Rolling Star

New Mexico
Supercomputing Challenge

Final Report
April 2, 2008

Team 008
Albuquerque Academy

Team Members:
Brandon Smith
Wesley Smalls

Sponsoring Teacher:
Jim Mims

Project Mentor:
Kiran Manne

Table of Contents

Cover Page	1
Table of Contents	2
Executive Summary	3
Problem Statement	4
Description of Major Calculation Methods	5
Description of Major Routines	9
Results	12
Conclusions	13
Project Achievements	14
Recommendations	15
Acknowledgements	16
References and Citations	17
Appendices	18
Appendix A: Program Screenshots	18
Appendix B: Program Code	30

Executive Summary

This Project attempts to model a star through many stages of its life, from the Bok Globule (or nebula) stage, all the way through to the star's nova (super or otherwise) event and its aftermath

Because the purpose of the project was to have a person be able to study a star from any angle and move freely around and within it, we chose to write the final program's code in C++ with the OpenGL graphics library. This allowed us to easily produce stunning 3-D representations of the star, its surroundings, and its internal workings.

The final program is based around the principle of taking large packets of mass and having them mimic the activity of an atom or molecule. We chose this method so that the simulation could be carried out without trying to calculate everything for every atom within a star.

The "macros" that the simulation is based around have several attributes that dictate their behavior. Using a traditional "Sweep" method in which every macro is examined individually, the Program applies all forces and effects to each macro through every time step, thereby creating an animation of the star.

The results of the testing show promising results. In almost every simulation, a star successfully forms and many of its more common phenomena can be easily seen. However, there is always room for improvement and any simulation can be made to be more exact. Ours is no exception.

Problem Statement

Stars have always held a large scientific interest for scientists in many different fields. Some of the reasons for this are: they emit multiple types of electromagnetic radiation, project charged particles from their surface creating effects such as the solar wind, and they produce some of the most powerful gravitational and electromagnetic effects in the universe. Because of this, scientists have yearned for ways to predict the many effects and phenomena caused by stars for reasons such as: weather prediction, safe space travel and study, to learn about how solar systems and galaxies are formed, and the simple advancement of astronomy and other fields.

Description of Major Calculation Methods

In our original prototype program, we strove to perfect the basic principles that would dictate how the macros would eventually interact. During the process of building the prototype program and testing it, we decided that there would need to be five main methods for dealing with all of the calculations pertaining to each individual macro. These methods are: dealing with gravity, dealing with electromagnetic forces, dealing with collisions and momentum, dealing with and transferring heat and energy, and finally dealing with fusion and its forces.

The method that we used for dealing with gravity is fairly simple and straight-forward. We use a loop to find the distance between two macros; use the gravitational force constant, the masses of the two macros and the distance between them to calculate a vector force on each macro, and apply that force to the macros in the form of acceleration. The loop then repeats this for every pair of macros in the system.

The method that we use for dealing with electromagnetic forces is very similar to the method used for dealing with gravity. It uses a loop to find the distance between two macros, and then uses this distance to calculate forces acting on each macro. However, because the electromagnetic force is so much stronger than gravity, and because the program only utilizes its effects as a

countermeasure against macros being able to appear inside one another in a time-step jump, the electromagnetic force is only recognized when two macros are reasonably close to one another. It is used as a repelling force only.

Collisions between macros are a very important aspect to the program. In real life, when a chunk of matter comes in contact with another chunk of matter, one would expect them to bounce off of each other (assuming they are moving and no external forces are being applied). The method that we developed for dealing with collisions between macros is a simple routine that conserves the energy of the two-macro system. In essence, it takes the sum of the vector quantities of the momentum from each macro, and divides it between them according to their masses.

When two macros collide, the law of conservation of momentum dictates that the total momentum of the system be conserved. However, the law of conservation of energy dictates that all of the system's energy must also be conserved. In order to handle this, the program converts all of the lost kinetic energy from a collision into stored or internal energy for the macros. This energy is realized through heat. The program then finds the sum of the two macros' energies, and splits it between the two on an energy-per-unit-of-mass basis. Until fusion begins within the star, collisions between macros are the only sources of energy.

In order to deal with fusion and its forces, the program must do several things, including keeping track of every macro's atomic number and state, their energy-per-unit-of-mass, and their proximity to other macros. Using all of

these factors, the program calculates whether or not a macro is capable of fusion. If it is capable, then when it comes in contact with another macro, the program calculates whether or not the new macro system is jointly capable of fusion. If it is, then the proper fusion reaction "takes place," and each of the macros goes their own separate ways with a new density, speed, and energy level.

Originally, we used a "Bleed off" method to deal with how the energy of fusion spread to the macros around a "fusing" macro. This method simply found all of the macros within a certain radius of the "fusing" macro and, based on their distances and the number of macros present, divided up the "fusing" macro's extra energy in small increments every time step. This had the effect of the macro "shining," and passing on its energy in the form of light. However, this method had several drawbacks: 1) it did not take into account how the macro would become excited and begin to move in a different way, 2) It did not take into account the effects of actual light and how it interacts with objects, 3) it assumed that all of the energy went to the macros around it and did not take into account the energy that simply leaked off into space.

In order to correct many of the "Bleed off" method's shortcomings, we devised a much simpler method to describe a macro's behavior while "fusing." The Wiggle method is simple: As the macro "fuses," it "wiggles," i.e. its velocity changes by the amount of energy that it releases every time step due to fusion. This method takes into account how the individual macro is affected by the forces of its own fusion. It also passes on this fusion energy by means of a

method that already exists within the program. Finally, it is much less resource intensive than the “Bleed off” method.

Description of Major Routines

All of the routines in the program are broken down into simple steps. There is at least one routine representing each of the methods in the program.

The first major routine that the program uses is the routine that initializes all of the graphics libraries and all of the other functions required by OpenGL in order to work properly. This routine, Init(), also sets up many of the needed machine states for OpenGL and, most importantly, creates all of the macros that will be active during the program's running time.

The routine that it calls to create the macros is CreateNewMacro(). This routine gives each macro a random position in 3-Dimensional space that is within half of a light year from the origin. It gives each macro a radius based on the total system mass, the number of macros, and the density of the "Hydrogen gas" that each of them is made up of. Also, it gives them a color of 100% red, and an initial velocity of 0. All of these values are stored in arrays.

During each of the time steps, the Display() routine is called. In this routine, the camera is positioned wherever the user specifies, and all of the macros are drawn in their respective position.

The routine to draw the macros is called DrawMacros(). This routine draws the macros in their correct positions in either a fully solid 3-D mode, a Wire Frame 3-D mode, or a point mode. The user can toggle between these modes by pressing the "m" key.

Because of how OpenGL operates, we decided to put all of the programs major calculations in the Idle() function. The Idle() function is called whenever the program is not drawing the scene. In this function, the camera position and orientation can be changed if the user has pressed on of the movement keys (w, e,a, s, d, f, j, k, l, ;, I, and o). After the camera is moved and or rotated, if the simulation has not been paused (space bar), it calls the major routines of the program: applyMacroForces(), moveMacros(), and checkMacroPositions().

The applyMacroForces() routine was originally going to be a a crossroads where all of the functions dealing with macro forces would be called, however, after some development of the code, the only function that it calls is applyGravity().

The moveMacros() function does exactly that, it moves the macros according to their current positions and speeds and generates new positions for them to occupy.

The checkMacroPositions() function is really one of the most important routines that the program calls. It checks the postions of all of the macros in relation with their radii, and if any of their radii are overlapping, then the conserveMomentum() routine is called.

The conserveMomentum() routine, again, does exactly what it says it does, it conserves the momentum n macro-to-macro systems. It compares the positions of macros that are overlapping, finds the sum of the two-macro-system, and divides it between them on a per-mass basis. From this routine,

the routines: fuseTogether(), transferHeat(), transferEnergy(), and moveSpecificMacros().

The transferHeat() routine transfers heat between macros. It finds the sum of the system's heat, and then divides it between them on a heat-per-mass basis.

The transferEnergy() routine does the same thing as the transferHeat() routine, except that it deals with the extraEnergy variable of the macros.

The moveSpecificMacros() routine simply moves each of the two overlapping macros far enough apart in opposite directions that they are no longer touching.

The fuseTogether() routine calculates all of the effects of fusion. It covers the compression of the macros' radii, and their increased energy output, which is represented by the macros increasing their speed randomly.

Results

In almost every one of the trial runs of the program, at all stages of development, the results that were generated by the program indicated that all of the benchmarked stages and routines were met and were functioning properly.

In the later stages of development, when we were able to add the effects of fusion to the macros and measure their increased energy output, we saw the effects of a stable, roughly circular sun. The center was represented by white-hot macros that had a smaller radius and a higher density. They were moving faster with much higher energy than were the cooler macros at the surface of the sun. We were able to distinguish evidence of “sun spots” and rough “solar flares” on the surface of the star that eventually formed.

Conclusions

In the prototypes of the program, we achieved a stable, navigable system for our simulation to run in. We then added macros and gravity, the basic force at work in the program. After gravity we added collision detection and the conservation of momentum. This led to the “generation of energy” from collisions, which led to fusion being possible between macros.

From the results that we obtained from running the program, we can safely assume that the simulation runs correctly at approximately the right time scale for whatever size star the program is simulating. It can be deemed a success.

Project Achievements

We decided upon a scientific topic for which we have developed a functional computational model. This model gives a visual representation of the events that happen during a star's life. The model takes the scientific principles and laws that we know govern the events of a star's life and give those laws a manifestation that further enables us to understand the effect the laws truly have.

Recommendations

Increase numbers and run on a computer that can efficiently handle the massive numbers needed for an accurate simulation. This will greatly improve the quality of the simulation and give a much more thorough representation of the laws involved in running the star's life.

Acknowledgements

We would like to first thank Mr. Jim Mims, our computer science teacher, for helping teach us everything needed to complete this project and helping us remember deadlines.

We would then like to thank Mr. Kiran Manne, our project mentor and Physics teacher, without whom we would never have been able to attempt this project.

Finally, we would like to thank all of our friends for putting up with us as we slowly (and likely very annoyingly) made our way through this project.

References

Written

ILLUSTRATED ENCYCLOPEDIA OF ASTRONOMY AND SPACE

Author: Thomas Y. Crowell, Published in: 1979 (first edition), Editor: Ian Ridpath

Internet

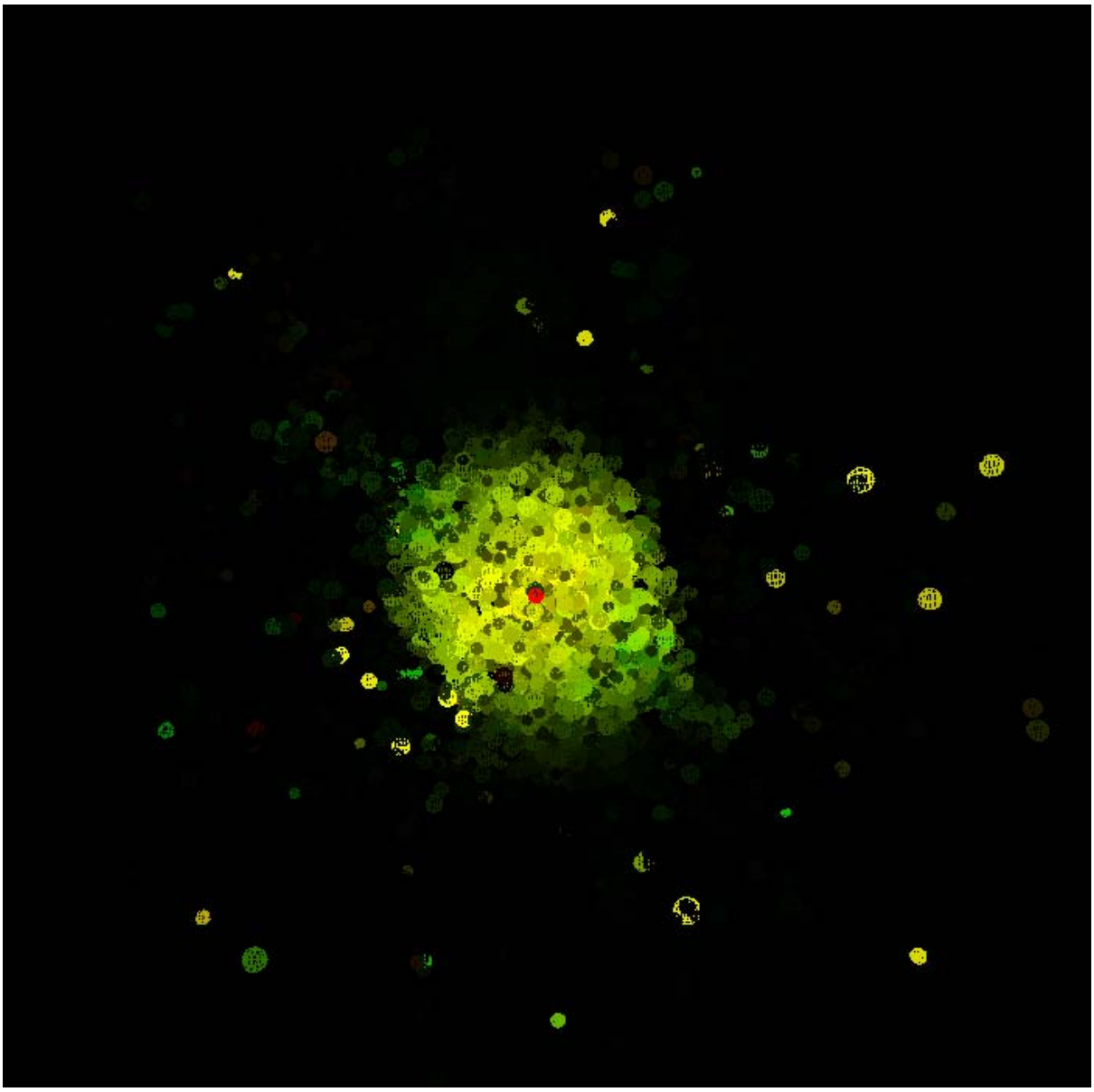
http://en.wikipedia.org/wiki/Nuclear_fusion

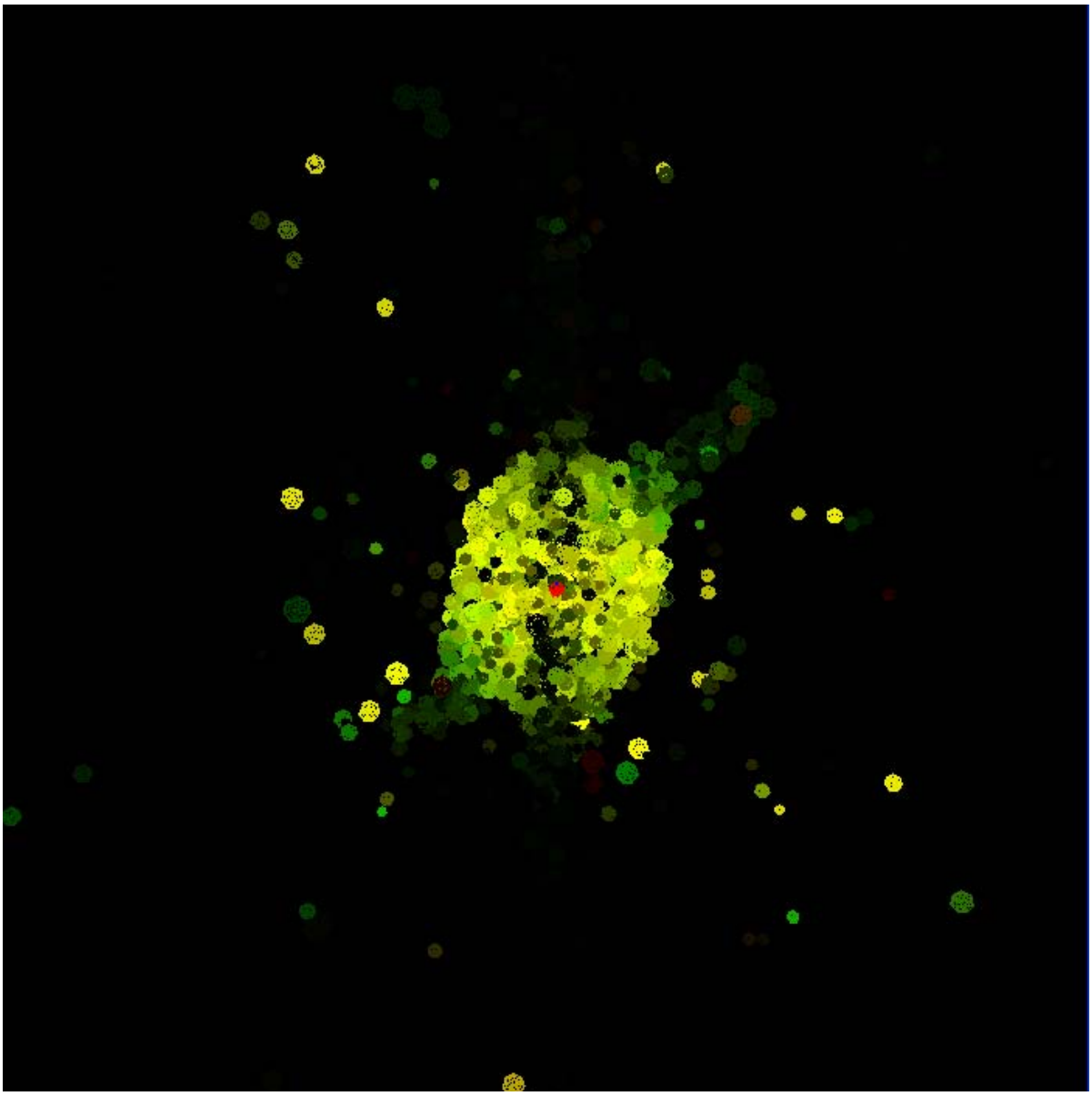
http://filer.case.edu/sjr16/advanced/stars_birth.html

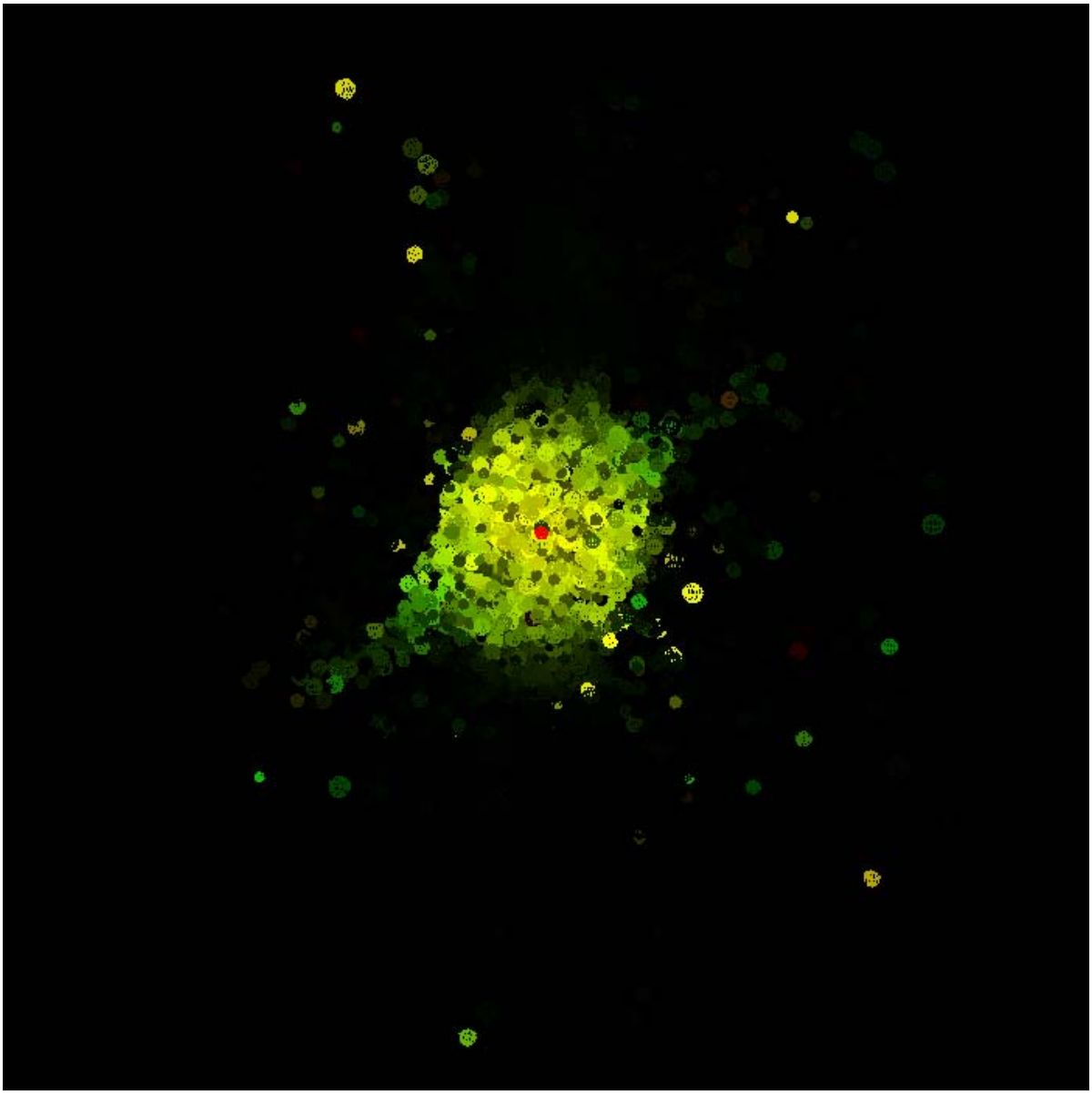
http://hyperphysics.phy-astr.gsu.edu/hbase/astro/snovcn.html

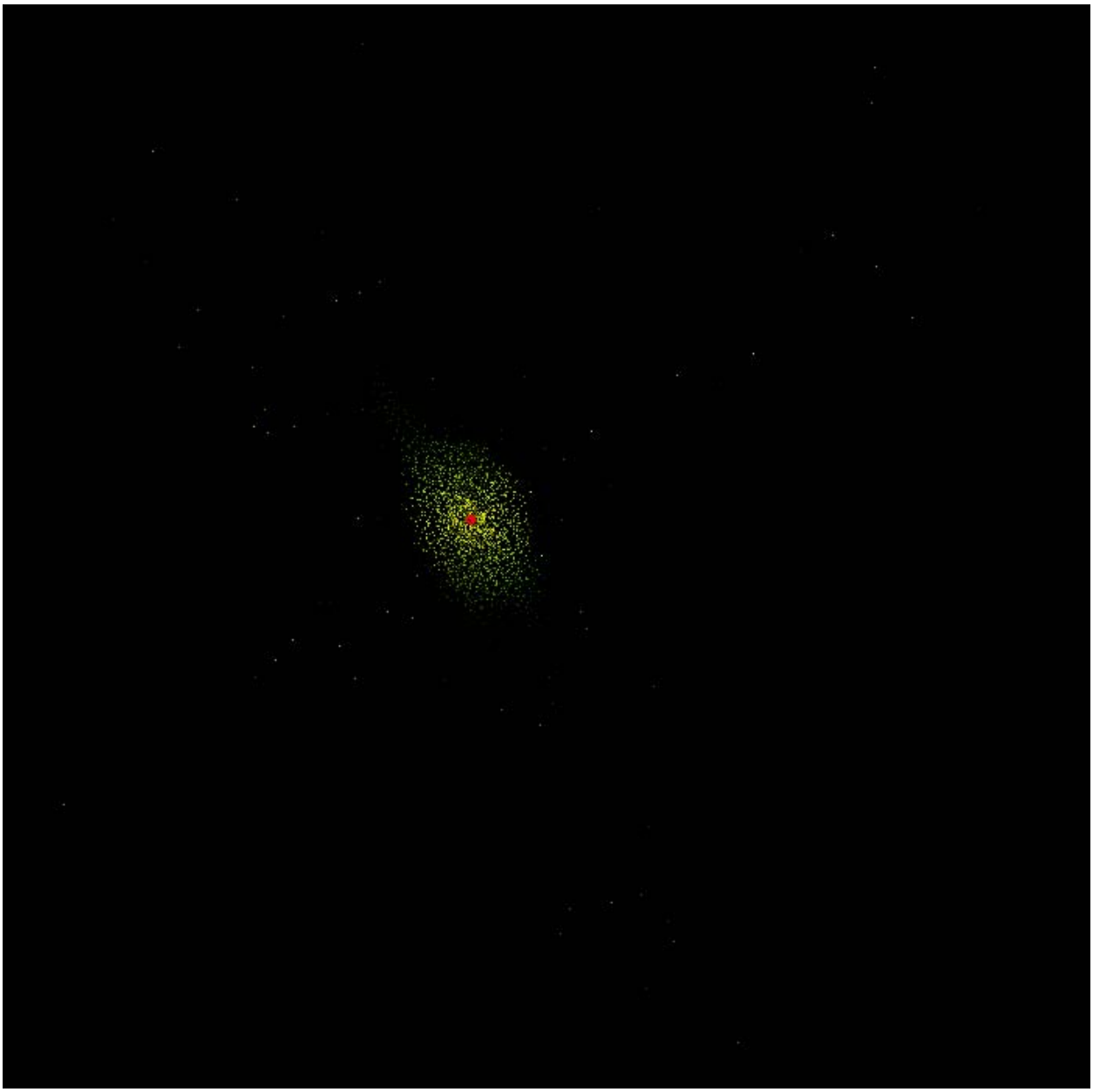
http://hyperphysics.phy-astr.gsu.edu/hbase/astro/nebula.html

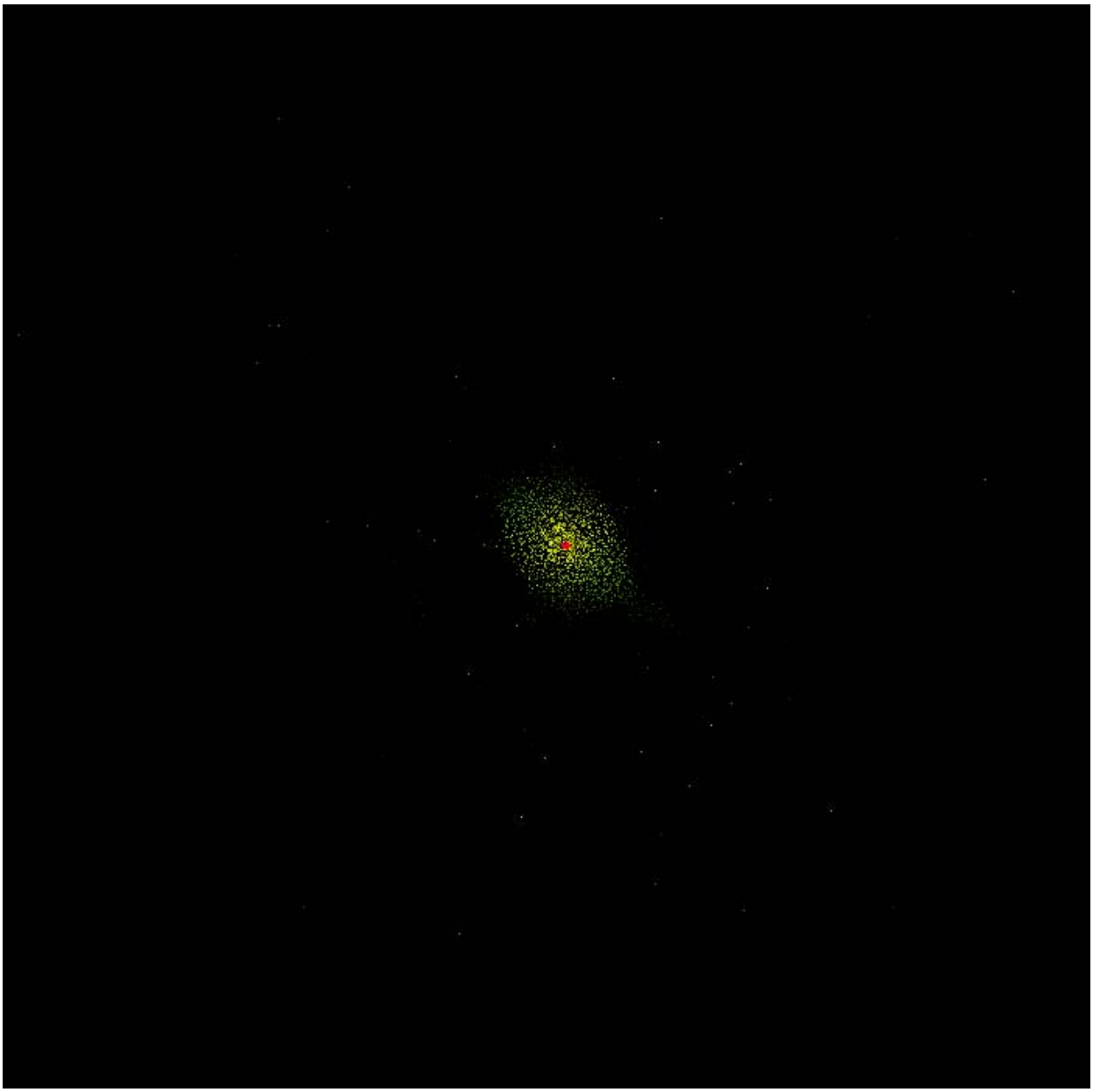
Appendix A: Project Screenshots

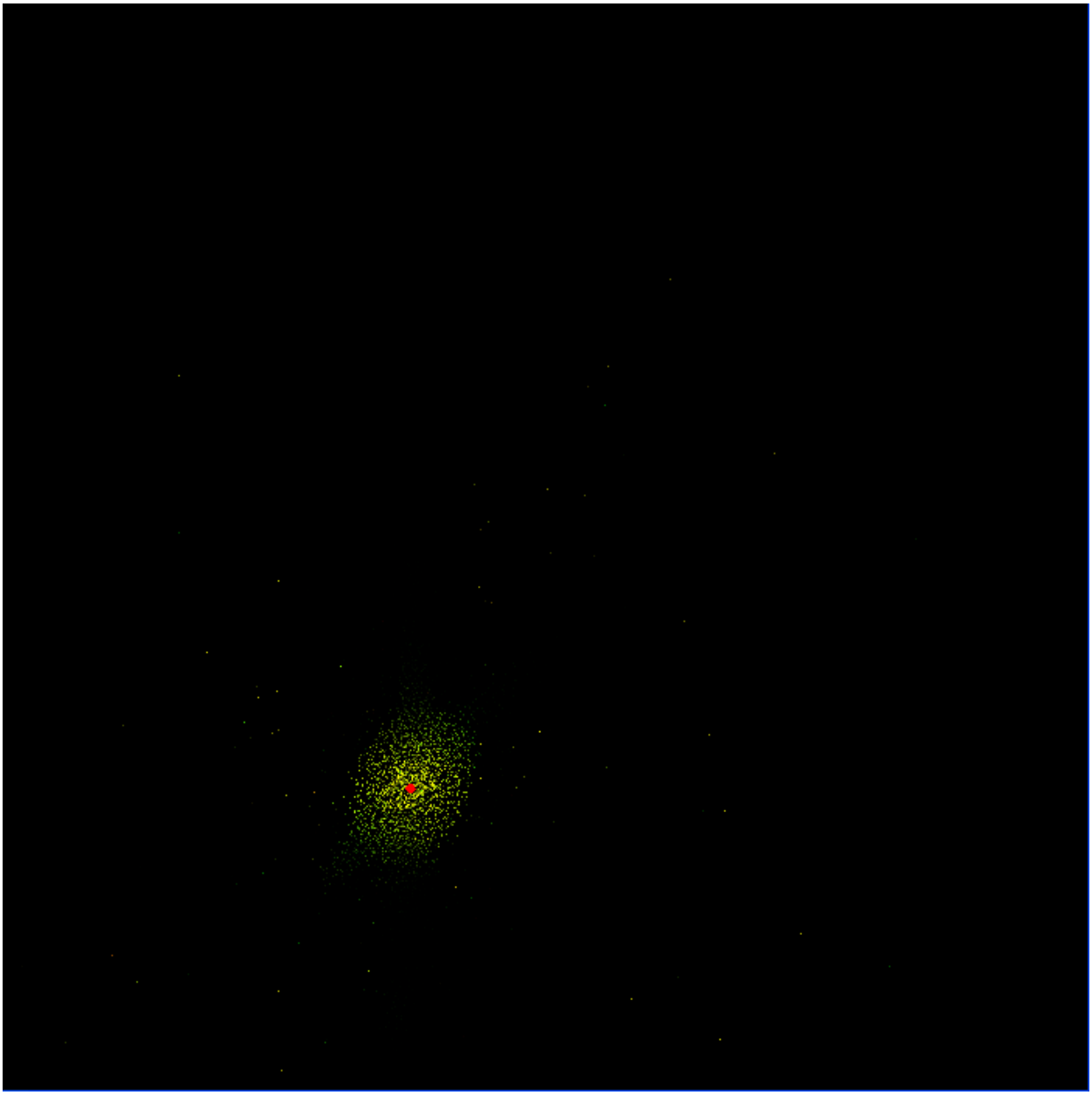


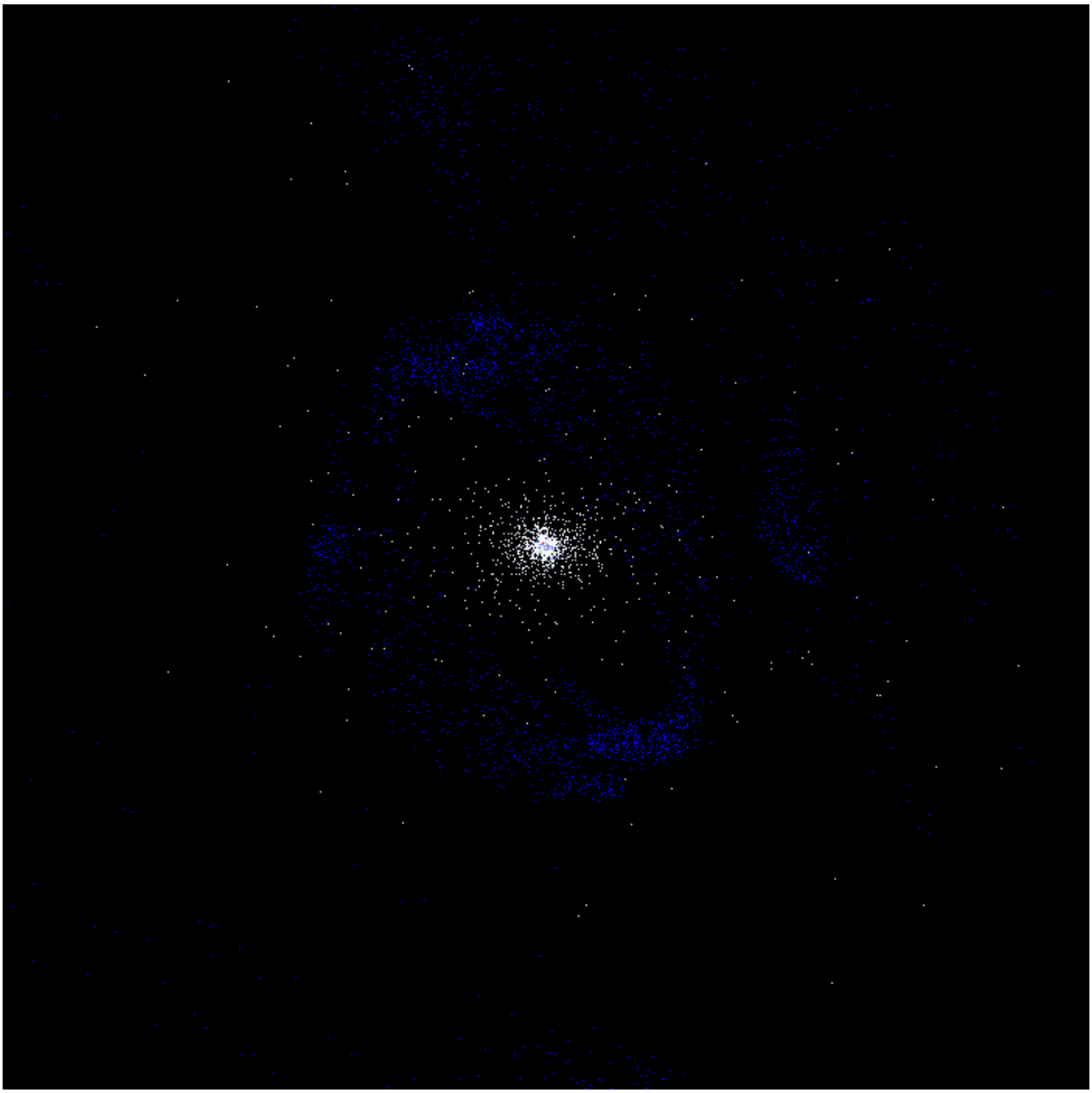


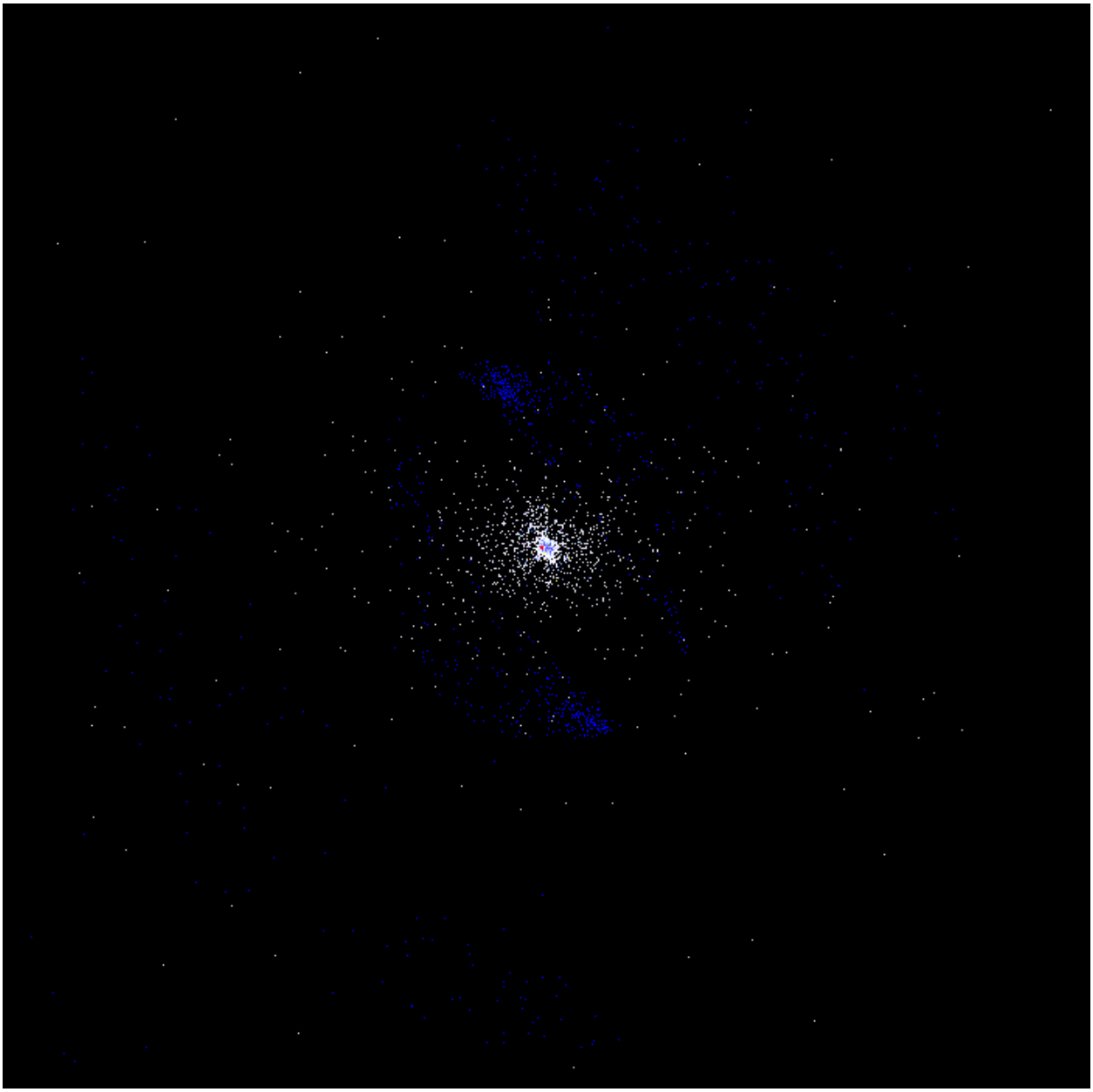


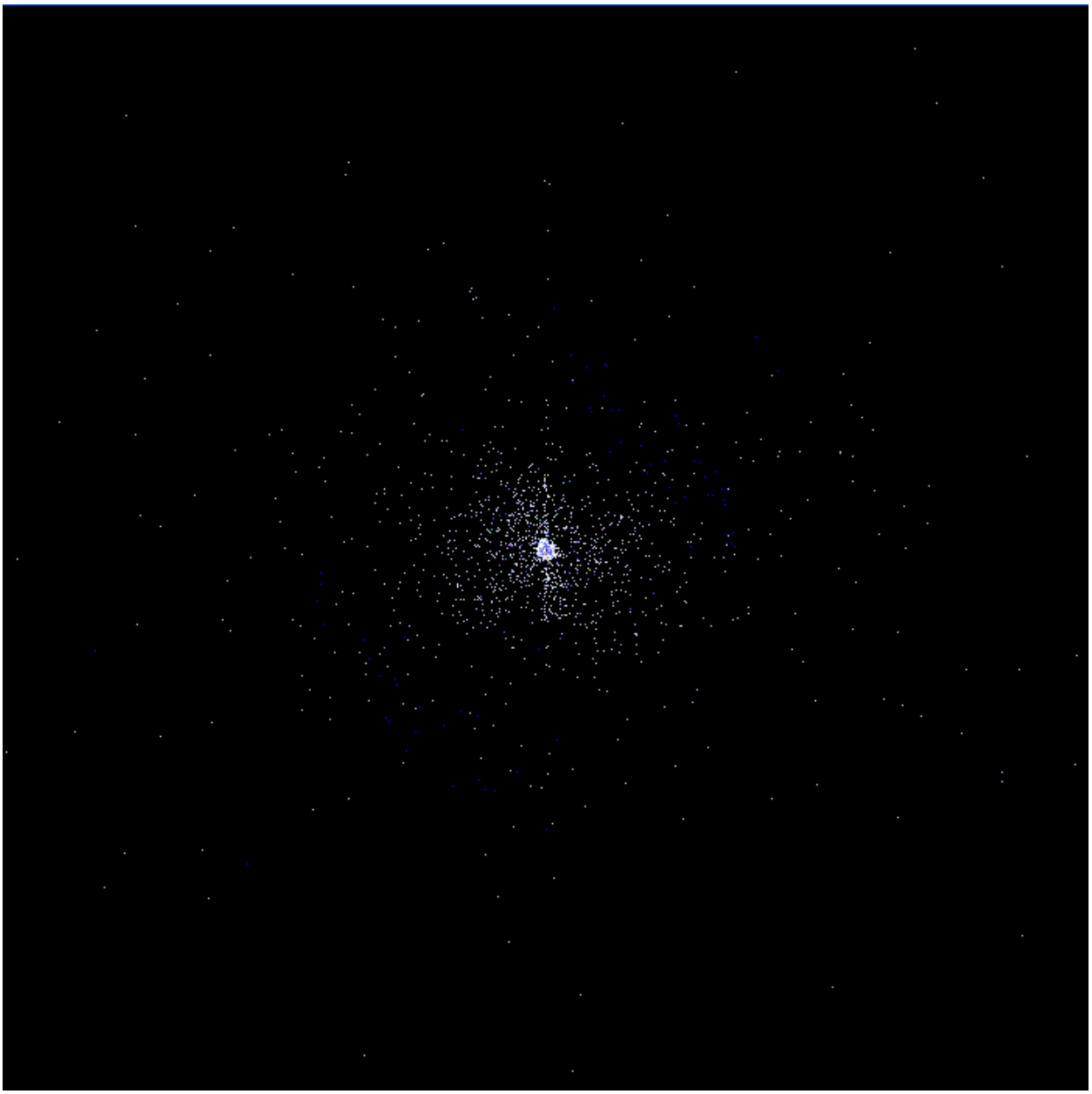


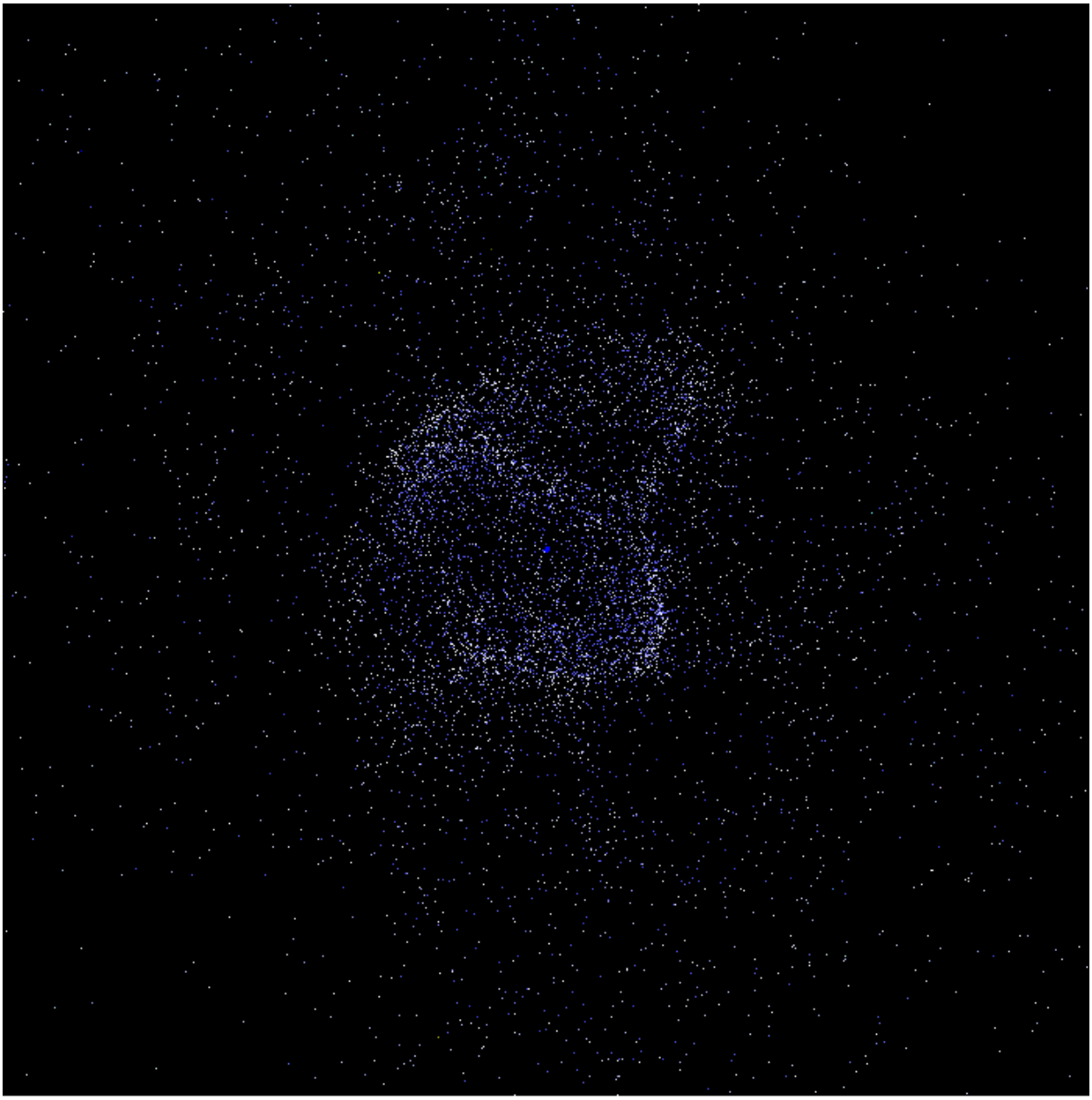


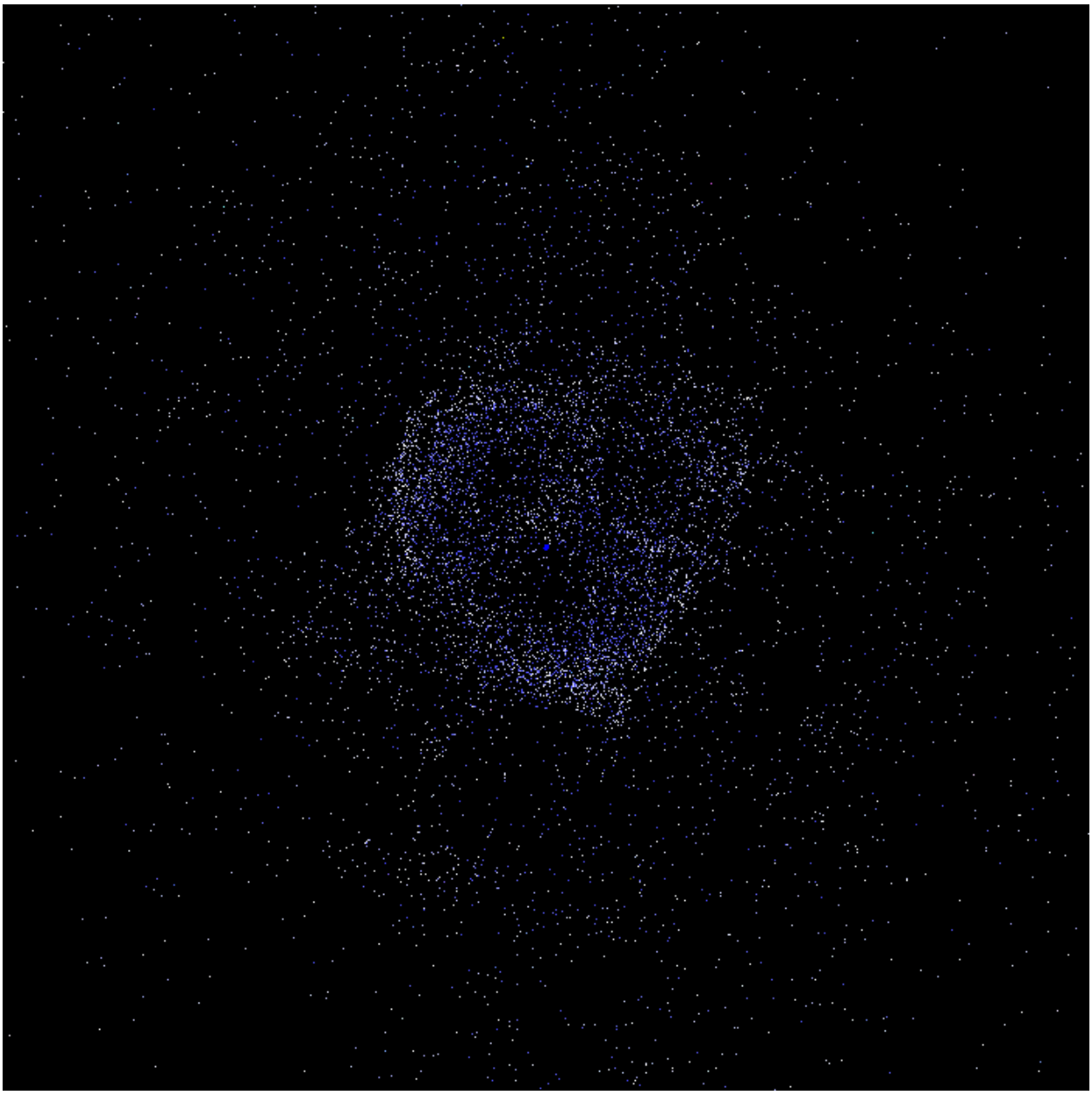


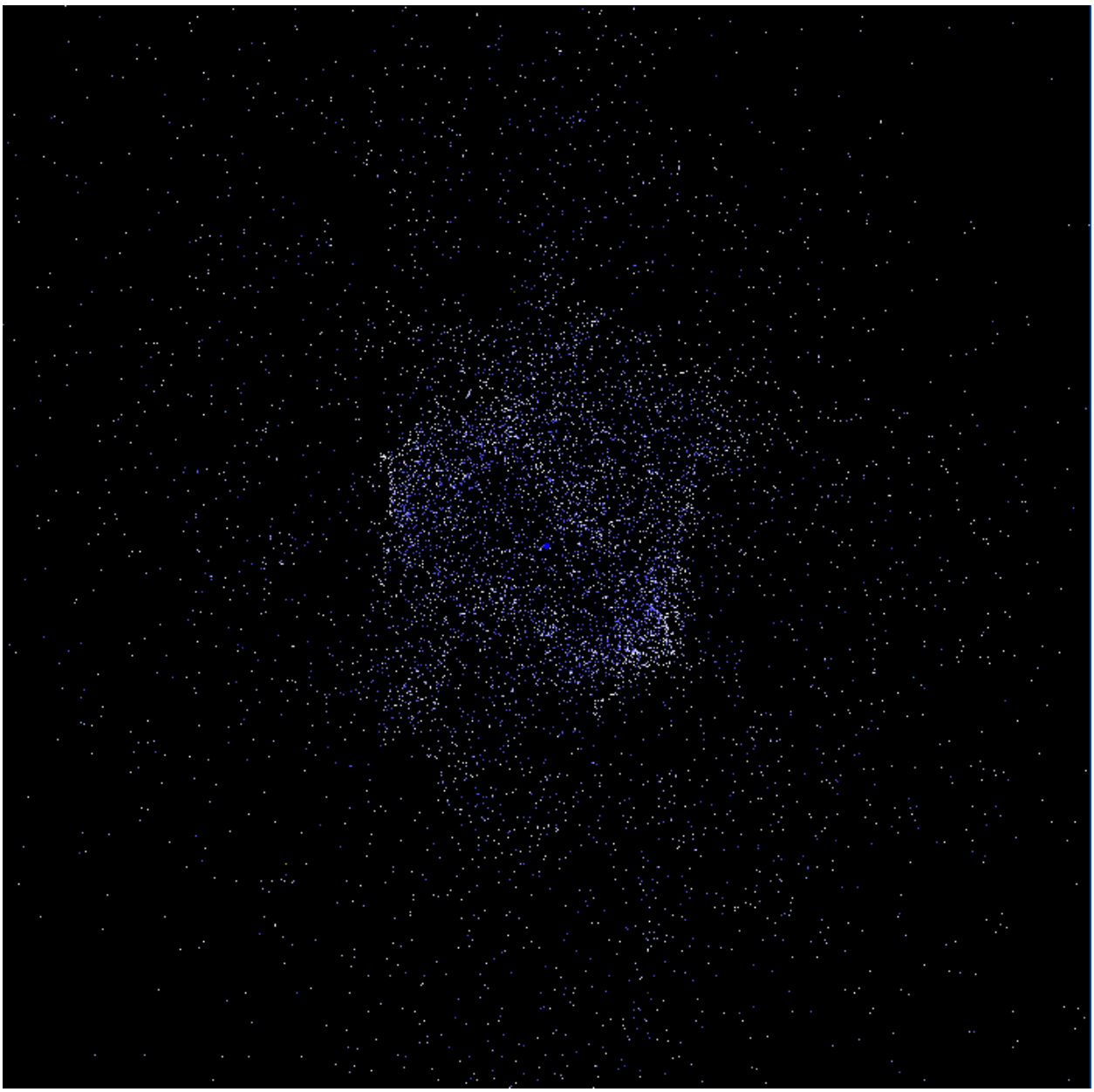












Appendix B: Project Code

```
#include <GL/glut.h>
#include <stdlib.h>
#include <ctime>

#include <string.h>
#include <iostream.h>
#include <math.h>

int mainScreenHeight = 700;
int mainScreenWidth = 700;



bool SimulationPaused = false;



int beginning_macro_number = 100000;


double time_step = 10000000; // in years


long beginning_cloud_radius = 1000;//4730.2642; // * 1000000000000; // meters = lightyear / 2    (cloud is lightyear
across)
```

```
double system_mass = 1.99 * 100000000000000 * 1000; //sun's mass * bok globule mass




int corona_radius = 9460.5284 * 1000000000000;




double camera_position[] = {0,0,30000000000000000};
double camera_rotation[] = {0,0,0};

int camera_rotating[] = {0,0,0,0,0,0};
int camera_moving[] = {0,0,0,0,0,0};

double camera_movement_speed = 50000000000000;
double camera_rotation_speed = .9;




int drawMode = 3;




double macro_xposition[100000];
double macro_yposition[100000];
double macro_zposition[100000];
double macro_xspeed[100000];
```

```
double macro_yspeed[100000];
double macro_zspeed[100000];
double macro_xaccel[100000];
double macro_yaccel[100000];
double macro_zaccel[100000];

double macro_mass[100000]; // in kilograms
double macro_radius[100000]; // in meters
double macro_wiggle_radius[100000];
//double macro_rcolor[10000];
double macro_gcolor[100000]; //temerature guage
double macro_bcolor[100000]; //extra energy guage
bool macro_fusing[100000]; //releasing fusion energy
int macro_nuclear_state[100000]; //Hydrogen, Helium, ...
double macro_extra_energy[100000];



double macro_centerofmass[3];
double macro_totalmass = 0;
double macro_massmultiplier = 0;









float macro_mat_specular[] = {.2,.2,0,1};
float macro_mat_shininess[] = {10};
float macro_mat_emission[] = {.2,.2,0,.05};
float macro_mat_Diffuse[] = {.2,.2,0,.1};



float light_position[] = {0,0,-50000000,0};
float white_light[] = {.5,.5,.5};
```

```
float ambient_light[] = {.2,.2,.2,.1};
```

```
#ifndef PI
#define PI 3.14159265358979323846264338327 95
#endif
#ifndef G
#define G 6.673 * (10 ^ (-11))
#endif
#ifndef E
#define E 9 * (10 ^ (9))
#endif
#ifndef A
#define A 6.022 * (10 ^ (23))
#endif
```

```
int randomInt(int range, bool neg)
{
        int theNumber = 0;
        float negNum = 0;
        theNumber = int(range * rand() / (RAND_MAX + 1.0));
        if (neg == true)
```

```
	{
		negNum = rand () % (2) + 1;
		if (negNum <= 1)
		{
			theNumber = -1 * theNumber;
		}
	}
	return theNumber;
}



void createNewMacro(int number)
{
	do
	{
		macro_xposition[number] = randomInt(beginning_cloud_radius, true);
		macro_yposition[number] = randomInt(beginning_cloud_radius, true);
		macro_zposition[number] = randomInt(beginning_cloud_radius, true);
	}
	while (double(sqrt((macro_xposition[number] * macro_xposition[number]) + (macro_yposition[number] *
macro_yposition[number]) + (macro_zposition[number] * macro_zposition[number]))) > beginning_cloud_radius);

	macro_xposition[number] = macro_xposition[number] * 1000000000000;
	macro_yposition[number] = macro_yposition[number] * 1000000000000;
	macro_zposition[number] = macro_zposition[number] * 1000000000000;



	// V = ( PI * (4 / 3) * (R * R * R))

	// density = 10000 particles per cubic centimeter
	// density = 10000000000000 particles per cubic meter (10000 * 100 * 100 * 100)

	// avogadro's number = A =
```

```
		// grams = ((particle #) * atomic mass units) / A = ((particle #) * amu) / A
		// kilograms = ((particle #) * amu * 1000) / A



		// v = m / d
		// 4PiRRR = 30000000000000m
		// RRR = 30000000000000m / 4Pi




		// R = (30000000000000m / 4Pi) ^ (1 / 3)
		// R = pow((30000000000000m / 4Pi), (1 / 3));




		macro_xspeed[number] = 0;
		macro_yspeed[number] = 0;
		macro_zspeed[number] = 0;
		macro_xaccel[number] = 0;
		macro_yaccel[number] = 0;
		macro_zaccel[number] = 0;
		macro_mass[number] = double(system_mass / beginning_macro_number);

		macro_radius[number] = double(pow(((30000000000000 * macro_mass[number]) / (4 * PI)), (1.0 / 3.0)) *
10000);

		//macro_rcolor[number] = 0;

		macro_gcolor[number] = 0;
		if (randomInt(2,false) == 1)
			macro_bcolor[number] = false;
		else
```

```
		macro_bcolor[number] =false; //false
	macro_fusing[number] = false;
	macro_nuclear_state[number] = 1;
	macro_extra_energy[number] = 0;
	//macro_bcolor[number] = randomInt(255,false);
}

void moveMacros(void)
{
	int e;
	for (e = 0; e <=  beginning_macro_number; e++)
	{
		macro_xspeed[e] = double(macro_xspeed[e] + macro_xaccel[e]);
		macro_yspeed[e] = double(macro_yspeed[e] + macro_yaccel[e]);
		macro_zspeed[e] = double(macro_zspeed[e] + macro_zaccel[e]);

		macro_xposition[e] = double(macro_xposition[e] + (macro_xspeed[e] * 31559328 * time_step)); //
number of seconds in a year
		macro_yposition[e] = double(macro_yposition[e] + (macro_yspeed[e] * 31559328 * time_step));
		macro_zposition[e] = double(macro_zposition[e] + (macro_zspeed[e] * 31559328 * time_step));

		macro_xaccel[e] = 0;
		macro_yaccel[e] = 0;
		macro_zaccel[e] = 0;


		macro_gcolor[e] = double(macro_gcolor[e] - (macro_gcolor[e] / 1000)); // 3x
		if (macro_gcolor[e] < 0)
			macro_gcolor[e] = 0;

	}
}
void moveSpecificMacros(int i, int e)
{
```

```
        double totalDistance, xDistance, yDistance, zDistance, rDistance, crossDistance;

        rDistance = double(macro_radius[i] + macro_radius[e]);

        xDistance = double(macro_xposition[i] - macro_xposition[e]);
        yDistance = double(macro_yposition[i] - macro_yposition[e]);
        zDistance = double(macro_zposition[i] - macro_zposition[e]);

        totalDistance = double(sqrt((xDistance * xDistance) + (yDistance * yDistance) + (zDistance * zDistance)));

        crossDistance = double(rDistance - totalDistance);

        macro_xposition[i] = double(macro_xposition[i] + (crossDistance * (xDistance / totalDistance)));
        macro_yposition[i] = double(macro_yposition[i] + (crossDistance * (yDistance / totalDistance)));
        macro_zposition[i] = double(macro_zposition[i] + (crossDistance * (zDistance / totalDistance)));


        xDistance = -xDistance;
        yDistance = -yDistance;
        zDistance = -zDistance;

        macro_xposition[e] = double(macro_xposition[e] + (crossDistance * (xDistance / totalDistance)));
        macro_yposition[e] = double(macro_yposition[e] + (crossDistance * (yDistance / totalDistance)));
        macro_zposition[e] = double(macro_zposition[e] + (crossDistance * (zDistance / totalDistance)));
}

void fuseTogether(int i, int e)
{
        double totalEnergy = double(macro_extra_energy[i] + macro_extra_energy[e]);
        double requiredEnergy = 0;


        switch (macro_nuclear_state[i])
          {
```

```
                        case 1:
                                requiredEnergy = 3;
                                break;
                        case 2:

                                break;
                        case 3:

                                break;
                        case 4:

                                break;
                        case 5:

                                break;
                        case 6:

                                break;
                        case 7:

                                break;
                        case 8:

                                break;
                        case 9:

                                break;
                }


        switch (macro_nuclear_state[e])
{
                case 1:
                        requiredEnergy = requiredEnergy + 3;
```

```
                    break;
                case 2:

                    break;
                case 3:

                    break;
                case 4:

                    break;
                case 5:

                    break;
                case 6:

                    break;
                case 7:

                    break;
                case 8:

                    break;
                case 9:

                    break;
        }

        if (totalEnergy >= requiredEnergy)
        {

        }
}

void transferHeat(int i, int e)
```

```
{
	double heatPerMass = double((macro_gcolor[i] + macro_gcolor[e]) / (macro_mass[i] + macro_mass[e]));
	double energyPerMass = double((macro_extra_energy[i] + macro_extra_energy[e]) / (macro_mass[i] +
macro_mass[e]));

	macro_gcolor[i] = macro_gcolor[i] + macro_extra_energy[i];
	if (macro_gcolor[i] > 1 )
	{
		macro_extra_energy[i] = macro_gcolor[i] - 1;
		macro_gcolor[i] = 1;
	}
	else
	{
		macro_extra_energy[i] = 0;
	}



	macro_gcolor[e] = macro_gcolor[e] + macro_extra_energy[e];
	if (macro_gcolor[e] > 1 )
	{
		macro_extra_energy[e] = macro_gcolor[e] - 1;
		macro_gcolor[e] = 1;
	}
	else
	{
		macro_extra_energy[e] = 0;
	}

	macro_gcolor[i] = double(macro_mass[i] * heatPerMass);
	macro_gcolor[e] = double(macro_mass[e] * heatPerMass);

	macro_extra_energy[i] = double(macro_mass[i] * energyPerMass);
	macro_extra_energy[e] = double(macro_mass[e] * energyPerMass);
```

```
}

void transferEnergy(int i, int e, double macro1LostEnergy, double macro2LostEnergy) //the crash between two macros generates heat/light and shares it between them.
{
	macro_gcolor[i] = double(macro_gcolor[i] + (macro1LostEnergy / 1000));
	if (macro_gcolor[i] > 1)
	{
		macro_extra_energy[i] = double(macro_extra_energy[i] + macro_gcolor[i] - 1);
		macro_gcolor[i] = 1;
	}
	else
	{
		if (macro_gcolor[i] < 1)
		{
			if (macro_extra_energy[i] >= (1 - macro_gcolor[i]))
			{
				macro_extra_energy[i] = double(macro_extra_energy[i] - (1 - macro_gcolor[i]));
				macro_gcolor[i] = 1;
			}
			else
			{
				macro_gcolor[i] = double(macro_gcolor[i] + macro_extra_energy[i]);
				macro_extra_energy[i] = 0;
			}
		}
	}



	macro_gcolor[e] = double(macro_gcolor[e] + (macro2LostEnergy / 1000));
	if (macro_gcolor[e] > 1)
	{
		macro_extra_energy[e] = double(macro_extra_energy[e] = macro_gcolor[e] - 1);
```

```
                macro_gcolor[e] = 1;
        }
        else
        {
                if (macro_gcolor[e] < 1)
                {
                        if (macro_extra_energy[e] >= (1 - macro_gcolor[e]))
                        {
                                macro_extra_energy[e] = double(macro_extra_energy[e] - (1 - macro_gcolor[e]));
                                macro_gcolor[e] = 1;
                        }
                        else
                        {
                                macro_gcolor[e] = double(macro_gcolor[e] + macro_extra_energy[e]);
                                macro_extra_energy[e] = 0;
                        }
                }
        }
}

void conserveMomentum(int i, int e, double mFactor)
{
        double xMomentum = 0, yMomentum = 0, zMomentum = 0;
        double macro1speed, macro2speed;
        double macro1EnergyI, macro1EnergyF;
        double macro2EnergyI, macro2EnergyF;
        double macro1LostEnergy, macro2LostEnergy;
        int f = 0;



        macro1speed = double(sqrt((macro_xspeed[i] * macro_xspeed[i]) + (macro_yspeed[i] * macro_yspeed[i]) +
(macro_zspeed[i] * macro_zspeed[i])));
```

```
macro2speed = double(sqrt((macro_xspeed[e] * macro_xspeed[e]) + (macro_yspeed[e] * macro_yspeed[e]) +
(macro_zspeed[e] * macro_zspeed[e])));

macro1EnergyI = double((macro_mass[i] * (macro1speed * macro1speed)) / 2);
macro2EnergyI = double((macro_mass[e] * (macro2speed * macro2speed)) / 2);



xMomentum = double(((macro_xspeed[i] * macro_mass[i]) + (macro_xspeed[e] * macro_mass[e])) / 2);
yMomentum = double(((macro_yspeed[i] * macro_mass[i]) + (macro_yspeed[e] * macro_mass[e])) / 2);
zMomentum = double(((macro_zspeed[i] * macro_mass[i]) + (macro_zspeed[e] * macro_mass[e])) / 2);



macro_xspeed[i] = double(xMomentum / (-macro_mass[i]));
macro_yspeed[i] = double(yMomentum / (-macro_mass[i]));
macro_zspeed[i] = double(zMomentum / (-macro_mass[i]));

macro_xspeed[e] = double(xMomentum / (-macro_mass[e]));
macro_yspeed[e] = double(yMomentum / (-macro_mass[e]));
macro_zspeed[e] = double(zMomentum / (-macro_mass[e]));



macro1speed = double(sqrt((macro_xspeed[i] * macro_xspeed[i]) + (macro_yspeed[i] * macro_yspeed[i]) +
(macro_zspeed[i] * macro_zspeed[i])));
macro2speed = double(sqrt((macro_xspeed[e] * macro_xspeed[e]) + (macro_yspeed[e] * macro_yspeed[e]) +
(macro_zspeed[e] * macro_zspeed[e])));

macro1EnergyF = double((macro_mass[i] * (macro1speed * macro1speed)) / 2);
macro2EnergyF = double((macro_mass[e] * (macro2speed * macro2speed)) / 2);
```

```
		macro1LostEnergy = double((macro1EnergyI - macro1EnergyF) );
		macro2LostEnergy = double((macro2EnergyI - macro2EnergyF) );

		if (macro1LostEnergy < 0)
			macro1LostEnergy = -macro1LostEnergy;

		if (macro2LostEnergy < 0)
			macro2LostEnergy = -macro2LostEnergy;



		if (macro_extra_energy[i] + macro_extra_energy[e] >= 6)//macros have enough energy to fuse
		{
			fuseTogether(i, e);
		}
		else
		{
			transferHeat(i, e);
			transferEnergy(i, e, macro1LostEnergy, macro2LostEnergy);
		}

		moveSpecificMacros(i, e);
		//moveMacros();
}

void repelEachOther(int i, int e, double wrDistance, double rDistance, double xDistance, double yDistance, double
zDistance, double totalDistance)
{
		double wRatio = macro_wiggle_radius[i] / macro_wiggle_radius[e];
		double macro1Force = 0, macro2Force = 0, macro1Accel = 0, macro2Accel = 0;

		macro1Force = double((E * (macro_mass[i] * macro_mass[e])) / double((macro_wiggle_radius[i] - rDistance +
wrDistance - totalDistance) * (macro_wiggle_radius[i] - rDistance + wrDistance - totalDistance)));
```

```
macro2Force = double((E * (macro_mass[i] * macro_mass[e])) / double((macro_wiggle_radius[e] - rDistance +
wrDistance - totalDistance) * (macro_wiggle_radius[i] - rDistance + wrDistance - totalDistance)));

macro1Accel = double((macro1Force / macro_mass[i]) + (macro2Force / macro_mass[i]));
macro2Accel = double((macro1Force / macro_mass[e]) + (macro2Force / macro_mass[e]));

if (macro_xspeed[i] >= 0)
        macro_xspeed[i] = double(macro_xspeed[i] - (macro1Accel * (xDistance / totalDistance)));
else
        macro_xspeed[i] = double(macro_xspeed[i] + (macro1Accel * (xDistance / totalDistance)));

if (macro_yspeed[i] >= 0)
        macro_yspeed[i] = double(macro_yspeed[i] - (macro1Accel * (yDistance / totalDistance)));
else
        macro_yspeed[i] = double(macro_yspeed[i] + (macro1Accel * (yDistance / totalDistance)));

if (macro_zspeed[i] >= 0)
        macro_zspeed[i] = double(macro_zspeed[i] - (macro1Accel * (zDistance / totalDistance)));
else
        macro_zspeed[i] = double(macro_zspeed[i] + (macro1Accel * (zDistance / totalDistance)));



if (macro_xspeed[e] >= 0)
        macro_xspeed[e] = double(macro_xspeed[e] - (macro2Accel * (xDistance / totalDistance)));
else
        macro_xspeed[e] = double(macro_xspeed[e] + (macro2Accel * (xDistance / totalDistance)));

if (macro_yspeed[e] >= 0)
        macro_yspeed[e] = double(macro_yspeed[e] - (macro2Accel * (yDistance / totalDistance)));
else
        macro_yspeed[e] = double(macro_yspeed[e] + (macro2Accel * (yDistance / totalDistance)));

if (macro_zspeed[e] >= 0)
```

```
                macro_zspeed[e] = double(macro_zspeed[e] - (macro2Accel * (zDistance / totalDistance)));
        else
                macro_zspeed[e] = double(macro_zspeed[e] + (macro2Accel * (zDistance / totalDistance)));









        //double((rDistance + wrDistance - totalDistance) / wrDistance);
}

void checkMacroPositions(void)
{
        int i, e;
        int COMDistance = 0;//center of mass, checks to see if macro is within corona radius
        double totalDistance, rDistance, wrDistance, xDistance, yDistance, zDistance;



        for (i = 0; i <= beginning_macro_number; i++)
        {
                e = i;
                for (e = 0; e <= beginning_macro_number; e++)
                {
                        if (e != i)
                        {
                                wrDistance = double(macro_wiggle_radius[i] + macro_wiggle_radius[e]);
                                rDistance = double(macro_radius[i] + macro_radius[e]);
                                xDistance = double(macro_xposition[i] - macro_xposition[e]);
                                yDistance = double(macro_yposition[i] - macro_yposition[e]);
                                zDistance = double(macro_zposition[i] - macro_zposition[e]);
                                totalDistance = double(sqrt((xDistance * xDistance) + (yDistance * yDistance) +
(zDistance * zDistance)));
```

```
                                        if (totalDistance <= wrDistance + rDistance)
                                        {
                                                if (totalDistance <= rDistance)
                                                        conserveMomentum(i, e, 1);
                                                else
                                                        repelEachOther(i, e, wrDistance, rDistance, xDistance, yDistance,
zDistance, totalDistance);
                                        }

                                }
                        }

                        COMDistance = double(sqrt(((macro_centerofmass[0] - macro_xposition[i]) * (macro_centerofmass[0]
- macro_xposition[i])) + ((macro_centerofmass[0] - macro_yposition[i]) * (macro_centerofmass[0] - macro_yposition[i]))
+ ((macro_centerofmass[0] - macro_zposition[i]) * (macro_centerofmass[0] - macro_zposition[i]))));
                        if (COMDistance >= corona_radius)
                        {
                                macro_xspeed[i] = -macro_xspeed[i];
                                macro_yspeed[i] = -macro_yspeed[i];
                                macro_zspeed[i] = -macro_zspeed[i];

                                macro_xposition[i] = macro_xposition[i] + macro_xspeed[i];
                                macro_yposition[i] = macro_yposition[i] + macro_yspeed[i];
                                macro_zposition[i] = macro_zposition[i] + macro_zspeed[i];

                                macro_xspeed[i] = double(macro_xspeed[i] / 5);
                                macro_yspeed[i] = double(macro_yspeed[i] / 5);
                                macro_zspeed[i] = double(macro_zspeed[i] / 5);
                        }

                        macro_bcolor[i] = double(double(macro_nuclear_state[i]) / 26); // iron
                }

}
```

```
void applyGravity(void)
{
	double totalDistance, xDistance, yDistance, zDistance;
	double totalAccel;
	int i, e;

	for (i = 0; i <= beginning_macro_number; i++)
	{
		e = i;
		for (e = 0; e <= beginning_macro_number; e++)
		{
			xDistance = double(macro_xposition[e] - macro_xposition[i]);
			yDistance = double(macro_yposition[e] - macro_xposition[i]);
			zDistance = double(macro_zposition[e] - macro_xposition[i]);

			totalDistance = double(sqrt((xDistance * xDistance) + (yDistance * yDistance) + (zDistance *
zDistance)));
			totalAccel = double(((G * (macro_mass[i] * macro_mass[e])) / (totalDistance * totalDistance))
/ (macro_mass[i] ));

			macro_xaccel[e] = double(macro_xaccel[e] + (totalAccel * (xDistance / totalDistance)));
			macro_yaccel[e] = double(macro_yaccel[e] + (totalAccel * (yDistance / totalDistance)));
			macro_zaccel[e] = double(macro_zaccel[e] + (totalAccel * (zDistance / totalDistance)));


			xDistance = -xDistance;
			yDistance = -yDistance;
			zDistance = -zDistance;

			macro_xaccel[i] = double(macro_xaccel[i] + (totalAccel * (xDistance / totalDistance)));
			macro_yaccel[i] = double(macro_yaccel[i] + (totalAccel * (yDistance / totalDistance)));
			macro_zaccel[i] = double(macro_zaccel[i] + (totalAccel * (zDistance / totalDistance)));
```

```
                }

//              macro_wiggle_radius[i] = double((macro_radius[i] * macro_extra_energy[i]) / (macro_mass[i]));
        }
}

void applyMacroForces(void)
{
        applyGravity();




}


void findCenterOfMass(void)
{
        int i = 0;

        macro_centerofmass[0] = 0;
        macro_centerofmass[1] = 0;
        macro_centerofmass[2] = 0;

        for (i = 0; i <=  beginning_macro_number; i++)
        {
                macro_centerofmass[0] = macro_centerofmass[0] + (macro_massmultiplier * macro_xposition[i] *
macro_mass[i]);
                macro_centerofmass[1] = macro_centerofmass[1] + (macro_massmultiplier * macro_yposition[i] *
macro_mass[i]);
                macro_centerofmass[2] = macro_centerofmass[2] + (macro_massmultiplier * macro_zposition[i] *
macro_mass[i]);
        }
}
```

```
void drawMacros(void)
{
	int i = 0;

	switch (drawMode)
	{
	case (1):
		glMaterialfv(GL_FRONT, GL_EMISSION, macro_mat_emission);
		glEnable(GL_LIGHTING);
		glEnable(GL_LIGHT0);
		glEnable(GL_BLEND);
		glDepthMask(GL_FALSE);
		for (i = 0; i <=  beginning_macro_number; i++)
		{
			glPushMatrix();
				float macro_mat_diffuse[] = {1,macro_gcolor[i],macro_extra_energy[i],.05};
				glTranslatef(macro_xposition[i], macro_yposition[i], macro_zposition[i]);
				glMaterialfv(GL_FRONT, GL_DIFFUSE, macro_mat_diffuse);
				glutSolidSphere(macro_radius[i] + macro_wiggle_radius[i],5,5);
			glPopMatrix();
		}
		glDisable(GL_BLEND);
		glDepthMask(GL_TRUE);
		glDisable(GL_LIGHTING);
		glDisable(GL_LIGHT0);
		break;
	case (2):
			for (i = 0; i <=  beginning_macro_number; i++)
			{
				glColor3f(1,macro_gcolor[i],macro_extra_energy[i]);
				glPushMatrix();
					glTranslatef(macro_xposition[i], macro_yposition[i], macro_zposition[i]);
					glutWireSphere(macro_radius[i] + macro_wiggle_radius[i],5,5);
				glPopMatrix();
```

```
                }
            break;
        case (3):
            glBegin(GL_POINTS);
                for (i = 0; i <=  beginning_macro_number; i++)
                {
                    glColor3f(1,macro_gcolor[i],macro_extra_energy[i]);
                    glVertex3f(macro_xposition[i],macro_yposition[i],macro_zposition[i]);
                }
            glEnd();
            break;
        }
}

void drawReferenceGalaxy(void)
{
        glColor3f(.7,0,.7);
        glutWireSphere(4730.2642 * 1000000000000000,25,25);
}

void idle(void)
{
        int i = 0;

        if (camera_moving[0] == 1)
        {
            camera_position[0] = camera_position[0] - camera_movement_speed;
        }
        if (camera_moving[1] == 1)
        {
            camera_position[0] = camera_position[0] + camera_movement_speed;
        }
        if (camera_moving[2] == 1)
        {
```

```
		camera_position[1] = camera_position[1] - camera_movement_speed;
}
if (camera_moving[3] == 1)
{
		camera_position[1] = camera_position[1] + camera_movement_speed;
}
if (camera_moving[4] == 1)
{
		camera_position[2] = camera_position[2] + camera_movement_speed;
}
if (camera_moving[5] == 1)
{
		camera_position[2] = camera_position[2] - camera_movement_speed;
}




if (camera_rotating[0] == 1)
{
		camera_rotation[0] = camera_rotation[0] + camera_rotation_speed;
}
if (camera_rotating[1] == 1)
{
		camera_rotation[0] = camera_rotation[0] - camera_rotation_speed;
}
if (camera_rotating[2] == 1)
{
		camera_rotation[1] = camera_rotation[1] - camera_rotation_speed;
}
if (camera_rotating[3] == 1)
{
		camera_rotation[1] = camera_rotation[1] + camera_rotation_speed;
}
if (camera_rotating[4] == 1)
```

```
	{
		camera_rotation[2] = camera_rotation[2] - camera_rotation_speed;
	}
	if (camera_rotating[5] == 1)
	{
		camera_rotation[2] = camera_rotation[2] + camera_rotation_speed;
	}




	if (SimulationPaused == false)
	{
		applyMacroForces();

		moveMacros();

		checkMacroPositions();

		findCenterOfMass();

	}

glutPostRedisplay();

}

void display(void)
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	glMatrixMode( GL_PROJECTION );
	glLoadIdentity();
	gluPerspective( 45, 1, 0, 30000 );
```

```
        // start fresh
        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity();



        //drawOrientationDisplay();


        // set up "camera"
        glRotated(-camera_rotation[0],0,1,0);
        glRotated(-camera_rotation[1],1,0,0);
        glRotated(-camera_rotation[2],0,0,-1);
        glTranslatef( -camera_position[0], -camera_position[1], -camera_position[2]);



        //drawReferenceGalaxy();

        drawMacros();

        //drawPhotons();


        //create smooth animation
        glutSwapBuffers();

        //draw scene
        glFlush();
}

void reshape(int w, int h)
```

```
{
        //Define the window and viewing rules
   glMatrixMode(GL_PROJECTION);
        gluPerspective(45,1,0,3000000);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glutPostRedisplay();
}


void keyboardDown(unsigned char key, int x, int y)
{
        switch (key)
  {
                case '-'://slow down movement speed
                        camera_movement_speed = camera_movement_speed / 1.5;
                        break;
                case '='://accelerate movement speed
                        camera_movement_speed = camera_movement_speed * 1.5;
                        break;
                case '['://slow down movement speed
                        camera_rotation_speed = camera_rotation_speed / 1.25;
                        break;
                case ']'://accelerate movement speed
                        camera_rotation_speed = camera_rotation_speed * 1.25;
                        break;
                case 'j'://turn heading left
                        camera_rotating[0] = 1;
                        break;
                case ';'://turn heading right
                        camera_rotating[1] = 1;
                        break;
                case 'k'://pitch head down
                        camera_rotating[2] = 1;
```

```
            break;
case 'l'://pitch head up
            camera_rotating[3] = 1;
            break;
case 'i'://roll left
            camera_rotating[4] = 1;
            break;
case 'o'://roll right
            camera_rotating[5] = 1;
            break;
case 'a'://move down neg. x axis
            camera_moving[0] = 1;
            break;
case 'f'://move up pos. x axis
            camera_moving[1] = 1;
            break;
case 's'://move down neg. y axis
            camera_moving[2] = 1;
            break;
case 'd'://move up pos. y axis
            camera_moving[3] = 1;
            break;
case 'w'://move down neg. z axis
            camera_moving[4] = 1;
            break;
case 'e'://move up pos. z axis
            camera_moving[5] = 1;
            break;
case 'm'://move up pos. z axis
            switch (drawMode)
            {
            case 1:
                        drawMode = 2;
                        break;
```

```
                        case 2:
                                drawMode = 3;
                                break;
                        case 3:
                                drawMode = 1;
                                break;
                        }
                        break;
                case ' '://pause/unpause simulation
                        if (SimulationPaused == false)
                                SimulationPaused = true;
                        else
                                SimulationPaused = false;
                        break;
                case 27:
                        exit(0);
                        break;
  }
        glutPostRedisplay();
}


void keyboardUp(unsigned char key, int x, int y)
{
        switch (key)
  {
                case 'j'://turn heading left
                        camera_rotating[0] = 0;
                        break;
                case ';'://turn heading right
                        camera_rotating[1] = 0;
                        break;
                case 'k'://pitch head down
                        camera_rotating[2] = 0;
```

```
                        break;
                case 'l'://pitch head up
                        camera_rotating[3] = 0;
                        break;
                case 'i'://roll left
                        camera_rotating[4] = 0;
                        break;
                case 'o'://roll right
                        camera_rotating[5] = 0;
                        break;
                case 'a'://move down neg. x axis
                        camera_moving[0] = 0;
                        break;
                case 'f'://move up pos. x axis
                        camera_moving[1] = 0;
                        break;
                case 's'://move down neg. y axis
                        camera_moving[2] = 0;
                        break;
                case 'd'://move up pos. y axis
                        camera_moving[3] = 0;
                        break;
                case 'w'://move down neg. z axis
                        camera_moving[4] = 0;
                        break;
                case 'e'://move up pos. z axis
                        camera_moving[5] = 0;
                        break;
  }
        glutPostRedisplay();
}

void mouse(int button, int state, int x, int y)
{
```

```
	switch (button)
	{
	case GLUT_LEFT_BUTTON:
				if (state == 0)
				{

				}
				else
				{

				}
				break;
	case GLUT_RIGHT_BUTTON:
				if (state == 0)
				{

				}
				else
				{

				}
				break;
	default:
		break;
	}
}

void init(void)
{
	//OpenGL initializing calls
	glEnable (GL_DEPTH_TEST);
	glShadeModel (GL_SMOOTH);
	glClearColor (0.0, 0.0, 0.0, 0.0);
```

```
	//Fog stuff
	//glEnable(GL_FOG);

	//blending stuff
	glBlendFunc(GL_SRC_COLOR, GL_DST_COLOR);

	//glMaterialfv(GL_FRONT, GL_SPECULAR, macro_mat_specular);
	//glMaterialfv(GL_FRONT, GL_SHININESS, macro_mat_shininess);



	//lighting stuff
	glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
	glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);
	glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_light);
	glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

	//Standard stuff
	glShadeModel (GL_SMOOTH);

	//create macros
	srand((unsigned)time(0));
	for (int number = 0; number <=  beginning_macro_number; number++)
	{
		createNewMacro(number);
		macro_totalmass = macro_totalmass + macro_mass[number];
	}
	macro_massmultiplier = 1 / macro_totalmass;
	macro_totalmass = macro_totalmass * 10000;
}

int main(int argc, char** argv)
{
  //Initialization calls
```

```
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize (mainScreenWidth, mainScreenHeight);
    glutInitWindowPosition (200, 50);
    glutCreateWindow (argv[0]);
    init();
    glutIdleFunc(idle);
    glutReshapeFunc (reshape);
    glutKeyboardFunc (keyboardDown);
    glutKeyboardUpFunc (keyboardUp);
    glutMouseFunc (mouse);
    glutDisplayFunc (display);
    glutMainLoop();
    return 0;
}
```