Shattered Glass:

An analysis of a brittle collision using Smooth Particle Hydrodynamics

AiS Challenge

Final Report

April 3, 2002

006

Albuquerque Academy

Team Members

Zach LaBry

Preston Dell

Eric Searle

Charles Whitehead

Hugh Wimberly

Teacher

Mr. Jim Mims

Project Mentor

Mr. Jim Mims

Table of Contents:

Executive summary:

For our AiS challenge project, we decided to use Smooth Particle Hydrodynamics (SPH) to calculate the state of a system during and after the collision between a brittle object and a resilient object. We took the complexity of glass and attempted to model it using a gridless system of many particles. To this end, we heavily researched SPH and the intrinsic physical properties of our materials. Our difficulties lay in many areas, primarily due to computing restraints of even the supercomputers Mode, Pi, and Theta. Our project is both impossible to test or even run, and so impossible to benchmark, considering its massive time requirements and file saving and reading difficulties. Our results are thus limited; our main conclusions merely a reflection of the difficulty of the problem. In fact, through this process, although we gained very helpful experience, it became clear that our results were limited to small statements about the manner in which small particles of glass act upon the entire system during the course of the collision. Regardless, we are relatively satisfied that we were able to progress so far into such a difficult project, even if we were not able to complete the project to our full desires.

Problem:

One of the most important fields in modern science is the study of material properties: by understanding how various materials react under different circumstances, chemical engineers can create materials that have the specific properties that the engineers need.

One primary objective in this field in creating something that can withstand high-force impacts and collisions: for instance, in space, a satellite may encounter objects moving in excess of 10^5 m/s. At these velocities, even an object of only a few grams can severely damage most terrestrial materials.

We decided at the outset of the project that we would analyze a simple collision system. We wanted to use a complex material as our initial object subject to the collision to simulate the difficulty of an actual system. Our intention was to create a general method of analyzing any collision system, by using a material that requires methods of many other substances. Our colliding object was intended to be able to simulate many different types of objects.

We briefly considered using a more stable and predictable solid, such as a polymer: at low temperatures, they shatter in an almost completely predictable manner, and even in normal kinetic conditions, polymers tend to stretch, bend and break in a nearly uniform manner. Very common and easily manufactured, glass is much more complex than might be assumed. First of all, glass is actually a class of materials, like polymers: depending on the chemical makeup, glass can have very different properties and internal bonding-structure. More importantly, glass is actually an amorphous solid:

having properties of a liquid in most kinetic conditions, glass actually flows slowly over time, because instead of a crystalline structure that would be normally indicated by its uniform fracture lines, glass particles are actually held together by very complex frictional bonds. Within glass particles, a more regular chemical bond exists, but even then, the bonds follow no particular trends; the structure of a glass “crystal” is essentially random, rather than conforming to specific cubic, hexagonal, or other crystalline structure that make common homogenous chemical solids like carbon so predictable. Due to the multiple complexities of glass, we then decided to use it as our base inertial object, considering both the fact that to our knowledge no one has ever written a SPH program using glass because of the inherent randomness of glass and its unpredictability, and if we are able to accurately predict the shatter pattern of glass, we can later eliminate several variables that make glass difficult to work with to approximate most other common materials.

We decided after choosing glass for our inertial object that we needed to merely choose a small, massive solid as our colliding object. We consulted physicists and found that in most cases the internal particle redistribution in a colliding object of relatively small volume and large mass was negligible and that we could ignore it for our purposes. Our final project was decided: to accurately and precisely predict the position of all the particles in a system consisting of a brittle object, glass, and a small massive object, after a collision between the two.

Method:

In conjunction with our decision to simulate a collision and accurately predict the final outcome, we also decide to implement Smooth Particle Hydrodynamics (SPH) so that we would be able to complete the project. SPH involves taking a grid of a system, calculating the various states of positions on the intersections of the grid, and then creating a grid inside each of the intersections, to “zoom in” and more accurately understand the state of the system. This process is done recursively until the grid ceases to exist as a grid and actually represents the states of the individual particles of the system. The primary ability of the theory of SPH is to allow a property of an object to be more accurately described. SPH is most often used in very complex problems involving kinetic (temperature) objects that don’t obey simple laws of kinetic theory at the macroscopic level or in situations involving sections of fluid that due to the specific substances and conditions are compressible or otherwise don’t follow the basic laws of fluid dynamics except at the microscopic level. SPH is also used in situations where extreme accuracy is required, since real-world objects essentially never follow the primary laws of physics perfectly: for instance, except in tightly controlled laboratory conditions, all known liquids are compressible to some degree, creating small margin of error, often less than .05%, when the primary laws of fluid mechanics are applied to the system as a whole. We decided to use SPH, since the kinematic laws do not allow for fractures or breaks; we are forced to look at the system as a set of particles rather than as a whole.

Using SPH as applied to a motion system allowing for chemical and frictional bonding allowed us to research and design our own system. SPH is normally applied to other types of systems, and we were unable to find an example of a true kinematic system that utilized SPH to account for material imperfection, requiring us to create the theory ourselves.

As soon as we decided to utilize SPH theory, we realized that our project would require an advanced programming language due to the number of particles that SPH would require to work. Bearing in mind the fact that we are all fairly fluent in Java and C++, we decided to construct our base code and the structure for our heavy computational work in C++, based on the fact that C++ is more efficient in processing raw data and heavy computational work. To accurately measure the condition of the system, our code would require several thousand particles regulated and reconstructed at a very small time interval, approximately 10^{-5} seconds. This is far too much data to interpret without a computer, so we decided to implement Java's graphical capabilities to graph the condition of the system.

Our first slightly successful version of the code ran a system of 8 x 10^4 particles, a 20 particle-per-side cubic structure. It utilized an extremely computationally strenuous recursive sequence, that in the end, combined with our original use of arrays instead of a more complex and less memory requiring set of pointer fields, created a program which could not possibly complete running in a reasonable amount of time on a normal computer. Even with our recent utilization of an original pointer field, we needed to use the supercomputers at the LANL facility in order to solve the problem, thus justifying our project as appropriate for an AiS Challenge.

Results:

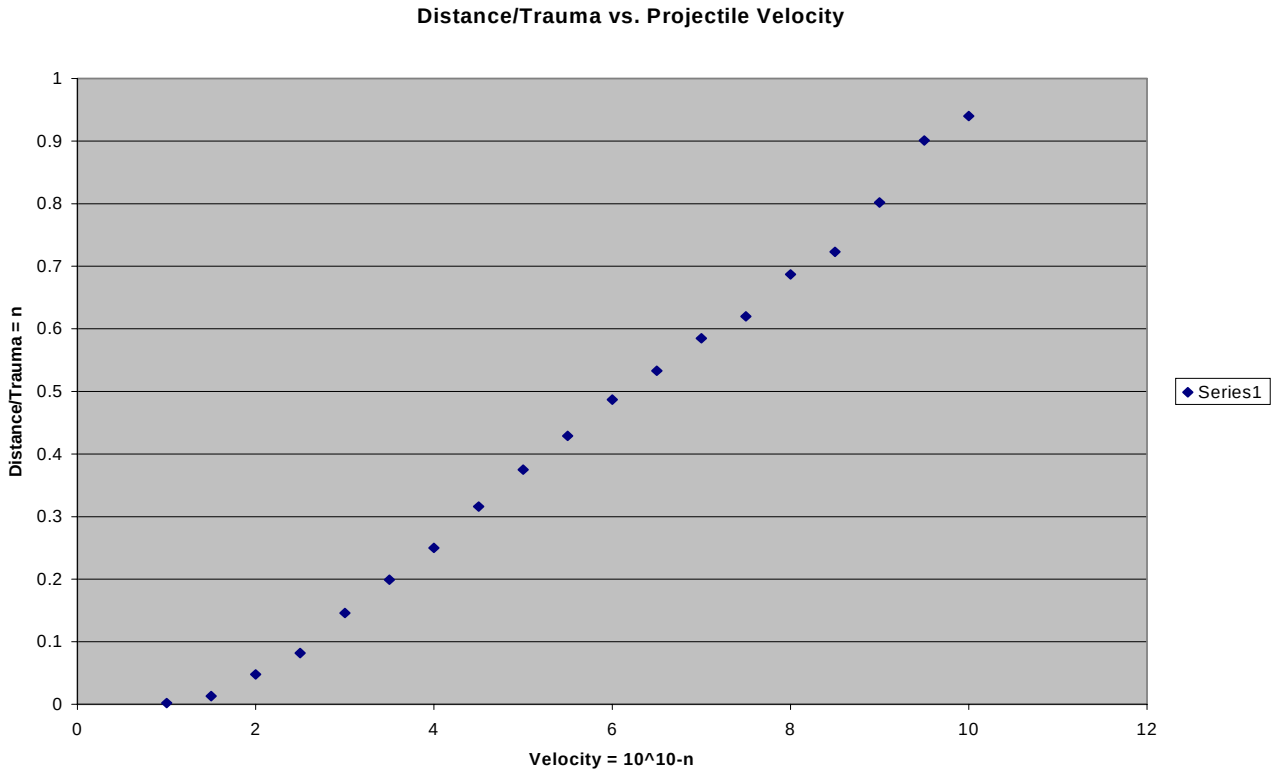
We found that in our simulation of a particle collision, that due to the constantly outward projected lines of force, the particle field separated in radial symmetry. Although this was unsurprising, it was interesting how the irregular particle distribution combined with the randomized bonding caused many particles to gain a velocity with a component opposite to the velocity of the initial colliding object. The system still obeyed the laws of conservation, demonstrated by our finding that after the collision, the system still had a net energy and momentum equal to the initial sate of the system. However, we found that this was the reason, rather than leverage, that glass shards spin upon separation from a pane of glass while it shatters.

We feel that it is important to note the conditions presumed by our program, and how those conditions differ from any real world situation would affect the result. If one were to shoot a bullet through a pane of glass, several factors such as gravity, and the force exerted upon the edges of the glass by the frame would have a significant impact on the result, and whether or not the pane shatters. We were unsure as to whether we could accurately account for such factors, as they would pertain to the collision, and ultimately decided to leave them out of the final code.

We found that the higher the velocity/mass ratio, the less trauma the particle field sustained. (See graph, Page 9) For instance, with a mass and velocity actually approximating a bullet, there were very few external fractures distant from the point of collision, but rather the bonds close to the point of impact broke readily and caused a

nearly uniform “hole” in the particle field, marking the trajectory of the projectile. For a lower velocity/mass ratio, the glass tends to fracture in long lines extending from the point of impact, as is most commonly associated with broken glass.

One of our most important findings was that the fracture lines of the glass were nearly independent of the randomized structure of the bonds because many bonds tended to group together to create larger groups similar to each other. In effect, randomizing bonds generally caused them to neutralize each other on a larger scale. Unless we guided the development of the bond structure and the initial kinetic state to formulate either a crystalline or heavily imbalanced molecular configuration, the result was much the same.



Graph Above: The Distance/Trauma is the average distance of the trauma (fractures in the frictional bonding) from the point of impact: note that the higher the velocity, the lower the Distance/Trauma- a faster projectile tends to make a more uniform hole, that tends to be localized around the point of impact.

Conclusion:

Our time requirement significantly impeded us, due to a massive recursion sequence that increased the length of the program's run time with a polynomial growth based on the number of particles in the system. A 10 by 10 by 10 grid of particles took about 25 minutes to run on the supercomputer Mode, and we estimate based on a 12 hour test run time of the 20 by 20 by 20 grid that it would take about 10 days to complete. Even for the 1000 particle system, the log file that was written was nearly 8,000 pages long, using only one line per log entry, an approximately 44.9mb .txt file. Given that the program follows a polynomial growth in time based on the number of particles, that means that a 2.7*10^5 particle system, to simulate individual molecules, would require about 5.8 years of run-time to complete calculations on only a single second of system time. Even realizing that our system rarely requires more than a second in system time to complete all vital collisions, the time requirement is far too great: to make our program useful, we should be able to complete the entire process in under a week.

Glass also turned out to be extremely difficult to work with. Although we intended to use glass from the start precisely because of its inherent complexities, only through significant research did we find out exactly how complex glass' internal structure is. Normal equations both for fluid dynamics and kinetic theory break down in their ability to even predict the properties of very small pieces of glass, and the irregularities that cause fracture lines in broken glass to appear random are essentially impossible to predict in any type of glass except that produced in laboratory conditions, under very high pressures and in controlled heat settings. We were forced to apply a random chance for fault in every section of the glass, resulting in our inability to measure any given sheet

of glass. Even under ideal conditions, we can model only a generalization of a breaking sheet of glass, and to debug our final code to make sure that it was not only running but that it was correctly predicting the outcome of the collision we will had to introduce human error and look at several different final results and compare them to similar sheets of glass in the real world. This created problems in identifying where a problem is, since the problem would probably not be a machine deficiency but an error in our assumptions of initial conditions or incorrectly derived equations.

Our actual results were limited by the numerical intensity of our program. Because of the size of our file that was generated during each run of the program, we were unable to efficiently use the data that it contained. The program could not be run off the supercomputers through telnet, because our telnet accounts were not even close to the required size to store even one version of a final run. The program ran far too slowly on our home computers, and we were unable in the end to bypass this blockade: our final program is merely a set of code that has theoretical functionality but cannot be run within time constraints. Even our run-outputs for the program using only a small number of particles are nearly useless to us, since we are unable to open files of that size with our Java reader to create our original concept of bitmaps. The most disappointing conclusion that we drew from this experience was that we simply didn't have the computing resources to complete our project as intended. However, we were able to use small portions of our outputs to generate interesting results, that led us to conclusions about the properties of shattering glass, such as our discovery of bond strength as an independent function of glass density.

Most significant achievement:

We had two important achievements through this project, both relating to how we bypassed difficulties. First of all, the most resource intensive part of our original program was calculating the states of the collision using arrays. Our original development of a pointer field, where each particle in the field was a pointer to every surrounding particle, as well as in indicator of the presence of a bond, allowed us to completely replace our arrays with much less memory intensive pointers.

Mathematically, deriving equations of state was a very difficult process, and we were forced to learn a method of taking partial derivatives in three space to allow our program to utilize more realistic angular collisions.

In both cases, our own initiative allowed us to overcome our initial difficulties, and in fact, without outside help of any kind in either case.

Acknowledgements:

First, our thanks to Jim Mims, our computer science instructor and sponsor, for suggesting the project originally.

Further thanks to Dr. Authur Payne and Dr. Agustin Kintanar, for directing us towards useful sources and helping us with necessary derivations of equations we used.

Final Code:

This the only product of our project that will wither fit in less than several hundred pages or be legible in any way, since our run-time outputs are both nearly gibberish too the human reader and sever thousand pages of text, and we were unable do to computing resources to create a bitmap capable of interpreting these outputs.

```
/**********************************
*       Created by: Charles Whitehead,  *
*       Preston Dell, and Hugh Wimberly*
*       Date created: 03-14-02          *
*       Last modified 04-02-02          *
*       SPH with pointers               *
*       Models particle collisions      *
**********************************/
////////////////////////////////////////////////////
//Cpp files/////////////////////////////////////////
////////////////////////////////////////////////////

//Sphp.cpp//
#include "Includes.h"

//Main driver//
int main()
{
```

```
	int Velo, massC,  massF;


	cout<<"Welcome to particle collider.\nPlease enter what mass you would like the
colliding object to have: ";
	cin>>massC;
	cout<<"Please enter the mass of the particles in the field: ";
	cin>>massF;
	cout<<"Please enter the velocity at which the particle is traveling: ";
	cin>>Velo;
	Start(10,100,1);		//Begins main part of program//
	return 0;
}

/*Main part of program*/
//Functions.cpp//

//Includes//
#include "Includes.h"
#include "Globals.h"



void Start(int Velo,int massC, int massF)
{
```

```
//Opens files for writing//
data.open("data.txt");
data1.open("AllMoving.txt");
//int Where = 0;
//int N = 0;
//Starts the colliding particle 10 units away from middle of field//
//Initializes the particle//
Colliding = new Particle;
Colliding->Label = 1000;
Colliding->Velox = -Velo;
Colliding->Veloy = 0;
Colliding->Veloz = 0;
Colliding->x = 10;
Colliding->y = 0;
Colliding->z = 0;

//Initializes the first particle in the center of the field//
Current = new Particle;
Current->Label = 0;
Current->x=0;
Current->y=0;
Current->z=0;
```

```
        Current->Velox=0;
        Current->Veloy=0;
        Current->Veloz=0;
        Current->left = NULL;
        Current->right =NULL;
        Current->top = NULL;
        Current->bottom = NULL;
        Current->back = NULL;
        Current->forward = NULL;
        //Temp = Current;

        //Initializes the field of particles//
        Initialize(Current,as);

        //Keeps old velocities for use in benchmarking//
        double velo = Colliding->Velox;

        while(Colliding->x >-3 /*|| (Colliding->Velox >0 && Colliding->Veloy >0 &&
Colliding->Veloz >0)*/)
        {
                //if(int(times)%100 == 0)
                //        data<<Colliding->x<<"\n" ;
                if(Colliding->Velox != velo)
```

```
		{
			velo = Colliding->Velox; //Benchmark
		}

		//Increments the Colliding particles position in space//
		Colliding->x += Colliding->Velox * .0001;
		Colliding->y += Colliding->Veloy * .0001;
		Colliding->z += Colliding->Veloz * .0001;

		//Increments the field's positions//
		Increment(Current,0);

		//Compares positions//
		Compare(Current, Colliding,  b, massC, massF, s, a,times,0);

		//Increments the time//
		times+=.0001;

	}

	//Benchmark//
	cout<<Colliding->collisions ;
}
```

```
//Not used//
void MakeFile(Particle Current)
{
	Current.Save<<Current.Label<<"\t";
	Current.Save<<Current.x<<"\t";
	Current.Save<<Current.y<<"\t";
	Current.Save<<Current.z<<"\n";

}


//Creates the particle field//
//Initialize.cpp//

#include "Includes.h"

//Storage//
ofstream Starting;
```

```
void Initialize(Particle *Current,int as)
{

	//In case the first particle is not already initialized//
	if(as == 0)
	{
		Starting.open("Start.txt");
		//Current->collisions = 0;
		Current->x=0;
		Current->y=0;
		Current->z=0;
		Current->Velox=0;
		Current->Veloy=0;
		Current->Veloz=0;
	}



	//Creates the neighboring particles//
	if(Current->left == NULL)
	{
		Particle *left = new Particle;
		left->right = Current;
```

```
            left->left =NULL;
            left->top = NULL;
            left->bottom = NULL;
            left->forward = NULL;
            left->back = NULL;
            Current->left = left;
            left->Label =Current->Label +1;
            left->x = Current->x;
            left->y = Current->y -1;
            left->z = Current->z;
            left->Velox = 0;
            left->Veloy = 0;
            left->Veloz = 0;
            //left->collisions = 0;
    }
    if(Current->right == NULL)
    {
            Particle *right = new Particle;
            right->left = Current;
            right->right = NULL;
            right->top = NULL;
            right->bottom = NULL;
            right->forward = NULL;
```

```
        right->back = NULL;
        Current->right = right;
        right->Label =Current->Label -1;
        right->x = Current->x;
        right->y = Current->y+1;
        right->z = Current->z;
        right->Velox = 0;
        right->Veloy = 0;
        right->Veloz = 0;
        //right->collisions = 0;
}
if(Current->top == NULL)
{
        Particle *top = new Particle;
        top->bottom = Current;
        top->left = NULL;
        top->right = NULL;
        top->top = NULL;
        top->forward = NULL;
        top->back = NULL;
        Current->top = top;
        top->Label =Current->Label +10;
        top->x = Current->x;
```

```
        top->y = Current->y;
        top->z = Current->z+1;
        top->Velox = 0;
        top->Veloy = 0;
        top->Veloz = 0;
        //top->collisions = 0;
}
if(Current->bottom == NULL)
{
        Particle *bottom = new Particle;
        bottom->top = Current;
        bottom->bottom = NULL;
        bottom->left = NULL;
        bottom->right = NULL;
        bottom->forward = NULL;
        bottom->back = NULL;
        Current->bottom = bottom;
        bottom->Label =Current->Label -10;
        bottom->x = Current->x;
        bottom->y = Current->y;
        bottom->z = Current->z-1;
        bottom->Velox = 0;
        bottom->Veloy = 0;
```

```
        bottom->Veloz = 0;
        //bottom->collisions = 0;
}
if(Current->forward == NULL)
{
        Particle *forward = new Particle;
        forward->back = Current;
        forward->top = NULL;
        forward->bottom = NULL;
        forward->left = NULL;
        forward->right = NULL;
        forward->forward = NULL;
        Current->forward = forward;
        forward->Label =Current->Label +100;
        forward->x = Current->x+1;
        forward->y = Current->y;
        forward->z = Current->z;
        forward->Velox = 0;
        forward->Veloy = 0;
        forward->Veloz = 0;
        //forward->collisions = 0;
}
if(Current->back == NULL)
```

```
	{
		Particle *back = new Particle;
		back->forward = Current;
		back->back = NULL;
		back->top = NULL;
		back->bottom = NULL;
		back->left = NULL;
		back->right = NULL;
		Current->back = back;
		back->Label =Current->Label -100;
		back->x = Current->x-1;
		back->y = Current->y;
		back->z = Current->z;
		back->Velox = 0;
		back->Veloy = 0;
		back->Veloz = 0;
		//back->collisions = 0;
	}

	//Stores the initial field so it can be graphed//
	Starting<<Current->Label<<"\t"<<Current->x<<"\t"<<Current-
>y<<"\t"<<Current->z<<"\n";
```

```
//Ends the recursion//
if(as >= 5)
{
	return;
}

//Increments to the next particle in the field//
if(Current->left != NULL)
{
	Initialize(Current->left,as+1);
}
if(Current->right != NULL)
{
	Initialize(Current->right,as+1);
}
if(Current->top != NULL)
{
	Initialize(Current->top,as+1);
}
if(Current->bottom != NULL)
{
	Initialize(Current->bottom,as+1);
}
```

```
	if(Current->forward != NULL)
	{
		Initialize(Current->forward,as+1);
	}
	if(Current->back != NULL)
	{
		Initialize(Current->back,as+1);
	}

}



//Increments the particles in the field//
//Increment.cpp//

#include "Includes.h"



void Increment(Particle *Current, int run)
{
	//Ending condition for the recursion//
	if(run >=4)
	{
```

```
        return;
}

//Increments the positions//
Current->x += Current->Velox * .0001;
Current->y += Current->Veloy * .0001;
Current->z += Current->Veloz * .0001;

//Shifts through the intire field//
if(Current->left != NULL)
{
        Increment(Current->left,run+1);
}
if(Current->right != NULL)
{
        Increment(Current->right,run+1);
}
if(Current->top != NULL)
{
        Increment(Current->top,run+1);
}
if(Current->bottom != NULL)
{
```

```
            Increment(Current->bottom,run+1);
        }
        if(Current->forward != NULL)
        {
            Increment(Current->forward,run+1);
        }
        if(Current->back != NULL)
        {
            Increment(Current->back,run+1);
        }

}


//Compares whether or not the particles are colliding//
//Compare.cpp//

#include "Includes.h"


void Compare(Particle *Current, Particle *Colliding, double &b, double massC, double
massF,double s,double a,double times,int run)
{
```

```
//Ends recursion//
if(run >=10)
{
	return;
}

//Tests to see if the outside particle and a field particle are colliding//
if(Current->Label != b)
{
	Test(Current,Colliding,massC, massF,.1,1000, times,a,b);
}

//Compares to see if particles in the field are colliding with eachother//
CompareOthers(Current,Current, massF,s,a,b,times);

//Increments number of runs//
run++;

//Increments to the next particle//
if(Current->left != NULL)
{
	Compare(Current->left,Colliding, b, massC, massF, s,a,times,run);
}
```

```
if(Current->right != NULL)
{
	Compare(Current->right,Colliding, b, massC, massF, s,a,times,run);
}
if(Current->top != NULL)
{
	Compare(Current->top,Colliding, b, massC, massF, s,a,times,run);
}
if(Current->bottom != NULL)
{
	Compare(Current->bottom,Colliding, b, massC, massF, s,a,times,run);
}
if(Current->forward != NULL)
{
	Compare(Current->forward,Colliding, b, massC, massF, s,a,times,run);
}
if(Current->back != NULL)
{
	Compare(Current->back,Colliding, b, massC, massF, s,a,times,run);
}
```

```
}

void CompareOthers(Particle *Current, Particle *Current2,double massF,double s,double
a,double &b,double times)
{
	//Ends recursion//
	if(s >= 10)
	{
		return;
	}

	//Tests to see if particles are colliding//
	//So long as these following conditions are true//
	if((Current != NULL || Current != 0)&& a!=Current2->Label && Current2-
>Label !=Current->Label &&(Current2 != NULL || Current2 != 0) && Current !=
Current2 && Current->Label != a && (Current2->Velox < 0 || Current2->Veloy <0 ||
Current2->Veloz <0) )
	{
		Test(Current, Current2,massF, massF,.1,1000, times,a,b);
	}

	//Increments the number of runs//
	s++;
```

```
//Tests the next particle//
if(Current->left != NULL)
{
	CompareOthers(Current->left,Current2,massF,s,a,b,times);
}
if(Current->right != NULL)
{
	CompareOthers(Current->right,Current2,massF,s,a,b,times);
}
if(Current->top != NULL)
{
	CompareOthers(Current->top,Current2,massF,s,a,b,times);
}
if(Current->bottom != NULL)
{
	CompareOthers(Current->bottom,Current2,massF,s,a,b,times);
}
if(Current->forward != NULL)
{
	CompareOthers(Current->forward,Current2,massF,s,a,b,times);
}
if(Current->back != NULL)
```

```
	{
		CompareOthers(Current->back,Current2,massF,s,a,b,times);
	}
}


//Test for collision and updates new speeds//
//Test.cpp//

#include "Includes.h"

//Storage//
ofstream CData;

void Test(Particle *Current, Particle *Colliding,double massC, double massF,double
radius,double strength, double times,double &a,double &b)
{
	//Opens file if not already open//
	if(!CData.is_open())
	{
		CData.open("Collisions.txt");
	}
```

```
//Makes sure the particle does not hit the same particle due to time
increments//
if((Colliding->time != times && Current->time != times&& Colliding-
>collisions != int(Current->x) && Colliding->collisions != Current->x)&& Current-
>Label != a && (Current->Label != Colliding->Label) && (Colliding->Label !=
Current->collisions ||Current->Label !=Colliding->collisions)  && (Current->x-.01 <
Colliding->x && Current->x+.01 > Colliding->x && Current->y-radius < Colliding-
>y&& Current->y+radius > Colliding->y && Current->z-.01 < Colliding->z&& Current-
>z+.01> Colliding->z))
{

//Tests to see if bonds are broken//
double needed = 0;
needed = massC * sqrt((Colliding->Velox*Colliding->Velox +
Colliding->Veloy*Colliding->Veloy+Colliding->Veloz*Colliding->Veloz));
//Randomizes bond strenght to simulate glass//
strength = (rand()%500)+700;

//If they are then break the bonds//
if(needed >=  (strength* 6))
{
Current->left = NULL;
Current->right = NULL;
```

```
                Current->top = NULL;
                Current->bottom = NULL;
                Current->forward = NULL;
                Current->back = NULL;

        }
        else    //The mass is equal to the bonded particles masses//
        {
                massF = massF*6;
        }


        //Prints out the time of collision what particles are colliding and
where//
        cout/*<<"Particle "<<*/<<times<<"\t"<<Colliding-
>Label<<"\t"/*"hit "<<*/<<Current->Label<<"\t"<</*" at: "<<*/int(Current-
>x)<<"\t"<<int(Current->y)<<"\t"<<int(Current->z)<<"\n";

        //Changes the velocity of both particles//
        Current->Velox = (2 *massC/(massC+massF))* Colliding->Velox;
        Colliding->Velox = ((massC-massF)/(massC+massF)) * Colliding-
>Velox;
```

```
                    Current->Veloy = (2 * massC/(massC+massF))* Colliding-
>Veloy;
                    Colliding->Veloy = ((massC-massF)/(massC+massF)) * Colliding-
>Veloy;
                    Current->Veloz = (2* massC/(massC+massF))* Colliding->Veloz;
                    Colliding->Veloz = ((massC-massF)/(massC+massF)) * Colliding-
>Veloz;

                    //Stores the collision to avoid errors//
                    Current->collisions  = int(Colliding->x);
                    Colliding->collisions = int(Current->x ) ;

                    //Stores time of collision to avoid having the particles colliding
multiple times//
                    Colliding->time = times;
                    Current->time = times;

                    //To keep from certain errors//
                    a=Colliding->Label;
                    b= Current->Label ;

               }
}
```

```
//////////////////////////////////////////////////////
//Header files/////////////////////////////////////
//////////////////////////////////////////////////////

//Includes.h//
//Needed files//

#include "Defines.h"
#include "Functions.h"

//Defines.h//
//Things needed to be included//

//Prevents multiple definitions//
#ifndef DEFINESGUARD
#define DEFINESGUARD

#include <iostream.h>
#include <fstream.h>
```

```
#include <conio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

//Structure of a bond//
struct Bond
{
	bool broke;
};

//Main structure used in the program to represent the particles//
struct Particle
{
	int Label;
	double x,y,z;
	double Velox, Veloy,Veloz;
	ofstream Save;
	Particle *right, *left,*top,*bottom,*forward,*back;
	double collisions;
	double time;
};
```

```
#endif

//Globals.h//

//Global variables needed//

int strength =  1000;    //Bond strength



//Defines the initial particles//

Particle *Colliding = NULL;

Particle *Current = NULL;

Particle *Temp = NULL;

//Defines the masses for a default setting//

double massC =100;

double massF =1;

//Markers//

int s =0;

int as =0;

//Timer//

double times;
```

```
//Used to avoid possible errors//
int a = 5000;
double b = 5000;

//Storage//
ofstream data, data1;

//Functions.h//
//All the functions used in the program//

//Begins the main part of the program//
void Start(int Velo,int massC, int massF);

//Compares particles//
void Compare(Particle *Current, Particle *Colliding, double &b, double massC, double
massF,double s,double a,double times,int run);
void CompareOthers(Particle *Current, Particle *Current2,double massF,double s,double
a,double &b,double times);

//Increments positions of particles of the field//
void Increment(Particle *Current,int run);
```

//Initializes the particle field//

void Initialize(Particle *Current,int as);

//Tests for collisions between particles//

void Test(Particle *Current, Particle *Colliding,double massC, double massF,double radius,double strength, double times,double &a,double &b);

//Not used//

void MakeFile(Particle);