

Simulation of deformation of colliding objects using particle systems
by Shirish Raghunath Koti

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Shirish Raghunath Koti (1989)

Abstract:
Physically-based modeling has become important as a useful tool in realistically animating the complex motions of objects. Much of the work centers around a mathematical development of the model and quite often involves the complexity associated with such a model. This project uses particle systems to model the objects whose deformation during collision is simulated. One of the greatest advantages of this approach is its mathematical simplicity. Since the object is considered as a collection of a large number of independent particles, connected to the corresponding neighboring particles by pseudosprings, the object is never viewed as one entity. This simplifies the process of calculating deformations at any part of the object. Another advantage is obvious when objects of different and arbitrary shapes have to be considered. Flexibility of shape of the objects is inherent to this model. Since it is the pseudo-springs that determine the deformation of the object, the characteristics of only the pseudo-springs need to be changed in order to simulate collisions of different types i.e., elastic or inelastic. The same holds true when objects with different physical properties have to be used in the simulation. In this sense, this model is far more general than many others.

SIMULATION OF DEFORMATION OF COLLIDING OBJECTS USING PARTICLE SYSTEMS

by

Shirish Raghunath Koti

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

August 1989

N378
K8491

APPROVAL

of a thesis submitted by

Shirish Raghunath Koti

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Aug 25th 89
Date

J. Dunbrigh Stanley
Chairperson, Graduate Committee

Approved for the Major Department

Aug 25th 89
Date

J. Dunbrigh Stanley
Head, Major Department

Approved for the College of Graduate Studies

August 31, 1989
Date

Henry L Parsons
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major Professor, or in his absence, by the Dean of Libraries when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Shirish R. Koli

Date Aug. 25, 89

TABLE OF CONTENTS

	Page
LIST OF FIGURES	v
ABSTRACT	vi
1. INTRODUCTION	1
2. SIMULATION MODEL	5
Modeling The Object	5
Role Of The Peripheral Particles	7
Attributes Of The Object	8
Modeling The Scene	9
Modeling The Deformation	9
3. IMPLEMENTATION	13
Major Data Structures	13
Creating Particles	14
Detecting Collision	16
Updating Particle Attributes Before Collision	17
Computing Time Step	18
Updating Particle Attributes During Collision	20
Checking For Errors	23
4. ADAPTING FOR 3-D	24
5. RESULTS AND CONCLUSIONS	26
General Observations	27
Conclusions	29
REFERENCES CITED	30
APPENDIX	32

LIST OF FIGURES

Figure Page

1. Creating Particles 6
2. Head-on Collision 28
3. Collision At An Angle 28
4. Program Listing 33
5. Input Sample - 1 86
6. Input Sample - 2 87

ABSTRACT

Physically-based modeling has become important as a useful tool in realistically animating the complex motions of objects. Much of the work centers around a mathematical development of the model and quite often involves the complexity associated with such a model. This project uses particle systems to model the objects whose deformation during collision is simulated. One of the greatest advantages of this approach is its mathematical simplicity. Since the object is considered as a collection of a large number of independent particles, connected to the corresponding neighboring particles by pseudo-springs, the object is never viewed as one entity. This simplifies the process of calculating deformations at any part of the object. Another advantage is obvious when objects of different and arbitrary shapes have to be considered. Flexibility of shape of the objects is inherent to this model. Since it is the pseudo-springs that determine the deformation of the object, the characteristics of only the pseudo-springs need to be changed in order to simulate collisions of different types i.e., elastic or inelastic. The same holds true when objects with different physical properties have to be used in the simulation. In this sense, this model is far more general than many others.

CHAPTER 1

INTRODUCTION

Over the past few years, physically-based modeling has become an important area of computer graphics research. In particular, many researchers have studied mechanisms for realistically animating colliding objects [1], [2] and [3]. An approach based on particle systems, a relatively new technique in computer graphics, is taken in this project to simulate deformation of colliding objects.

Collision of two or more objects is an extremely common phenomenon in day-to-day life. A game of billiards or tennis, collision of molecules in a fluid being heated, vehicle accidents are all examples of collision. A collision is invariably accompanied by deformation of the objects involved. The shape, extent and duration of the deformation depend on the properties of the objects and the nature of the collision. Although deformation of the ball during a game of tennis is not likely to evoke much interest from the players, analysis and prediction of deformation nevertheless are necessary - at least useful - in many cases. Machine shop operations, tool design, vehicle accidents and protective ordnance are some applications where an analysis finds immense use. A graphical representation of the results of such an analysis greatly enhances the usefulness of the entire exercise.

Most of the earlier approaches to solving this problem of predicting deformation involve severe mathematical complexity. The deforming object is modeled [4] as a body in an inertial frame of reference. The energy stored in the body due to the deformation is expressed in terms of the displacement of the geometric points in the body using differential equations. To solve for displacement of any point, a set of these equations needs to be solved. The complex theory involved in deriving and using these differential equations is quite often a serious limitation.

Particle systems were first introduced by Reeves [5] to animate such fuzzy objects as explosions, fire, cloud and smoke. A particle in such a system is an extremely small part of the object having certain attributes as color, position and velocity. Attributes of a particle are determined stochastically, and every particle has a finite life-span which is less than the life-span of the object i.e., particles are created and destroyed during the animation of the object. Randomness is inherent to a particle system of this nature.

The purpose of this project is to simulate deformation of two colliding objects using particle systems. The concept of particle system is modified for this purpose by eliminating the randomness of the attribute values. These particles are created when the simulation begins and they exist as long as the simulation lasts. The attribute values acquired by the particles are not random but definite.

The object consists of a large number of particles. Each particle is independent in that it can have its own values for such attributes as position and velocity. Each particle is connected to all its neighbors by pseudo-springs which provide the restoring forces. This is the basic model used here. The movement of the particles as a result of the impact forces and the restoring forces determines the deformation of the object when viewed in its entirety.

The fact that the pseudo-springs control the deformation of the object and that the shape of the object is inconsequential allows the objects to take arbitrary shapes. The characteristics of the pseudo-springs determine the type of the collision, i.e. elastic or plastic, and also simulate the physical properties of the objects like modulus of elasticity. For example, increasing the stiffness of the pseudo-spring amounts to increasing the rigidity of the object. The ability to simulate a wide variety of conditions by varying only one parameter, viz. the stiffness and characteristics of the pseudo-springs, makes this model very general.

Absence of mathematical complexity is probably the greatest advantage of this model. Changes to positions and velocities of the particles due to the interplay of restoring and impact forces are determined by using elementary laws of motion which further makes the model simpler. Since deformation is simulated by the movements of the individual particles, the overall shape of the object is unimportant. Consequently, objects of arbitrary shape can be used in the simulation without making any changes in the model. The large number of

particles, however makes this model inherently computationally expensive.

A detailed description of the model is provided in Chapter 2. The process of translating the model into an executable system is explained in Chapter 3. The model used in this project simulates objects in two dimensions. Chapter 4 covers the modifications necessary to extend the model to three dimensional objects. Finally, Chapter 5 discusses results and conclusions.

CHAPTER 2

SIMULATION MODEL

The objects, scene, and the deformation all have their own models. Though these models are related to each other one way or the other, they will be dealt with separately in the subsequent sections. The models and implementations discussed in this project are for two dimensional objects, although extending them for three dimensional objects is relatively straight forward. In fact, work to that effect is already in progress.

Modeling The Object

The object is assumed to be composed of a large number of particles. A particle is an entity resembling a geometric point, having a definite position and its own mass, and other attributes. In this respect, all particles are identical. The particles are formed by conceptually placing a grid over the object and selecting those grid points that fall on or inside the object boundary. A grid of a predetermined size is chosen for this purpose. Although it is not required during modeling, a square grid is used to simplify implementation. Hereafter, the word grid always implies a square grid. The grid is big enough to completely enclose the entire object when placed over it. Figure 1 shows the

process of creating particles pictorially. The grid is shown to be very coarse (small grid size) in Figure 1 for the purpose of clarity.

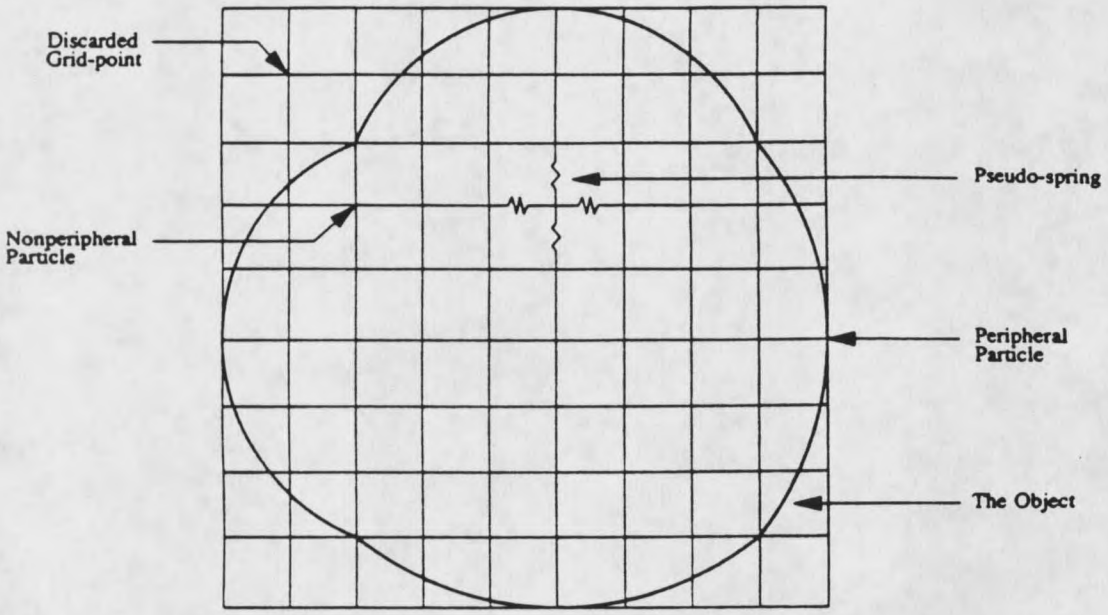


Figure 1
Creating Particles

Every grid point is checked to see if it falls outside, on or inside the object boundary. All the grid points falling on or inside the object form particles belonging to the object; the rest are discarded. Particles which lie inside the boundary of the object are the nonperipheral particles, and those which lie on the boundary of the object are the peripheral particles. Peripheral particles have a special role which will be discussed in the following section. The status of a particle as to whether it is peripheral or nonperipheral stays unchanged in most

cases. Under special circumstances, however, where the object in collision undergoes fracture, a nonperipheral particle may become peripheral.

Each particle, whether peripheral or nonperipheral, has its own set of attributes. Identity of the particle, which is unique in a given object, position, velocity and acceleration of the particle, and status of the particle as to whether the particle is peripheral or nonperipheral are some of the attributes necessary for every particle.

Every particle is conceptually connected to four other particles - its left, right, top and the bottom neighbors. Each connection is a pseudo-spring. The spring offers certain resistance to compression and elongation, and in general the values for compression and elongation are different. The same springs can also be made to withstand forces up to a limit without having any deformation and then completely fail for any force above this limit. This kind of a behavior can be used to model objects like an egg shell. These springs have no use until a collision with some other object occurs since until then there is no relative displacement between any two particles. Once a collision occurs, however, these springs prove to be the heart of simulation of deformation.

Role Of Peripheral Particles

Peripheral particles are no different from nonperipheral particles as far as the attribute values or the change in the attribute values are concerned.

However, they do find a special role in detection of collision and such other tasks. A collision can occur only if boundaries (i.e. peripheral particles) of two objects come in contact. It is therefore only necessary to check if the peripheral particles are colliding. Since an object has a very small number of peripheral particles compared to nonperipheral particles, a considerable amount of computation is saved. A similar saving in computation is achieved when only peripheral particles are used to check if an object is leaving the scene. Another advantage of identifying a particle as peripheral or nonperipheral is evident while determining the neighbors of a particle. A nonperipheral particle must have all the four neighbors while a peripheral particle can have one, two or at most three neighbors.

Attributes Of The Object

The impact time, impact forces and consequently, the extent and shape of deformation are largely governed by physical properties of the material of the object, such as modulus of elasticity and Poisson's ratio [6]. The size of the object, mass of the object and initial velocity of the object also make a significant difference in the final results. Another important attribute which directly influences the end result is the stiffness of the pseudo-spring connecting two particles. There are several other attributes like color of the object which do not affect the simulation in a significant way, but are nevertheless necessary.

Modeling The Scene

The lower and upper extents of the scene define its boundary. The scene is divided into a predetermined number of square boxes. It may take several boxes to contain an object completely, and a given box may contain parts of several objects including none at all. Size of the box is small enough not to contain an object completely in one box or else the savings in computations achieved by dividing the scene would not be worth the effort.

Peripheral particles of an object in a given box are considered for detection of collision with peripheral particles of another object only in that box. In other words, a check to detect a collision is made from one box at a time, and only peripheral particles of different objects contained in that box are considered. All the boxes which contain peripheral particles of only one object or no object at all are skipped. This way a great deal of computation is saved while checking for occurrence of collision. This method is very similar to that of space-division adopted by Glassner [7].

Modeling The Deformation

Once a collision occurs, it lasts for a very small but definite amount of time. This contact duration for collision of two spheres is given by [8]

$$\tau = (4.53)\left[\left(\frac{1-\mu_1^2}{E_1\pi}+\frac{1-\mu_2^2}{E_2\pi}\right)\frac{m_1m_2}{m_1+m_2}\right]^{\frac{2}{5}}\left[\frac{R_1+R_2}{v_0R_1R_2}\right]^{\frac{1}{5}} \quad (2.1)$$

where μ_1, μ_2 are Poisson's ratio for the materials of the objects, E_1, E_2 are moduli of elasticity for the material of the objects, m_1, m_2 are the masses of the objects, R_1, R_2 are the radii of the objects (spheres), and v_0 is the relative velocity between the two objects.

The contact duration is divided into a large number of intervals and for every interval, position and velocity of each particle are updated taking into account the external and the internal forces acting on each of the particles. The new position and the velocity are given by

$$w = ut + \frac{1}{2}at^2 \quad (2.2)$$

and

$$v = u + at \quad (2.3)$$

where w is the change in the value of the x or y component of the position, u is the initial value of the corresponding component of velocity, a is the value of the corresponding component of acceleration, and t is the time interval for which updating is carried out.

The direction of the external force is given by the position vector joining the two peripheral particles in collision. The magnitude depends on the time that has elapsed since collision started. The magnitude of this force is given by [8]

$$f_e = \frac{1.14 m_1 m_2 v_0^2}{(m_1+m_2)\alpha_m} \sin\left[\frac{1.068 v_0 t}{\alpha_m}\right] \qquad \left[0 \le t \le \frac{\pi\alpha_m}{1.068 v_0}\right] \tag{2.4}$$

where

$$\alpha_m = \left[\frac{15\pi v_0^2\left(\frac{1-\mu_1^2}{E_1\pi}+\frac{1-\mu_2^2}{E_2\pi}\right) m_1 m_2}{16(m_1+m_2)}\right]^{\frac{2}{5}} \left[\frac{R_1+R_2}{R_1 R_2}\right]^{\frac{1}{5}}$$

and all the other symbols have the same meaning as in equation 2.1.

As can be seen from the equation, the external force is absent at the start and the end of collision, and it varies sinusoidally during the collision.

Before a collision occurs, all particles in an object move with the same velocity at any given time. When a collision occurs, those peripheral particles involved in the collision experience the impact forces. At this time, the rest of the particles are not even aware of a collision. Because of the impact forces, which by definition oppose the initial motion, an acceleration opposing the velocity is introduced in the peripheral particles involved in collision. This acceleration changes the velocity, and consequently the displacement of these particles compared to that of the rest of the particles. The distances between a particle involved in collision and its neighbors are not the same as they were under no-collision conditions. The deviation from the normal distance causes the pseudo-spring to stretch or compress as the case may be, and the particle in collision and its neighbors experience the restoring forces of the pseudo-springs.

These forces are called internal forces.

The direction of the internal force is given by the position vector of the two particles in question. The magnitude is given by

$$f_i = k\Delta l \tag{2.5}$$

where k is the spring constant with the pseudo-spring in compression or elongation and Δl is the change in the length of the pseudo-spring

The internal forces are experienced by every particle of the object at some point in time or the other. In fact, it is due to the internal forces that a particle far away from a peripheral particle in collision and initially unaware of the collision is eventually affected by the collision.

Thus, in the duration that a collision lasts, all the particles experience internal forces of different magnitudes and consequently experience a deviation from their normal displacements. The deviation from the normal displacement is more pronounced for particles closer to the particles involved in collision, and progressively decreases for particles away from the particles in collision. A macroscopic view of the deviation of each particle from the normal displacement results in simulation of the deformation of the object.

CHAPTER 3

IMPLEMENTATION

The entire program is divided into different logical modules consistent with the tasks performed within each module. Some of the data structures are common to all the modules. These data structures will be discussed here in brief before describing the functioning of the various modules.

Major Data Structures

The scene has a structure, struct scene_st, associated with it that contains information like the upper and lower extents of the scene and the number of objects in the scene.

Every object has information like the physical properties, dimensions, initial velocity, number of particles in that object, and serial number of that object. All this information is stored in a structure, struct obj_st. This structure also has a pointer to where information on all its particles is stored dynamically. Another part of this structure is a pointer to a structure, struct rowptr_st, which is basically a linked list of addresses of rows of particles along with the first and the last column beyond which a particle cannot be found in that row. This linked list is useful in determining the neighbors of a given particle.

Every particle of an object has a structure, struct part_st, associated with it to contain information about position, velocity, acceleration and status of the particle. Status tells whether a particle is peripheral or nonperipheral.

When a collision is detected, information about the objects in collision and the particular particles in collision is stored in a structure, struct coll_st. A linked list of this structure is maintained if more than one pair of peripheral particles are in collision.

As mentioned in the earlier chapter, the entire scene is divided into a number of boxes. The information about the peripheral particles of all the objects in a given box is maintained as a multidimensional array of linked lists of a structure, struct box_st.

Creating Particles

The function of this module is to determine which particles belong to the given object and to assign values to all the attributes of every particle.

The overall dimensions of the object, its initial velocity and the grid size to be used for that object are known through the data file supplied as input to the program. The top, left, right and bottom extremes of the object are determined.

As seen earlier in Figure 1, imaginary horizontal scan-lines are drawn through the top extreme of the object progressing towards and including the

bottom extreme. Vertical scan-lines are drawn through the left extreme progressing towards and including the right extreme. An intersection of a horizontal scan-line with a vertical scan-line constitutes a grid point. All the scan-lines are equally spaced as a result of which the distance between any two neighboring grid points is the same. The number of horizontal scan-lines is the same as that of vertical scan-line and are both equal to the grid size for the object.

Every grid point is checked to see if it lies outside, on or inside the object boundary. For an object that is a circle, this reduces to solving the equation of the circle using the grid point as a point. All the grid points that lie outside the object boundary are discarded. If a grid points falls on the boundary, its status is set to peripheral. A given row of particles must have at least one peripheral particle. If the grid point under consideration is the first peripheral particle in that row then the address of this new row is added to the linked list of row-addresses, along with the first and last valid columns of this row.

The status of a particle that lies inside the boundary is set to nonperipheral. Irrespective of its status, every particle initially has the same velocity as that of the entire object. The position of each particle is set to the x and y coordinates of the corresponding grid point. The acceleration is set to zero for every particle in this implementation, although including initial acceleration for the object and hence for all the particles is a trivial modification. The identity of every particle is obtained by assigning two numbers to the particle. The two

numbers indicate the row and the column used to obtain the grid point corresponding to this particle. This scheme of identifying particles enables each particle to be identified uniquely in a given object.

While attributes are assigned to every particle, the linked list of peripheral particles in a given box is also extended if the particle under consideration is peripheral. In other words, if a particle is peripheral then the box in which this particle lies is determined. There is one linked list for every object whose peripheral particles lie in this box. If the peripheral particle under consideration is the first one for this object in this box, a new linked list is created. Otherwise, the existing linked list is simply extended.

When all the grid points are considered, the number of particles in the object is known. With the mass of the object known through the data file, the mass of each particle is determined.

Detecting Collision

The function of this module is to check every object against every other object in the scene to determine if the two objects have collided. A proper value is returned depending on whether or not a collision is detected.

The distance between a peripheral particle of one object in a given box and a peripheral particle of another object in the same box is determined using their

positions. If this distance is less than a predetermined limit, a collision is assumed to have occurred. If the distance is greater, another pair is tested. All possible pairs of peripheral particles in the same box are tested. Two peripheral particles of the same object are not tested even if they are in the same box, since an object can only collide with another object, and never with itself. If all combinations in a box are exhausted, the process is repeated for other boxes. If all boxes are exhausted and no collision is detected, an appropriate value indicating this condition is returned.

If a collision is detected, the serial numbers of the two objects in collision, pointers to the two peripheral particles involved, and the box in which collision is detected are stored. If more than one pair are involved in a collision, a linked list is created. Thus, if a collision is detected, a linked list of this kind with one or more elements is created. A pointer to this linked list is returned when a collision is detected.

Updating Particle Attributes Before Collision

This is a relatively straight-forward function. A time step is chosen as discussed in the following section. Positions of all the particles are updated using this time step and their velocities. Since no collision has occurred, all particles have the same velocities. Consequently, every particle undergoes the same displacement when time is advanced through this time step.

When particle positions are updated, some peripheral particles might cross the box boundaries. It is therefore necessary to update the box information for every peripheral particle. The linked lists of peripheral particles are completely recreated, instead of being updated each time this function is invoked.

In addition to the above which causes permanent changes to attribute values, this module also performs the task of temporarily updating the peripheral particles while checking to make sure two objects do not intersect each other.

Computing Time Step

The decision as to whether a collision has occurred is based solely on whether the smallest distance between any two peripheral particles of different objects is less than a predetermined limit. If the time step is not small enough, two objects could intersect and still a collision wouldn't be detected, or a collision would be detected but particles in collision would be incorrectly identified. Such a situation would be catastrophic to the entire simulation and should be avoided at any cost. It is therefore extremely important to have a time step that is small enough. On the other hand, if a time step is too small, too many computations are involved for a precision that is not required.

This function takes a big time step initially. With this time step, all the peripheral particles are updated which is a temporary change. It is not

necessary to update the nonperipheral particles since they do not take part in collision detection. After updating peripheral particles for all the objects, a test as described below is made to see if two objects have intersected. If any two objects have intersected, the initial time step is halved and the entire process repeated until no two objects intersect. At this point, attributes of all the peripheral particles are restored to their original values and the latest value of time step is returned.

If two particles cross their paths and are in opposite planes after an update then both their x and y direction ratios change signs. For example, if two particles are to the left and right of a plane and after update go to the right and left of the plane respectively then the sign of x and y directions ratios after the update would be the opposite of their counterparts before the update. It is this fact that is used to determine if two objects have intersected. The magnitude of direction ratios or distance between particles are not useful here.

The x and y direction ratios of every pair of peripheral particles belonging to different objects but in the same box are computed before and after the temporary update. If there is a change in sign of both the x and y direction ratios for any pair then the objects are intersecting, indicating that the time step is too big.

Updating Particle Attributes During Collision

As soon as collision is detected, the contact duration is computed using equation 2-1, even before this module is executed.

When a collision is detected, the relevant information about all the particles involved in the collision is made available to this module. All the particles involved in the collision for the two objects are marked to be peripheral, which they must be, and affected. At this point, the contact duration is divided into a predetermined number of intervals. The following procedure is then repeated for every interval.

First, the positions and velocities of all the particles - peripheral or nonperipheral - are updated using equations 2-2 and 2-3 respectively. All the particles that have not been affected yet by the collision have the same velocity and no acceleration. It is computationally economical to compute the new positions of these particles separately.

Next, all the particles that are neighbors of an affected particle are marked as affected. The structure struct rowptr_st mentioned earlier stores information about the starting address of each row and the first and last useful column in that row. This data structure makes the task of finding neighbors of a given particle easier and more efficient. The grid size used in creating particles determines the maximum number of particles in a row or a column. When the

number of intervals numerically exceeds twice the grid size, all of the particles in an object are guaranteed to be affected by the collision.

At this stage, accelerations of all the particles need to be calculated. All of the particles which are involved in the collision experience the impact force resulting in acceleration. This acceleration is termed as the external acceleration. By virtue of the pseudo-springs, all the particles experience a force resulting in an acceleration. This acceleration is termed as internal acceleration.

To obtain external acceleration, first the force due to impact is calculated using equation 2-4. The magnitude of this force is divided by as many pairs of peripheral particles as the ones involved in the collision. This is then further divided by the mass of a particle. This is the magnitude of external acceleration experienced by any particle in collision. An assumption that the total force of impact is equally divided among all the particles of an object in collision is implicit here. The direction of external acceleration is obtained by computing the direction cosines of the position vector between the pair of particles under consideration. The direction cosines then readily produce the x and y components of external acceleration, with the magnitude already known.

Internal acceleration is caused by the change in the length of the pseudo-spring. A particle is connected to four neighbors and therefore to four springs. Each spring has its own force acting on the particle. The distance between the

particle under consideration and one of its neighbors is calculated using their positions. The difference between this distance and the normal distance between any two particles is the compression or elongation of the pseudo-spring. The force is then obtained using equation 2-5. The spring constant chosen for this depends on whether the pseudo-spring is in compression or tension. This force is then divided by mass of the particle to obtain magnitude of internal acceleration. The position vector between the particle and the neighbor determines direction of the acceleration. The direction cosines then readily produce the x and y components of the internal acceleration, with the magnitude already known. It is important to choose an appropriate position vector depending on whether the pseudo-spring is in compression or tension. This procedure is repeated for all the neighbors of the particle. The x and y components of internal accelerations due to each of the neighbors are added. This then becomes the internal acceleration of the particle.

Once the process of updating positions and velocities, marking the particles newly affected by collision and computing external and internal accelerations is completed for all the particles of both the objects, the particles are displayed. This entire cycle is then repeated for the next interval.

Display is carried out on HP SRX-350 work station using the HP-UX Starbase graphics package.

Checking For Errors

The primary function of this module is to make sure that the program is terminated whenever an object leaves the scene. Checks are also made in the beginning as soon as the data file is read to ensure that there are no inconsistencies in the data. To ensure that every object is within the scene, the positions of the peripheral particles are checked against the scene boundaries. Since no nonperipheral particle can leave the scene before a peripheral particle leaves, it is not necessary to check positions of nonperipheral particles.

CHAPTER 4

ADAPTING FOR 3-D

Currently this program has been implemented for the two dimensional case. However, there are hardly any conceptual differences between the two and the three dimensional cases. The minor differences in the implementation are discussed below.

The attributes such as position, velocity and the acceleration will now require the z component. Modules used for computing the time step and the accelerations make use of direction ratios and direction cosines of the position vector. They now will have to take the z coordinate also into account.

When particles are created, scan-lines are drawn from the top to the bottom and from the left to the right of the object to determine the grid points that constitute particles. In the three dimensional case, similar scan-lines will have to be drawn but not just in one plane. Rather, every plane between the front and the back will have to be covered. The distance between two successive planes can be the same as that between two successive scan-lines. The scene is currently divided into boxes of equal size. In the two dimensional case, these should rather be called squares. The same needs to be extended by dividing the three dimensional scene into cubes or boxes of equal size (i.e. voxels).

Some of the modules will need modification. The module that gets a neighbor of a given particle should be capable of finding a neighbor in the front of or behind the particle, since the particle will now have six neighbors instead of four. Similarly, the module that computes the internal acceleration will have contribution from two more neighbors. Some changes will be necessary in the module that displays the objects since a three dimensional rather than a two dimensional picture will have to be displayed.

The need to modify the program for a three dimensional case can hardly be understated. If the simulation has to have any practical value, then objects must be three dimensional. The two dimensional simulation serves as a good model and makes visualization easier. As pointed out in the preceding sections, extending the model to the three dimensional case is relatively trivial.

CHAPTER 5

RESULTS AND CONCLUSIONS

As discussed earlier, the efforts in this project have been focused on two dimensional objects. Simulation was carried out on two circular objects by using different values for the various parameters like stiffness of the pseudo-springs, velocities, masses, positions, and the material properties of the objects.

The simulation was done on a DECstation 3100 and displayed on a HP SRX-350 workstation. The entire software is written in C. A listing of the program appears in Figure 4 in the Appendix.

Figure 2 shows a picture of the two objects colliding head-on with each other. The values used for the various parameters are obtained through the data file supplied as input to the program. Relevant portions of this data file are shown in Figure 5 in the Appendix.

Figure 3 shows a picture of the two objects in collision such that the velocity vectors are not parallel, i.e. the collision is not head-on. Figure 6 in the Appendix shows relevant portions of the data file used for this simulation.

Deformation can clearly be observed in both the objects in both of the cases above. As expected, deformation is limited to a very small depth from the surface compared to the dimensions of the objects. A conical pattern with the apex towards the center of the object can be observed on both the objects. This

pattern indicates the path taken by the wave due to the impact as it travels from one end of the object to the other. The aliasing effect seen in both the pictures was not eliminated in order to preserve the above mentioned conical pattern.

General Observations

As it is to be expected, the more the steps into which the impact time is divided, the better are the results. Obviously, the amount of computation increases, although linearly, with the increased number of steps. An appropriate trade-off is therefore necessary. The same holds true for the grid size which ultimately determines the number of particles in the object. An increased number of particles has another drawback - increased memory requirement.

Although it appears that the number of steps into which the impact time is divided and the grid size are unrelated quantities, simulation shows that number of steps should be at least an order of magnitude greater than the grid size. The relation between the two becomes obvious if the following is taken into account: a small number of time steps means a larger time interval. Since the impact force depends on time, as can be seen from equation 2-4, this means that larger impact forces give rise to larger displacements of particles. If the displacement of a particle with respect to another particle is much larger than the normal distance between the two, i.e. if the compression or elongation of the pseudo-spring is much larger than its normal length, the spring model breaks down.

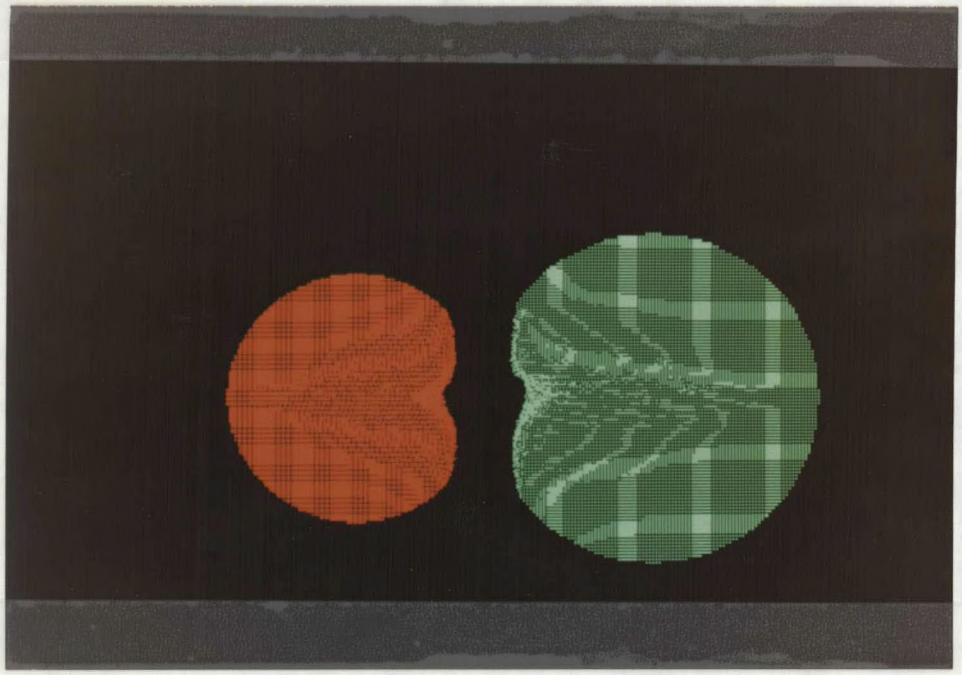


Figure 2
Head-on Collision

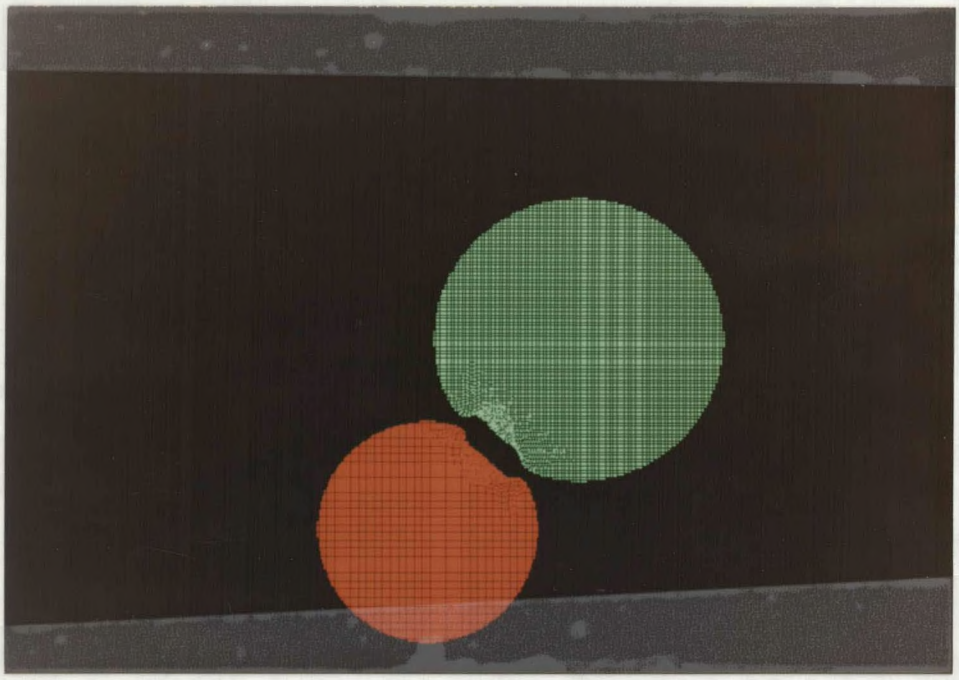


Figure 3
Collision At An Angle

Conclusions

The goal of this project was to use a simpler model to simulate deformation of colliding objects. This model is conceptually one of the simplest and is far more general than other models, and the simulation based on this model has produced promising results. There is still a lot of effort needed before collision of three dimensional objects of arbitrary shape can be simulated to obtain accurate deformations.

REFERENCES CITED

REFERENCES

1. Hahn J.K., Realistic Animation Of Rigid Bodies, Computer Graphics, Vol.22, No.4, Aug 88, pp299-308.

2. Platt J.C., Barr A.H., Constraint Methods For Flexible Models, Computer Graphics, Vol.22, No.4, Aug 88, pp279-288.

3. Moore M., Wilhelms J., Collision Detection And Response For Computer Animation, Computer Graphics, Vol.22, No.4, Aug 88, pp289-298.

4. Terzopoulos D., Fleischer K., Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture, Computer Graphics, Vol.22, No.4, Aug 88, pp269-278.

5. Reeves W.T., Particle Systems - A Technique For Modeling A Class Of Fuzzy Objects, ACM Transactions On Graphics, Vol.2, No.2, April 83, pp359-376.

6. Crandall S.H., Dahl N.C., Lardner T.J., An Introduction To The Mechanics Of Solids, McGraw-Hill International Book Company, 1978, pp280-286.

7. Glassner A.S., Space Subdivision For Fast Ray Tracing, IEEE Computer Graphics And Applications, Vol.4, No.10, Oct 84, pp15-22.

8. Goldsmith W., Impact, Edward Arnold Publishers Ltd., 1960, pp82-91.

APPENDIX

Figure 4
Program Listing

```
/*********************************************************************
 *                                                                   *
 * This file  : collide.c (main)                                     *
 * Other files: parts.c                                              *
 *              updates.c                                            *
 *              display.c                                            *
 *              globals.h                                            *
 *                                                                   *
 * This file contains functions main(), readdata(), null_pointers(),*
 * error_check(), and compute_impact_force().                        *
 *                                                                   *
 *********************************************************************
/




#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <strings.h>
#include "globals.h"


main(argc,argv)
  int   argc;
  char  *argv[];
{

   int      i;
   int      fildes;                 /* file descriptor for display */
   int      srno_fst, srno_sec;     /* sr.no. of objects in collision
*/
   char     ans[5];
   double   delta_time, dummy;
   obj_s    object[MAXOBJ], coll_object[2];
   coll_s   *save_coll, *collptr;

   coll_s   *collision_detect();
   part_s   *get_neighbour();
   void     readdata(), create_particles(), null_pointers(), error_ch
eck();
   void     before_collision_update(), during_collision_update();
```

Figure 4 (contd.)

```
double   compute_time_step(), compute_impact_force();
int      display_initialize();
void     display_picture();




if (argc < 2)
   fprintf(stderr,"\nUsage : %s <datafile>\n\n",argv[0]), exit(1);

readdata(argv[1],object);

error_check();

for (i=0;i<MAXOBJ;i++)
   null_pointers(i);          /* all pointers to null */

  /* STD_DELTA_TIME is the initial 'seed' to compute delta_time */
delta_time = STD_DELTA_TIME;


for (i=0;i<scene.nobjects;i++)
   create_particles(&object[i]);


fildes = display_initialize();



  /* while(1) simulates the motion forever until one of the object
s goes
      out of the screen.  This condition may be changed if simulati
on is
      desired for a specific period.
  */

while (1) {

   collision_in_progress_global = FALSE;

   delta_time = compute_time_step(object,delta_time);

   if ((save_coll = collptr = collision_detect()) == (coll_s *)NUL
L) {

      for (i=0;i<scene.nobjects;i++) {
         before_collision_update(object[i],delta_time,PERMANENT);
         error_check();
      }
```

Figure 4 (contd.)

```
            /* i=0 => clear the screen */
           for (i=0;i<scene.nobjects;i++)
              display_picture(fildes,object[i],i);
        }

        else {     /* collision has occured */

            /* the two objects in collision */
           srno_fst = collptr->srno_fst;
           srno_sec = collptr->srno_sec;

            /* now, compute only impact time => interval = ZERO (dummy
 not used)
                (this function also sets collision_in_progress_global t
o TRUE) */
           dummy = compute_impact_force(object[srno_fst],object[srno_se
c],ZERO);


            /* for all objects other than the ones in collision */
           for (i=0;i<scene.nobjects;i++) {
              if ((object[i].srno != collptr->srno_fst) &&
                  (object[i].srno != collptr->srno_sec)) {
                 before_collision_update(object[i],delta_time,PERMANENT
);
                 error_check();
              }
           }

            /* now, display the objects not in collision */
           for (i=0;i<scene.nobjects;i++)
              if ((object[i].srno != collptr->srno_fst) &&
                  (object[i].srno != collptr->srno_sec))
                 display_picture(fildes,object[i],i);


            /* for the two objects involved in collision, update the p
ositions
                and display, continuously throughout the impact duratio
n
                (display_picture() is invoked in during_collision_upda
te()) */
           for (i=0;i<scene.nobjects;i++) {
              if (object[i].srno == collptr->srno_fst)
                 coll_object[0] = object[i];
              if (object[i].srno == collptr->srno_sec)
                 coll_object[1] = object[i];
           }
```

Figure 4 (contd.)

```
            during_collision_update(fildes,coll_object,collptr);
            error_check();


              /* now that impact simulation is over, do a normal update
on all the
                 objects so that the two objects in collision move suffi
ciently far
                 away from each other: this way collision_detect() will
not wrongly
                 announce another collision.
          */
            delta_time = RECVRY_FCTR*impact_time_global;
            for (i=0;i<scene.nobjects;i++) {
               before_collision_update(object[i],delta_time,PERMANENT);
               error_check();
            }

            for (i=0;i<scene.nobjects;i++)
               display_picture(fildes,object[i],i);

              /* null the pointer, too */
            collptr = (coll_s *)NULL;

            delta_time = STD_DELTA_TIME;

         }

      }

   }
```

```
  /************************  End of main()  ************************
**/
```

```
/*********************************************************************
 *                                                                   *
 * Name      : readdata()                                            *
 * Input     : Name of data file                                     *
 *             Pointer to an array of objects                        *
```

Figure 4 (contd.)

```
 * Output    : None                                                    *
 * Function : Read data file                                           *
 *                                                                     *
 ***********************************************************************/


void readdata(fname,obj)
  char    *fname;
  obj_s   *obj;
{

   int      i;
   char     buf[85];
   FILE     *fp;



   if ((fp = fopen(fname,"r")) == NULL)
      fprintf(stderr,"\nCan't open %s\n\n",fname), exit(1);


     /* get rid of all the text: data starts after the $ sign */
   do {
      fgets(buf,85,fp);
      buf[ strlen(buf) - 1 ] = '\0';
   } while (strcmp(buf,"$") != 0);

     /* lower extents of the scene */
   fscanf(fp,"%f%f%f",&(scene.lowex.x),&(scene.lowex.y),&(scene.lowex
.z));
   fgets(buf,85,fp);

     /* upper extents of the scene */
   fscanf(fp,"%f%f%f",&(scene.uppex.x),&(scene.uppex.y),&(scene.uppex
.z));
   fgets(buf,85,fp);

     /* no. of objects */
   fscanf(fp,"%d",&(scene.nobjects));
   fgets(buf,85,fp);

     /* no. of intervals into which impact time should be divided */
   fscanf(fp,"%ld",&num_intervals_global);
   fgets(buf,85,fp);

     /* real-time simulation (1) or store data on disk (0) */
   fscanf(fp,"%d",&real_time_global);
   fgets(buf,85,fp);

     /* now info about each of the objects */
```

Figure 4 (contd.)

```
   for (i=0;i<scene.nobjects;i++,obj++) {

        /* data for an object follows the $ line: get rid of the line
 */
        fscanf(fp,"%s",buf);

        /* name of the object (discard here), and object id */
      fscanf(fp,"%s",buf);
      fscanf(fp,"%d",&(obj->id));
      fgets(buf,85,fp);
      obj->srno = i;          /* start sr.no. from 0 */

        /* centre and the radius */
      fscanf(fp,"%f%f%f",&(obj->centre.x),&(obj->centre.y),&(obj->cen
tre.z));
      fscanf(fp,"%f",&(obj->radius));
      fgets(buf,85,fp);

        /* colour of the object : red, green, blue */
      fscanf(fp,"%f%f%f",&(obj->colour[0]),&(obj->colour[1]),&(obj->c
olour[2]));
      fgets(buf,85,fp);

        /* grid width : this determines no. of particles in the objec
t.
           For symmetry purposes, this must be even.  If not, force i
t ! */
      fscanf(fp,"%d",&(obj->grid));
      fgets(buf,85,fp);
      if (obj->grid % 2 == 1)
         obj->grid += 1;

        /* normal distance between two particles (of the same object)
 */
      obj->dnot = (2 * obj->radius)/(obj->grid);

        /* spring constant in compression and extension */
      fscanf(fp,"%f%f",&(obj->k_comp),&(obj->k_extn));
      fgets(buf,85,fp);

        /* initial velocity of the object */
      fscanf(fp,"%f%f%f",&(obj->vel.x),&(obj->vel.y),&(obj->vel.z));
      fgets(buf,85,fp);

        /* mass of the object */
      fscanf(fp,"%f",&(obj->mass));
      fgets(buf,85,fp);

        /* Poisson's ratio for the material of the object */
      fscanf(fp,"%f",&(obj->mu));
```

Figure 4 (contd.)

```
      fgets(buf,85,fp);

        /* Modulus of elasticity for the material of the object */
      fscanf(fp,"%f",&(obj->elast));
      fgets(buf,85,fp);

        /* initialize the pointer to null */
      obj->start = (part_s *)NULL;

   }

}



        /*****************  End of readdata()  **************/




/*********************************************************************
 *                                                                   *
 * Name      : null_pointers()                                       *
 * Input     : Serial number of object                               *
 * Output    : None                                                  *
 * Function : Null the linked list of peripheral particles in all    *
 *            boxes                                                  *
 *                                                                   *
 *********************************************************************/


void null_pointers(obj_srno)
  int  obj_srno;                /* for which object is this to be done
 */
{

   register int    i, j, k;


   /* REMEMBER: before_collision_update() also uses this function */


     /* for error-checking purposes, i,j,k should go to BOXNUM+1 */

   for (i=0;i<BOXNUM+1;i++)
      for (j=0;j<BOXNUM+1;j++)
```

Figure 4 (contd.)

```
     /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
       ^ NOTE : Change this ONE to BOXNUM for 3-d     ^
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
         for (k=0;k<ONE+1;k++)
            box[i][j][k].start[obj_srno] = (part_s *)NULL;

}



  /*********************  End of null_pointers()  *******************
/




/*********************************************************************
 *                                                                   *
 * Name      : error_check()                                         *
 * Input     : None                                                  *
 * Output    : None                                                  *
 * Function : Check for data inconsistencies and to see if any of    *
 *            the objects leaves the scene                           *
 *                                                                   *
 *********************************************************************/


void error_check()
{

   static int     first_time=TRUE;
   int            i, j, k, m;
   float          x, y, z;
   part_s         *perphptr;


     /* when this function is invoked for the first time, check if th
e data
        furnished by the user has any inconsistency */

   if (first_time) {
      if ( (fabs(scene.lowex.x - scene.uppex.x) < EPSILON) ||
```

Figure 4 (contd.)

```
          (fabs(scene.lowex.y - scene.uppex.y) < EPSILON) ||
          (fabs(scene.lowex.z - scene.uppex.z) < EPSILON) )
        fprintf(stderr,"\nImproper scene extents: program terminated
.\n\n"), exit(1);

      if (scene.nobjects > MAXOBJ)
        fprintf(stderr,"\nToo many objects in the scene: program ter
minated.\n\n"), exit(1);

      if (scene.nobjects <= 0)
        fprintf(stderr,"\nNo objects in the scene: program terminate
d.\n\n"), exit(1);

      if (BOXNUM == 0)
        fprintf(stderr,"\nBOXNUM should be nonzero: program terminat
ed.\n\n"), exit(1);

      first_time = FALSE;

   }


     /* check if after update any of the peripheral particles of any
of the
        objects has gone out of scene */
   else {

     /* for error-checking purposes, i,j,k should go to BOXNUM+1 */

      for (i=0;i<BOXNUM+1;i++)
        for (j=0;j<BOXNUM+1;j++)
             /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^
               ^ NOTE : Change the ONE to BOXNUM for the 3-d case
 ^
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^*/
           for (k=0;k<ONE+1;k++)

              /* for every object in the scene */
             for (m=0;m<scene.nobjects;m++) {

               perphptr = box[i][j][k].start[m];

               while (perphptr != (part_s *)NULL) {

                  x = perphptr->posn.x;
                  y = perphptr->posn.y;
                  z = perphptr->posn.z;
```

Figure 4 (contd.)

```
                    if ( (x < scene.lowex.x || x > scene.uppex.x) ||
                         (y < scene.lowex.y || y > scene.uppex.y) ||
                         (z < scene.lowex.z || z > scene.uppex.z) )
                       fprintf(stderr,"\nObject %d leaving scene: pr
ogram terminated.\n\n",m+1), exit(1);
                    perphptr = perphptr->next;

                }

            }

    }

}
```

```
  /******************  End of error_check()  ******************/
```

```
/*****************************************************************
 *                                                               *
 * Name      : compute_impact_force()                            *
 * Input     : The two objects involved in collision             *
 *             The interval for which impact force is desired    *
 * Output    : None, when this function is invoked for the first time*
 *             Magnitude of impact force, on subsequent invocations  *
 * Function  : When invoked for the first time,                  *
 *               calculate the impact time                       *
 *               set the parameter to say collision is in progress   *
 *               calculate all quantities to calculate forces later  *
 *             When invoked for the second time or later,        *
 *               calculate the magnitude of impact force on every    *
 *               particle involved in collision                  *
 *                                                               *
 * Theory    : Force and the contact-duration on impact          *
 *                                                               *
 *                          1.14*v_not^2          1.068*v_not*totime   *
 *       contact force = ------------- * sin (--------------------) *
```

Figure 4 (contd.)

```
 *                      k_one*alpha_m              alpha_m        *
 *                                                                *
 *                                                                *
 *            (Ref. page 83-90 of IMPACT by Werner Goldsmith)     *
 *                                                                *
 ******************************************************************
/


double compute_impact_force(object_one,object_two,interval)
  obj_s  object_one, object_two;
  int    interval;
{

   static double    k_one;
   static double    alpha_m, v_not;
   static double    t_const, f_const;
   double           delta_one, delta_two;
   double           totime;          /* total time since collision s
tarted */
   double           force_magn=0.0;   /* magnitude of the force */
   double           m1, m2, r1, r2, mu1, mu2, e1, e2, temp;


     /* Collision is in progress: calculate the impact force now */
   if (collision_in_progress_global) {
      totime = t_const * interval;
      force_magn = f_const * sin(totime);
   }

     /* Collision is just detected: calculate the impact time now */
   else {

      collision_in_progress_global = TRUE;

        /* calculate the various quantities to get t_const and f_cons
t */
      mu1 = object_one.mu;
      mu2 = object_two.mu;
      m1 = object_one.mass;
      m2 = object_two.mass;
      r1 = object_one.radius;
      r2 = object_two.radius;
      e1 = object_one.elast;
      e2 = object_two.elast;

        /* v_not is the relative speed between the two objects */
      v_not =  pow( (double)(object_one.vel.x - object_two.vel.x), 2.
0 )
```

Figure 4 (contd.)

```
            + pow( (double)(object_one.vel.y - object_two.vel.y), 2.
0 )
            + pow( (double)(object_one.vel.z - object_two.vel.z), 2.
0 );
      v_not = pow(v_not,0.5);

      delta_one = (1 - mu1*mu1)/(e1*PI);
      delta_two = (1 - mu2*mu2)/(e2*PI);

      k_one = (m1 + m2)/(m1*m2);
      temp = ((15*PI*v_not*v_not*(delta_one+delta_two)*m1*m2)/(16*(m1
+m2)));
      temp = pow(temp,2/5.0);
      alpha_m = temp * pow((r1+r2)/(r1*r2),1/5.0);

      impact_time_global = 2.9432*alpha_m/v_not;

      t_const = PI/num_intervals_global;

      f_const = ((1.140*v_not*v_not) / (k_one*alpha_m));

   }

   return(force_magn);

}



  /******************  End of compute_impact_force()  **************
****/
```

Figure 4 (contd.)

```
/*********************************************************************
 *                                                                   *
 * This file   : parts.c                                             *
 * Other files: collide.c (main)                                     *
 *              updates.c                                            *
 *              display.c                                            *
 *              globals.h                                            *
 *                                                                   *
 * This file contains functions create_particles(),                  *
 * collision_detect(), and distance().                               *
 *                                                                   *
 *********************************************************************/



#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include "globals.h"


  /*****************************************************************
   *                                                               *
   * Name      : create_particles()                                *
   * Input     : Pointer to a structure containing info about the  *
   *             object under consideration                        *
   * Output    : None                                              *
   * Function : Create all particles for the given object by       *
   *             assigning the position and other attributes to each*
   *             particle                                          *
   *             Create a linked list of all peripheral particles in*
   *             every box                                         *
   *                                                               *
   *****************************************************************/


void  create_particles(obj)
  obj_s  *obj;
{

   int       row, col, plane;     /* of the particle */
```

Figure 4 (contd.)

```
   int       brow, bcol, bplane;  /* of the box to which particle bel
ongs */
   int       times_visited;       /* how many times scan line entered
 circle */
   float     x, y, z;             /* of a grid point - a prospective
particle */

   int       obj_srno;
   long      grid_points;
   float     obj_centre_x, obj_centre_y, obj_centre_z, obj_radius;
   float     scene_lowex_x, scene_uppex_y, scene_uppex_z;

   float     xmin, ymin, zmin, xmax, ymax, zmax;
   float     delta_x, delta_y, delta_z;
   float     value;
   float     boxsize_x, boxsize_y, boxsize_z;
   part_s    *blockptr;
   part_s    *latest;
   rowptr_s  *rowptr;


      /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^
        ^ NOTE : 3-d version needs modification from this point to th
e end ^
        ^        of this function.
      ^
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^*/


     /* with start, middle and end-points included, obj->grid +1 poin
ts (or
        grid nodes) for each of the three dimensions */
   grid_points = (obj->grid +1)*(obj->grid +1)*(ONE);


     /* malloc space to store the particle info */
   blockptr = (part_s *)malloc( sizeof(part_s) * grid_points);
   if (blockptr == (part_s *)NULL)
      fprintf(stderr,"\nCouldn't malloc space for particle info: prog
ram terminated.\n"), exit(1);


   grid_points = (obj->grid +1)*(ONE);
     /* malloc space to store the addresses of every row in every pla
ne */
   rowptr = (rowptr_s *)malloc(sizeof(rowptr_s)*grid_points);
   if (rowptr == (rowptr_s *)NULL)
```

Figure 4 (contd.)

```
      fprintf(stderr,"\nCouldn't malloc space for row-address info: p
rogram terminated.\n"), exit(1);


   obj->start  = blockptr;
   obj->rowptr = rowptr;


     /* size of the box : scene is divided into (BOXNUMxBOXNUMxBOXNUM
) voxels
        Note : this is reciprocal of actual boxsize (to speed up the
loop) */
   boxsize_x = BOXNUM/(scene.uppex.x - scene.lowex.x);
   boxsize_y = BOXNUM/(scene.uppex.y - scene.lowex.y);
   boxsize_z = BOXNUM/(scene.uppex.z - scene.lowex.z);


     /* reduce access time ! */
   scene_lowex_x = scene.lowex.x;
   scene_uppex_y = scene.uppex.y;
   obj_centre_x = obj->centre.x;
   obj_centre_y = obj->centre.y;
   obj_centre_z = obj->centre.z;
   obj_radius = obj->radius;
   obj_srno = obj->srno;


     /* determine extents of the circle */
   xmin = obj_centre_x - obj_radius;
   ymin = obj_centre_y - obj_radius;
   zmin = obj_centre_z - obj_radius;
   xmax = obj_centre_x + obj_radius;
   ymax = obj_centre_y + obj_radius;
   zmax = obj_centre_z + obj_radius;


   delta_x = (xmax - xmin)/(obj->grid);
   delta_y = (ymax - ymin)/(obj->grid);
   delta_z = (zmax - zmin)/(obj->grid);

   obj->pcount = 0;


   for (row=0;row<=obj->grid;row++) {    /* to include end point, row<
=    */

      y = ymax - (row * delta_y);

        /* expecting first intersection : a peripheral particle */
      times_visited = 0;
```

Figure 4 (contd.)

```
      for (col=0;col<=obj->grid;col++) {   /* to include end point, c
ol<=  */
         x = xmin + (col * delta_x);

         value  = (x - obj_centre_x)*(x - obj_centre_x);
         value += (y - obj_centre_y)*(y - obj_centre_y);
         value -= (obj_radius)*(obj_radius);

           /* if point is on or inside the circle */
         if (value < EPSILON) {

            times_visited++;

            blockptr->id[0] = row;
            blockptr->id[1] = col;
            blockptr->id[2] = ZERO;     /* for the 2-D version, id[2]=
ZERO  */

            blockptr->posn.x = x;
            blockptr->posn.y = y;
              /* initially, all particles have the same velocity as o
bject */
            blockptr->vel = obj->vel;

              /* initially, all particles have zero acceleration */
            blockptr->accn.x = blockptr->accn.y = blockptr->accn.z =
0.0;

              /* box no. in which the particle lies. Box (0,0) is lef
t top */
            brow   = blockptr->boxno[0] = (int)((x - scene_lowex_x)*b
oxsize_x);
            bcol   = blockptr->boxno[1] = (int)((scene_uppex_y - y)*b
oxsize_y);
            bplane = blockptr->boxno[2] = ZERO;

              /* first intersection ? if yes, peripheral particle */
            if (times_visited == 1) {

                /* all row addresses to be stored in a 1-D array.  1
 plane has
                   obj->grid+1 rows in it.
                   i'th row in j'th plane => (obj->grid+1)*j + i
 */
               rowptr[ ((obj->grid+1)*ZERO) + row ].next = blockptr;
               rowptr[ ((obj->grid+1)*ZERO) + row ].fst_col = col;

               blockptr->status = PERIPHERAL;
```

Figure 4 (contd.)

```
                /* add to the linked list of peripheral particles  *
/

                /* First, find out the latest peripheral particle */
              if ((latest = box[brow][bcol][ZERO].start[obj_srno]) !
= (part_s *)NULL) {
                while (latest->next != (part_s *)NULL)
                   latest = latest->next;

                  /* now, add the new peripheral particle to the li
st */
                latest->next = blockptr;

              }

                /* else : for this box-and-object, this is the first
                   peripheral particle  */
              else
                box[brow][bcol][ZERO].start[obj_srno] = blockptr;

            }

              /* else : this is not a peripheral particle */
            else
               blockptr->status = NON_PERIPHERAL;

              /* peripheral or not, null the link for now */
            blockptr->next = (part_s *)NULL;

            (obj->pcount)++;    /* no. of particles in the object */

            blockptr++;         /* ready for the next particle */

         }

      }


        /* last one in this row (unless this row is the first one or
the last
           one among the scan rows) was a peripheral particle */
      if (times_visited > 1) {

         (blockptr-1)->status = PERIPHERAL;

          /* create or extend the linked list of peripheral particle
s :
             Note : brow,bcol are from previous iteration, but still
 valid */
```

Figure 4 (contd.)

```
          latest = box[brow][bcol][ZERO].start[obj_srno];

          if (latest == (part_s *)NULL)     /* first one : create the l
ist */
             box[brow][bcol][ZERO].start[obj_srno] = blockptr-1;

          else {    /* list already exists : extend it  */

               /* First, find out the latest peripheral particle */
             while (latest->next != (part_s *)NULL)
                latest = latest->next;

               /* now, add the new peripheral particle to the list */
             latest->next = blockptr-1;
          }

       }

         /* store last column in this row */
       rowptr[ ((obj->grid+1)*ZERO) + row ].last_col = (blockptr-1)->i
d[1];

    }


    /* mass of each of the particles */
    obj->part_mass = (obj->mass)/(obj->pcount);


      /* although grid_points blocks were malloced, not all are used.
         free the unused blocks.  */
    free((char *)blockptr);


    return;


}




  /********************  End of create_particles()  ****************
***/
```

Figure 4 (contd.)

```
/*****************************************************************
 *                                                               *
 * Name      : collision_detect()                                *
 * Input     : None                                              *
 * Output    : Pointer to a linked list containing collision info *
 *             Null pointer if no collision is detected          *
 * Function : Check if any two objects in the scene have collided *
 *                                                               *
 *****************************************************************
/


coll_s *collision_detect()
{

   int        i, j, k, m, n, d;
   int        first_time=TRUE;
   coll_s     *ret_ptr=(coll_s *)NULL;  /* return pointer */
   coll_s     *current, *newblock;
   part_s     *ppfst, *ppsec;     /* the two (potentially) colliding p
articles */
   double     distance();


     /* These 7 loops ! : The outermost 3 for loops are to cover all
the boxes.
        The following for and while loop are to take care of an objec
t and all
        peri. part.s in that object (in a particular box) resp.  The
next for
        and while loop do the same for another object with which coll
ision is
        checked for.
        */


   for (i=0;i<BOXNUM;i++) {

     for (j=0;j<BOXNUM;j++) {


     /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^
       ^  NOTE : Change the ONE to BOXNUM for the 3-d case.
^
```

Figure 4 (contd.)

```
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^*/
      for (k=0;k<ONE;k++) {

        for (m=0;m<scene.nobjects-1;m++) {

            /* addr. of 1st peri. part. of object m in box[i][j][k]
 */
          ppfst = box[i][j][k].start[m];

          while (ppfst != (part_s *)NULL) {

             for (n=m+1;n<scene.nobjects;n++) {


               /* addr. of 1st peri. part. of object n in box[i][
j][k] */
              ppsec = box[i][j][k].start[n];

              while (ppsec != (part_s *)NULL) {

                 if (distance(ppfst->posn,ppsec->posn) < DNOT) {
                    newblock = (coll_s *)malloc(sizeof(coll_s));
                    if (newblock == (coll_s *)NULL)
                       fprintf(stderr,"\n%s\n","Couldn't malloc s
pace for\
                        storing collision info: program terminate
d."), exit(1);
                    if (first_time) {
                       first_time = FALSE;
                       ret_ptr = current = newblock;
                    }
                    else {  /* another collision : extend the lin
ked list */
                       current->next = newblock;
                       current = current->next;
                    }

                     /* fill in the collision info */
                    current->srno_fst = m;  /* srno of first obje
ct */
                    current->srno_sec = n;  /* srno of second obj
ect */
                    current->ppfst = ppfst;  /* ptr to particle i
nfo */
                    current->ppsec = ppsec;  /* ptr to particle i
nfo */

                    current->box_id[0] = i;
                    current->box_id[1] = j;
```

Figure 4 (contd.)

```
                        current->box_id[2] = k;
                          /* no more collisions : stop the linked lis
t */
                        current->next = (coll_s *)NULL;
                      }
                    ppsec = ppsec->next;  /* try next peri. part. */
                  }
                }
               ppfst = ppfst->next; /* try next peri.part. */
             }
           }
         }
       }
   }
   return(ret_ptr);
}


  /********************  End of collision_detect()  ******************
***/




  /*****************************************************************
***
   *
  *
   * Name      : distance()
  *
```

Figure 4 (contd.)

```
   * Input     : The two points in question(structs, with x y z member
s)*
   * Output    : The distance
   *
   * Function : Compute the distance between the two given points
   *
   *
   *
   ************************************************************************
***/


double distance(p_one,p_two)
  point  p_one, p_two;
{

   double     result;
   double     pow();

   result =    pow((double)(p_one.x - p_two.x),2.0)
             + pow((double)(p_one.y - p_two.y),2.0)
             + pow((double)(p_one.z - p_two.z),2.0);

   result = pow(result,0.5);

   return(result);

}



  /********************  End of distance()  ********************/
```

Figure 4 (contd.)

```
/*********************************************************************
 *                                                                   *
 * This file  : updates.c                                            *
 * Other files: collide.c (main)                                     *
 *              parts.c                                              *
 *              display.c                                            *
 *              globals.h                                            *
 *                                                                   *
 * This file contains functions before_collision_update(),           *
 * during_collision_update(), internal_acceleration(), get_          *
 * neighbour(), external_acceleration(), and compute_time_step().    *
 *                                                                   *
 *********************************************************************/



#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include "globals.h"


/*********************************************************************
 *                                                                   *
 * Name      : before_collision_update()                             *
 * Input     : Struct containing info about the object in question   *
 *             Time step to be used for updating                     *
 *             Mode, whether change is to be permanent or temporary  *
 * Output    : None                                                  *
 * Function  : (This function is invoked when there is no collision) *
 *             Update positions of all particles                     *
 *             Recreate linked list of peri. particles in each box   *
 *             If mode is                                            *
 *               PERMANENT, above changes are permanent              *
 *               FORWARD_TEMP, above changes are temporary           *
 *               BACKWARD_TEMP, above changes are temporary and      *
 *                  nullify changes due to FORWARD_TEMP              *
 *                                                                   *
 *********************************************************************/


void before_collision_update(object,delta_time,mode)
```

Figure 4 (contd.)

```
   obj_s   object;
   double  delta_time;
   int     mode;          /* all particles to be updated (permanently),
only

                             peripheral forward or peripheral backward (
temp) */
{

   int        i, sign;
   int        brow, bcol, bplane;  /* of the box to which particle bel
ongs */
   float      disp_x, disp_y, disp_z;
   float      boxsize_x, boxsize_y, boxsize_z;
   float      scene_lowex_x, scene_uppex_y, scene_uppex_z;
   float      x, y, z;
   part_s     *blockptr, *latest;
   void       null_pointers();


   blockptr = object.start;

   sign = (mode == PERMANENT || mode == FORWARD_TEMP) ? 1: -1;

     /* all particles still have the same vel => same displacement */
   disp_x = (object.vel.x) * delta_time * sign;
   disp_y = (object.vel.y) * delta_time * sign;
   disp_z = (object.vel.z) * delta_time * sign;


     /* size of the box : scene is divided into (BOXNUMxBOXNUMxBOXNUM
) voxels
        Note : this is reciprocal of actual boxsize (to speed up the
loop) */
   boxsize_x = BOXNUM/(scene.uppex.x - scene.lowex.x);
   boxsize_y = BOXNUM/(scene.uppex.y - scene.lowex.y);
   boxsize_z = BOXNUM/(scene.uppex.z - scene.lowex.z);
   scene_lowex_x = scene.lowex.x;
   scene_uppex_y = scene.uppex.y;
   scene_uppex_z = scene.uppex.z;


     /* NULL all the box[][][].start[] : linked list of PERIPHERAL in
 a box */
   null_pointers(object.srno);


   for (i=0; i<object.pcount; i++,blockptr++) {

       /* in case of temporary updating, skip all non-peripheral par
ticles */
```

Figure 4 (contd.)

```
      if ((mode != PERMANENT) && (blockptr->status != PERIPHERAL))
         continue;


      x = blockptr->posn.x += disp_x;
      y = blockptr->posn.y += disp_y;
      z = blockptr->posn.z += disp_z;

        /* update box number of the particle */
      brow   = blockptr->boxno[0] = (int)((x - scene_lowex_x)*boxsize
_x);
      bcol   = blockptr->boxno[1] = (int)((scene_uppex_y - y)*boxsize
_y);
      bplane = blockptr->boxno[2] = ZERO;

        /* if PERIPHERAL, add/create the linked list of peripheral pa
rticles  */
      if (blockptr->status == PERIPHERAL) {

        /* First, find out the latest peripheral particle */
         if ((latest = box[brow][bcol][ZERO].start[object.srno]) != (
part_s *)NULL) {
            while (latest->next != (part_s *)NULL)
               latest = latest->next;

              /* now, add the new peripheral particle to the list */
            latest->next = blockptr;

         }

           /* else : for this box-and-object, this is the first
                     peripheral particle  */
         else
            box[brow][bcol][ZERO].start[object.srno] = blockptr;

           /* null the link for now */
         blockptr->next = (part_s *)NULL;

      }

   }


}


  /*********************  End of before_collision_update()  ********
```

Figure 4 (contd.)

```
*******/



/*********************************************************************
 *                                                                   *
 * Name      : during_collision_update()                             *
 * Input     : File descriptor for the display device                *
 *             The two objects involved in collision (their structs)*
 *             Linked list having info about particles in collision *
 * Output    : None                                                  *
 * Function  : (This function is invoked when collision has occured)*
 *             Update positions of all particles                     *
 *             Calculate the velocities and accn. of each particle   *
 *             Repeat these two steps for num_intervals_global times*
 *                                                                   *
 *********************************************************************/


void during_collision_update(fildes,object,collptr)
  int     fildes;              /* file descriptor for display device */
  obj_s   object[];            /* the two objects -info structure */
  coll_s  *collptr;            /* pointer to collision info structure *
/
{


   int         i, interval, up_lim[2], side;
   int         ccount, sr;
   char        ans[5];
   double      tv;
   float       disp_x[2], disp_y[2], disp_z[2];
   point       temp_accn;
   part_s      *blockptr, *nebrptr;
   coll_s      *save_coll;
   point       external_acceleration(), internal_acceleration();
   part_s      *get_neighbour();
   void        display_picture();



   save_coll = collptr;

   ccount = 0;
   while (collptr != (coll_s *)NULL) {
      ccount++;                        /* no. of particle-pairs in colli
```

Figure 4 (contd.)

```
sion */
      collptr = collptr->next;
   }

   collptr = save_coll;        /* restore it !! */


     /* store collision info. for both the objects (ppfst and ppsec)
*/

   while (collptr != (coll_s *)NULL) {
      collptr->ppfst->status = PERI_FECTED;   /* PERIpheral and afFEC
TED */
      collptr->ppfst->accn = external_acceleration(object[0],collptr,
ccount,ONE);
      collptr = collptr->next;
   }

   collptr = save_coll;        /* restore it !! */

   while (collptr != (coll_s *)NULL) {
      collptr->ppsec->status = PERI_FECTED;
      collptr->ppsec->accn = external_acceleration(object[1],collptr,
ccount,ONE);
      collptr = collptr->next;
   }


     /* Divide the impact time in num_intervals_global intervals.  Up
date
        position of every particle.  Then display. Then mark the part
icles
        which are neighbours of the AFFECTED particles.  Then find ac
celeration
        for all the AFFECTED particles.  Back into the loop to update
 positions.  */

     /* tv used repeatedly in the loop  */
   tv = impact_time_global/num_intervals_global;


     /* if particle is not AFFECTED, its velocity is going to be the
same as
     that of the entire object                */
    for (sr=0;sr<2;sr++) {
       disp_x[sr] = object[sr].vel.x * tv;
       disp_y[sr] = object[sr].vel.y * tv;
       disp_z[sr] = object[sr].vel.z * tv;
    }
```

Figure 4 (contd.)

```
   /* max.no. of loops needed for all particles to get AFFECTED */
      /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        ^ NOTE: Change  '2 * ....' to  '3 * ....'        for 3-D   ^
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
  for (sr=0;sr<2;sr++) {
     up_lim[sr] = 2 * object[sr].grid;                           /* for
2-D */
  }


  for (interval=1; interval<=num_intervals_global; interval++) {

    for (sr=0;sr<2;sr++) {


        /* Update all the positions */
      for (i=0,blockptr=object[sr].start; i<object[sr].pcount; i++,b
lockptr++) {
    if (blockptr->status == AFFECTED || blockptr->status == PERI_FECTED
) {

         /* update the position using Newton's law: s = ut + (at^2)/2
*/
       blockptr->posn.x += (blockptr->vel.x)*tv + (blockptr->accn.x)*tv
*tv/2;
       blockptr->posn.y += (blockptr->vel.y)*tv + (blockptr->accn.y)*tv
*tv/2;
       blockptr->posn.z += (blockptr->vel.z)*tv + (blockptr->accn.z)*tv
*tv/2;

         /* now, update the velocities using  v = u + at  */
       blockptr->vel.x += (blockptr->accn.x)*tv;
       blockptr->vel.y += (blockptr->accn.y)*tv;
       blockptr->vel.z += (blockptr->accn.z)*tv;



    }

    else {        /* particle is not AFFECTED: no acceleration yet */

       blockptr->posn.x += disp_x[sr];
       blockptr->posn.y += disp_y[sr];
       blockptr->posn.z += disp_z[sr];

    }

      }
```

Figure 4 (contd.)

```
          /* Mark the particles which are neighbours of AFFECTED parti
cle
             To avoid newly marked AFFECTED particles as originally AF
FECTED
             particles, first mark all the neighbours as AFFECTED_TEMP
   */

          /* after 'up_lim' loops, all particles are AFFECTED: skip fo
r loop */
        if (interval > up_lim[sr])
           ;

        else {    /* some particles may not have been AFFECTED yet */

           for (i=0,blockptr=object[sr].start; i<object[sr].pcount; i+
+,blockptr++) {

         if (blockptr->status == AFFECTED || blockptr->status == PERI_F
ECTED)

                for (side=RIGHT; side<=OUTWARD; side++) {

                   nebrptr = get_neighbour(object[sr],blockptr,side);
                   if (nebrptr == (part_s *)NULL)
                      ;
                   else
                      if ( (nebrptr->status != PERI_FECTED) &&
                           (nebrptr->status != AFFECTED) )
                         nebrptr->status = AFFECTED_TEMP;

                }

          }

        }

          /* mark all the temporarily affected (AFFECTED_TEMP) particl
es as
             AFFECTED
       */
        if (interval <= up_lim[sr]) {

           for (i=0,blockptr=object[sr].start; i<object[sr].pcount; i+
+,blockptr++)
              if (blockptr->status == AFFECTED_TEMP)
                 blockptr->status = AFFECTED;
        }
```

Figure 4 (contd.)

```
        /* Find acceleration of all the AFFECTED particles */

        /* First, external acceleration of all PERI_FECTED particles
 */
      collptr = save_coll;       /* restore it !! */

      if (collptr->srno_fst == object[sr].srno) {
         while (collptr != (coll_s *)NULL) {
           collptr->ppfst->accn = external_acceleration(object[sr],c
ollptr,ccount,interval);
           collptr = collptr->next;
        }
      }

      else {
        while (collptr != (coll_s *)NULL) {
       collptr->ppsec->accn = external_acceleration(object[sr],collp
tr,ccount,interval);
       collptr = collptr->next;
        }
      }

        /* now, internal accn. of all AFFECTED and PERI_FECTED parti
cles */
      for (i=0,blockptr=object[sr].start; i<object[sr].pcount; i++,b
lockptr++) {

    if (blockptr->status == AFFECTED)
       blockptr->accn = internal_acceleration(object[sr],blockptr);


      /* PERI_FECTED particles already have an accn. (from impact) */
    if (blockptr->status == PERI_FECTED) {

       temp_accn = internal_acceleration(object[sr],blockptr);

       blockptr->accn.x += temp_accn.x;
       blockptr->accn.y += temp_accn.y;
       blockptr->accn.z += temp_accn.z;

    }

      }


  }
```

Figure 4 (contd.)

```
      /* now, display both the objects (sr = 0 => clear screen) */
    for (sr=0;sr<2;sr++)
       display_picture(fildes,object[sr],sr);


  }

}



/*****************  End of during_collision_update()  ***************
****/
```

```
/*********************************************************************
 *                                                                   *
 * Name      : internal_accleration()                                *
 * Input     : Struct for the object in which this particle lies     *
 *             Pointer to particle info of this object               *
 * Output    : Structure containing x y z components of acceleration*
 * Function : Find net acceleration experienced by a particle due    *
 *             to its neighbours                                     *
 *                                                                   *
 *********************************************************************/


point internal_acceleration(object,blockptr)
  obj_s   object;
  part_s  *blockptr;
{

   int       side;
   float     k_val;                  /* spring constant */
   float     x_dc, y_dc, z_dc;       /* direction cosines */
   double    dist, delta_dist;
   double    min_dist;
   float     accn_magn;              /* acceleration magnitude */
```

Figure 4 (contd.)

```
   point    accn;                  /* acceleration vector */
   part_s   *nebrptr;              /* neighbouring-particle pointer */
   double   distance();            /* see file containing create_parti
cles() */
   part_s   *get_neighbour();


   accn.x = accn.y = accn.z = 0.0;

     /* used in the loop to avoid a round-off error */
   min_dist = object.dnot/1000;

   for (side=RIGHT; side<=OUTWARD; side++) {

        /* check if a neighbour exists on the side 'side' */
      if ((nebrptr = get_neighbour(object,blockptr,side)) == (part_s
*)NULL)
         ;

        /* neighbour on this side does exist */
      else {

          /* get distance between the particle and its neighbour */
        dist = distance(blockptr->posn,nebrptr->posn);

          /* dist>object.dnot => the spring is in extension: select
the right
             spring constant, and get the direction cosines
             (signs important !) */
        if (dist > object.dnot) {
           k_val = object.k_extn;
           x_dc = (nebrptr->posn.x - blockptr->posn.x)/dist;
           y_dc = (nebrptr->posn.y - blockptr->posn.y)/dist;
           z_dc = (nebrptr->posn.z - blockptr->posn.z)/dist;
        }

          /* dist<object.dnot => the spring is in compression: selec
t the
             right spring constant, and get the direction cosines
             (signs important !) */
        else {
           k_val = object.k_comp;
           x_dc = (blockptr->posn.x - nebrptr->posn.x)/dist;
           y_dc = (blockptr->posn.y - nebrptr->posn.y)/dist;
           z_dc = (blockptr->posn.z - nebrptr->posn.z)/dist;
        }

          /* change in spring length */
        delta_dist = fabs(dist - (double)object.dnot);
```

Figure 4 (contd.)

```
            /* instead of 'delta_dist == 0.0': k_val is so high, it ma
kes
               a difference even if 0.0 is represented is 0.0000001 !
  */
          if (delta_dist < min_dist)
             ;
          else {
               /* magnitude of the acceleration */
             accn_magn = (k_val * delta_dist)/object.part_mass;

               /* add the acceleration components */
             accn.x += (accn_magn * x_dc);
             accn.y += (accn_magn * y_dc);
             accn.z += (accn_magn * z_dc);


          }


       }

    }

    return(accn);

}




  /*********************  End of internal_acceleration()  **********
*****/









/*********************************************************************
 *                                                                   *
 * Name      : get_neighbour()                                       *
```

Figure 4 (contd.)

```
 * Input     : Structure containing info about the object in question*
 *             Pointer to particle info of this object                *
 *             Side for which neighbour is desired                    *
 * Output    : Pointer to structure on particle which is the desired *
 *             neighbour                                              *
 * Function : Find the neighbour for the particle in question on     *
 *             the side desired                                       *
 *                                                                    *
 **********************************************************************
/


part_s *get_neighbour(object,blockptr,side)
  obj_s   object;       /* object structure */
  part_s  *blockptr;    /* particle in question */
  int     side;         /* which side neighbour: RIGHT, LEFT, etc. */
{


   int         indx;
   rowptr_s    rowptr;



      /* first row and last row require special attention: there is o
nly one
         particle each in these rows, and blockptr-1 & blockptr+1 (re
spectively)
         for these cases would not be meaningful
   */

     /* for first (0'th) row, RIGHT, LEFT or TOP neighbour can't exis
t */
   if ((blockptr->id[0] == 0) &&
       (side == RIGHT || side == LEFT || side == TOP))
      return((part_s *)NULL);

     /* for last (grid+1'th) row, RIGHT, LEFT or BOTTOM neighbour can
't exist */
   if ((blockptr->id[0] == object.grid) &&
       (side == RIGHT  | side == LEFT || side == BOTTOM))
      return((part_s *)NULL);



     /* now, rest of the particles */

   switch (side) {

                  /* if a particle has a neighbour to its RIGHT, row no. of
```

Figure 4 (contd.)

```
                        the next particle (id[0]) should be the same  */
     case RIGHT    :  if ((blockptr+1)->id[0] == blockptr->id[0])
                        return(blockptr+1);
                    return((part_s *)NULL);

                     /* if a particle has a neighbour to its LEFT, row no. of
                        the previous particle (id[0]) should be the same  */
     case LEFT     :  if ((blockptr-1)->id[0] == blockptr->id[0])
                        return(blockptr-1);
                    return((part_s *)NULL);

                        /* this particle is on plane id[2], row id[0]:
  every
                           plane contains grid+1 rows:  address of thi
s row is
                           therefore indx'th element in object.rowptr[
] array
                           TOP => previous row => indx-1                *
/
     case TOP      :  indx = (object.grid+1)*(blockptr->id[2]) + block
ptr->id[0];
                      rowptr = object.rowptr[indx-1];

                        /* check if indeed there is a particle vertica
lly above
                           this one !
    */
                      if ( (blockptr->id[1] < rowptr.fst_col) ||
                           (blockptr->id[1] > rowptr.last_col) )
                     return((part_s *)NULL);

                        /* now, col is id[1]: add the difference in ad
dresses */
                      return(rowptr.next + (blockptr->id[1] - rowptr.f
st_col));

                        /* (see comment for TOP) BOTTOM => next row =>
 indx+1 */
     case BOTTOM   :  indx = (object.grid+1)*(blockptr->id[2]) + block
ptr->id[0];
                      rowptr = object.rowptr[indx+1];

                        /* check if indeed there is a particle vertica
lly below
                           this one !
    */
                      if ( (blockptr->id[1] < rowptr.fst_col) ||
                           (blockptr->id[1] > rowptr.last_col) )
                     return((part_s *)NULL);
```

Figure 4 (contd.)

```
                        /* now, col is id[1]: add the difference in ad
dresses */
                       return(rowptr.next + (blockptr->id[1] - rowptr.f
st_col));

      /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^
        ^ NOTE: Cases INWARD and OUTWARD need to be modified for 3-D
  ^
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^*/
     case INWARD  :  break;

     case OUTWARD :  break;

   }

   return ((part_s *)NULL);

}


  /*********************  End of get_neighbour()  ********************
***/




/*********************************************************************
 *                                                                   *
 * Name      : external_acceleration()                               *
 * Input     : Structure containing info about the object in question*
 *             Pointer to linked list of particles in collision      *
 *             Number of pairs of particles involved in collision    *
 *             Interval (max. = num_intervals_global)                *
 * Output    : Structure containing x y z components of acceleration *
 * Function  : Compute acceleration on each particle involved in     *
```

Figure 4 (contd.)

```
 *            collision due to the impact                          *
 *                                                                 *
 *******************************************************************
/


point external_acceleration(object,collptr,ccount,interval)
  obj_s   object;          /* one of the two objects in collision */
  coll_s  *collptr;        /* ptr to linked list of collision info */
  int     ccount;          /* no. of particles in collision of a give
n object */
  int     interval;        /* how many'th interval is this */
{

   double   accn_magn;     /* magnitude of accn. per particle per int
erval) */
   double   dist;          /* distance between the two particles in c
ollision */
   float    x_dc, y_dc;    /* x, y, z direction cosines */
   float    z_dc;
   point    accn;          /* acceleration vector (of every particle)
 */
   obj_s    dummy;         /* no input needed, but function expects i
t ! */
   double   distance();
   double   compute_impact_force();


     /* assumption: net acceleration (magnitude) in a given interval
is equally
        divided into all the particles involved in collision.      */
   accn_magn = ((compute_impact_force(dummy,dummy,interval)/object.pa
rt_mass)/ccount);

   dist = distance(collptr->ppfst->posn,collptr->ppsec->posn);


     /* direction of accn. depends on which object is being considere
d */

   if (collptr->srno_fst == object.srno) {
      x_dc = (collptr->ppfst->posn.x - collptr->ppsec->posn.x)/dist;
      y_dc = (collptr->ppfst->posn.y - collptr->ppsec->posn.y)/dist;
      z_dc = (collptr->ppfst->posn.z - collptr->ppsec->posn.z)/dist;
   }
   else {
      x_dc = (collptr->ppsec->posn.x - collptr->ppfst->posn.x)/dist;
      y_dc = (collptr->ppsec->posn.y - collptr->ppfst->posn.y)/dist;
      z_dc = (collptr->ppsec->posn.z - collptr->ppfst->posn.z)/dist;
   }
```

Figure 4 (contd.)

```
    /* now, the components */
    accn.x = (accn_magn * x_dc);
    accn.y = (accn_magn * y_dc);
    accn.z = (accn_magn * z_dc);


    return(accn);

}



  /******************** End of external_acceleration()  ************
******/
```

```
/*****************************************************************
 *                                                               *
 * Name      : compute_time_step()                               *
 * Input     : Struct containing info about the object in question  *
 *             Time step used as a 'seed'                        *
 * Output    : Time step to be used by before_collision_update   *
 * Function : Compute a time step small enough that two objects  *
 *             will not go into each other                       *
 *                                                               *
 *****************************************************************/


double compute_time_step(object,delta_time)
  obj_s   object[];
  double  delta_time;
{

    int        i, j, k, m, n, d, count=0;
    int        found_pair[2];              /* a pair found in a box */
    double     drx[2], dry[2], drz[2];     /* direction ratios in x, y
and z */
    double     dist, smallest[2];
    coll_s     *clostpr;             /* closest pair (of peripheral part
```

Figure 4 (contd.)

```
icles) */
   part_s    *ppfst, *ppsec;     /* the two (potentially) colliding p
articles */
   double    distance(), fabs();
   void      before_collision_update();


     /* just two-pairs-of-particles info needs to be stored */
   if((clostpr = (coll_s *)malloc(2*sizeof(coll_s))) == (coll_s *)NUL
L)
      fprintf(stderr,"\nCouldn't malloc: program terminated\n"), exit
(1);


     /* measure the smallest distance between any of the pairs of per
ipheral
        particles, temporarily update only the peripheral particles a
nd measure
        the distance again.  (the two pairs of particles - before the
 temporary
        update and after - don't have to be the same).  check the dir
ection
        ratios to make sure objects have not gone into each other.
     */

   while (count < 2) {      /* count=0 => before temp.update, 1 => aft
er */

      smallest[count] = INFINITY;
      found_pair[count] = FALSE;


        /* These 7 loops!: The outermost 3 for loops are to cover all
 the boxes.
           The following for and while loops are to take care of an o
bject and
           all peri. part.s in that object (in a particular box) resp
.  The next
           for and while loops do the same for another object with wh
ich
           collision is checked for.
      */



      for (i=0;i<BOXNUM;i++) {

        for (j=0;j<BOXNUM;j++) {
```

Figure 4 (contd.)

```
        /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^
          ^  NOTE : Change the ONE to BOXNUM for the 3-d case.
   ^
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^*/
          for (k=0;k<ONE;k++) {

            for (m=0;m<scene.nobjects-1;m++) {

                /* addr. of 1st peri. part. of object m in box[i][j][
k] */
              ppfst = box[i][j][k].start[m];

              while (ppfst != (part_s *)NULL) {

                for (n=m+1;n<scene.nobjects;n++) {

                    /* addr. of 1st peri. part. of object n in box[i]
[j][k] */
                  ppsec = box[i][j][k].start[n];

                  while (ppsec != (part_s *)NULL) {

                    if ((dist = distance(ppfst->posn,ppsec->posn)) <
smallest[count]) {
                      smallest[count] = dist;
                      found_pair[count] = TRUE;

                        /* fill in the 'closest pair' info */
                      clostpr->srno_fst = m;  /* srno of first object
 */
                      clostpr->srno_sec = n;  /* srno of second objec
t */
                      clostpr->ppfst = ppfst;  /* ptr to particle inf
o */
                      clostpr->ppsec = ppsec;  /* ptr to particle inf
o */

                      clostpr->box_id[0] = i;
                      clostpr->box_id[1] = j;
                      clostpr->box_id[2] = k;

                        /* not really necessary, but just in case */
                      clostpr->next = (coll_s *)NULL;

                    }

                    ppsec = ppsec->next;  /* try next peri. part. */
```

Figure 4 (contd.)

```
                    }
                  }
                  ppfst = ppfst->next; /* try next peri.part. */
                }
              }
            }
          }
        }
          /* temporary update needed only once ! */
        if (count == 0) {
          clostpr++;
            /* temporarily update all the peripheral particles to make
sure that
               in the next update the objects don't go into each other
               FORWARD_TEMP => advance: update particles
    */
          for (i=0;i<scene.nobjects;i++)
             before_collision_update(object[i],delta_time,FORWARD_TEMP)
;
        }
        count++;
    }

      /* if a pair of particles from different objects is not found in
any
         of the boxes both before temporary update and after, this
         delta_time is definitely ok: return it  */
    if ( !found_pair[0] && !found_pair[1] ) {
        for (i=0;i<scene.nobjects;i++)
           before_collision_update(object[i],delta_time,BACKWARD_TEMP);
        return(delta_time);
    }
```

Figure 4 (contd.)

```
     /* if a pair of particles is not found in a box before update, (
but
        is found after update) then compute direction ratios by calcu
lating
        positions of the closest particles one delta_time before
     */
   if ( !found_pair[0] ) {

      drx[0] =  (clostpr->ppfst->posn.x - clostpr->ppfst->vel.x * del
ta_time)
               -(clostpr->ppsec->posn.x - clostpr->ppsec->vel.x * del
ta_time);

      dry[0] =  (clostpr->ppfst->posn.y - clostpr->ppfst->vel.y * del
ta_time)
               -(clostpr->ppsec->posn.y - clostpr->ppsec->vel.y * del
ta_time);

      drz[0] =  (clostpr->ppfst->posn.z - clostpr->ppfst->vel.z * del
ta_time)
               -(clostpr->ppsec->posn.z - clostpr->ppsec->vel.z * del
ta_time);

   }


   else {

      clostpr--;

      drx[0] = clostpr->ppfst->posn.x - clostpr->ppsec->posn.x;
      dry[0] = clostpr->ppfst->posn.y - clostpr->ppsec->posn.y;
      drz[0] = clostpr->ppfst->posn.z - clostpr->ppsec->posn.z;

      clostpr++;

   }

     /* direction ratios after time-increment */

   drx[1] = clostpr->ppfst->posn.x - clostpr->ppsec->posn.x;
   dry[1] = clostpr->ppfst->posn.y - clostpr->ppsec->posn.y;
   drz[1] = clostpr->ppfst->posn.z - clostpr->ppsec->posn.z;


     /* if the pair of objects having closest distance before the tem
porary
```

Figure 4 (contd.)

```
        update is not the same as that after, lot more work is requir
ed to
        determine a "good" time-step.  for now, delta_time/10.0 serve
s as a
        good assumption(?) to take care of this very rare possibility
         (for this to happen, closest pair must be found before and a
fter) */
   if ( ( (clostpr->srno_fst != (clostpr-1)->srno_fst)||
          (clostpr->srno_sec != (clostpr-1)->srno_sec)  ) &&
        ( found_pair[0] && found_pair[1] )  ) {

      for (i=0;i<scene.nobjects;i++)
         before_collision_update(object[i],delta_time,BACKWARD_TEMP);
      return(delta_time/10.0);
   }


     /* as long as at least one of x, y or z direction ratios hasn't
changed
        sign, it's ok to have this delta_time: return it  */
   if ( !( ( (drx[0] >= 0.0 && drx[1] <  0.0) ||
             (drx[0] <  0.0 && drx[1] >= 0.0) ||
             (fabs(drx[0]) < 0.00001 && fabs(drx[1]) < 0.00001) ) &&
           ( (dry[0] >= 0.0 && dry[1] <  0.0) ||
             (dry[0] <  0.0 && dry[1] >= 0.0) ||
             (fabs(dry[0]) < 0.00001 && fabs(dry[1]) < 0.00001) ) ) )
 {
         /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^
           ^ NOTE: In case of 3-d, include the following condition
   ^
           ^
   ^
           ^ && ( (drz[0] >= 0.0 && drz[1] <  0.0) ||
   ^
           ^       (drz[0] <  0.0 && drz[1] >= 0.0) ||
   ^
           ^       (fabs(drz[0]) < 0.00001 && fabs(drz[1]) < 0.00001)
 ) ^
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^*/

      for (i=0;i<scene.nobjects;i++)
         before_collision_update(object[i],delta_time,BACKWARD_TEMP);
      return(delta_time);
   }

     /* all direction ratios have changed => objects pass through eac
h other.
```

Figure 4 (contd.)

```
        restore particle positions, halve delta_time, and
        lo! behold!  recursion!
  */
   else {

        /* restore all the peripheral particles to their original pos
itions
           FALSE => restore
   */
      for (i=0;i<scene.nobjects;i++)
         before_collision_update(object[i],delta_time,BACKWARD_TEMP);
      free(clostpr-1);
      return(compute_time_step(object,delta_time/2.0));
   }

}



  /*********************  End of compute_time_step()  **************
*******/
```

Figure 4 (contd.)

```
/*********************************************************************
 *                                                                   *
 * This file  : display.c                                            *
 * Other files: collide.c (main)                                     *
 *              parts.c                                              *
 *              updates.c                                            *
 *              globals.h                                            *
 *                                                                   *
 * This file contains functions display_initialize(), and            *
 * display_picture().                                                *
 *                                                                   *
 *********************************************************************/




#include <stdio.h>
#include <math.h>
#include <fcntl.h>
#include <starbase.c.h>
#include "globals.h"



/*********************************************************************
 *                                                                   *
 * Name     : display_initialize()                                   *
 * Input    : None                                                   *
 * Output   : None                                                   *
 * Function : Initialize the display device if program is to be      *
 *            run in real time                                       *
 *                                                                   *
 *********************************************************************/


int display_initialize()
{

   int           fildes;


   if (real_time_global) {

      printf("\33h\33J"), fflush(stdout);
```

Figure 4 (contd.)

```
      if ((fildes = gopen("/dev/crt",OUTDEV,"hp98721",INIT | THREE_D)
) == -1)
          fprintf(stderr,"\nCouldn't open output device\n"), exit(-1);

      view_window(fildes,-0.05*scene.uppex.x,-0.05*scene.uppex.y,
                         1.05*scene.uppex.x,1.05*scene.uppex.y);

      return(fildes);

   }

   else
      return(-1);


}



  /******************  End of display_initialize()  ***************
******/




/*********************************************************************
**
 *
 *
 * Name      : display_picture()
 *
 * Input     : File descriptor of the display device
 *
 *             Structure containing info about the object in question
 *
 *             Mode, to determine if screen should be cleared
 *
 * Output    : None
 *
 * Function : Display the object in question
 *
 *
 *
 *********************************************************************
**/
```

Figure 4 (contd.)

```
void display_picture(fildes,object,mode)
  int     fildes;
  obj_s   object;
  int     mode;
{

   register int     i;
   static int       frame=0, filenum=0;
   int              limit;
   static char      filename[20];
   char             comnd[100];
   float            x, y, z;
   float            delta;
   part_s           *blockptr;
   static int       fdes;
   static FILE      *fp;



   if (object.srno == 0)
      frame++;

     /* display only 50 frames */
   limit = num_intervals_global/50;
   if ( (collision_in_progress_global) && (frame%limit != 0) )
      return;

   if (real_time_global) {

      if (mode == 0)
         clear_view_surface(fildes);

      line_color(fildes,object.colour[0],object.colour[1],object.colo
ur[2]);

      blockptr = object.start;
      delta = DNOT/100;                    /* some small distance */

      for (i=0; i<object.pcount; i++,blockptr++) {

         move2d(fildes,blockptr->posn.x,blockptr->posn.y);

         x = blockptr->posn.x + delta;
         y = blockptr->posn.y + delta;

           /* check is already made to see that no particle falls
              outside the scene extents                          */
         draw2d(fildes,x,y);
```

Figure 4 (contd.)

```
      }
      make_picture_current(fildes);

   }


     /* not real time: data needs to be stored on file */
   else {

        /* open new file if the first (0 th) object is being dealt wi
th */
      if (object.srno == 0) {
         filenum++;
         sprintf(filename,"c_p.%d",filenum);

         if ((fp = fopen(filename,"r")) == (FILE *)NULL)
            fp = fopen(filename,"w");
         else
            fprintf(stderr,"%s exists",filename), exit(1);

      }

        /* since every frame is stored in different file, pcount is t
he only
           info needed to separate one object from the other  */
      fprintf(fp,"%ld\n",object.pcount);

      blockptr = object.start;

      for (i=0; i<object.pcount; i++,blockptr++)
         fprintf(fp,"%.3f %.3f\n",blockptr->posn.x,blockptr->posn.y);

      if (object.srno == scene.nobjects-1) {
         fclose(fp);
         sprintf(comnd,"compress %s",filename);
         system(comnd);
      }

   }

}


  /*********************  End of display_picture()  **************
********/
```

Figure 4 (contd.)

```
/*********************************************************************
 *                                                                   *
 * This file  : globals.h                                            *
 * Other files: collide.c (main)                                     *
 *              parts.c                                              *
 *              updates.c                                            *
 *              display.c                                            *
 *                                                                   *
 * This file contains defines, typedefs and variables common to all*
 * parts of the program.                                             *
 *                                                                   *
 *********************************************************************/



#ifndef  MULTI_DEFINE

#define  TRUE              1
#define  FALSE             0
#define  ZERO              0
#define  INFINITY          1.0e50
#define  ONE               1
#define  PI                3.14159265358979

  /* constants denoting status of a particle */
#define  PERIPHERAL        1
#define  NON_PERIPHERAL    2
#define  AFFECTED          3
#define  AFFECTED_TEMP     4
  /* PERIpheral and afFECTED */
#define  PERI_FECTED       5


  /* a max of 6 neighbours (in 3-D) possible on these sides
     CAUTION: don't alter these! (this order is used in for loops) */
#define  RIGHT             1
#define  LEFT              2
#define  TOP               3
#define  BOTTOM            4
#define  INWARD            5
#define  OUTWARD           6


  /* mode for updating: function before_collision_update() */
#define  PERMANENT         1
#define  FORWARD_TEMP      2
```

Figure 4 (contd.)

```
#define  BACKWARD_TEMP     3


  /* two particles (of different objects) are closer than this => col
lision */
#define  DNOT              0.1


  /* distance (tolerance) to decide particle in or out */
#define  EPSILON           0.01


  /* standard delta-time: as long as objects don't pass through each
other,
     this is used as the time-step */
#define  STD_DELTA_TIME    0.05


  /* recovery factor: used just after one collision is completely dea
lt with */
#define  RECVRY_FCTR       100


  /* maximum number of objects permitted in the scene */
#define  MAXOBJ            2


  /* 3-d scene is divided into (BOXNUM x BOXNUM x BOXNUM) voxels */
#define  BOXNUM            8



struct point_st {
  float  x;
  float  y;
  float  z;
};

typedef struct point_st point;


  /* structure containing information about a given particle */

struct part_st {
  int     id[3];     /* row, column, plane of particle : unique in an
object */
  point   posn;      /* position, changes with time */
  point   vel;       /* velocity, changes with time */
```

Figure 4 (contd.)

```
  point    accn;      /* acceleration, changes with time */
  char     status;    /* whether this particle is PERIPHERAL, AFFECTED
etc. */
  int      boxno[3]; /* row, col, plane of the box in which this parti
cle is */

  /* pointer to the next peripheral particle of this object in the
same box */
  struct part_st  *next;

};

typedef struct part_st part_s;


  /* structure containing starting address of every row of particles
and related
     info.: useful in finding the neighbours of a given particles  */

struct rowptr_st {
  part_s     *next;      /* pointer to the starting of a row of particl
es */
  int        fst_col;    /* first particle in this row starts at this c
olumn */
  int        last_col;   /* last particle in this row ends at this colu
mn */
};

typedef struct rowptr_st rowptr_s;


  /* structure containing information about an object
     NOTE : This structure needs modification in location details if
anything
            other than circle (or sphere) is used. */

struct obj_st {
  int      id;           /* e.g. circle = 1, rectangle = 2 etc. */
  int      srno;         /* how many'th object is this in the scene */
  point    centre;
  float    radius;
  float    colour[3];    /* r,g,b : of the object */
  int      grid;         /* size of cuboid (gridXgridXgrid) to create p
articles */
  double   dnot;         /* normal distance between two particles */
  float    k_comp;       /* value of spring constant in compression */
  float    k_extn;       /* value of spring constant in extension */
  point    vel;          /* initial velocity of the object */
```

Figure 4 (contd.)

```
  long    pcount;       /* no. of particles in this object */
  part_s  *start;       /* pointer to where particle info is stored */
  float   mass;         /* mass of the object */
  double  part_mass;    /* mass of one particle (object.mass/object.pc
ount) */
  float   mu;           /* Poisson's ratio for the material of this ob
ject */
  float   elast;        /* E, Modulus of elasticity for the material *
/

    /* 1-D array of pointers such that rowptr[(obj->grid)*j + i] poin
ts at the
       beginning of i'th row of particles in j'th plane.  (for the 3-
D case)  */
  rowptr_s   *rowptr;

};

typedef struct obj_st obj_s;



  /* structure containing information about the scene */

struct scn_st {
  int     nobjects;     /* no. of objects */
  point   lowex;        /* lower extent i.e. xmin, ymin, zmin */
  point   uppex;        /* upper extent i.e. xmax, ymax, zmax */
};

typedef struct scn_st scn_s;



  /* structure containing information about each box in the scene :
     pointer to the first peripheral particle of an object is stored
in
     start[i].  Remaining peripheral particles of that object are sto
red
     as a linked list using part_s.next.
  */

struct box_st {
  part_s  *start[MAXOBJ];   /* one for each of the objects */
};

typedef struct box_st box_s;
```

Figure 4 (contd.)

```
  /* structure containing info about collision once it is detected */

struct coll_st {
  int              srno_fst;    /* first object in collision */
  part_s           *ppfst;      /* which particle (of this object) in c
ollision */
  int              srno_sec;    /* id of the second object in collision
 */
  part_s           *ppsec;      /* which particle (of this object) in c
ollision */
  int              box_id[3];   /* box in which collision is detected *
/
  struct coll_st   *next;       /* if more than one pair of particles c
ollide */
};

typedef struct coll_st coll_s;



  /* scene info : stays the same throughout the program */
scn_s     scene;


  /* box info : common to the entire program, although changes with t
ime
      (dimensions for array box need to be BOXNUM+1 rather than BOXNUM
 for
       error-checking purposes)
     */
  /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^
    ^ NOTE : Change ONE to BOXNUM for the 3-d case
        ^
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^*/
box_s     box[BOXNUM+1][BOXNUM+1][ONE+1];



int       collision_in_progress_global;
double    impact_time_global;     /* when two bodies collide, the impac
t time */
long      num_intervals_global;   /* no. of steps to divide the impact
time */
int       real_time_global;       /* 1 (or 0)whether real-time simulati
on desired (or not) */


#endif
```

Figure 5
Input Sample - 1

```
$
0.0 0.0 0.0          /*  lower extents of the scene  */
16.0 16.0 16.0       /*  upper extents of the scene  */
2                    /*  number of objects           */
5000                 /*  no. of intervals            */
0                    /*  0 = on file, 1 = real-time  */
$
Object 1
6.6 5.2 0.0 1.5      /*  centre x y z, and radius    */
1.0 0.0 0.0          /*  colour components           */
125                  /*  grid size                   */
5.0e9 5.0e9          /*  spring stiffness            */
1.0  0.0 0.0         /*  velocity                    */
3.1e6                /*  mass                        */
0.28                 /*  Poisson's ratio             */
200.0e9              /*  Modulus of elasticity       */
$
Object 2
10.8 5.2 0.0 2.0     /*  centre x y z, and radius    */
0.0 1.0 0.0          /*  colour components           */
125                  /*  grid size                   */
6.0e9 6.0e9          /*  spring stiffness            */
-2.5 0.0 0.0         /*  velocity                    */
3.61e6               /*  mass                        */
0.28                 /*  Poisson's ratio             */
200.0e9              /*  Modulus of elasticity       */
```

Figure 6
Input Sample - 2

```
$
0.0 0.0 0.0          /*  lower extents of the scene  */
16.0 16.0 16.0       /*  upper extents of the scene  */
2                    /*  number of objects           */
5000                 /*  no. of intervals            */
0                    /*  0 = on file, 1 = real-time  */
$
Object 1
6.6 5.2 0.0 1.5      /*  centre x y z, and radius    */
1.0 0.0 0.0          /*  colour components           */
125                  /*  grid size                   */
5.0e8 5.0e8          /*  spring stiffness            */
4.0  1.0 0.0         /*  velocity                    */
3.1e6                /*  mass                        */
0.28                 /*  Poisson's ratio             */
200.0e9              /*  Modulus of elasticity       */
$
Object 2
10.8 8.2 0.0 2.0     /*  centre x y z, and radius    */
0.0 1.0 0.0          /*  colour components           */
125                  /*  grid size                   */
6.0e8 6.0e8          /*  spring stiffness            */
-2.5 0.0 0.0         /*  velocity                    */
3.61e6               /*  mass                        */
0.28                 /*  Poisson's ratio             */
200.0e9              /*  Modulus of elasticity       */
```

MONTANA STATE UNIVERSITY LIBRARIES
3 1762 10062089 5