Solar Flares and Their Interactions with the Earth's Geomagnetic Sphere

New Mexico AiS Challenge

Final Report

Team 007

Albuquerque Academy

Team Members:

Paul Boyle

Phillip Coleman

Alfredo Davila Jr.

Bill Knoop

Teacher and Mentor:

Jim Mims

Executive Summary

The effects of solar particle discharge on the Earth's geomagnetic sphere cannot be ignored. Upon impact with Earth's magnetosphere these particles, traveling at relativistic speeds, orbit around the magnet field lines while being pulled towards one of the poles. While interacting with the field lines the particles continue to accelerate towards the magnetic pole they are attracted to while at the same time being repelled by particles of a similar charge within a certain radius. Our project has been designed to simulate these interactions through a complex particle system.

Our program was written in C++ utilizing multiple classes to interact with the particle system. We first model the origin of the solar flare using equations to determine the dispersion of particles throughout the corona. Secondly we have compartmentalized equations that deal with the effects of a magnetic field on charged particles traveling at high velocity. Using these equations we effectively modeled the geomagnetic sphere.

We then utilized an OpenGL application to help demonstrate the effects on the particles. The application will effectively map the particles position to a color dot in a 3D environment. These dots will help model the dispersion and speed of individual particles. The data will also be demonstrated through the use of color to represent charge, energy, and delta energy.

Analyzing the data we are confident of the accuracy and utility of our program. The results are very promising of future uses of our simulation to predict the effects of a solar flare on the Earth. Further work utilizing this simulation would undoubtedly be useful in predicting the further impacts on human structures.

Project Description

This year's project was to simulate the effect that a solar flare would have on the geomagnetic sphere surrounding the Earth. We planned to first accurately model the Earth's magnetic field, and then use several physics equations we had not yet learned to simulate the effects of different kinds of solar flares. We planned to use many different situations to test the program, and then, after verifying the accuracy of the program, to run a variety of simulations, varying the size, energy, in the solar flare, and the position at which it hit the Earth. We then planned to implement, through OpenGL, a graphic summary of the process, assigning different colors to different ranges of energy in the geomagnetic sphere. If time had permitted we would have liked to research what effects these changing energy levels would cause, such as effects on satellite communications. However, time did not permit for this research at the writing of this report.

Progress

The progress thus far has an included the perfection of the physics model, which allow us to model the interactions between the Earth's magnetic field and the solar flares. Because the physics model didn't seem completely accurate, we decided that we should look into special relativity, and increasing the mass of the particles of the solar flare as they approach relativistic speeds. The aspect of the program has been done separately, as we were unsure as to its accuracy, and we are currently testing the two models against each other in order to see which seems closer to reality, if need be, this module can be removed from the code without any effect on the rest of the program. In addition to this we have created an accurate simulation of the Earth's magnetic field. The OpenGL

implementation of the data is coming along nicely, and by the time this report is submitted it should be complete. As mentioned before, the idea of looking at the effects on the satellites had to be dropped at the time being, however, we hope to be able to look at this information before the final judging.

Challenges

At interim all of judges expressed concern about the state of our project. After the judging our team decided exactly what needed to be done still, and what order would be best to complete the tasks in. The physics model came first, which wasn't as difficult as it seemed it would be at the beginning of the year, as three of the team members took electro-magnetic physics this semester. The relativity was something that we were not planning on learning until college, but lucky one of our team members, Phillip Coleman, took an interest in this subject and did some independent research. As mentioned before, he has been working on the implementation of several relativity equations to make the model even more accurate. The next part was what most concern was voiced about, the accurate modeling of the Earth's electromagnetic sphere. Indeed, this was a difficult task, and we considered throwing out the idea of using the Earth's electromagnetic sphere and just using a generic charged space, until we came upon several interesting equations that allowed us to create a very close approximation of the Earth. Now that we have the heart of the program together, we are working on getting the face ready, the OpenGL implementation has turned out to be one of the more difficult tasks, as none of us had any previous knowledge of implementing graphics with a C++ program. Although this has been difficult, we feel that it will be done in time. The most trying aspect of this project has not been the actual coding, but actually finding information about the subject and

assembling a complete model. We had to bring together man tiny little pieces of science and assemble a whole.

Implementations

Our program begins by calculating the energy output of a solar flare. This is determined in two ways – the use of pre-assigned constants and calculation of remaining variables in differential equations. Essentially, a solar flare compromises a two dimensional area of length *l* and width *w*, which is subdivided into pixels. Each pixel contains a certain amount of minimum energy (one of the pre-determined constants) which is used to determine the flare area, maximum height, and emission rate – or the rate at which energy is released. In our program, each cell is initialized with a base temperature, which is modified over a given area so that the energy in the flare radiates outward. What results, and is used in our program to determine the flare energy and output over distance *x* and time *t*, is a grid of dimensions [*l, w]* with subdivisions each having resultant energy *E*, velocity *v*, mass *m*, and density *D*.

Our program then moves the grid across (theoretically) the distance between the earth and the sun, moving at a v = 3.00 * 10^5, or the speed of light in km / s ^2. For each iteration, our program computes the amount of energy lost to space over the distance traveled, using the equation : This continues until the flare (grid) comes within range of the particle field, determined by the outmost particle.

The particle system is initialized so that the particles themselves are inert; they experience no velocity, forces, or have any charge, until the solar flare hits. For each iteration that the flare is within the particle field, each particle is checked against the field. If the particle is within the field, energy is transferred to the particle, along with a

charge and velocity. The energy transferred is equal to the electric field of the particle – if there is more than one particle in the vicinity, then the energy is distributed equally. For each iteration thereafter, the particle's velocity is determined by two factors.

The first is the earth's magnetic field. Because we've assumed that the field is both a) constant and b) wholly elliptical, the particle is drawn either to the north or south pole. (We do recognize that the magnetic north is actually south, but it is just for reference that we code it in reverse.) We also assume that if a particle is drawn to one pole, it is repeled by the other. However, it is also repeled by the total particles that are in the sphere above it – the combined fields are set against each other.

The second factor used is the magnetic fields of the particles within a certain radius of the particle itself. Our program determines where the radius *r* of each particle is – i.e. where the magnetic field itself is zero outside the particle. It then determines which particles are within the given radius, and computes their magnetic fields based upon either :

```
Forcemag = (mu / 4 *pi) * ((charge1 * charge2)) / radius^2) * velocity1
x (velocity2 x radius^[unit vector])
Forcemag = radius x velocity x magnetic field]
```

Once the solar flare (grid) has passes beyond the furthest particle in the field, several more iterations are allowed in order to incorporate

Results

Our results look promising. As we haven't quite got the OpenGL interpreter ready yet, we are only able to extrapolate the accuracy of the program from looking at masses of numbers it generates. We have noticed trends in the data that the electromagnetic sphere interacts with the solar flare, and increases the energy associated

with a certain piece of space. If we increase the power of the flare and leave all other variables constant we see a greater change in the energy. It is impossible to look at our distribution function fully, as looking at trends through the interactions between the thousands of particles after the solar flare would require a fairly powerful interpreter to be running in one's head.

We are also unable to determine the accuracy associated with the special relativity equations we are running. The data output doesn't seem to vary much between the program without relativity, and the program with relativity, but there is a minimal change. Much of the important information will be available to us in the next week as we get the interpreter working, and then we will truly be able to perfect the code. We are confident that a week should be more than enough for us to work out the final bugs we are experiencing with our OpenGL, and will have extremely promising results by the time we get to the final presentation in Los Alamos.

Conclusions

Thus far we can conclude that the angle of incidence and the size of the solar flare directly affect the amount of energy transferred to the Earth's electromagnetic field. Although this is a simple conclusion, it is what we have been able to notice from the thousands of numbers that we looked through. We will have a conclusion regarding the distribution of the energy across the electromagnetic field at the final presentation. We have simplified the project to not include other sources of radiation, other than the normal electromagnetic field that the Earth has, and the contribution from the solar flare. We also eliminated gravity from the simulation, as then the particles around the Earth would be moving too much for us to deal with, especially when we don't have the graphics up

and running yet. We are nonetheless confident that in Los Alamos we will have these graphics and have far more thorough and concrete results and conclusions.

Code

```
/************************************************

  Author :              Bill Knoop, Phillip Coleman,
                              Alfredo Davila, Paul Boyle

  Date Created :  2/13/04

  File :                Source.cpp

  Comments :            The main file of the program -
                              contains all essential defines
                              functions, and constants needed
                              to create the viewing window,
                              perform     calculations, and
                              ensure program execution

************************************************/

/*
      Commands :

*/

#define WIN32_LEAN_AND_MEAN
#define WIN32_EXTRA_LEAN

#define SUCCESS 1
#define FAIL 0

#define NO_GL_ERROR 1.0
#define FATAL_GL_ERROR 0.0

#define SIN 2
#define COS 3
#define TAN 4

/* compiler and system includes */
/* iostream is not supported */
#include <fstream>
#include <cstdlib>
#include <cstdio>
#include <cmath>
#include <ctime>
#include <windows.h>
#include <winuser.h>
#include <windowsx.h>
#include <conio.h>
#include <string>
```

```
/* opengl includes */
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>

/* project includes */
#include "Random.h"
#include "Elementary_Particle.h"
#include "Base_Particle_System.h"
#include "Magnetic_Field.h"

/* graph utilities */
// #include "Graph.h"
// #include "GAConverter.h"

// using namespace std;

HDC g_HDC; // global device context
float lightPosition[] = {0.0f, -1.0f, 0.0f, 0.0f}; // position of light
bool keys[256]; // the array used for keyboard input
const int MAX_PARTICLES = 129600; // total number of particles [360^2]
bool simulation = false; // is simulation running ?
ofstream logFile("AiSProject.log", ios::out); // file for writing to
program log
bool fatal_error_encountered = false; // check for fatal errors
char *fileNames[4] = {"energy.dat",
                                "dispersion.dat",
                                "charge.dat",
                                "solar_energy.dat"};
char *discriptors[4] = {"E",
                                    "dE",
                                    "q",
                                    "T"};
char *varNames[4] = {"Particle Energy",
                           "Delta Energy [Loss]",
                           "Particle Charge",
                           "Thermal Energy"};

/* stops program in case of premature crash */
void END()
{
      MessageBox(NULL, "Fatal error encountered - program cannot
continue and will now exit", "OPENGL WARNING", MB_OK);
      abort();
}

/* adding a log entry */
int LogEntry(const char *txt, double dw_Flag)
{
      if(dw_Flag == FATAL_GL_ERROR)
      {
            logFile << txt << endl;
            END();
      }
      else if(dw_Flag == NO_GL_ERROR)
      {
```

```
            logFile << txt << endl;
      }
      return SUCCESS;
}

/* close log */
int CloseLog()
{
      logFile.close();
      return SUCCESS;
}
/* windows support functions */
void Initialize()
{
      glEnable(GL_DEPTH_TEST); // enable the depth buffer
      glEnable(GL_CULL_FACE); // do not render hidden surfaces
      glEnable(GL_LIGHTING); // enable lighting
      glEnable(GL_LIGHT0); // enable default light
      glEnable(GL_COLOR_MATERIAL); // switch materials to colors
      glShadeModel(GL_SMOOTH); // enable smooth shading
      glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // clear background to
black
}
/*
void FileFormat()
{
      Data *x, *y;
      GraphAnalysis *ga;
      for(int i = 0; i < 4; i++)
      {
            for(int j = 0; j < 4; j++)
            {
                  if(j != i)
                  {
                        char *fName = "graph" + char(j * i);
                        x = new Data(fileNames[i]);
                        y = new Data(fileNames[j]);
                        ga = new GraphAnalysis();
                        ga->LoadData(x, y);
                        ga->SetData((const char *)varNames[i], (const
char *)varNames[j], (const char *)descriptors[i], (const char
*)descriptors[j]);
                        ga->WriteToFile((const char *)fName);
                  }
            }
      }
}
*/
/* SetupPixelFormat */
void SetupPixelFormat(HDC hDC)
{
      int pixelFormat; // pixel format index
      static PIXELFORMATDESCRIPTOR pfd = {
            sizeof(PIXELFORMATDESCRIPTOR),
            1,
            PFD_DRAW_TO_WINDOW |
            PFD_SUPPORT_OPENGL |
```

```
            PFD_DOUBLEBUFFER,
            PFD_TYPE_RGBA,
            32,
            0, 0, 0, 0, 0, 0,
            0,
            0,
            0,
            0, 0, 0, 0,
            16,
            0,
            0,
            PFD_MAIN_PLANE,
            0,
            0, 0, 0};
      pixelFormat = ChoosePixelFormat(hDC, &pfd); // find best pixel
format
      SetPixelFormat(hDC, pixelFormat, &pfd); // apply pixel format to
window
}
/* WndProc */
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
      static HGLRC hRC; // rendering context
      static HDC hDC; // device context
      int g_width, g_height; // window dimensions

      switch(message)
      {
            /* create the OPENGL contexts and initial window */
            case WM_CREATE:
                  LogEntry("creating window", NO_GL_ERROR);
                  hDC = GetDC(hwnd);
                  g_HDC = hDC;
                  SetupPixelFormat(hDC);
                  hRC = wglCreateContext(hDC);
                  wglMakeCurrent(hDC, hRC);
                  break;
            /* close window and deselect all OPENGL contexts */
            case WM_CLOSE:
                  LogEntry("closing window", NO_GL_ERROR);
                  wglMakeCurrent(hDC, NULL);
                  wglDeleteContext(hRC);
                  PostQuitMessage(0);
                  CloseLog();
                  break;
            /* resize window and aspect ratio */
            case WM_SIZE:
                  LogEntry("resizing window", NO_GL_ERROR);
                  g_height = HIWORD(lParam);
                  g_width = LOWORD(lParam);
                  if(g_height == 0)
                        g_height = 1;
                  /* OPENGL reset viewport */
                  glViewport(0, 0, g_width, g_height);
                  glMatrixMode(GL_PROJECTION);
                  glLoadIdentity();
```

```
                  gluPerspective(54.0f, (GLfloat)g_width /
(GLfloat)g_height, 1.0f, 1000.0f);
                  glMatrixMode(GL_MODELVIEW);
                  glLoadIdentity();
                  break;
            /* a key is pressed */
            case WM_KEYDOWN:
                  keys[wParam] = true;
                  break;
            /* a key is released */
            case WM_KEYUP:
                  keys[wParam] = false;
                  break;
            /* refer to Default Window Procedure [do nothing] */
            default:
                  break;
      }
      return DefWindowProc(hwnd, message, wParam, lParam);
}
/* WinMain */
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nShowCmd)
{
      WNDCLASSEX window; // window class
      HWND hwnd; // window handle
      MSG msg; // window message
      bool done; // completion flag

      /* create instance of window */
      window.cbSize = sizeof(WNDCLASSEX);
      window.style = CS_HREDRAW | CS_VREDRAW;
      window.lpfnWndProc = WndProc;
      window.cbClsExtra = 0;
      window.cbWndExtra = 0;
      window.hInstance = hInstance;
      window.hIcon = LoadIcon(NULL, IDI_APPLICATION);
      window.hCursor = LoadCursor(NULL, IDC_ARROW);
      window.hbrBackground = NULL;
      window.lpszMenuName = NULL;
      window.lpszClassName = "OpenglWindow";
      window.hIconSm = LoadIcon(NULL, IDI_WINLOGO);

      /* register window */
      if(!RegisterClassEx(&window))
            return 0;
      LogEntry("window class registered", NO_GL_ERROR);

      /* create window */
      hwnd = CreateWindowEx(NULL,
                                          "OpenglWindow",
                                          "AiS Project",
                                          WS_OVERLAPPEDWINDOW |
WS_VISIBLE |
                                          WS_SYSMENU | WS_CLIPCHILDREN
|
                                          WS_CLIPSIBLINGS,
```

```
                                                100, // x coordinate of upper
lefthand corner
                                                100, // y coordinate of upper
lefthand corner
                                                800, // width
                                                600, // height
                                                NULL, // no parent(s)
                                                NULL, // no menu(s)
                                                hInstance, // application
instance
                                                NULL); // no extraneous flags
        if(!hwnd)
        {
                MessageBox(NULL, "OPENGL could not create window context",
"AiS Project", MB_OK);
                return 0;
        }
        ShowWindow(hwnd, SW_SHOW);
        UpdateWindow(hwnd);
        LogEntry("window created and operational", NO_GL_ERROR);
        done = false;
        /* call Initialize function */
        ShowCursor(FALSE); // hide the cursor
        while(!done)
        {
                PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);
                if(msg.message == WM_QUIT)
                {
                        done = true; // exit the application
                }
                else if(keys[VK_UP] == true)
                {

                }
                else if(keys[VK_DOWN] == true)
                {

                }
                /* begin simulation */
                else if(keys[VK_S] == true)
                {
                        if(simulation != true)
                        {
                        }
                }
                /* pause simulation */
                else if(keys[VK_P] == true)
                {

                }
                /* energy color code */
                else if(keys[VK_E] == true)
                {

                }
                /* charge color code */
                else if(keys[VK_Q] == true)
```

```
            {

            }
            /* dispersion color mode */
            else if(keys[VK_W] == true)
            {

            }
            /* if message is not recongized, pass on */
            else
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
      }
      /* call cleanup function */
      ShowCursor(TRUE); // reinstate the cursor
      return msg.wParam;
}
/************************************************

  Author :              Bill Knoop, Phillip Coleman,
                              Alfredo Davila, Paul Boyle

  Date Created :  2/13/04

  File :                Base_Particle_System.h

  Comments :            An abstract class used as a base
                              for the particle systems used in
                              this program.
                              NOTE : This class was taken from
                              OPENGL Game Programming
                              protected under the GNU license

************************************************/

#ifndef BASE_PARTICLE_SYSTEM_H
#define BASE_PARTICLE_SYSTEM_H

// using namespace std; // namespace standard

class Base_System
{
      public:
            Base_System(int maxParticles, quantum_vector startPos);
            virtual void Update(float elapsedTime) = 0;
            virtual void Render()                        = 0;
            virtual int Emit(int numParticles);
            virtual void InitializeSystem();
            virtual void DestroySystem();

      protected:
            virtual void InitializeAll(int index)     = 0;
            Q_Particle *particle_list;
            int num_particles;
            quantum_vector start_pos;
```

```
            int max_particles;
            float elapsed_time;
            quantum_vector force_vector;
};

Base_System::Base_System(int maxParticles, quantum_vector startPos)
{
      max_particles = maxParticles;
      start_pos = startPos;
      particle_list = NULL;
}

int Base_System::Emit(int numParticles)
{
      while(numParticles && (num_particles < max_particles))
      {
            InitializeAll(num_particles++);
            --numParticles;
      }
      return numParticles;
}
void Base_System::InitializeSystem()
{
      if(particle_list)
      {
            delete[] particle_list;
            particle_list = NULL;
      }
      particle_list = new Q_Particle[max_particles];
      num_particles = 0;
      elapsed_time = 0.0f;
}

void Base_System::DestroySystem()
{
      if(particle_list)
      {
            delete[] particle_list;
            particle_list = NULL;
      }
      num_particles = 0;
}

#endif
/************************************************

  Author :              Bill Knoop, Phillip Coleman
                              Alfredo Davila, Paul Boyle

  Date Created :  April 4, 2004

  File :                EarthField header file

  Comments :

************************************************/
```

```
#ifndef EARTH_H
#define EARTH_H

using namespace std;

const float earth_radius = (float)(6.3 * pow(10, 6)); // radius of the
earth in km
const float earth_field_strength = (float).6; // Gauss

class Earth
{
      public:
            Earth()
                  : height(earth_radius), radius(100),
fieldstrength(earth_field_strength)
            {};
            Earth(float _height, float _width, float _radius, float
_fieldStrength)
                  : height(_height), radius(_radius),
fieldstrength(fieldStrength)
            {};
            ~Earth();

            /* computational functions */
            float NorthernForce() const { return fieldstrength;};
            float SoutherForce() const { return fieldstrength * -1};
            void MagneticFieldLines(Q_Particle *pList);
            /* rendering functions */
            void RenderEarth();
      private:
            float ComputeTotalMagneticForce(Q_Particle q_charge);
            float ComputeNorthernForce(q_vector radius, q_vector
velocity);
            float ComputeSouthernForce(q_vector radius, q_vector
velocity);
            float height;
            float radius;
            float fieldstength;
            q_vector northern_pole;
            q_vector southern_pole;
};

/* returns the magnetic force of the northern magnetic pole */
float Earth::ComputeNorthernForce(q_vector radius, q_vector velocity)
{
      velocity = velocity.CrossProduct(radius.UnitVector());
      return fieldstrength * velocity.Normalize();
}

/* returns the magnetic force of the southern magnetic pole */
float Earth::ComputeSouthernForce(q_vector radius, q_vector velocity)
{
      velocity = velocity.CrossProduct(radius.UnitVector());
      return fieldstrength * velocity.Normalize();
}

float Earth::ComputeTotalMagneticForce(Q_Particle q_charge)
```

```
{
	float mag_force;
	/* if charge is positive */
		/* determine if particle is past equator
		   [+height for north, -height for south]
		   */
		/* compute distance from northern magnetic pole */
		/* also compute distance from southern magnetic pole */
		q_vector northern_radius = RadiusVector(northern_pole,
q_charge.position);
		q_vector southern_radius = RadiusVector(souther_pole,
q_charge.position);
		mag_force = ComputeNorthernForce(northern_radius,
q_charge.velocity);
		mag_force = mag_force +
ComputeSouthernForce(southern_radius, q_charge.position);
	}
	else if(q_charge.charge < 0)
	{
		/* compute distance from southern magnetic pole */
		/* also compute distance from northnern magnetic pole */
		q_vector southern_radius = RadiusVector(southern_pole,
q_charge.position);
		q_vector northern_radius = RadiusVector(northern_pole,
q_charge.position);
		mag_force = ComputeSouthernForce(southern_radius,
q_charge.velocity);
		mag_force = mag_force +
ComputeNorthernForce(northern_radius, q_charge.position);
	}
	return mag_force;
}

void Earth::MagneticFieldLines(Q_Particle *pList, int numParticles)
{
	/* check for valid reference */
	if(pList == NULL)
		return;
	/* effect all particle velocities with the earth's magnetic field
*/
	for(int m = 0; m < numParticles; m++)
	{
		float mgForce = ComputeTotalMagneticForce(pList[m]);
		pList[m]->velocity.x += mgForce;
		pList[m]->velocity.y += mgForce;
		pList[m]->velocity.z += mgForce;
	}
}

#endif
/************************************************

  Author :              Bill Knoop, Phillip Coleman,
                              Alfredo Davila, Paul Boyle

  Date Created :  2/13/04
```

```
  File :                Elementary_Particle.h

  Comments :            Includes the basic structs and
                              functions needed for the magnetic
                              field, solar flare(s), and other
                              graphical representations

************************************************/

#ifndef ELEMENTARY_PARTICLE_H
#define ELEMENTARY_PARTICLE_H

// using namespace std;
/*
void floatcpy(float list[], float cpy[], int index)
{
      try
      {
            for(int c = 0; c < index; c++)
            {
                  list[c] = cpy[c];
            }
      }
      catch(CException, e)
      {
            /* trace exception, most likely an array index out of
bounds /
            TRACE("Exception Found in FloatCpy(float *[], float *[],
int *");
            /* delete exception /
            e->Delete();
      }
}*/

struct quantum_vector
{
      float x;
      float y;
      float z;
      quantum_vector()
            : x(0.0f), y(0.0f), z(0.0f)
      {
      }
      quantum_vector(int a, int b, int c)
            : x(a), y(b), z(c)
      {
      }
      /* vector operators */
      /* equality */
      const bool operator == (const quantum_vector &qv) const
      {
            return ((x == qv.x) && (y == qv.y) && (z == qv.z));
      }
      /* inequality */
      const bool operator != (const quantum_vector &qv) const
      {
            return !(*this == qv);
```

```
	}
	/* increment */
	const quantum_vector& operator += (const quantum_vector &qv)
	{
		x += qv.x;
		y += qv.y;
		z += qv.z;

		return *this;
	}
	const quantum_vector& operator -= (const quantum_vector &qv)
	{
		x-= qv.x;
		y -= qv.y;
		z -= qv.z;

		return *this;
	}
	/* addition */
	quantum_vector operator + (const quantum_vector &qv) const
	{
		return quantum_vector(qv.x + x, qv.y + y, qv.z + z);
	}
	quantum_vector operator + (const float &sc) const
	{
		return quantum_vector(sc + x, sc + y, sc + z);
	}
	/* subtraction */
	quantum_vector operator - (const quantum_vector &qv) const
	{
		return quantum_vector(x - qv.x, y - qv.y, z - qv.z);
	}
	quantum_vector operator - (const float &sc) const
	{
		return quantum_vector(x - sc, y - sc, z - sc);
	}
	float Normalize()
	{

	}
};

struct Q_Particle
{
	quantum_vector position;
	quantum_vector previous_position;
	quantum_vector velocity;
	quantum_vector acceleration;
	float energy;
	float delta_energy;
	float life_span;
	float mass;
	float delta_mass;
	float weight;
	float delta_weight;
	float color[3];
	float color_delta[4];
```

```
	float start_time;
	float charge;
	int status;
	/* functions used for updating particle position
	   and evalutating energy transfer and graphical
	   representation
	*/
	int UpdatePosition(float deltaTime, int _funcX, int coeffX, int
_funcY, int coeffY, int _funcZ, int coeffZ)
	{
		/* update to new position :: this function is used
		   in the simulation of the magnetic field - each particle
		   moves in an eliptical orbit, therefore the position
		   < x, y, z > corresponds to the following set of
parametric
		   equations :
		   x = 2 * sin t
		   y = 4 * cos t
		   z = 2 * sint
		   which yield an elipse whose length is greatest along the
		   y - axis
		      */
		previous_position = position;
		if(_funcX != NULL && coeffX != NULL)
		{
			if((int)_funcX == SIN)
			{
				position.x = (int)coeffX * sin((int)deltaTime);
			}
			else if((int)_funcX == COS)
			{
				position.x = (int)coeffX * cos((int)deltaTime);
			}
			else if((int)_funcX == TAN)
			{
				position.x = (int)coeffX * tan((int)deltaTime);
			}
			else
			{
				return FAIL;
			}
		}
		else if(_funcY != NULL && coeffY != NULL)
		{
			if((int)_funcY == SIN)
			{
				position.y = (int)coeffY * sin((int)deltaTime);
			}
			else if((int)_funcY == COS)
			{
				position.y = (int)coeffY * cos((int)deltaTime);
			}
			else if((int)_funcY == TAN)
			{
				position.y = (int)coeffY * tan((int)deltaTime);
			}
			else
```

```
                  {
                        return FAIL;
                  }
            }
            else if(_funcZ != NULL && coeffZ != NULL)
            {
                  if((int)_funcZ == SIN)
                  {
                        position.x = (int)coeffZ * sin((int)deltaTime);
                  }
                  else if((int)_funcZ == COS)
                  {
                        position.x = (int)coeffZ * cos((int)deltaTime);
                  }
                  else if((int)_funcZ == TAN)
                  {
                        position.z = (int)coeffZ * tan((int)deltaTime);
                  }
                  else
                  {
                        return FAIL;
                  }
            }

            return SUCCESS;
      }
      int UpdatePosition()
      {
            return SUCCESS;
      }
};

/*
      The textures for each particle are included in their respective
      generators - otherwise more memory per particle will be used
      than is necessary
            GLuint texture; // particle texture template
*/

#endif
/************************************************

  Author :              Bill Knoop, Phillip Coleman,
                              Alfredo Davila, Paul Boyle

  Date Created :  4/4/04

  File :                Environment.h

  Comments :            Incorporates the particle system
                              and the earth into a manageable
                              system - controls all aspects of
                              the program

************************************************/

#ifndef ENVIRONMENT_H
```

```
#define ENVIRONMENT_H

using namespace std; // namespace standard

union DISTANCE_INDEX{
	bool mkilometer;
	bool kilometer;
	bool meter;
	bool centimeter;
};

class Environment : public ForceCalc
{
	public:
		Environment();
		Environment(float deltaTime);
		~Environment();

		void InitializeSystem(HDC *hDC);
		void UpdateSystem(float deltaTime);
		void UpdateTimer(float deltaTime);
		void RenderEnvironment();
		void DestroyEnvironment();

		void CodeEnergy() {pColor->SetIndex(PC_ENERGY);}
		void CodeDisp() {pColor->SetIndex(PC_DISPERSION);}
		void CodeCharge() {pColor->SetIndex(PC_CHARGE);}
		void CodeHeat() {pColor->SetIndex(PC_HEAT);}

	private:
		void ColorCodeParticles();
		void InitializeEarth();
		void InitializeField();
		void InitializeColor();
		void InitializeFont();
		void UpdateDistance();
		bool within_earth_boundaries;
		Field *magField;
		Earth *earth;
		ParticleColor *pColor;
		BitmapFont *font;
		DISTANCE_INDEX *envDistance;
		HDC *hDC;
};

Environment::Environment()
{
}

void Environment::InitializeFont()
{
	font->BuildFont("Times New Roman", 32);
	SelectObject(hDC, font->Font());
	wglUseFontBitmaps(hDC, 32, 96, font->FontBase());
}

void Environment::CalcParticleForces()
```

```
{

}

float Environment::SingleParticleField(int index, float radius)
{
	return (mu / (4*pi)) * (magField->IndexAt(index).charge /
(float)pow((double)radius, 2));
}

float Environment::AllParticleField(int index, float deltaTime)
{
	float radial_range = RadialZero(index, deltaTime);
	Q_Particle *temp = magField->IndexAt(index);
	float field = 0.0f;
	for(int i = 0; i < magField->Particles; i++)
	{
		if(magField->ParticleField(magField->IndexAt(i).position.x,
magField->IndexAt(i).position.y, magField->IndexAt(i).position.z,
radial_range.Normalize() == true)
		{
			field += Magnetic_Force(temp, magField->IndexAt(i));
		}
	}
}

quantum_vector Environment::RadialZero(int index, float deltaTime)
{
	/* compute the distance at which the B field is 0
	   or at least approximately 0
	   */
	/* retreive particle */
	quantum_vector radial_output;
	Q_Particle *nParticle = magField->IndexAt(index);
	/* B = mu / 2 * pi * q / r * t */
	float r = 1.0f;
	float B = 0.0f;
	float diff = .9999999f;
	float first_term = mu / (2 * pi);
	while(B > diff)
	{
		B = first_term * (float)(nParticle->charge / (nParticle-
>velocity.Normalize() * deltaTime));
		r *= 100;
	}
	radial_output.x = r;
	radial_output.y = r;
	radial_output.z = r;
	return radial_output;
}

void Environment::InitializeEarth()
{
	earth = new EarthField();
}

void Environment::InitializeField()
```

```
{
	quantum_vector origin;
	origin.x = 0.0f;
	origin.y = 0.0f;
	origin.z = 0.0f;
	magField = new Field(1000, origin;
}

void Environment::InitializeColor()
{
	pColor = new ParticleColor();
	pColor->SetIndex(PC_NEUTRAL);
}

void Environment::ColorCodeParticles()
{
	pColor->ParticleColorIndex(magField->ParticleList(), magField-
>Particles());
}

void Environment::RenderEnvironment()
{
	earth->Render();
	magField->Render();
}

void Environment::UpdateSystem(float deltaTime)
{
	float particle_magnetic_forces;
	float flare_magnetic_forces;
	/* upadate particles in the magnetic field lines */
	earth->MagneticFieldLines(magField->ParticleList(), magField-
>Particles());
	for(int p = 0; p < magField->Particles(); p++)
	{
		particle_magnetic_forces = AllParticleField(p, deltaTime);
		magField->IndexAt(p).velocity.x +=
particle_magnetic_forces;
		magField->IndexAt(p).velocity.y +=
particle_magnetic_forces;
		magField->IndexAt(p).velocity.z +=
particle_magnetic_forces;
	}
}

#endif
/************************************************

  Author :		Bill Knoop, Phillip Coleman,
			Alfredo Davila, Paul Boyle

  Date Created :  4/4/04

  File :		Font.h

  Comments :		class to create fonts for
			text output in winAPI
```

```
                              environment
                              NOTE : Most of the functions
                              used in this class were directly
                              used or derived from
                              OPENGL Game Programming
                              protected under the GNU license

**************************************************/

#ifndef FONT_H
#define FONT_H

class BitmapFont
{
      public:
            BitmapFont();
            BitmapFont(const char *_fontName, int _fontSize);
            ~BitmapFont();

            void BuildFont(const char *_fontName, int _fontSize);
            void Print(float x, float y, float z, char *txt);
            void Print(q_vector position, char *txt);
            HFONT Font() {return glFont;};
            unsigned int FontBase() {return fontBase;};
            void ClearList();
      private:
            void CreateBitmapFont();
            int fontSize;
            const char *fontName;
            unsigned int fontBase;
            HFONT glFont;
};

BitmapFont::BitmapFont()
{
}

BitmapFont::BitmapFont(const char *_fontName, int _fontSize)
{
      strcpy(fontName, _fontName);
      fontSize = _fontSize;
      CreateBitmapFont();
}

BitmapFont::~BitmapFont()
{
      ClearList();
}

void CreateBitmapFont()
{
      /* create storage for 96 characters */
      fontBase = glGenLists(96);
      if(stricmp(fontName, "symbol") == 0)
      {
            glFont = CreateFont(fontSize,
                                          0,
```

```
                                          0,
                                          0,
                                          FW_BOLD,
                                          FALSE,
                                          FALSE,
                                          FALSE,
                                          SYMBOL_CHARET,
                                          OUT_TT_PRECIS,
                                          CLIP_DEFAULT_PRECIS,
                                          ANTIALIASED_QUALITY,
                                          FF_DONTCARE | DEFAULT_PITCH,
                                          fontName);
        }
        else
        {
                        glFont = CreateFont(fontSize,
                                          0,
                                          0,
                                          0,
                                          FW_BOLD,
                                          FALSE,
                                          FALSE,
                                          FALSE,
                                          ANSI_CHARSET,
                                          OUT_TT_PRECIS,
                                          CLIP_DEFAULT_PRECIS,
                                          ANTIALIASED_QUALITY,
                                          FF_DONTCARE | DEFAULT_PITCH,
                                          fontName);
        }
        /* check for initialized font */
        if(!glFont)
              return;
}

void BitmapFont::Print(float x, float y, float z, char *txt)
{
        if((fontBase == 0) || (txt == NULL))
              return;
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glTranslatef(x, y, z);
        glPushAttrib(GL_LIST_BIT)
              glListBase(fontBase - 32);
              glCallLists(strlen(txt), GL_UNSIGNED_BYTE, txt);
        glPopAttrib();
}

void BitmapFont::Print(q_vector position, char *txt)
{
        if((fontBase == 0) || (txt == NULL))
              return;
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glTranslatef(position.x, position.y, position.z);
        glPushAttrib(GL_LIST_BIT)
              glListBase(fontBase - 32);
```

```
            glCallLists(strlen(txt), GL_UNSIGNED_BYTE, txt);
      glPopAttrib();
}

void BitmapFont::ClearList()
{
      if(fontBase != 0)
            glDeleteLists(fontBase, 96);
}

#endif
/*********************************************

  Author :              Bill Knoop, Phillip Coleman
                              Alfredo Davila, Paul Boyle

  Date Created :  April 4, 2004

  File :                ForceCalc header file

  Comments :

*********************************************/

#ifndef FORCECALC_H
#define FORCECALC_H

class ForceCalc
{
      public:
            ForceCalc() {};
            ~ForceCalc() {};

            virtual void CalcAllForces();

      protected:
            virtual void CalcParticleForces();
            virtual float SingleParticleField(int index, float radius)
= 0;
            virtual float AllParticleField(int index, float radius) =
0;
            virtual q_vector RadialZero(int index);
};

#endif
#ifndef GACONVERTER_H
#define GACONVERTER_H

string GA_HEADER = "<Document>
  <Version>3.0 FC4</Version>
  <ApplicationBuildDateTime>  Mar 12 2002 09:48:28
</ApplicationBuildDateTime>
  <Copyright>Copyright (c) 2002 Vernier Software &
Technology</Copyright>
  <FileName>Z:\test.ga3</FileName>
  <FileIDString>Thu Jan 24 09:30:02 2002Jessica
Fink575816838</FileIDString>
```

```
  <CreatorOS>2</CreatorOS>
  <Radians>1</Radians>
  <NumDerivativePoints>3</NumDerivativePoints>
  <NumSmoothingPoints>5</NumSmoothingPoints>
  <CreationDateTime>Thu, Jan 24, 2002 9:30:02 AM</CreationDateTime>
  <ModifiedDateTime>3/31/2004 16:14:46</ModifiedDateTime>
  <PageView>0</PageView>
  <PageBounds>0 0 10.6133 6.45333</PageBounds>
  <PrinterOrientation>1</PrinterOrientation>
  <MathType>0</MathType>
  <AllowOptionsDialogs>1</AllowOptionsDialogs>
  <AllowAutoCurveFit>1</AllowAutoCurveFit>
  <DataBrowserVisible>0</DataBrowserVisible>
  <DataBrowserSelected>0 </DataBrowserSelected>
  <DataBrowserExpanded>0 </DataBrowserExpanded>
  <AutoscaleFromZero>0</AutoscaleFromZero>
  <Page>
    <ID>101 </ID>
    <BroadcastTo>0 </BroadcastTo>
    <PageTitle></PageTitle>
    <PageColor>0xcccc 0xcccc 0xffff</PageColor>
    <PageDisplayPageInfo>0 </PageDisplayPageInfo>
    <PagePageInfo></PagePageInfo>
    <PageGraphPaper>0 </PageGraphPaper>
    <PageGraphPaperLineSpacing>0.25 </PageGraphPaperLineSpacing>
    <PageGraphPaperLineColor>0x8484 0x7070
0xffff</PageGraphPaperLineColor>
    <PageGraphCartesian>
      <ID>105 </ID>
      <BroadcastTo>0 </BroadcastTo>
      <Bounds>3.52854 0.244939 10.3538 6.20841 </Bounds>
      <SubordinateToID>0</SubordinateToID>
      <SubordinateOffset>0 0</SubordinateOffset>
      <BackgroundTransparent>0</BackgroundTransparent>
      <BackgroundColor>0xffff 0xffff 0xffff</BackgroundColor>
      <AddNewData>0</AddNewData>
      <Locked>0</Locked>
      <GraphTitle></GraphTitle>
      <GraphTitleColor>0x0 0x0 0x0</GraphTitleColor>
      <GraphShowPointProtectors>1</GraphShowPointProtectors>
      <GraphShowPoints>0</GraphShowPoints>
      <GraphInterpolate>0</GraphInterpolate>
      <GraphShowTangent>0</GraphShowTangent>
      <GraphShowLegend>0</GraphShowLegend>
      <GraphPlotBaseColumnID>103 </GraphPlotBaseColumnID>
      <GraphPlotTraceIDPairs>1 103 104 </GraphPlotTraceIDPairs>
      <GraphShowTraceErrorBars>1</GraphShowTraceErrorBars>
      <GraphShowBaseErrorBars>1</GraphShowBaseErrorBars>
      <GraphConnectLines>1</GraphConnectLines>
      <GraphBarGraph>0</GraphBarGraph>
      <GraphShowCursorStatus>1</GraphShowCursorStatus>
      <GraphMajorTickStyle>1</GraphMajorTickStyle>
      <GraphMajorTickColor>0x8080 0x8080 0x8080</GraphMajorTickColor>
      <GraphMinorTickStyle>3</GraphMinorTickStyle>
      <GraphMinorTickColor>0xc0c0 0xc0c0 0xc0c0</GraphMinorTickColor>
      <GraphPlotYLabel></GraphPlotYLabel>
      <GraphPlotXAutoscale>0</GraphPlotXAutoscale>
```

```
    <GraphPlotYAutoscale>0</GraphPlotYAutoscale>
    <GraphPlotXMin>0</GraphPlotXMin>
    <GraphPlotXMax>11</GraphPlotXMax>
    <GraphPlotYMin>0</GraphPlotYMin>
    <GraphPlotYMax>110</GraphPlotYMax>
  </PageGraphCartesian>
  <PageTable>
    <ID>106 </ID>
    <BroadcastTo>0 </BroadcastTo>
    <Bounds>0.27675 0.261269 3.23796 3.95822 </Bounds>
    <SubordinateToID>0</SubordinateToID>
    <SubordinateOffset>0 0</SubordinateOffset>
    <BackgroundTransparent>0</BackgroundTransparent>
    <BackgroundColor>0xffff 0xffff 0xffff</BackgroundColor>
    <AddNewData>1</AddNewData>
    <Locked>0</Locked>
    <TableColumn> 103 1 </TableColumn>
    <TableColumn> 104 1 </TableColumn>
    <StartIndex> 0 </StartIndex>
    <Font>
      <Size> 12 </Size>
      <Justification> 0 </Justification>
      <Styles> 0 0 0 0 0 0 </Styles>
      <Family> 3 </Family>
      <Color>0x0 0x0 0x0</Color>
      <Name>Arial</Name>
    </Font>
    <DisplayBlackAndWhite> 0 </DisplayBlackAndWhite>
  </PageTable>
  <PageText>
    <ID>107 </ID>
    <BroadcastTo>0 </BroadcastTo>
    <Bounds>0.387448 4.16724 3.15494 6.0745 </Bounds>
    <SubordinateToID>0</SubordinateToID>
    <SubordinateOffset>0 0</SubordinateOffset>
    <BackgroundTransparent>0</BackgroundTransparent>
    <BackgroundColor>0xffff 0xffff 0xffff</BackgroundColor>
    <AddNewData>0</AddNewData>
    <Locked>0</Locked>
    <Font>
      <Size> 12 </Size>
      <Justification> 0 </Justification>
      <Styles> 0 0 0 0 0 0 </Styles>
      <Family> 3 </Family>
      <Color>0x0 0x0 0x0</Color>
      <Name>Arial</Name>
    </Font>
    <TextText>Notes:</TextText>
    <TextReadOnly> 0 </TextReadOnly>
    <TextMakeAllTextVisible> 1 </TextMakeAllTextVisible>
  </PageText>
</Page>
<DataSet>
  <ID>102 </ID>
  <BroadcastTo>1 106 </BroadcastTo>
  <DataSetName>Data Set</DataSetName>
  <DataSetComments></DataSetComments>
```

```
    <DataSetCreationTime>Thu, Jan 24, 2002 9:30:02
AM</DataSetCreationTime>
    <DataSetSourceName></DataSetSourceName>
    <DataSetShowInDataBrowser>1</DataSetShowInDataBrowser>
    <DataSetHistogram>0</DataSetHistogram>
    <DataSetFFT>0</DataSetFFT>
    <DataColumn>
      <ID>103 </ID>
      <BroadcastTo>2 105 106 </BroadcastTo>";

const string GA_MIDDLE_X = "<DataObjectColor>0x0 0x0
0x0</DataObjectColor>
      <DataObjectDataSource>0</DataObjectDataSource>
      <DataObjectPrecisionDecimal>1</DataObjectPrecisionDecimal>
      <DataObjectPrecision>3</DataObjectPrecision>
      <ColumnUnits></ColumnUnits>
      <ColumnProtected>0</ColumnProtected>
      <ColumnTreatAsText>0</ColumnTreatAsText>
      <ColumnPPStyle>1</ColumnPPStyle>
      <ColumnDisplayPPInterval>1</ColumnDisplayPPInterval>
      <ColumnUseErrorCalculations>0</ColumnUseErrorCalculations>
      <ColumnErrorFixed>1</ColumnErrorFixed>
      <ColumnErrorConstant>1</ColumnErrorConstant>
      <ColumnErrorValue>0</ColumnErrorValue>
      <ColumnErrorColumnID>0</ColumnErrorColumnID>
      <ColumnCells>";
const string GA_MIDDLE_Y_1 = "</ColumnCells>
    </DataColumn>
    <DataColumn>
      <ID>104 </ID>
      <BroadcastTo>2 105 106 </BroadcastTo>";
const string GA_MIDDLE_Y_2 = "<DataObjectColor>0x9999 0x0
0x0</DataObjectColor>
      <DataObjectDataSource>0</DataObjectDataSource>
      <DataObjectPrecisionDecimal>1</DataObjectPrecisionDecimal>
      <DataObjectPrecision>3</DataObjectPrecision>
      <ColumnUnits></ColumnUnits>
      <ColumnProtected>0</ColumnProtected>
      <ColumnTreatAsText>0</ColumnTreatAsText>
      <ColumnPPStyle>3</ColumnPPStyle>
      <ColumnDisplayPPInterval>1</ColumnDisplayPPInterval>
      <ColumnUseErrorCalculations>0</ColumnUseErrorCalculations>
      <ColumnErrorFixed>1</ColumnErrorFixed>
      <ColumnErrorConstant>1</ColumnErrorConstant>
      <ColumnErrorValue>0</ColumnErrorValue>
      <ColumnErrorColumnID>0</ColumnErrorColumnID>
      <ColumnCells>";
const string GA_END = "</ColumnCells>
    </DataColumn>
  </DataSet>
  <DataSourceServer>
    <ID>7 </ID>
    <BroadcastTo>0 </BroadcastTo>
  </DataSourceServer>
</Document>";

class GraphAnalysis : public Graph
```

```
{
public:
	GraphAnalysis()
	{
		g_file = new ofstream();
	}
	~GraphAnalysis() {}

	int WritetoFile(const char *fileName)
	{
		string line_1, line_2, line_3, line_4;
		g_file.open(fileName, ios::out);
		if(data_loaded)
		{
			line_1 = "<DataObjectName>" + datName1 +
"</DataObjectName>";
			line_2 = "<DataObjectName>" + datName2 +
"</DataObjectName>";
			line_3 = "<DataObjectShortName>" + datSymbol1 +
"</DataObjectShortName>";
			line_4 = "<DataObjectShortName>" + datSymbol1 +
"</DataObjectShortName>";
			/* output in the following order
				< FILE >
				GA_HEADER
				line_1
				line_3
				GA_MIDDLE_X
				[elements of] row [<< endl]
				GA_MIDDLE_Y_1
				line_2
				line_4
				GA_MIDDLE_Y_2
				[elements of] columns [<< endl]
				GA_END
			*/
			g_file << GA_HEADER;
			g_file << line_1 << endl;
			g_file << line_3 << endl;
			g_file << GA_MIDDLE_X;
			/* print out first column data */
			for(int i = 0; i < row.DataElements(); i++)
			{
				g_file << (float)row.ElementAt(i) << endl;
			}
			g_file << GA_MIDDLE_Y_1;
			g_file << line_2 << endl;
			g_file << line_4 << endl;
			g_file << GA_MIDDLE_Y_2;
			/* print out second column data */
			for(i = 0; i < column.DataElements(); i++)
			{
				g_file << (float)column.ElementAt(i) << endl;
			}
			g_file << GA_END;
		}
		else
```

```
		{
			return FAIL;
		}
		return SUCCESS;
	}

	int ConfirmFile(const char *fileName)
	{
		FILE *fptr;
		if((fptr = fopen(fileName, "rb")) == NULL)
		{
			return FAIL;
		}
		else
		{
			free(fptr);
			return SUCCESS;
		}
	}

	int LoadDataSet(Data *_dat1, Data *_dat2)
	{
		row = _dat1;
		column = _dat2;
	}

	void SetData(const char *_dname1, const char *_dname2, const char
*_dsymbol1, const char *_dsymbol2)
	{
		strcpy(datName1, dname1);
		strcpy(datName2, dname2);
		strcpy(datSymbol1, dsymbol1);
		strcpy(datSymbol1, dsymbol1);
	}

private:
	const char *datName1;
	const char *datName2;
	const char *datSymbol1;
	const char *datSymbol2;
};

#endif
#ifndef GRAPH_H
#define GRAPH_H

class Data
{
public:
Data()
{
g_data = new float[0];
}
Data(const char *fileName)
{
int g_c = LoadData(fileName);
if(g_c == SUCCESS)
```

```
      g_state = true;
else
      g_state = false;
;
~Data()
{
if(g_data)
      delete[] g_data;
}

int DataElements() const
{
      return num_elements;
}

float ElementAt(int index)
{
      if(index > -1 && index < num_elements)
      {
            return g_data[index];
      }
}

Data operator& = (Data *dat)
{
      g_data = NULL;
      num_elements = dat-<DataMemebers();
      g_data = new float[num_elements];
      for(int i = 0; i < dat->DataMembers(); i++)
      {
            g_data[i] = dat->ElementAt(i);
      }
}

private:
int LoadData(const char *fileName)
{
FILE *fPtr;
if((fPtr = fopen(fileName, "rb")) == NULL)
      return FAIL;
/* break */
ifstream infile(fileName, ios::in);
infile >> num_data;
g_data = new float[num_data];
int c = 0;
float temp;
/* load all data */
try
{
      while(!infile.fail())
      {
            infile >> temp;
            g_data[c] = (float *)temp;
            c++;
      }
}
catch(CException *cE)
```

```
{
	return FAIL;
}
return SUCCESS;
}
bool DataLoaded()
{
return g_state;
}
private:
float *g_data;
bool g_state;
int num_elements;
};

class Graph
{
public:
	Graph();
	Graph(Data *_dat1, Data *_dat2);
	~Graph();

	virtual int WritetoFile(const char *fileName) = 0;
	virtual int ConfirmFile(const char *fileName) = 0;
	virtual int LoadDataSet(Data *_dat1, Data *_dat2) = 0;

protected:
	Data *row;
	Data *column;
	ofstream g_file;
	bool data_loaded;
};

class Statistics
{
Statistics();
~Statistics();
int LoadStats(const char *fileNames[], int num_data)
{
int c = 0;
num_stats = num_data;
stats = new Data[num_stats];
try
{
	while(fileNames[c] != NULL)
	{
		stats[c].LoadData(fileNames[c]);
		c++;
	}
}
catch(CException cE)
{
return FAIL;
}
};
private:
int num_stats;
```

```
Data *stats;
};

#endif
{\rtf1\ansi\ansicpg1252\deff0\deflang1033{\fonttbl{\f0\fswiss\fcharset0
Arial;}}
{\*\generator Msftedit 5.41.15.1503;}\viewkind4\uc1\pard\f0\fs20
#ifndef GRAPHDATA_H\par
#define GRAPHDATA_H\par
\par
#define SUCCESS 1\par
#define FAIL 0\par
\par
class Data\line\{\par
public:\par
Data()\line\{\par
g_data = new float[0];\line\};\par
Data(const char *fileName)\line\{\par
int g_c = LoadData(fileName);\par
if(g_c == SUCCESS)\par
\tab g_state = true;\par
else\par
\tab g_state = false;\line\};\par
~Data()\line\{\par
if(g_data)\par
\tab delete[] g_data;\line\};\par
\par
int DataElements() const\line\{\par
\tab return num_elements;\line\};\par
\par
float *ElementAt(int index)\line\{\par
\tab if(index > -1 && index < num_elements)\par
\tab\{\par
\tab\tab return g_data[index];\par
\tab\}\par
\tab else\par
\tab\{\par
\tab\tab return NULL;\par
\tab\}\line\};\par
\par
private:\par
int LoadData(const char *fileName)\line\{\par
FILE *fPtr;\par
if((fPtr = fopen(fileName, "rb")) == NULL)\par
\tab return FAIL;\par
/* break */\par
ifstream infile(fileName, ios::in);\par
infile >> num_data;\par
g_data = new float[num_data];\par
int c = 0;\par
float temp;\par
/* load all data */\par
try\line\{\par
\tab while(!infile.fail())\line\tab\{\par
\tab\tab infile >> temp;\par
\tab\tab g_data[c] = (float *)temp;\par
\tab\tab c++;\par
```

```
\tab\}\par
\}\par
catch(CException *cE)\par
\{\par
\tab return FAIL;\line\}\par
return SUCCESS;\par
\};\par
bool DataLoaded()\line\{\par
return g_state;\line\}\par
private:\par
float *g_data;\par
bool g_state;\par
int num_elements;\line\};\par
\par
class Graph\line\{\par
public:\par
\tab Graph();\par
\tab Graph(Data *_dat1, Data *_dat2);\par
\tab ~Graph();\par
\par
\tab virtual int WritetoFile(const char *fileName) = 0;\par
\tab virtual int ConfirmFile(const char *fileName) = 0;\par
\tab virtual int LoadDataSet(Data *_dat1, Data *_dat2) = 0;\par
\par
protected:\par
\tab Data *row;\par
\tab Data *column;\par
\tab ofstream g_file;\par
\tab bool data_loaded;\line\};\line\par
class Statistics\par
\{\par
Statistics();\par
~Statistics();\par
int LoadStats(const char *fileNames[], int num_data)\par
\{\par
int c = 0;\par
num_stats = num_data;\par
stats = new Data[num_stats];\par
try\par
\{\par
\tab while(fileNames[c] != NULL)\line\tab\{\par
\tab\tab stats[c].LoadData(fileNames[c]);\par
\tab\tab c++;\line\tab\}\par
\}\par
catch(CException cE)\par
\{\par
return FAIL;\line\}\line\};\par
private:\par
int num_stats;\par
Data *stats;\line\};\par
\par
#endif\par
}
/************************************************

  Author :              Bill Knoop, Phillip Coleman,
                          Alfredo Davila, Paul Boyle
```

```
  Date Created :  2/13/04

  File :                Field.h

  Comments :            Contains the class instantiation
                              of the magnetic field - inherits
                              from the base_system class,
                              therefore all the initialization
                              and   destruction of < the system >
                              is handled by the base

**************************************************/

#ifndef FIELD_H
#define FIELD_H

#include <fstream>
// using namespace std; // namespace standard

/* define the origin on the <x,y,z> plane */
/* define starting mass (size) */
const quantum_vector ORIGIN (0.0f, 0.0f, 0.0f);
const float MAX_MASS = .0075f;

/* output files */

class Field : public Base_System
{
      public:
            Field(int maxParticles, float height, float width, float
depth)
                  : :   height(pHeight), width(pWidth), depth(pDepth),
Base_System(maxParticles, ORIGIN)
            {}
            void Update(float deltaTime)
            {
                  /* no particles are "killed", simply updated */
                  for(int c = 0; c < num_particles; c++)
                  {
                        particle_list[c].UpdatePosition(elapsedTime,
(int *)SIN, (int *)2, (int *)COS, (int *)4, (int *)SIN, (int *)2);
                  }
            }
            void Render()
            {
                  glEnable(GL_BLEND);
                  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
                  GLUquadricObj *sphere = gluNewQuadric(); // create
base for
                  // sphere
                  glPushAttrib(GL_CURRENT_BIT);
                  for(int c = 0; c < num_particles; c++)
                  {
                        glPushMatrix();
                              glMateriali(GL_FRONT_AND_BACK,
GL_SHININESS, rand() % 128);
```

```
                              glColor3f(particle_list[c].color[0],
particle_list[c].color[1], particle_list[c].color[2]);
                              glTranslatef(particle_list[c].position.x,
particle_list[c].position.y, particle_list[c].position.z);
                              gluSphere(sphere, (double)MAX_MASS, 16,
10);
                        glPopMatrix();
                  }
                  glPopAttrib();
                  // delete gl sphere object
                  if(sphere)
                        gluDeleteQuadric(sphere);
                  glDisable(GL_BLEND);
            }

            int Emit(int numParticles) { return 0;}

            /* magnetic field specific functions */
            /* accessors */
            void InitializeSystem()
            {
                  // let Base_System do initialization
                  Base_System::InitializeSystem();
            }

            void DestroySystem()
            {
                  heat_file.close();
                  energy_file.close();
                  dispersion_file.close();
                  // call Base_System destructor
                  Base_System::DestroySystem();
            }

            Q_Particle* IndexAt(int index)
            {
                  if(index >= 0 && index < num_particles)
                        return *&particle_list[index];
                  else
                        return NULL;
            }

            bool ParticleAt(float x, float y, float z, int index)
            {
                  /* returns true if the particle is there */
                  if(index >= 0 && index < num_particles)
                  {
                        if(particle_list[index].position.x == x &&
particle_list[index].position.y == y && particle_list[index].position.z
== z)
                              return true;
                        else
                              return false;
                  }
                  else
                  {
                        return false;
```

```
                }
        }

        bool ParticleField(float x, float y, float z, float disp,
int index)
        {
                /* returns true if a [index] particle is within
                   the < dx, dy, dz > defined space between
                   < x, y, z > and <x + dx, y + dy, z + dz >
                   */
                if(index < 0 || index >= num_elements)
                        return false;
                float dx, dy, dz;
                dx = (float)abs((int)particle_list[index].position.x)
+ disp;
                dy = (float)abs((int)particle_list[index].position.y)
+ disp;
                dz = (float)abs((int)particle_list[index].position.z)
+ disp;
                if(particle_list[index].position.x == x &&
particle_list[index].position.y == y && particle_list[index].position.z
== z)
                                return true;
                /* determine if within 3D space < dx, dy, dz > */
                else if(particle_list[index].position.x < dx &&
particle_list[index].position.y < dy && particle_list[index].position.z
< dz)
                        return true;
                else if(particle_list[index].position.x > (-1)*dx &&
particle_list[index].position.y < dy && particle_list[index].position.z
< dz)
                        return true;
                else if(particle_list[index].position.x >(-1)*dx &&
particle_list[index].position.y > (-1)*dy &&
particle_list[index].position.z < dz)
                        return true;
                else if(particle_list[index].position.x > (-1)*dx &&
particle_list[index].position.y > (-1)*dy &&
particle_list[index].position.z > (-1)*dz)
                        return true;
                else if(particle_list[index].position.x < dx &&
particle_list[index].position.y > (-1)*dy &&
particle_list[index].position.z < dz)
                        return true;
                else if(particle_list[index].position.x < dx &&
particle_list[index].position.y > (-1)*dy &&
particle_list[index].position.z > (-1)*dz)
                        return true;
                else if(particle_list[index].position.x < dx &&
particle_list[index].position.y < dy && particle_list[index].position.z
> (-1)*dz)
                        return true;
                else
                        return false;
        }

        int WriteDataToFile()
```

```
            {
                  for(int f = 0; f < num_particles; f++)
                  {
                        energy_file << (float)particle_list[f].energy
<< endl;
                        dispersion_file <<
(float)particle_list[f].delta_energy << endl;
                  }
                  return SUCCESS;
            }
            Q_Particle *ParticleList() {return *&particle_list;)

      protected:
            void InitializeAll(int index)
            {
                  particle_list[index].position.x = 2;
                  particle_list[index].position.y = 4;
                  particle_list[index].position.z = 2;
                  particle_list[index].mass = MAX_MASS;
                  particle_list[index].energy = 0; // no starting
energy
                  particle_list[index].charge = 0.0f; // neutral charge
            }
            ofstream heat_file; // heat change over time [optional]
            ofstream energy_file; // energy change over time
            ofstream dispersion_file; // delta energy
            ofstream charge_file; // charge change over time
            float height;
            float width;
            float depth;
            float strength_of_field;
};

#endif
/***********************************************

  Author :              Bill Knoop, Phillip Coleman
                              Alfredo Davila, Paul Boyle

  Date Created :  April 4, 2004

  File :                MFMath header file

  Comments :

***********************************************/

#ifndef MFMATH_H
#define MFMATH_H

using namespace std; // namespace standard

const float pi = 3.1415; // pi
const float mu = 4 * pi * pow(10, -7); // permeability of free space
const float epsilon = 9 * pow(10, 9); // coulomb's constant

/* function uses the equation :
```

```
   Fmag = (mu / 4 *pi) * ((q1 * q2)) / r^2) * v1 x (v2 x r^[unit
vector])
   */
float Magnetic_Force(Q_Particle *q1, Q_Particle *q2)
{
      q_vector result =
q2.velocity.CrossProduct(RadiusVector(q1.position, q2.position));
      result = result.CrossProduct(q1.velocity);
      float magnetic_force = (q1.charge * q2.charge) /
(float)pow((double)RadiusVector(q1.position, q2.position).Normalize(),
.5);
      magnetic_force = magnetic_force / RadiusVector(q1.position,
q2.position);
      magnetic_force *= mu / 4 * pi;
}

/* functions finds the radius vector [distance] between two objects */
float RadiusVector(q_vector pos_1, q_vector pos_2)
{
      return q_vector(pos_1.x - pos_2.x, pos_1.y - pos_2.y, pos_1.z -
pos_2.z);
}

/* function uses the basic equation
   [Fmag = radius x velocity x magnetic field]
   */
float MagneticFieldStrength(q_vector radius, q_vector velocity, float
b_field)
{
      radius = radius.CrossProduct(velocity);
      float fieldStrength = radius.Normalize() * b_field;
      return fieldStrength;
}

#endif
/*************************************************

  Author :              Bill Knoop, Phillip Coleman,
                              Alfredo Davila, Paul Boyle

  Date Created :  4/4/04

  File :                ParticleColor

  Comments :            Minor utility used to help
                              render the particle systen
                              and magnetic field

*************************************************/

#ifndef PARTICLECOLOR_H
#define PARTICLECOLOR_H

using namespace std; // namespace standard

#define PC_NEUTRAL            1
#define PC_CHARGE       2
```

```
#define PC_DISPERSION   3
#define PC_ENERGY       4
#define PC_HEAT             5

struct GLCOLOR{
      float color[3];
};

GLCOLOR colorVector[8] = {
                                          1.0f, 0.0f, 0.0f, // red
                                          1.0f, 0.5f, 0.0f, // orange
                                          1.0f, 1.0f, 0.0f, // yellow
                                          0.0f, 1.0f, 0.0f, // green
                                          0.0f, 0.0f, 1.0f, // blue
                                          1.0f, 0.0f, 1.0f, // purple
                                          0.5f, 0.5f, 0.5f, // grey
                                          1.0f, 1.0f, 1.0f, // white
                                          0.0f, 0.0f, 0.0f // black
};

union COLOR_INDEX{
      bool neutral;
      bool charge;
      bool dispersion;
      bool energy;
      bool heat;
};

class ParticleColor
{
      public:
            ParticleColor();
            ~ParticleColor();

            /* accessor functions */
            int ParticleColorIndex(Q_Particle *pList, int
numParticles);
            /* modifier functions */
            void SetIndex(int pcState);

      private:
            float Max(Q_Paricle *pList, int numParticles);
            float Min(Q_Particle *pList, int numParticles);
            COLOR_INDEX *pc_Index;
};

void ParticleColor::SetIndex(int pcState)
{
      /* set pc_Index to correct color index allocation */
      if(pcState == PC_NEUTRAL)
            pc_Index->neutral = true;
      else if(pcState == PC_CHARGE)
            pc_Index->charge = true;
      else if(pcState == PC_ENERGY)
            pc_Index->energy = true;
      else if(pcState == PC_DISPERSION)
            pc_Index->dispersion = true;
```

```
	else if(pcState == PC_HEAT)
		pc_Index->heat = true;
	/* if pcState out of bounds - no effect is made */
}

int ParticleColor::ParticleColorIndex(Q_Particle *pList, int
numParticles)
{
	if(pList == NULL)
		return FAIL;
	/* determine ranges */
	float min = Min(pList, numParticles);
	float max = Max(pList, numParticles);
	/* set to abs of min for calculation */
	float abs_max = (float)abs((int)max);
	float abs_min = (float)abs((int)min);
	/* find the increment (dr) over which to determine
	   color ranges [max - min] / numColors
	   */
	float dr = (abs_max - abs_min) / 8;
	float range[8];
	/* begin calculation of intervals
	   using r = min + dr - assumes min
	   is negative
	   */
	range[0] = min + dr;
	for(int u = 1; u < 8; u++)
	{
		range[u] = range[u - 1] + dr;
	}
	for(int ci = 0; ci < numParticles; ci++)
	{
		if(pc_Index->neutral == true)
		{
			/* make invisible to background */
			floatcpy(pList[ci].color, colorVector[7], 3);
		}
		/* calculate indexs for charge, energy, dispersion */
		else if(pc_Index->charge == true)
		{
			if(pList[ci].charge =< range[0])
				floatcpy(pList[ci].color, colorVector[0], 3);
			for(int cp = 1; cp < 8; cp++)
			{
				if(pList[ci].charge =< range[cp] &&
pList[ci].charge > range[cp - 1])
					floatcpy(pList[ci].color,
colorVector[cp], 3);
			}
		}
		else if(pc_Index->energy == true)
		{
			if(pList[ci].energy =< range[0])
				floatcpy(pList[ci].color, colorVector[0], 3);
			for(int cp = 1; cp < 8; cp++)
			{
```

```
                        if(pList[ci].energy =< range[cp] &&
pList[ci].charge > range[cp - 1])
                              floatcpy(pList[ci].color,
colorVector[cp], 3);
                  }
            }
            else if(pc_Index->dispersion == true)
            {
                  if(pList[ci].dispersion =< range[0])
                        floatcpy(pList[ci].color, colorVector[0], 3);
                  for(int cp = 1; cp < 8; cp++)
                  {
                        if(pList[ci].dispersion =< range[cp] &&
pList[ci].charge > range[cp - 1])
                              floatcpy(pList[ci].color,
colorVector[cp], 3);
                  }
            }
      }
      return SUCCESS;
}

float ParticleColor::Min(Q_Particle *pList, int numParticles)
{
      if(pList == NULL)
            return (float)FAIL;
      float min = 0;
      for(int pc = 0; pc < numParticles; pc++)
      {
            /* determine index state */
            if(pc_Index->charge == true)
            {
                  if(pList[pc].charge < temp)
                        min = pList[pc].charge;
            }
            else if(pc_Index->energy == true)
            {
                  if(pList[pc].charge < temp)
                        min = pList[pc].energy;
            }
            else if(pc_Index->dispersion == true)
            {
                  if(pList[pc].charge < temp)
                        min = pList[pc].delta_energy;
            }
      }
      return min;
}

float ParticleColor::Max(Q_Particle *pList, int numParticles)
{
      if(pList == NULL)
            return (float)FAIL;
      float max = 0;
      for(int pc = 0; pc < numParticles; pc++)
      {
            /* determine index state */
```

```
            if(pc_Index->charge == true)
            {
                  if(pList[pc].charge > temp)
                        max = pList[pc].charge;
            }
            else if(pc_Index->energy == true)
            {
                  if(pList[pc].charge > temp)
                        max = pList[pc].energy;
            }
            else if(pc_Index->dispersion == true)
            {
                  if(pList[pc].charge > temp)
                        max = pList[pc].delta_energy;
            }
      }
      return max;
}

#endif
#ifndef RANDOM_H
#define RANDOM_H

class Random
{
      public:
            Random()
            {
                  time_t seconds;
                  time (&seconds);
                  srand ((unsigned int) seconds);
            }
            Random(unsigned int seed)
            {
                  time_t seconds;
                  time (&seconds);
                  srand ((unsigned int) seed);
            }
            ~Random() {}

            float nextFloat(int val)
            {
                  return (float)(rand() % val);
            }
};

#endif
#ifndef SOLARFLARE_H
#define SOLARFLARE_H

class SolarFlare : public Sun
{
      public:
            SolarFlare();
            ~SolarFlare();

            float ComputeFlareEnergy()
```

```
            {
                  return FlareEnergy(solar_temperature, solar_emission,
solar_area, solar_height);
            }
            float ComputeFlareArea()
            {
                  return AreaTotal(a, solar_energy, alpha);
            }
            float ComputeEmissionIncrease()
            {
                  return EmissionIncreaseTotal(c, solar_energy,
lambda);
            }

            void UpdateGridPosition()
            {
                  for(int l = 0; l < length; l++)
                  {
                        for(int h = 0; h < height; h++)
                        {
                              solar_flare[l][h].position.x +=
solar_flare[l][h].velocity.h;
                        }
                  }
                  distance_from_earth -= 1000;
            }

            void UpdateSolarFlare(float deltaTime)
            {

            }

            void FlareEruption()
            {
                  /* calculate flare height */
                  solar_height = Height(b, max_energy, beta);
                  /* calculate flare area */
                  solar_area = AreaTotal(a, max_energy, alpha);
                  float dimensions = (float)pow((double)solar_area,
.5);
                  length = dimensions;
                  height = dimensions;
                  InitializeGrid(0.0f, 0.0f, length, height, 0.0f);
                  /* calculate total flare energy */
                  solar_energy = FlareEnergy(solar_temperature,
solar_emission, solar_area, solar_height);
                  bool erupted = false;
                  for(int i = 0; i < length; i++)
                  {
                        for(int a = 0; a < height; a++)
                        {

      solar_flare[i][a].SetPixelEnergy(minEnergy);
                        }
                  }
                  while(erupted != true)
                  {
```

```
                        /* allocate all variables on solar grid */
                        for(int l = 0; l < length; l++)
                        {
                              for(h = 0; h < height; h++)
                              {

      solar_flare[l][h].SetPixelEmission(solar_flare[l][h].PixelEmissio
n() + Pixel_funcEmissionIncrease(solar_flare[l][h].PixelArea(),
solar_area, solar_emission);

      solar_flare[l][h].SetPixelEnergy(solar_flare[l][h].PixelEnergy()
+ Pixel_funcEnergyIncrease(alpha, solar_flare[l][h].PixelArea(),
solar_area, solar_flare[l][h].PixelEnergy());
                                    if(solar_flare[l][h].PixelEnergy()
> maxEnergy)
                                          erupted = true;
                              }
                        }
                  }
            }

            bool WithinRange()
            {
                  if((distance_from_earth < 6250) ||
((distance_from_earth - 1000) < 6250))
                        return true;
                  else
                        return false;
            }

            void SetEnergyLevels(float min, float max) {minEnergy =
min; maxEnergy = max;}

            float DistanceFromEarth() {return distance_from_earth;};

            void SetConstants(float _a, float _b, float _c, float
_alpha, float _beta, float _lambda, float _sigma)
            {
                  a = _a;
                  b = _b;
                  c = _c;
                  alpha = _alpha;
                  beta = _beta;
                  lambda = _lambda;
                  sigma = _sigma;
            }

            float EnergyAt(float x, float y, float z, float l, float h)
            {
                  for(int len = 0; len < length; len++)
                  {
                        for(int ht = 0; ht < height; ht++)
                        {
                              if(solar_flare[len][ht].position.x == x
&& solar_flare[len][ht].position.y == y &&
solar_flare[len][ht].position.z == z)
                              {
```

```
                                    return
solar_flare[len][ht].PixelEnergy();
                                }
                        }
                }
        }

    private:
        void InitializeGrid(float z, float y, float length, float
height, float x)
        {
            height = (int)height;
            length = (int)length;
            float *tempFlare = (float *)malloc(length * height *
sizeof(float));
            if(temptempFlare == NULL)
            {
                #define FLARE_GRID_CREATION_ERROR
                // DEBUG();
                return;
            }
            solar_flare = (float **)malloc(length * sizeof(float
*));
            for(int l = 0; l < length; r++)
                solar_flare[l] = tempFlare + (l * height);
            for(int l = 0; l < length; l++)
            {
                for(int h = 0; h < height; h++)
                {

    solar_flare[l][h].SetPixelX(astronomical_unit);
                }
            }
        };
        SolarGrid **solar_flare;
        int num_pixels;
        int length, height;
        float distance_from_earth;
        float maxEnergy, minEnergy;
};

#endif
#ifndef SOLARGRID_H
#define SOLARGRID_H

class SolarPixel
{
    public:
        SolarPixel();
        ~SolarPixel();

        void SetPixelVelocity(q_vector pVelocity) {pixel_velocity =
pVelocity;}
        void SetPixelMass(float pMass) {pixel_mass = pMass;}
        void SetPixelEnergy(float pEnergy) {pixel_energy =
pEnergy;}
        void UpdatePixelPosition() {pixel_position.x += 1;}
```

```
            void SetPixelEmission(float pEmission) {pixel_emission =
pEmission;}
            void SetPixelDensity{float pDensity) {pixel_density =
pDensity;}
            void SetPixelX(float x) {pixel_position.x = x;}
            void SetPixelPosition(quantum_vector pixel_vec)
{pixel_velocity = pixel_vec;}
            float Energy() {return pixel_energy;}
            void UpdatePixelEnergyLoss()
            {
                  if(pixel_charge < 0)
                  {
                        pixel_energy = pixel_energy -
EnergyLossPerElectron((float)(1.6 * pow(10, -19)), pixel_mass,
atom_density, 1.0f, pixel_density, pixel_mass, pixel_KE,
pixel_velocity.Normalize(), (float)(1.38 * pow(10, -10)));
                  }
                  else
                  {
                        pixel_energy = pixel_energy -
EnergyLossPerProton(1.0f, (float)(1.6 * pow(10, -19)), pixel_charge,
pixel_mass, pixel_velocity.Normalize(), atom_density, 1.0f,
pixel_density, (float)(1.38 * pow(10, -10)));
                  }
            }
            float PixelEmission() {return pixel_emission;}
            float PixelArea() {return pixel_area;}
            float PixelMass() {return pixel_mass;}
            float PixelDensity() {return pixel_density;}
      private:
            float area;
            float pixel_emission;
            float pixel_energy;
            quantum_vector pixel_position;
            quantum_vector pixel_velocity;
            float pixel_mass;
            float pixel_density;
};

#endif
#ifndef SOLARMATH_H
#define SOLARMATH_H

#define MKILOMETER 1
#define KILOMETER 2
#define METER 3
#define CENTIMETER 4

const float Boltmanns_Constant = (float)(1.3806505 * pow(10, -23)); //
J / K

float distance_mod = 1;

void SetModifier(DISTANCE_INDEX *dIndex)
{
      if(dIndex->mkilometer == true)
            distance_mod = (float)pow(10, 6);
```

```
	else if(dIndex->kilometer == true)
		distance_mod = (float)pow(10, 3);
	else if(dIndex->meter == true)
		distance_mod = (float)pow(10, 2);
	else if(dIndex->centimeter == true)
		distance_mod = (float)pow(10, 1);
}

float FlareEnergy(float temperature, float emmision_increase, float
area, float height)
{
	return (3 * Boltzmanns_constant * temperature) *
(float)(pow((double)(emission_increase * area * height), .5));
}

float FlareFrequency(float flare_energy, float flare_events, float
constant_of_proportion)
{
	return flare_events * (float)(pow((double)flare_energy,
(double)constant_of_proportion));
}

float EmissionIncreaseTotal(float c, float energy, float lambda)
{

}

float AreaTotal(float a, float energy, float alpha)
{

}

float Height(float b, float energy, float beta)
{

}

float _u(float lambda, float alpha)
{

	return (float)(lambda / alpha);
}

float _m(float alpha, float u)
{
	return (float)(speed_of_light * (float)(pow((double)alpha,
(double)u)));
}

float EnergyLossPerProton(float z, float q, float zq, float M, float v,
float N, float Z, float NZ, float I)
{
	float top_fraction = 4 * pi * z * z * q * q * q * q * NZ *
(float)(pow((3 * pow(10, 9)), 4);
	float bottom_fraction = (M * v * v) * (float)(16 * pow(10, -6));
	float first_ln = log((2 * M * v * v) / I);
```

```
      float second_ln = log(1 - ((v * v) / (speed_of_light *
speed_of_light)));
      float last_part = (v * v) / (speed_of_light * speed_of_light);
      return (top_fraction / bottom_fraction) * (first_ln * second_ln *
last_part);
}

float EnergyLossPerElectron(float q, float M, float N, float Z, float
NZ, float eM, float eK, float v, float I)
{
      float top_fraction = 2 * pi * q * q * q * q * NZ * (float)pow((3
* pow(10, 9), 4));
      float bottom_fraction = eM * (v / speed_of_light) *
(float)pow((16 * pow(10, -6)), 2);
      float large_ln = log((eM * eK * (v / speed_of_light) * (v /
speed_of_light)) / (I * I *(1 - ((v / speed_of_light) * (v /
speed_of_light)))));
      return (top_fraction / bottom_fraction) * large_ln - ((v /
speed_of_light) * (v / speed_of_light));
}

float _funcM(float u, float EA, float m)
{
      return m * (float)(pow((double)EA, (double)u));
}

float M_functT(float c, float E, float lambda, float time, float s,
float deltaTime)
{
      return c * (float)(pow((double)E, (double)lambda)) * g_time *
(float)(pow((double)E, (double)(-1)*s)) * deltaTime;
}

float Pixel_funcEnergyIncrease(float alpha, float pixel_area, float
sigma, float pixel_energy)
{
      float dA = (1 / (1 - alpha));
      float dS = (2 - sigma) / (1 - alpha);
      float dAp = (float)pow((double)(alpha / pixel_area), (double)dS);
      float dSp = (-1)*((sigma - (2*alpha)) / (1 - alpha));
      float dE = (float)pow((double)pixel_energy, (double)dSp);
      return dA * dAp * dE;
}

float Pixel_funcEnergyIncrease(float pixel_area, float area, float
energy)
{
      return ((pixel_area / area) * energy);
}

float Pixel_funcEmissionIncrease(float pixel_area, float area, float
emission_rate)
{
      return (pixel_area / (area * emission_rate));
}
```

```
#endif
#ifndef SUN_H
#define SUN_H

class Sun
{
	public:
		Sun();
		~Sun();

		virtual float ComputeFlareEnergy();
		virtual float ComputeEmissionMeasureIncrease();
		virtual float ComputeFlareArea();

	private:
		virtual float
		float solar_height;
		float solar_area;
		float solar_energy;
		float solar_emission;
		float solar_temperature;
		/* predefined constants */
		float a, b, c;
		float alpha, beta, lambda, sigma;
};

#endif
/************************************************

  Author :		Bill Knoop, Phillip Coleman,
			Alfredo Davila, Paul Boyle

  Date Created :  4/4/04

  File :		TimerAPI.h

  Comments :		class that makes use of the high
			resolution system timer - used
			instead of the win32 timer for
			better performance

************************************************/

#ifndef TIMERAPI_H
#define TIMERAPI_H

class TimerAPI
{
	public:
		TimerAPI() {};
		~TimerAPI() {};

		bool LoadSystemSupport();
		float SecondsPassed(unsigned long frames = 1);
		float FramesPerSecond(unsigned long frames = 1);
		float LockFrames(unsigned char FPS);
	private:
```

```
            LARGE_INTEGER start_time;
            LARGE_INTEGER last_time;
            LARGE_INTEGER tps;
};

bool TimerAPI::LoadSystemSupport()
{
      if(QueryPerformanceFrequency(&tps))
      {
            return false;
      }
      else
      {
            QueryPerformanceCounter(&start_time);
      }
}

float TimerAPI::SecondsPassed(unsigned long frames = 1)
{
      last_time = start_time;
      LARGE_INTEGER current_time;
      QueryPerformanceCounter(&current_time);
      float seconds = ((float)current_time.QuadPart -
(float)last_time.QuadPart) / (float)tps.QuadPart;
      last_time = current_time;
      return seconds;
}

float TimerAPI::FramesPerSecond(unsigned long frames = 1)
{
      last_time = start_time;
      LARGE_INTEGER current_time;
      QueryPerformanceCounter(&current_time);
      float fps = (float)frames * (float)tps.QuadPart /
((float)current_time.QuadPart - (float)last_time.QuadPart);
      last_time = current_time;
      return fps;
}

float TimerAPI::LockFrames(unsigned char FPS)
{
      if(FPS == 0)
            FPS = 1;
      last_time = start_time;
      LARGE_INTEGER current_time;
      float fps;
      do
      {
            QueryPerformanceCounter(&current_time);
            fps = (float)tps.QuadPart / ((float)(current_time.QuadPart
- last_time.QuadPart));
      } while(fps > (float)FPS);
      last_time = current_time;
      return fps;
}

#endif
```

Bibliography

http://science.nasa.gov/ssl/pad/solar/
http://farside.ph.utexas.edu/teaching/plasma/lectures/node65.html
http://hesperia.gsfc.nasa.gov/sftheory/flare.htm
http://sohowww.nascom.nasa.gov/explore/faq/flare.html#FLARE_STARS
http://www.rel.org/docs/thesis/chap2.html
http://casa.colorado.edu/~wcash/APS3730/chapter5.pdf
http://arxiv.org/PS_cache/astro-ph/pdf/0104/0104218.pdf