:
          temp = map(float, open("C:\\scc\\planetdata.txt").read().split() )
          n = int(temp[0])
          print "Number of Planets: " , n
          del temp

#number of frames (x,y,z coords) we want to run #
#numframes = 1000
if Blender.Get("curframe") == 1:
          temp = map(float, open("C:\\scc\\planet0.dat").read().split() )
          numframes = (len(temp) / 3 )
          print "Number of Frames: " , numframes
          del temp

#cnt holds the position in that big coordinate list #
cnt = (Blender.Get("curframe") - 1)          #start at 0, like a list
if cnt > (numframes - 1):             #stop going at the last frame
          cnt = (numframes-1)

#get a string of length N for use with for loops #
length = range(n)

#make sure you only do this the first time #
```

```
if Blender.Get("curframe") == 1:

        #make an list of Blender Object Handles #
        obj = [1] * n

        #assign handles i.e. Planet.0 --> obj[0] #
        for counter in length:
                obj[counter] = Blender.Object.Get("Planet."+(str(counter)))

        #make an list for the radii (FLOATS) planet[x] = radii[x]#
        data = map(float, open("c:\scc\planetdata.txt").read().split())

        #apply radii to planets and skew the size
        for counter in length:

                radius = data[3 + counter*8 + 2]

                obj[counter].SizeX = (radius / 100)
                obj[counter].SizeY = (radius / 100)
                obj[counter].SizeZ = (radius / 100)

        del data

        #make a list of double arrays for each planet
        coord = [array('d')] * n

        #for every planet, load it's data into a temp list, then map it to
        #the appropriate array.
        for counter in length:
                temp = map(float, open("c:\\scc\\planet"+str(counter)+".dat").read().split())
                coord[counter] = array('d',map(float,temp))


#normal execution... #

#loop to tell which planet we're working on #
for counter in length:

        #now get the XYZ for the current frame and store it in X,Y,Z
        x = coord[counter][cnt * 3]
        y = coord[counter][(cnt * 3) + 1]
        z = coord[counter][(cnt * 3) + 2]

        #reposition the planet at this new location devided by 1000
        obj[counter].LocX = (x / 1000)
        obj[counter].LocY = (y / 1000)
        obj[counter].LocZ = (z / 1000)

sys.stdout.flush()
Blender.Redraw()
```

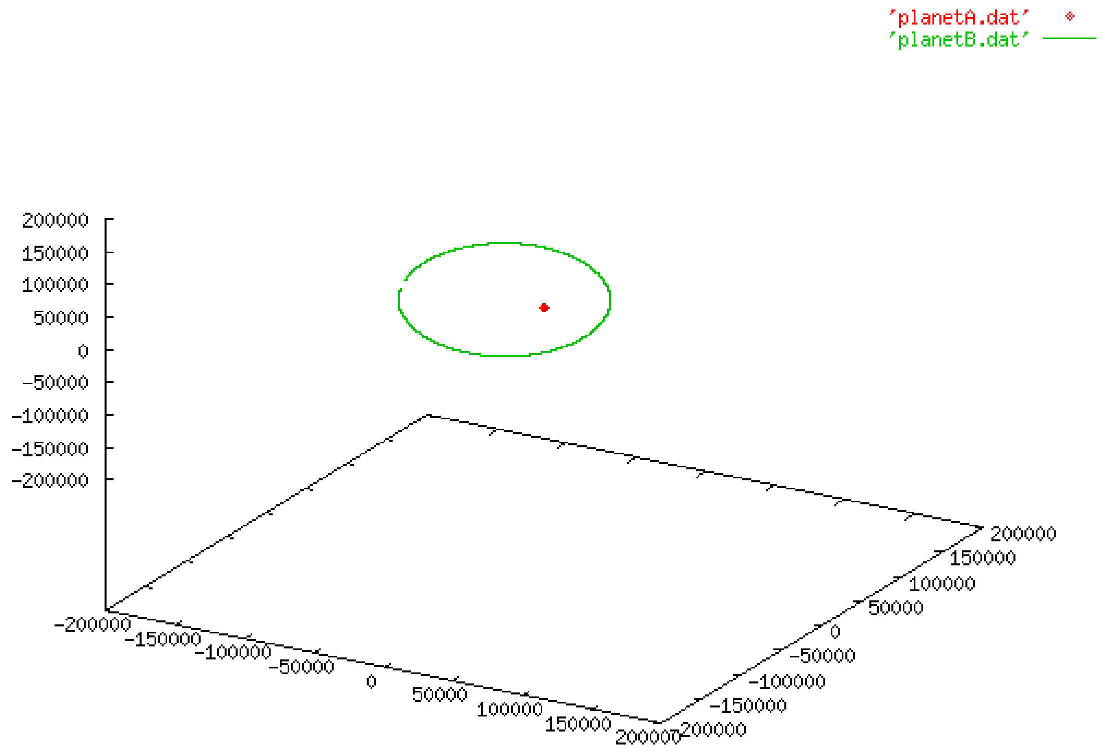
Appendix B
(Pictures)



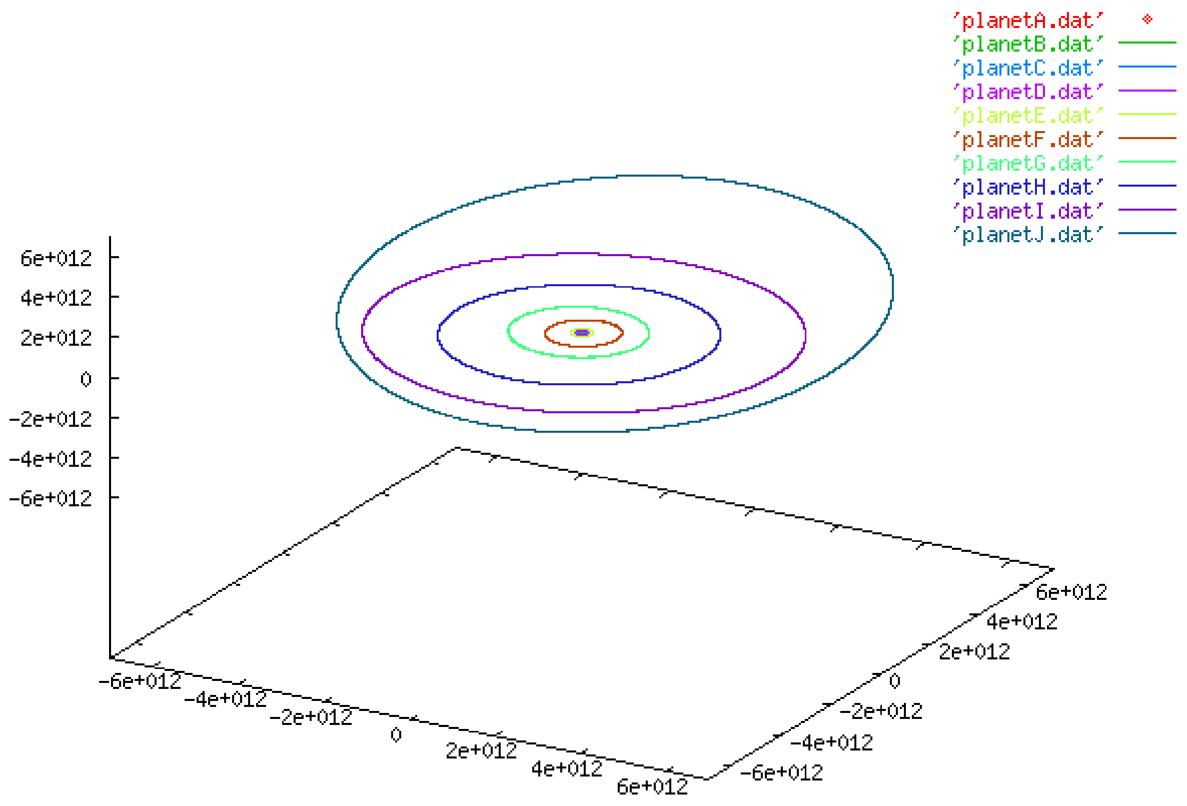
Our original two-dimensional test. Two planets of different masses are launched from the left and right sides of the grid, and the planet with the lesser mass loops around the other. Range -100 000 to 100 000.



This was our second and more advanced 2D test. We advanced to a 3-body project, and launched all the planets towards the center from the bottom, left and right sides of the graph. The extreme masses of these planets made them spiral around each other before launching into outer space. Range –100 000 to 100 000.



This 3D plot shows the results of our first orbit. Notice how the Green Planet makes an elliptical orbit around the Red Sun. Range –200 000 to 200 000.



Our master simulation: all nine planets and the sun of our solar system. Note the pitched up orbit of Pluto and the great change in scale between the inner solar system and outer solar system. Range –6e12 to 6e12.