Point of Impact

A Simulation of Temperature Changes Caused by Meteorite Impacts

New Mexico Adventures in Supercomputing Challenge

Final Report

April 3, 2003

Team 013
Albuquerque Academy

Team Members

Shaun Ballou
Paul Boyle
Bill Knoop
Ryan McGowan

Teacher

Jim Mims

Table of Contents

Executive Summary

Problem

Recently, many scientists, authors, and filmmakers have concerned themselves with the possibility of a disastrous meteorite impact. It has been suggested that a meteorite may have been the main catalyst that led to the extinction of the dinosaurs. The aim of this project is to discern how severe a threat a meteorite impact really is. In particular, we wish to focus on the effects of an impact on global oceanic temperature. If a large enough temperature change occurred in the oceans, it is conceivable that radical alterations could be made to local marine animal and plant life and global climate and temperature.

Method

We attempted to solve this problem by ramming an asteroid, with size, mass, and other characteristics defined by the user, into the Earth, which we assumed was completely covered by water to simplify calculations. We calculate the kinetic energy of the asteroid upon collision and assume that all of it becomes heat energy, which is then transferred to the nearby water. We then model the heat's flow from the impact site in order to determine the effects on global temperature and write the output to a file. This file is later used to create a graphical output that is more readable than the raw data.

Results

Our results were surprising, and at first confusing, although they make sense. Regardless of the size, velocity, impact location, and mass of the asteroid, no change was observed in local temperature at all. At first we tried a few tweaks to the program, but when we checked the computer's calculation of the initial temperature change at the impact point, we found that even the biggest asteroid raised local temperature by less than .03 degrees Celsius. Global temperature was virtually unaffected by the crash. However, over time heat from the warmer areas of the globe, like the Equator, flowed into the cooler areas such as the North and South Poles.

Conclusions

Although meteorites can be extremely massive and travel at high velocities, the energy created by an oceanic impact is not enough to affect temperature in any significant way. We hypothesize that though the energy created by the impact is tremendous, the sheer volume of water that the energy is dissipated through is too large for the energy to have any effect. At least in this regard, it seems that meteorites pose no real threat. If meteorites are really as dangerous as some think them to be, then it must be some other factor, such as debris clouds or heat transfer through the atmosphere. The flow of heat from the Equator to the Poles is probably due to the fact that there is nothing to prevent the world's oceans from becoming a uniform temperature in our simulation.

Introduction

In many ways, especially in our media, meteorites tend to be hailed as the harbingers of death. Simply look at the number of books and movies made about the subject. In most of these stories, though, the sky-dwelling menace is stopped just in time to prevent it from hitting our planet. However, what would happen if the hero or heroine failed to stop the asteroid? Numerous theories have been advanced, from tidal waves to mass extinction. Our project this year boils down to one main question: Is it really that big of a deal? We seek to model an asteroid impact in order to determine its effects on oceanic temperatures. This will allow us to determine, if only partially, the nature of the threat posed by stray celestial objects.

Although ocean temperature seems like a strange thing to choose when dealing with meteorite impacts, we chose this aspect for two main reasons.

First., ocean temperature is of some consequence. Were oceanic temperatures to be altered by something such as a meteorite, it could have drastic effects on marine life and global climate, as well as ocean current systems. It also simplifies the problem to a point where it becomes manageable. To fully model an impact and all of its resultant effects would require years of dedicated research and work, and we do not have the time, money, or manpower needed for such a project. We believe the scope that we cover in our project is small enough to be attainable, yet large enough to be significant.

Second, the project offers to help us learn more about the subject regardless of the results. If the results show that meteorites truly are a threat to human life, then we will know more about the range and severity of the damage caused by a meteor relative to its velocity, mass, and so

forth. If the results show that meteorites pose little threat with regard to ocean temperature, then that leaves us free to explore other ways in which a collision could have an impact on the viability of human life on this planet.

Description

We limited ourselves to an entirely oceanic planet without gravity, and assumed that all kinetic energy is turned into heat energy. We made these simplifications in order to make the problem more manageable and to make the simulation easier to code.

Our program runs in several steps. First, we receive input from the user concerning the meteor's velocity, direction, mass, and so on. The meteor's speed and direction are used to calculate when and where the asteroid lands. Since longitude has no real effect on oceanic temperature, we hold the longitude of the impact constant and determine only the latitude of the impact site. We represent our virtual world with a two-dimensional array, with each cell representing an area of the Earth two degrees latitude high and two degrees longitude wide. We convert the asteroid's energy to heat and raise the temperature of the grid cell representing the impact point. We then calculate the heat distribution over several timesteps. Each timestep produces a file containing the temperature of each cell in the grid. These files are then inserted into a raytracing program that creates a map of our virtual world with each point colored with respect to its temperature; for example, the coldest points are blue, and the warmest are red. This format allows us to better interpret the data output by our program. To view the exact code we wrote to achieve this goal, please refer to Appendix A.

Mathematical Model

C++

The mathematical model of our project consists of two parts: the model involving the asteroid's approach to Earth and the model involving what happens during and after impact.

Before the impact, we receive information about the meteor's initial position, velocity, and angle of movement. A Cartesian grid is used to represent the Earth and the space around it, with the origin at the center of the Earth. The meteor is placed on this grid based on its initial coordinates. The meteor's movement is then tracked over several timesteps until it strikes the Earth. The horizontal and vertical movement is determined by the functions,

$$\mathbf{X = V\cos\theta}$$
$$\mathbf{Y = V\sin\theta}$$

Where **V** is the meteor's speed and **θ** is the meteor's angle of movement. The program detects when the meteor has hit the Earth by determining the meteor's distance from Earth using the distance formula,

$$\mathbf{D = \sqrt{((y - y_1)^2 + (x - x_1)^2)}}$$

where $\mathbf{y_1}$ and $\mathbf{x_1}$ are zero because the center of the Earth, which we measure the distance from, is the origin of the coordinate system. This is then compared to the Earth's radius, which is approximately six thousand three hundred kilometers. The y-coordinate of the meteor is used to determine the latitude of the impact site. The heat energy generated by the meteor is determined by the equation

$$\mathbf{E = \tfrac{1}{2} MV^2}$$

where **E** is the total energy, **M** is the meteor's mass, and **V** is the meteor's velocity. The energy is divided by the amount of energy needed to raise the temperature of the body of water containing the impact site one degree, and get the temperature change generated by the impact. We then create a Cartesian grid to represent the world itself, and have each cell represent an area two degrees latitude high and two degrees longitude wide. The temperature values for each cell are initialized according to a function of their latitudes, which is given by

$$(3.90625 * 10^{-6})L^4 - (6.77008333 * 10^{-4})L^3 + .0359375L^2 - .97916666L + 30$$

where **L** is the latitude of the cell. We add the initial temperature change to the appropriate grid cell based on the impact site's latitude, and then distribute the heat over several timesteps. For each timestep, the heat flow in or out of a cell to neighboring cells is given by

$$.61A * (T_1 - T_2) / D$$

where A is the area of the surfaces exchanging heat, T_1 is the temperature of the cell, T_2 is the temperature of the cell that the first cell is interacting with, and D is the distance between the two cells.

POV-Ray™

The only real mathematical model involved in our graphics, which are done by POV-Ray™, a free downloadable raytracing program, is that of the equations used to determine a point's color as a function of its temperature. These are as follows:

The red value is given by

$$R(x) = \{ |\sin(5\pi x/2)| \quad (.4 \leq x \leq .6)$$

$$1 \qquad (x > .6) \}.$$

The green value is given by

$$G(x) = \{ 1 \qquad (.2 \leq x \leq .6)$$

$$|\sin(5\pi x/2)| \qquad (.6 < x < .8) \}.$$

The blue value is given by

$$B(x) = \{\ 1 \qquad (0 \le x \le .2)$$

$$\sin(5\pi x/2) \qquad (.2 < x \le .4)$$

$$\sin(5\pi x/2) \qquad (.8 \le x \le 1) \}.$$

Results

Our results were very surprising and did not come out at all like we had expected. Instead of raising the temperature of the ocean by great amounts, it instead did not raise the temperature by one degree. Even after trying various sizes and velocities, the results were the same. At first we thought or program might have had an error in it, but we manually calculated the change in the temperature and the manual results matched that of the computational results. To see the graphical output of our program which leads us to these results, see Figures 1 and 2 in the Figures section. We learned that a meteor impact of any realistic size and velocity would not change the temperature of the water at all.

We also learned several rather un-scientific but useful lessons over the course of this project. For example, we learned not to procrastinate and to perform preliminary research and calculations before choosing a project, so that we will have a better idea of what our results should be before we actually get any results. We learned that if we started earlier, we would have more time and not be quite as pressed as deadlines approach. By performing these preliminary precautions, we hope to improve our ability to complete projects like this one successfully, on time, and hopefully less stressfully.

Conclusions

It seems to be that meteorites have no real impact on local temperature, much less global temperature. While we at first thought that something was wrong, by monitoring the energy and temperature values our program produces at it runs, we performed some calculations and determined that the results actually made sense. Our virtual ocean contains over five hundred billion cubic kilometers of water. According to our calculations, it would take about four quadrillion joules of energy to raise the global ocean temperature a single degree Celsius. However, running even the largest meteor; a seventy-ton meteorite going two hundred sixty thousand kilometers per hour; only a few hundred billion joules are produced. It seems that though the energy released by the impact is enormous, it is not anywhere near enough to overcome the sheer volume of water contained in the ocean. The problem becomes compounded as water's relatively high specific heat (the energy required to change a substance's temperature) is taken into account. Thus, it makes sense that the impact has virtually no effect on ocean temperature.

As to the nature of the threat meteorites pose to human and other life with regard to oceanic temperature, it seems that we can safely say "none". Therefore, it appears that either meteorites are not actually capable of the mass destruction that is sometimes portrayed in the media or that some other aspect of the impact is responsible; perhaps the evaporated water or debris clouds created by the impact. More accurate and detailed simulations would have to be run in order to provide a more definite answer to the question.

Recommendations

Our project experienced several difficulties and limitations. In order for the effects of meteorite impacts to be more thoroughly and accurately simulated, we recommend several things. First of all, we recommend that more research be done on the equations governing conductive heat transfer through water and on the physics of the impact. We also recommend that various other aspects of the impact be implemented in order to produce a more accurate representation of the impact, such as gravity, air resistance, water evaporation and displacement, tidal waves, and three-dimensional heat transfer. Implementing land impacts would also be useful, as well as modeling the craters, shockwaves, debris clouds, and atmospheric heat transfers involved in such an impact. The composition and density of the asteroid could also be taken into account, as well as non-spherical asteroids. Overall, this project has vast potential for expansion.

Bibliography

Arnett, Bill. "Earth". http://seds.lpl.arizona.edu/nineplanets/nineplanets/earth.html. Accessed January 29, 2003; last updated October 21, 2002.

"Momentum". http://physics.bu.edu/py105/notes/Momentum.html. Accessed January 29, 2003.

Benson, Tom. "Forces on a Falling Object (with air resistance)". http://www.grc.nasa.gov/WWW/K-12/airplane/falling.html. Accessed January 29, 2003; last modified January 9, 2002.

Shelquist, Richard. "Air Density and Density Altitude Calculations". http://wahiduddin.net/calc/density_altitude.htm. Accessed January 29, 2003.

Sawyer, Donald M. "Moon Fact sheet". http://nssdc.gsfc.nasa.gov/planetary/factsheet/moonfact.html. Accessed January 29, 2003; last updated January 16, 2003.

Koehler, Kenneth R. "Friction and Drag". http://www.rwc.uc.edu/koehler/biophys/2d.html. Accessed January 30, 2003.

"How fast do meteors go?". http://www.meteorobs.org/maillist/msg21596.html. Accessed February 17, 2003.

"Geology 150". http://earth.usc.edu/~geol150/variability/deepocean.html. Accessed March 18, 2003.

"Thermal conductivity of water and sediment types". http://www-odp.tamu.edu/publications/194_IR/chap_02/c2_t6.htm. Accessed March 18, 2003.

"Meteors and Meteorites." http://www.infoplease.com/ipa/A0004503.html. Accessed March 18, 2003.

"Missions2005: The Atlantis Projects". http://web.mit.edu/12.000/www/m2005/a1/Robotics/prop.htm. Accessed March 18, 2003; last updated December 5, 2001.

Nave, R. "Heat Transfer". http://hyperphysics.phy-astr.gsu.edu/hbase/thermo/heatra.html. Accessed March 18, 2003.

"AP String Class". AP College Board.

Figures

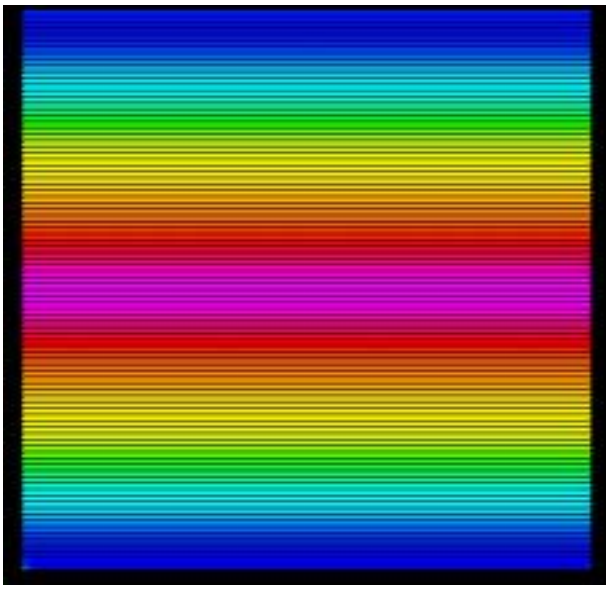

Figure 1: A map displaying heat distribution before the impact.

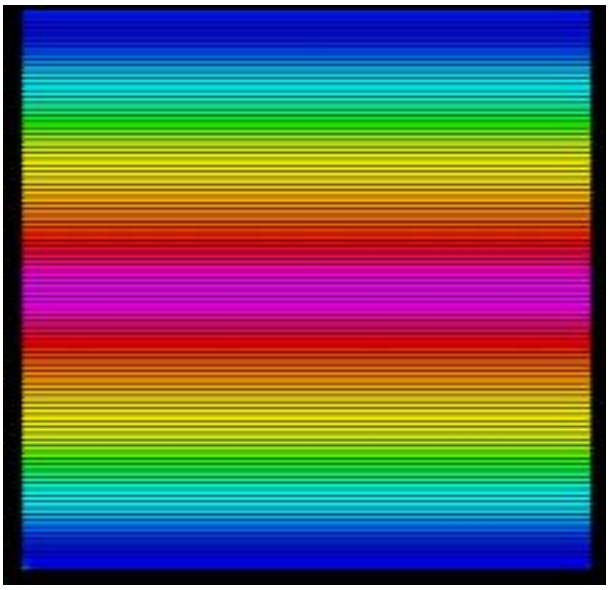

Figure 2: A map displaying heat dispersion after the impact. The impact occurred in the upper blue region.

Appendix A: Code

Source Code : pointOfImpact1.cpp

```
//Supercomputing Team 13: Albuquerque Academy
//Ryan McGowan, Bill Knoop, Shaun Ballou, Paul Boyle
//January 29, 2003

//Point Of Impact: Version 1
//Assumes a water impact and calculates heat energy released into
a system through meteorite
//impact.

#include <math.h>       //mathematic functions
#include <iostream.h>   //cerr and cin
#include <fstream.h>    //file writing functions
#include "apstring.h"   //string!
#include "world.h"      //the grid containing temperature
information
#include "meteor.h"     //contains info about the meteorite
#include "temp.h"


//Function Implementations


/*
getInfo(): gets the mass, velocity, and relative position of the
incoming object and returns
a boolean. If the function returns true, then the function
successfully completed; otherwise
there was an error.
*/


bool getInfo( double &pVelocity, double &pMass, double &pXDist,
double &pYDist, double &pAngle, double &pRadius )
{

     cerr << "Enter the initial velocity of the meteorite in
kilometers per hour.\n Most meteors travel between 40 and 260
thousand km/h.\n";
     cin >> pVelocity;

     //convert km/h to km/s...
     pVelocity /= 3600.0;

     if (pVelocity < 0)
     {
          cerr << ">>>ERROR: The meteorite's velocity must be
positive.\n";
          return false;
     }
```

```
     cerr << "Enter the initial mass of the meteorite in
kilograms.\nMeteorites can weigh from 1 to 64 000 kilograms.\n";
     cin >> pMass;

     if ( pMass < 0 )
     {
          cerr << ">>>ERROR: The meteorite must have a
nonnegative, nonzero mass.\n";
          return false;
     }
     cerr << "Enter the meteorite's initial horizontal position
relative to the\n center of the Earth.\n";
     cin >> pXDist;

     cerr << "Enter the meteorite's initial vertical position
relative to the center\n of the Earth.\n";
     cin >> pYDist;


     cerr << "Enter the meteorite's angle of movement in degrees
\nNegative angles are OK.\n";
     cin >> pAngle;

     cerr << "Enter the meteorite's radius in meters.\n";
     cin >> pRadius;

     if ( pRadius <= 0 )
     {
        cerr << ">>>ERROR: The meteorite must have a positive
radius!\n";
        return false;
     }

     pAngle = ( PI * pAngle ) / 180.0; //convert degrees to
radians

     return true;
}



/*
calculateEnergy: takes a meteor object and returns the object's
kinetic energy (to simplify the problem it is assumed
that all kinetic energy is converted to heat energy)
*/


double calculateEnergy( Meteor pMeteor )
{
     cerr << "Energy Calculated!\n"; //DEBUG
     system( "PAUSE" );
```

```
    return ( 0.5 * pMeteor.getMass() * pow( pMeteor.getVelocity(),
2 ) );
     //Kinetic Energy = .5 * mass * velocity ^ 2
}



/*
hitEarth: moves the meteor towards Earth; records the latitude at
which the meteor strikes
*/


bool hitEarth( Meteor & pMeteor, lat & pLatitude )
{
     double seconds = 0;
     while ( pMeteor.distanceFromEarth() > 0 ) //each loop is one
second
     {
          //update the meteorite's position, accounting for
gravity
          pMeteor.setPos( pMeteor.getX() + pMeteor.getVelocity()
* cos( pMeteor.getAngle() ),
                                  pMeteor.getY() +
pMeteor.getVelocity() * sin( pMeteor.getAngle() ));
          seconds++;

          if ( pMeteor.distanceFromEarth() > MOON_DISTANCE )
          {
               cerr << ">>>ERROR: The meteoroid missed the
Earth!\n\nWhile this COULD be considered a good thing, it's
difficult to collect data for a event that doesn't happen...\n";
               return false;
          }


     }
    pLatitude.setLat( pMeteor.getY() * 90 / RADIUS_EARTH );

    cerr << "The meteorite has hit Earth!\n"; //DEBUG
    system( "PAUSE" );

     return true;
}

double calcInitialHeat( double pEnergy, lat pLatitude )
{
    double amountOfWater = 0;
    amountOfWater = ( pLatitude.vertLength() / 1000.0 ) *
( pLatitude.horizLength() * 1000.0 );  //area 1 meter deep times
the size of the chunk contained within the lat & long
    amountOfWater *= 1000000; //convert to cubic centimeters of
water and multiply by the density of water (namely, 1)
```

```
    cerr << "Inital Heat Calculated!\n"; //DEBUG
    system( "PAUSE" );
    return pEnergy / ( amountOfWater * SPECIFIC_HEAT_WATER);
//divide the available energy by how many Joules are needed to
raise the temperature of the entire body of water one degree

}

void writeFile( world &pWorld , int fileNum )
{
    ofstream writer;
    double mini = pWorld.theWorld[0][0].getTemp(), maxi =
pWorld.theWorld[0][0].getTemp();
    apstring fileName = "Z:\\SCC\\Output Files\\test";
    apstring number = "";
    double youAreASTUPIDPROGRAM = 0;
    while ( fileNum > 0 )
    {
        number += char('0' + fileNum % 10);
        fileNum /= 10;
    }

    apstring r;
    r = "";

    for (int k = number.length() - 1; k >= 0; k--)
    {
        r += number[k];
    }

    fileName += r; //add number to string!
    fileName += ".poi";
    writer.open(fileName.c_str());

    for (int y = 0; y < 90; y++)
    {
        for (int x= 0; x < 180; x++)
        {
                youAreASTUPIDPROGRAM =
pWorld.theWorld[y][x].getTemp();

                if (youAreASTUPIDPROGRAM > maxi)
                     maxi = youAreASTUPIDPROGRAM;
                if (youAreASTUPIDPROGRAM < mini)
                     mini = youAreASTUPIDPROGRAM;
            writer << youAreASTUPIDPROGRAM << ",\n";
        }
    }
    writer.close();
    writer.open("Z:\\SCC\\Output Files\\scale.pos");
     writer << mini << "," << maxi;
     writer.close();
    cerr << "File " + fileName + " written...\n";
}
```

```
world distributeHeat( world &pWorld, int pTime)
{
    double initHeat[90][180];
    double tempChange[90][180];
    int left = 0;
    int right = 0;
    int top = 0;
    int bottom = 0;
    int longi = 0;

    for (int random = 0; random < 90; random++)
    {
        for (int random2 = 0; random2 < 180; random2++)
        {
            initHeat[random][random2] =
pWorld.theWorld[random][random2].getTemp();
            tempChange[random][random2] = 0.0;   //initialize the
array
        }
    }

    cerr << "Array containing temperature change values
initialized!\n"; //DEBUG

    for (int timer = 0; timer < pTime; timer++)
    {
        for (int y = 0; y < 90; y++)
        {
            for (int x = 0; x < 180; x++)
            {
                left = x - 1;
                right = x + 1;
                top = y - 1;
                bottom = y + 1;

                if (left < 0)
                    left = 179;

                if (right > 179)
                    right = 0;

                if (top < 0)
                    {
                         top = 0;
                         longi = 90 - (x - 90);
                    }
                    else
                         longi = x;

                if (bottom > 89)
                {
                         bottom = 89;
```

```
                    longi = 90 - (x - 90);
                }
                else
                    longi = x;


                tempChange[y][x] =
transferHeat( pWorld.theWorld[y][x].vertLength(), .01,
pWorld.theWorld[y][x].horizLength() *
pWorld.theWorld[y][x].vertLength(),
pWorld.theWorld[y][x].getTemp() - initHeat[y][x],
pWorld.theWorld[top][longi].getTemp() - initHeat[top][longi]);
                tempChange[y][x] +=
transferHeat( pWorld.theWorld[y][x].vertLength(), .01,
pWorld.theWorld[y][x].horizLength() *
pWorld.theWorld[y][x].vertLength(),
pWorld.theWorld[y][x].getTemp() - initHeat[y][x],
pWorld.theWorld[bottom][longi].getTemp() -
initHeat[bottom][longi]);
                tempChange[y][x] +=
transferHeat( pWorld.theWorld[y][x].horizLength(), .01,
pWorld.theWorld[y][x].horizLength() *
pWorld.theWorld[y][x].vertLength(),
pWorld.theWorld[y][x].getTemp() - initHeat[y][x],
pWorld.theWorld[y][left].getTemp() - initHeat[y][left]);
                tempChange[y][x] +=
transferHeat( pWorld.theWorld[y][x].horizLength(), .01,
pWorld.theWorld[y][x].horizLength() *
pWorld.theWorld[y][x].vertLength(),
pWorld.theWorld[y][x].getTemp() - initHeat[y][x],
pWorld.theWorld[y][right].getTemp() - initHeat[y][right]);
            }//x
        }//y

        for (int y1 = 0; y1 < 90; y1++)
        {
            for (int x1 = 0; x1 < 180; x1++)
            {

pWorld.theWorld[y1][x1].setTemp(pWorld.theWorld[y1][x1].getTemp()
+ tempChange[y1][x1] );
            }//x1
        }//y1


        writeFile(pWorld, timer );
    }//timer
    cerr << "Heat distribution over requested time completed!\n";
//DEBUG

    return pWorld;
}//distributeHeat
```

```
//the Program


int main()
{
     double vel = 0, mass = 0, xPos = 0, yPos = 0, angle = 0,
energy = 0, radius = 0;
     lat storeLat;
     world myWorld;

     if ( !getInfo( vel, mass, xPos, yPos, angle, radius ) )
     {
         system ( "PAUSE" );
          return 0;
     }

     Meteor collide( vel, mass, xPos, yPos, angle, radius );

     if ( collide.distanceFromEarth() <= 0 )
     {
          cerr << ">>>ERROR: The meteorite cannot start out
inside of the Earth!\n";
          system ( "PAUSE" );
          return 0;
     }

     if ( collide.distanceFromEarth() > MOON_DISTANCE )
     {
          cerr << ">>>ERROR: The meteorite starts too far away.
Bring it within 385 000 km of the Earth.\n";
          system ( "PAUSE" );
          return 0;
     }

     bool hits = hitEarth( collide, storeLat );

     if (!hits)
     {
        system( "PAUSE" );
        return 0;
    }

    energy = calculateEnergy( collide );
     storeLat.setTemp( calcInitialHeat( energy, storeLat ) );
//initalize world temperatures
     myWorld.theWorld[(int)storeLat.getLat()][90].setTemp(storeLa
t.getTemp());  //set the impact to the middle of the world
     writeFile(myWorld, 1000);  //write initial world to file
     cerr << "Initial Heat Stored to WorldMatrix!\n"; //DEBUG
     system("PAUSE");
     distributeHeat( myWorld, 10 );
```

```
	return 0;
}
```

Header Files

Constants.h

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

//Constants
const double MASS_EARTH = 5.972 * pow(10.0, 24.0); //the mass of
the Earth in kg, in case we need it
const double SPECIFIC_HEAT_WATER = 4.18; // specific heat of
water in J / ( g * degrees Celsius )
const double PI = 4 * atan(1.0); //our favorite mathematical
constant, defined as 4 * arctangent(1)
const double MOON_DISTANCE = 384467.0; //the moon's average
distance from Earth
const double WATER_THERMAL_CONDUCTIVITY = 0.61; //the thermal
conductivity of water
const double RADIUS_EARTH = 6378.0; //the Earth's radius

#endif
```

lat.h

```
#ifndef LAT_H
#define LAT_H

#include "constants.h"

class lat
{
public:
     lat();
     lat(double l);
     void adjustTemp(double lati);
     double getTemp();
     void setTemp(double t);
     double getLat();
     void setLat( double l );
     double horizLength();
     double vertLength();
private:
     double latitude;
     double temperature;
};

void lat::setLat( double l )
```

```
{
     latitude = l;
}
lat::lat(double l)
{
     latitude = l;
     const double RADIUS_EARTH = 6378.0; //the Earth's radius
     const double PI = 4 * atan(1.0); //our favorite mathematical
constant, defined as 4 * arctangent(1)
     temperature = 0;
}

void lat::adjustTemp(double lati)
{
     lati = abs((int)lati);
     temperature = 3.90625 * pow (10, -6) * pow(lati, 4) -
6.770833333 * pow (10, -4) * pow (lati, 3) + .0359375 * pow (lati,
2) - .97916666 * lati + 30;
}

lat::lat()
{
     temperature = 0;
     latitude = 0;
}


double lat::getTemp()
{
     return temperature;
}

void lat::setTemp(double t)
{
     temperature = t;
}

double lat::getLat()
{
     return latitude;
}

double lat::horizLength()
{
    return ( ( ( 90.0 - latitude )  / 90.0 ) * RADIUS_EARTH) /
360.0 ;
}

double lat::vertLength()
{
    return PI * RADIUS_EARTH / 180.0;
}
#endif
```

meteor.h

```
#ifndef METEOR_H
#define METEOR_H

#include "constants.h"
//a struct to represent the Meteorite
class Meteor
{
public:
     //constructors
     Meteor();
     Meteor(double pVelocity, double pMass, double pX, double pY,
double pAngle, double pRadius);
     //accessors
     double getMass();
     double getVelocity();
     double getX();
     double getY();
     double distanceFromEarth();
     double getAngle();
     double getRadius();
     //modifiers
     bool setMass( double pMass );
     bool setVelocity( double pVelocity );
     void setPos( double pX, double pY );
     bool setRadius( double pR );

private:
     double sVelocity;
     double sMass;
     double sXPos;
     double sYPos;
     double sAngle;
     double sRadius;
};

//Constructors for Struct Meteor
Meteor::Meteor()
{
     sVelocity = 0;
     sMass = 0;
     sXPos = 0;
     sYPos = 0;
}

Meteor::Meteor(double pVelocity, double pMass, double pX, double
pY, double pAngle, double pRadius)
{
     sVelocity = pVelocity;
     sMass = pMass;
     sXPos = pX;
```

```
     sYPos = pY;
     sAngle = pAngle;
     sRadius = pRadius;
}

//Accessor functions for struct Meteor

//getMass(): returns the mass of the meteorite.
double Meteor::getMass()
{
     return sMass;
}

//getVelocity(): returns the velocity of the meteorite.
double Meteor::getVelocity()
{
     return sVelocity;
}

//getRadius(): returns the radius of the meteorite
double Meteor::getRadius()
{
    return sRadius;
}

//getAngle(): returns the angle of the meteorite.
double Meteor::getAngle()
{
     return sAngle;
}

//getX(): returns the horizontal position of the meteorite
relative to the center of the Earth
double Meteor::getX()
{
     return sXPos;
}

//getY(): returns the vertical position of the meteorite relative
to the center of the Earth.
double Meteor::getY()
{
     return sYPos;
}

//distanceFromEarth(): returns the meteorite's distance from the
Earth
double Meteor::distanceFromEarth()
{
     return ( sqrt( pow(sXPos, 2) + pow(sYPos, 2) ) -
RADIUS_EARTH );
}

//Modifier functions for struct Meteor
```

```
/*
setMass(): takes a double representing the new mass for the
meteorite, and returns
a boolean stating whether the received parameter was valid or not.
Sets the meteorite's
mass.
*/
bool Meteor::setMass( double pMass )
{
     if ( pMass > 0 )
     {
          sMass = pMass;
          return true;
     }
     else
          return false;
}

/*
setVelocity(): takes a double representing the new velocity for
the meteorite, and returns
a boolean stating whether the receive parameter was valid or not.
Sets the meteorite's velocity.
*/
bool Meteor::setVelocity( double pVelocity )
{
     if ( pVelocity >= 0 )
     {
          sVelocity = pVelocity;
          return true;
     }
     else
          return false;
}

/*
setPosition(): takes two doubles representing the new x and y
coordinates of the meteorite.
If only one needs to be set, the struct's getX() or getY()
function may be passed as the parameter that doesn't change.
*/
void Meteor::setPos( double pX, double pY )
{
     sXPos = pX;
     sYPos = pY;
}

/*
setRadius(): takes a double representing the radius of the meteor.
Returns false if the value isn't valid.
*/
bool Meteor::setRadius( double pR )
{
```

```
    if (pR > 0)
    {
        sRadius = pR;
        return true;
    }
    else
    {
        return false;
    }
}
//END STRUCT
#endif
```

temp.h

```
#ifndef TEMP_H
#define TEMP_H


class temp
{
public:
    temp();
    double transferHeat( double distance, double time, double
area, double firstTemp, double secondTemp);
};

temp::temp()
{
}

double transferHeat( double distance, double time, double area,
double firstTemp, double secondTemp)
{

    double transferred = 0;
    transferred = ( (WATER_THERMAL_CONDUCTIVITY * area *
(firstTemp - secondTemp) )/ distance ) * time;
     //cerr << "\nfirst temp:" << firstTemp << "\ndistance:" <<
distance << "\narea:" << area << "\nsecondTemp:" << secondTemp <<
"\ntransferred energy:" << transferred;
    //system("PAUSE");
    return transferred;
}

#endif
```

world.h

```
#ifndef WORLD_H
#define WORLD_H

#include "lat.h"
```

```
//A class to represent our virtual world
class world
{
public:
     world();
     lat theWorld[90][180];
};

world::world()
{
     for (int y = 0; y < 90; y++)
     {
          for (int x = 0; x < 180; x++)
          {
               theWorld[y][x].adjustTemp(abs(90 - 2*y));
               theWorld[y][x].setLat(2*y - 90);
          }
     }
}

#endif
```

Appendix B: POV-Ray™ and Raytracing

POV-Ray™, the program with which we created our graphics, uses a method called raytracing to produce its output. Raytracing is a math-intensive process, and is very much based on the way the human eye sees objects. In the real world, a light source emits photons. As these photons interact with objects, they are refracted, reflected, or absorbed. We can see because our eyes detect these photons and construct an image based on the wavelengths and amplitudes of these photons. Raytracing works by performing this exact process, only ***backwards***. A camera and light source are defined, as well as any number of objects of various shapes, sizes, and colors. The scene's camera shoots many hundreds of thousands of rays in all directions in which it can see and follows the rays as they interact with objects. Depending on the nature of the objects with which a ray collides and whether or not the ray eventually ends up colliding with a light source, a color is calculated for the pixel or pixels corresponding to that particular ray. This allows us to create very detailed and realistic graphics.