Modeling the deformation of colliding objects
by Chance Reed Younkin

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
There have been a number of studies in the area of collisions recently but all approaches have been special purpose, or mathematically complex. We are looking for a general solution that will function in any collision environment. The model we use will allow arbitrary shapes, and will function in two or three dimensions. In the model, an object is considered as a collection of particles connected to one another by generalized springs. This approach greatly simplifies the mathematics in the simulation because it is just the force through a spring that determines the acceleration, velocity, and therefore the position of an affected particle, as well as its fracture properties. Newton's Laws are used to repeatedly update the positions of all the particles, and thus the form of the entire object. This system is an effective method for visualizing the behavior of colliding objects.

MODELING THE DEFORMATION OF COLLIDING OBJECTS

by

Chance Reed Younkin

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

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

July 1991

APPROVAL

of this thesis submitted by

Chance Reed Younkin

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.

July 24, 1991
Date

J. Dunbigh Stanley
Chairperson, Graduate Committee

Approved for the Major Department

July 24, 1991
Date

J. Dunbigh Stanley
Head, Major Department

Approved for the College of Graduate Studies

August 12, 1991
Date

Henry J. 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 Chance Reed Younken

Date 7-24-91

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

There have been a number of studies in the area of collisions recently but all approaches have been special purpose, or mathematically complex. We are looking for a general solution that will function in any collision environment. The model we use will allow arbitrary shapes, and will function in two or three dimensions. In the model, an object is considered as a collection of particles connected to one another by generalized springs. This approach greatly simplifies the mathematics in the simulation because it is just the force through a spring that determines the acceleration, velocity, and therefore the position of an affected particle, as well as its fracture properties. Newton's Laws are used to repeatedly update the positions of all the particles, and thus the form of the entire object. This system is an effective method for visualizing the behavior of colliding objects.

CHAPTER 1

INTRODUCTION

The phenomenon of objects colliding occurs everywhere in our universe. On a scale invisible to the naked eye, atoms collide to produce nuclear energy, and water molecules collide in a boiling pot to eventually produce steam. On a much larger scale, meteors, comets, planets, stars, and galaxies collide with one another to endlessly change our universe. In the middle of the spectrum billiard balls collide in a game of pool, and automobiles collide with just about anything. Since these phenomena affect every area of our lives, there is a need to learn more about what actually occurs during a collision.

A collision is nearly always followed by deformation of the objects involved. Though the deformation of a baseball as it comes off the bat will have no effect on the outcome of the game, there are many cases where deformations are of great interest. Vehicle accidents, protective ordnance and tool manufacturing are all applications where studying collisions and deformations would be extremely useful.

There have been a number of studies in the area of collisions recently, but many approaches have been special purpose solutions, for example, Hahn's method of modeling collisions of rigid bodies [4]. The problem here is that with rigid bodies, no deformation takes place. The altered velocities and positions of the bodies in the scene can be modeled effectively, but the realism of a deforming object is not present at all.

Physically based modeling has been used to simulate the deformation of colliding objects, but most of these approaches have involved intense mathematical

complexity. For example, Terzopoulos and Fleischer model the deforming object as a body in an inertial frame of reference [13]. The energy stored in the object due to deformation is represented by a system of partial differential equations. To calculate the position of a point on the object, this system of equations needs to be solved. The theory involved in deriving and solving these equations can be very complex.

The method presented in this paper is a general and simple solution that will work in any collision environment. The system is called A Collision Modeling Environment (ACME). This work is an extension of a thesis by Koti which allowed only colliding circles in two dimensions to be simulated [8]. With ACME, arbitrary three dimensional shapes, including spline surfaces, can be modeled.

In the model, an object is considered to be a system of particles connected to one another by generalized springs. Particle systems were introduced by Reeves to animate fire, smoke, clouds, fireworks, and grass [10], [11]. In Reeves' systems, a particle has properties such as color, position, velocity, and life span. A single particle exists for a very short time, relative to the life span of the entire object. These properties are assigned to a particle stochastically, so there is a degree of randomness in a particle system of this nature.

Randomness is eliminated from the particle system in ACME. Particles are created initially to take the shape and properties of a user-defined object and will not be destroyed during the simulation. The property values for particles will change, but only by well-defined forces acting on the springs between the particles in question.

Objects consist of a large number of independent particles. Each particle can have its own position, velocity, and acceleration. Due to this independence, an object will exhibit deformation when some particles are displaced more than others.

This approach greatly simplifies the mathematics involved because it is just the force through a spring that determines the acceleration, velocity, and therefore the position of an affected particle. Newton's laws are used repeatedly to update the positions of the particles, and thus the form of the entire object.

The characteristics of the spring are determined by the material of the object, and in turn determine the behavior of an object during a collision. For example, increasing the stiffness of the springs in an object will directly increase the rigidity of that object.

Denting and fracturing of objects is also possible with this system. Denting is modeled by comparing the distance between two particles with a user supplied threshold. If the threshold is reached, the equilibrium length of the spring is changed to a smaller or larger value. Fracturing is modeled in much the same way. If the threshold is reached, then the spring is destroyed, thus disconnecting the two particles. If enough springs are destroyed, then an object can be separated into two or more pieces.

Although the mathematics is simplified, the computational cost is significant. An object can consist of thousands of particles, all of which have to be continually updated. For each particle, a force due to all neighbors must be calculated. That force is then used to find the acceleration, velocity, and position of that particle. Doing these computations for each particle in each object is inherently expensive.

There are several distinct parts to ACME. A description of object creation is provided in Chapter 2, and collision detection is explained in Chapter 3. Chapter 4 discusses the work involved in computing timesteps. In Chapter 5, the details of the simulation model are covered. Chapter 6 is a description of how the simulation is

animated. Finally, Chapter 7 describes some future areas of study and some conclusions.

CHAPTER 2

OBJECT CREATION

Objects in ACME are created and defined by the system based on user specified control data. The user provides object attributes which include a name, an initial velocity, a color, and a mass. Young's Modulus of Elasticity, the coefficient of restitution, a dent threshold and a fracture threshold are also supplied (see Chapter 5). Based on these attributes, ACME determines other object properties such as spring constants, dentability, the number of particles comprising the object, and the mass of a particle. As objects are created, they are stored in a doubly linked list. The object list provides dynamic access to the objects for things such as collision detection, position updates, additions and deletions.

Object creation is a two step process, the first of which is creating a grid that represents the particle system. This grid can be either two or three dimensional, depending on the type of object being created. The second step is to define the object by transforming the grid into a linked list of particles and then finding all the neighbors for each particle.

Creating Particle Systems

Objects must be created as particle systems before anything else can be done. The principle used is the same as that of a cookie cutter, but in two or three dimensions. The idea is to create a grid that divides the space an object will occupy. The number of cells for the largest extent (x, y, or z direction) of a particular object is supplied by the user. ACME calculates the number of cells for the remaining directions ensuring that all cells are the same size and thus an equal distance apart.

Grid cells represent particles and the separation between cells represents the initial equilibrium spring length. The dimension of the grid (2D or 3D) is dependent upon the type of object being created. The grid is placed on the object like a cookie cutter such that it is exactly the size of the extents of the object. Figure 1 shows a two dimensional example.

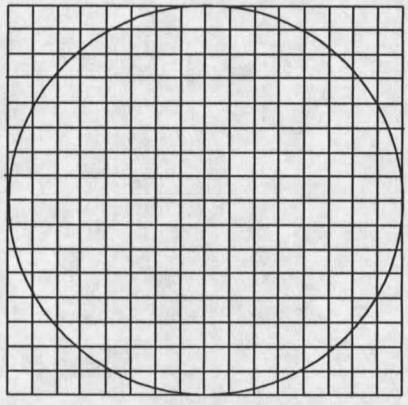

Figure 1

Cookie Cutter Grid Placed On A Circle

The next task is to determine which cells will actually become particles in the object. The first step is to create a shell of particles defining the outer boundaries of the object. All cells in the grid initially have values of *undefined*, which will later be classified as *peripheral* for cells on the edge of the object, *interior* for cells inside the object, or will remain *undefined* for cells outside the object.

Once the shell is defined, a *seed* cell is found which is guaranteed to be inside the boundary shell. The seed is found by extending a line from a given cell to infinity in any one direction and counting the number of intersections of that line and the shell. If the number of intersections is odd, then the cell is inside the shell and thus a valid seed. This algorithm was derived from [3]. The seed can then be used by a

flood-fill algorithm [12] to fill the interior of the shell.

Depending on the object type, ACME uses the corresponding module to find the peripheral cells in the grid. The flood-fill algorithm is then used for all object types to find the interior cells. The user supplies an identification number that describes the type of object to be created. Table 1 shows the types and corresponding identification numbers available in ACME. The following sections provide a description of each object type and how the particle system for that type is created.

Table 1 - Object Types And Identification Numbers In ACME

Circle	0
Sphere	1
Polygon	2
Polygonal Mesh	3
Spline Curve	4
Spline Surface	5

Circles and Spheres

To create a particle system for a two dimensional circle (id 0), a radius and an (x,y) pair representing the center of the circle is supplied by the user. The number of rows and columns in the grid is then calculated by finding the width and height of the circle and dividing each by the user-defined grid size.

A scan-line intersection approach is taken to create the shell of the circle. Each row of the grid represents a horizontal scan-line and each column represents a vertical scan-line. For each horizontal scan-line the y value is calculated, and the intersections with the circle are found. These intersection points are rounded to the nearest integer, say n, and the n^{th} cell in that row becomes a peripheral cell. In

Figure 2, the shaded cells represent two such cells. The same method is employed for each vertical scan-line except that the x value is calculated and the n^{th} cell in the column becomes a peripheral cell. Because we have used every row and every column in the grid and the grid extents are exactly the same as the circle extents, every peripheral cell can be found, thus guaranteeing a closed circle. This is important because the flood-fill will fail miserably otherwise.

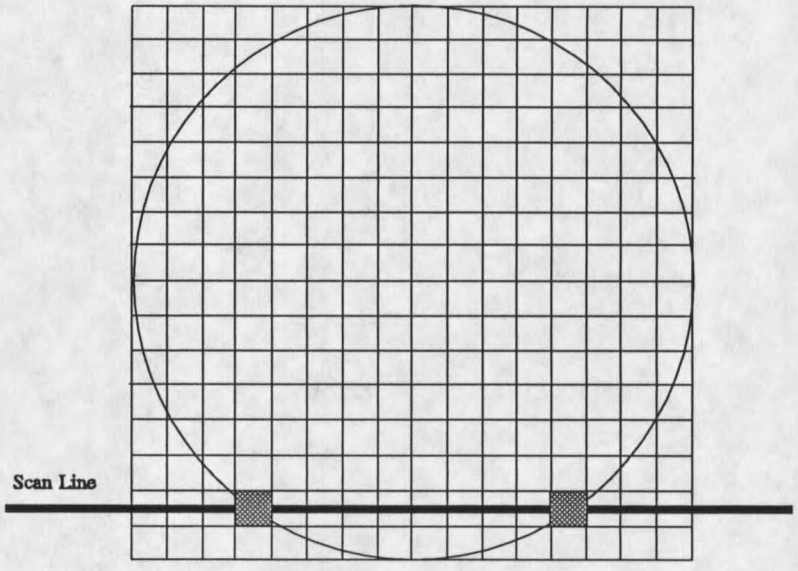

Figure 2

Horizontal Scan Line Intersecting A Circle

Creating a particle system for a three dimensional sphere (id 1) is done the same way with one additional step. New scan-lines called depth scan-lines must be intersected with a sphere to get all peripheral cells from front to back. The flood-fill algorithm has been adapted for three dimensions so it is used again.

Polygons and Polygonal Meshes

The approach to creating particle systems for polygons (id 2) is quite similar to that of creating circles and spheres. In this case, the data supplied by the user is in the form of polygon vertices. The grid is again created by calculating the extents of the object and dividing by the grid size. Horizontal and vertical scan-lines are again used, but here they are intersected with polygon edges rather than with circles or spheres. The intersection points are rounded to the nearest integer to determine which cells are peripheral. The same seed and flood-fill algorithms are used to create particle systems for polygons.

Particle systems for polygonal meshes (id 3) are created similarly, but the control data is much different. A mesh is input by first specifying all vertices in the mesh. Following the vertex list is a list of edges. An entry in the edge list consists of pointers to both the vertices comprising the edge and the polygons containing the edge. The data is stored as three linked lists: a vertex list, an edge list, and a polygon list. Horizontal, vertical, and depth scan-lines are all intersected with polygons in the mesh to create the shell. Once this is done the seed and flood-fill algorithms are used again.

Spline Curves and Spline Surfaces

To create particle systems for objects described by two dimensional spline curves (id 4), the scan-line technique is abandoned. This is for the simple reason that spline curves are generated by computing points on the curve, and therefore no scan-line intersections are needed. The increment used for the values in the parametric equation is chosen to be small enough so that the curve will be closed, ensuring that

the flood-fill algorithm will function properly.

A particle system for an object described by a three dimensional spline surface (id 5) is also created by generating all points on the surface, thus defining the shell of the object. The flood-fill algorithm once again works here. Non-uniform rational B-splines are used for both curves and surfaces. For a detailed discussion of splines, see [1] or [6].

For all types of objects, multiple shapes are allowed. For example, an egg shell can be created by first supplying control points for an outer surface, and then control points for an inner surface, in this case very near to the outer surface. Since this will define two closed surfaces, the flood-fill algorithm will fill only cells between the two.

Defining Objects

Once a particle system has been created for an object it is necessary to define the object fully. This entails creating a linked list of particles (as opposed to the grid form we have now) and creating a linked list of peripherals. After the lists are created, pointers to all the neighbors of each particle must be properly set. In addition, values for initial positions, velocities, and accelerations of each particle are set, and the mass of a particle is calculated.

The reason for going from a grid (2D or 3D array) of cells to a linked list of particles is to save memory. Consider the egg shell example above. Almost all cells in the grid remain undefined and thus waste memory. With the linked list scheme, only memory for the exact number of particles will be used.

Each node in the linked list of particles contains all the information necessary to describe that particle. This includes pointers to all six neighbors (or four neighbors

if the object is 2D), and pointers to the previous and next particles in the particle list. Each node also indicates the initial position, velocity, and acceleration of the particle, as well as its classification. If the particle is peripheral, then there is also a pointer to the next peripheral particle in the list. The remaining data associated with a particle are the lengths of the six (or four) springs connecting it to its neighbors. The lengths of the springs are initially equal and are calculated by dividing the largest extent of the object by the number of cells specified by the user for the grid size. These spring lengths can later change independently, based on what happens in the simulation (see the section on denting in Chapter 5).

When an object is defined in this manner, the control data, object extents, grid size, and even object identification number can be forgotten. These attributes become irrelevant in the collision environment. The only things to keep track of are the particles and the springs, which are fully described in the particle list. Once all objects have been defined, they are free to move about the scene. Because each particle is an independent entity, everything that happens to the objects is simply due to the behavior of the particles which interact with each other via the springs.

CHAPTER 3

COLLISION DETECTION

We now turn our attention toward detecting collisions. In ACME two objects are considered to be in contact if any number of pairs of particles is within a predefined tolerance. The first thing to do is determine what this tolerance should be.

Consider two objects, A containing peripheral particle a and B containing peripheral particles b_1 and b_2. Particle a could meet object B half way between b_1 and b_2 as shown in Figure 3. A collision is taking place, but if the tolerance is smaller than the distances $|ab_1|$ and $|ab_2|$, then no collision will be announced and the objects will be allowed to pierce each other. To guarantee that this won't happen, the maximum particle separation of all objects is chosen as a collision tolerance. In Figure 3, the largest $|ab_1|$ and $|ab_2|$ can be is 1/2 the particle separation for object B, say a distance d. In general, the largest d can ever be is 1/2 the maximum particle separation of all objects. Therefore, setting the collision tolerance to the maximum particle separation will ensure proper collision detection. This is essential because modeling deformation does not take place until a collision is detected.

A serious problem with collision detection is the time it takes to check the distance between all particles of all objects to see if any are colliding. This is where peripheral particles play a special role. By definition, only peripheral particles can be on the edge of an object. In other words, an object's peripheral particles will always be closer to other objects than its interior particles will be. Therefore, only distances between peripheral particles of different objects need to be checked for collision. The number of peripheral particles is very small relative to the total number of particles in an object, so considerable time is saved.

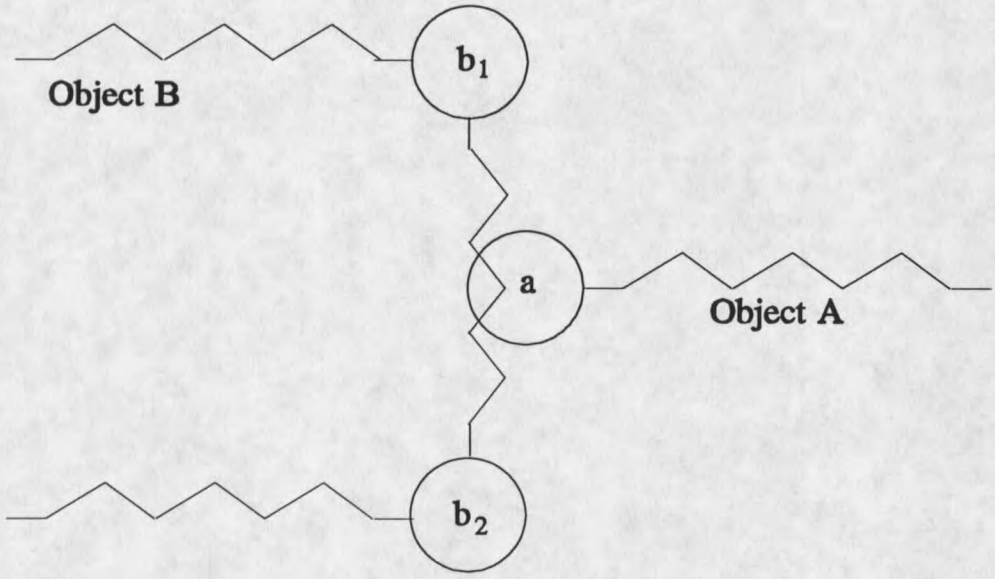

Figure 3

Collision Half Way Between Particles

Another time saving method ACME employs to detect collisions is to divide the scene into a number of distinct boxes called *voxels*. The technique we use is very similar to space subdivision [2]. The voxels can then be used to check only certain parts of the scene for collision, eliminating areas where no objects are present.

In the ACME system, the user defines the extents of the scene, normally a cube large enough to hold all subsequent objects. The size of each voxel is determined and the scene is then divided up into a three dimensional array of voxels, each voxel occupying its own section of the scene. The number of voxels in one dimension is a predefined number (currently eight). For example, if the user specifies a scene to be

40x40x40, then a voxel will be 5x5x5 and there will be 8^3 or 512 voxels comprising the scene.

Each voxel contains several linked lists of peripheral particles, one list for each object in the scene. Every peripheral particle in every object is checked to see which voxel it is in, and the particle is then inserted into the proper list. The existence of a list indicates the presence of an object in that voxel. For example, if there are three objects in the scene, every voxel will have three distinct lists. If an object is not present in a particular voxel, then the corresponding list will be empty.

Once the peripheral particles are placed in their respective voxel lists, collisions can be detected by stepping through the array of voxels. If a voxel and its 26 neighbors in 3-space contain lists from only the same object, then the particles in those lists do not need to be checked for collision. On the other hand, if a voxel or any of its neighbors contains lists from different objects, then particles in those lists need further examination. Only then will the distance between the pairs of particles be compared with the collision tolerance. This will significantly reduce the number of peripheral particles being examined because only the existence of a list is checked initially.

A linked list called a *collision list* is created to keep track of all collisions. An element in this list contains pointers to the two objects involved as well as a list of pairs of particles that are colliding. Any pair of particles whose distance is within the collision tolerance is placed on this list (called a *collision pair list*). The status of these particles is immediately changed to *colliding peripheral*. A new element in the collision list is created any time particles from different objects are colliding. Several of these may be created during one check if there is more than one collision. When

a collision is detected, ACME sets a flag to announce it, and the objects involved are marked as *colliding* objects. The complete collision list is then used to begin the deformation process.

CHAPTER 4

TIMESTEP COMPUTATION

The overall behavior of ACME is based on time. Objects are created and placed into the environment, but nothing happens until time is incorporated. Objects move based on their velocities and time. Collisions occur after a certain amount of time. The amount of deformation that takes place is dependent on the elapsed time. Fractures and dents depend on forces which build up over time.

Time is of essence in ACME, and computing the proper timestep is crucial for the simulation to behave properly. The positions, velocities, and accelerations of particles are updated once per timestep, and then the collision detector is called. Though we accounted for the piercing problem in collision detection, the collision detector only checks the objects between timesteps. It has no control over what happens *during* an interval of time.

A large timestep could allow objects to pierce one another before the collision detector has a chance to prevent it. A large timestep could also cause individual particles to touch or penetrate one another. If this happens, then the spring between these particles no longer behaves properly. It is clear that an efficient, dependable method of computing the timestep is necessary.

The strategy employed by ACME is to ensure that particles only move a certain distance during an interval of time. This distance is a predefined percentage ρ of the smallest existing separation between two particles, a distance d_{min}. Consider the following example: if ρ is 0.5 and d_{min} is currently 0.002, then no particle can travel more than 0.001 units of distance during a particular time interval t. Initially, all particles in an object are equidistant so d_{min} can be found quickly by comparing the

particle separations of each object. From that point on, any time particles are updated, d_{min} is compared with current separations and changed as needed. Having a maximum allowable distance will prevent particles from touching. It will also prevent objects as a whole from penetrating each other because d_{min} could in fact be a separation of particles belonging to different objects. The objective now is to find the worst case scenario and find a timestep based on that. That timestep is satisfactory for all cases.

The worst case scenario is one in which two particles approach each other with maximum speed and acceleration. Initially, all particles in an object have equal velocities, so the maximum relative speed s_{max} can be found quickly by comparing particle velocities of each object. From that point on, each time particles are updated, s_{max} can be compared with current speeds and changed as needed. The accelerations depend on the forces through the connecting springs; therefore, the greatest possible spring force is needed to find the maximum acceleration. Newton's Law states that:

$$a = \frac{f}{m} \tag{4.1}$$

where f is a force, m is a mass, and a is an acceleration. Notice that if the maximum force and the minimum mass are used, we can compute the maximum acceleration. The problem of computing a timestep is now reduced to finding the following: the maximum possible speed s_{max} between two particles, the maximum possible spring force f_{max}, and the minimum particle mass m_{min}. The process for finding m_{min} is a matter of looking at the object list and comparing object properties. The maximum

force, however, must be found by using Hooke's Law:

$$f = kx \tag{4.2}$$

where f is the force due to the spring, k is the spring constant, and x is the displacement of the spring.

To compute the maximum force, we now need the largest spring constant k_{max} and the largest displacement of a spring. The value of k_{max} can be found by comparing the spring constants of the objects (this is an attribute for each object in the object list). The largest displacement can be determined by considering the greatest distance between two particles, say d_{max}, as an equilibrium spring length and the smallest distance between two particles (d_{min} discussed above) as an amount of compression of a spring. Now, using

$$x = d_{\max} - d_{\min} \tag{4.3}$$

will yield a displacement that will be larger than any existing in the system. Now we can make the proper substitutions into Equation 4.2 to obtain the maximum force f_{max}:

$$f_{\max} = k_{\max}(d_{\max} - d_{\min}) \tag{4.4}$$

The maximum acceleration a_{max} can be found by using f_{max} and m_{min} in Equation 4.1. Now we can use a_{max}, s_{max}, d_{min}, and the predefined percentage ρ to find a timestep. An initial time t is used in one of Newton's Laws of motion to compute the displacement l:

$$l = s_{\max}t + \frac{1}{2}a_{\max}t^2 \tag{4.5}$$

If l is less than $\rho * d_{min}$, then t is an acceptable timestep. If it is larger than $\rho * d_{min}$ then t is halved and a new displacement l is computed. The process is repeated until a t is found such that l is small enough to prevent particles from touching.

Note that this worst case scenario does not necessarily exist in the environment. The maximum speed s_{max} may be determined from different particles than d_{min} which in turn may come from different particles than d_{max}. However, this scenario will result in the largest possible displacement of a particle, and will result in a satisfactory timestep. Consequently, objects will never pierce one another before a collision is announced because the closest particles will always be some distance apart, albeit quite small. The same is true for particles within an object. Since they will not be allowed to touch, the springs can be safely used to model the displacements of all particles.

CHAPTER 5

SIMULATION

The previous chapters have basically described the tasks necessary for ACME to model collisions properly. The crux of the system, however, is the simulation of all motion in the scene. Objects are created, collisions are detected, and the timestep is accurately computed, but the purpose of ACME is to model what happens after all that.

When two objects collide, several things happen. Forces due to impact will be generated and then propagate throughout the colliding objects. There will be a mechanical loss of energy or *damping* due to inherent restoring forces. Objects may undergo permanent denting, stretching, or cracking. Cracks may lead to an object fracturing into two or more pieces. This chapter explains how each of these components is modeled.

Computing Forces

There are two kinds of forces that can act on particles in an object. External forces due to the impact of a collision act on peripheral particles, and internal forces due to the displacements of the springs act on all particles (including peripherals). When a collision is announced and the collision list is created, it is the external forces that begin the deformation process. These forces cause the internal spring forces that propagate throughout an object.

Before any forces through a spring can be computed, a spring constant must be defined for each object. The spring constant is determined by using Young's

Modulus of Elasticity Y:

$$k = Yl \tag{5.1}$$

where l is the initial particle separation distance for the object [5]. The larger Y is, the larger k is and thus the more rigid the object will be. Steel, for example, has a Modulus of Elasticity of 200×10^9 N/m^2 [5] while the Modulus of Elasticity for hard rubber is 2070×10^6 N/m^2 [9].

Computing External Forces

An important characteristic of a peripheral particle is that it does not have springs connecting it to neighbors in all six directions. If it did, it wouldn't be a peripheral particle. Instead a *bumper spring* is attached to them (see Figure 4). Bumper springs are compressed when they come in contact with other bumper springs, but they are never extended beyond their equilibrium length because they are not attached at one end. As stated in Chapter 3, each collision has a collision pair list associated with it. An external impact force is computed for each pair of peripheral particles using the bumper springs. Consider objects A and B in Figure 4. Object A has spring constant k_A and Object B has a spring constant k_B. Particles a and b are a colliding pair in the collision pair list. The external impact force between a and b is simply due to the bumper springs connecting them. Because there are two springs connected in series, there will be a new constant k_{new} based on the two constants k_A and k_B:

$$k_{new} = \frac{k_A k_B}{k_A + k_B} \tag{5.2}$$

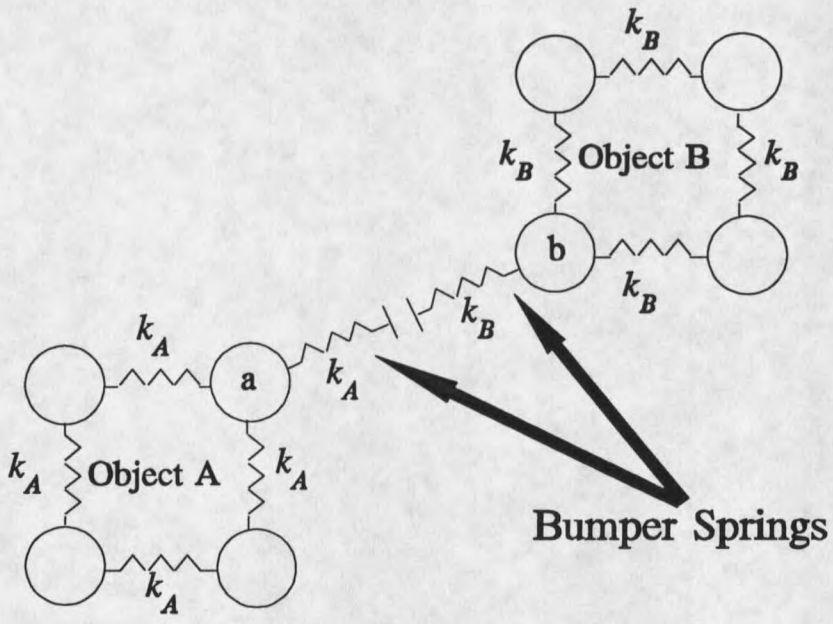

Figure 4

Bumper Springs Of A Colliding Pair

From Hooke's Law, the magnitude of the force f_{ab} on particle a due to particle b is given by:

$$|\vec{f}_{ab}| = k_{new}(d_t - d) \tag{5.3}$$

where d_t is the collision tolerance (used as the equilibrium position of the two springs in series) and d is the distance between particles a and b. The force f_{ba} on particle b due to particle a is simply $-f_{ab}$.

The new acceleration, position, and velocity for particle a are given by the following equations respectively:

$$\vec{a}_a = \vec{a}_a + \frac{m_A}{\vec{f}_{ab}} \tag{5.4}$$

$$\vec{p}_a = \vec{p}_a + \vec{v}_a t + \frac{1}{2}\vec{a}_a t^2 \tag{5.5}$$

$$\vec{v}_a = \vec{v}_a + \vec{a}_a t \tag{5.6}$$

where a_a, p_a, and v_a are the acceleration, position, and velocity of particle a, and t is the computed timestep. The accelerations of all particles are updated before any positions and velocities are updated. If this was not done a particle would be displaced before its neighbors were displaced; thus altering the separations between the particles prematurely.

Finding the forces and updating the particles is done for all pairs of particles in the collision pair list. Initially these particles are the only ones in the given object that are affected by the collision.

Computing Internal Forces

Computing the internal force is a slightly more involved task. For a given particle, the forces of all of the springs connecting it to its neighbors will contribute to the new acceleration of the particle. Therefore, the acceleration is a sum of up to six accelerations.

The magnitude of each force is given by:

$$|\vec{f}_i| = k|l_i - d_i| \tag{5.7}$$

where f_i is the force from the i^{th} neighbor of the particle, k is the spring constant for the object in question, l_i is the equilibrium spring length for the i^{th} neighbor, and d_i is the distance between the particle and the i^{th} neighbor. Figure 5 shows an example of this in two dimensions. For three dimensions, particle P would have two more neighbors, one into the page, and one coming out of the page. The total acceleration of the particle is then:

$$\vec{a} = \vec{a} + \sum \frac{\vec{f}_i}{m} \tag{5.8}$$

where m is the mass of a particle for the given object. Once the total accelerations for all particles are found, the positions and velocities are updated using Equations 5.5 and 5.6 respectively.

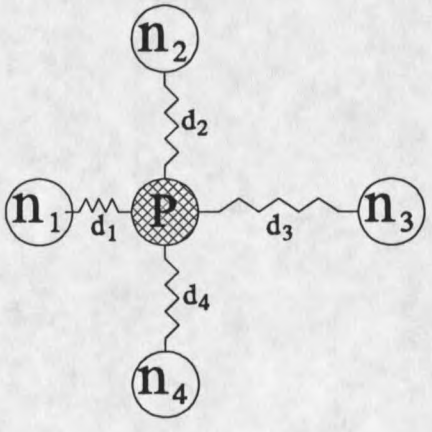

Figure 5

Forces On A Particle From Neighbors

Particles not in the collision pair list that experience internal forces are classified as either *affected interior* or *affected peripheral*, depending on whether or not they are edge particles. Particles classified as *colliding peripheral* (the particles in the collision pair list) will have accelerations that arise from both external forces and internal forces. A particle with this classification will have an acceleration due to the force through its bumper spring and an acceleration due to the force from its neighbors. These three classifications all represent particles that have been affected by a collision.

Marking Newly Affected Particles

For the object being updated, the linked list of particles is traversed and each affected particle is updated based on the forces acting on it. During the first time interval of a collision, all particles are unaffected except for those particles classified as *colliding peripheral.* During the next time interval, those same particles will remain affected but all of their neighbors will become affected also. The algorithm for finding newly affected particles after an update simply traverses the linked list of particles looking for particles whose classification is either *colliding peripheral*, *affected peripheral*, or *affected interior*. A particle with one of these three classifications was affected during the previous time interval, and therefore its neighbors will be affected during the next time interval. The neighbors are marked as either *affected peripheral temporary* or *affected interior temporary*. The *temporary* classification is used so that a newly affected particle's neighbors will not be marked as affected (they won't be affected until later). Once all affected particles' neighbors have been classified as affected, the list is retraced to drop the temporary

classification. After this step ACME is ready to compute a new timestep and continue to the next interval.

Accounting For Energy Loss

In the real world, a deforming object will eventually return to its original shape unless something drastic happens. For example, when a racquet hits a tennis ball, the ball is deformed dramatically and then oscillates until finally it returns to its original spherical shape. Inherent restoring forces cause the oscillations to be *damped.*

The damping forces are modeled in ACME by using two spring constants: k_{comp} when a spring is compressed, and k_{extn} when a spring is extended. The value for k_{comp} is the one derived using Young's Modulus of Elasticity (already discussed). The value for k_{extn} is derived from the coefficient of restitution of the object and is given by:

$$k_{extn} = \frac{k_{comp}}{e^2} \tag{5.9}$$

where e is the coefficient of restitution for the given object [7]. Now, e is always less than one, so k_{extn} will always be larger than k_{comp}. Because of this fact, the spring will always be more rigid during extension than during compression. By the very nature of a spring, if it is more rigid, it is more difficult to cause displacement. Therefore the forces in ACME will be damped by this mechanism. Whenever the distance between two particles is less than their equilibrium spring length, k_{comp} is used to compute the force, otherwise k_{extn} is used.

Allowing For Permanent Dents And Stretches

A dent in an object is caused when the object is deformed by compression and does not return to its original shape. A stretch is caused when the object is deformed by extension and does not return to its original shape. In ACME these phenomena are modeled by allowing the equilibrium lengths of the springs to change.

The criterion for altering the equilibrium spring lengths is a user-supplied *dent threshold.* The dent threshold is a percentage of the spring length. If the distance between two connected particles changes by more than this percentage, then the spring length is changed to the new distance. If the equilibrium spring lengths of a large group of particles are all made smaller, then the appearance will be that of a dent. By the same token, a stretch will appear if a large group of spring lengths are made larger. The threshold is dependent on the type of material. For instance, tin would have a fairly small dent threshold because once the forces are enough to cause displacement, they should be permanent. Jello$_{TM}$ on the other hand would virtually have no threshold whatsoever. Jello$_{TM}$ always bounces back!

Allowing For Fractures

The strategy for modeling fractures is analogous to that used for dents and stretches. The user supplies a *fracture threshold* and any particles whose separation changes by this amount are disconnected. They are disconnected by removing the spring between them, which means the two particles are no longer neighbors and forces can no longer propagate between them. This in essence, is a fracture on the particle level.

If two interior particles are disconnected in this way, they may need to be

reclassified as *peripheral*. This is true if both interior particles are attached to peripheral particles. Besides being reclassified, the new peripheral particles must also be placed in the proper voxel-particle list, and be placed in the object's list of peripheral particles. The total number of peripheral particles will need to be updated if this happens.

A large chain of particles whose springs have been removed will give the appearance of a crack in the object. If enough of these particle level fractures occur, an object may break into separate pieces. Any time a spring is destroyed, there is a possibility that the object has split. Therefore a flag is set for the object in question so that it can be tested for separation.

Testing An Object For Separation

Testing an object to see if it has broken into two or more pieces is very important. If pieces of an object are flying about the scene independently, yet still considered to be the same object, many problems arise. Pieces of an object could head in opposite directions, and they could collide with other objects. Because these pieces are incorrectly considered as a single entity, collisions from opposite sides of the scene could be announced for the same object. The pieces could also collide with each other, but no collision would be announced because the pieces are considered to be the same object. Because of these instances, it is necessary to check to see if an object has broken into multiple pieces.

The method used is a chain algorithm. Consider Figure 6, where the object has experienced three broken springs: the ones connecting particles 2 and 3, 6 and 7, and 10 and 11 in Figure 6a. Because these springs have broken, the chain algorithm will

check to see if the object has broken into pieces.

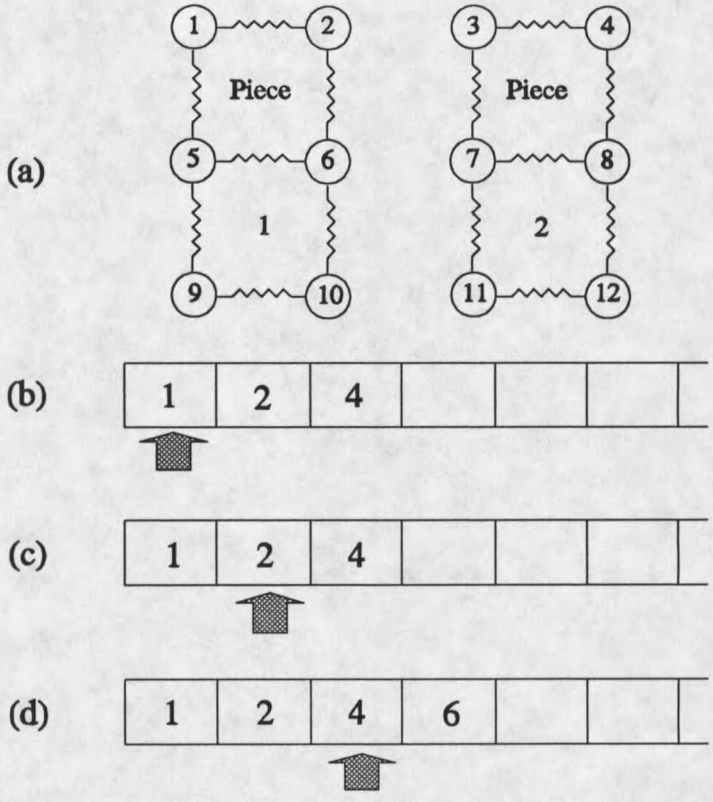

Figure 6

Chain Algorithm To Check If Object Has Broken

The first step is to place any peripheral particle on a queue, in this case, particle 1. A pointer is set to indicate that this is the *current particle*, and its neighbors are placed on the queue, as shown in Figure 6b. The pointer is then moved one place to the right (Figure 6c), and the new current particle's neighbors are placed on the queue (Figure 6d). Although particle 1 is a neighbor of particle 2, it is not put on the queue because it was inserted previously at location one. After the current

particle's neighbors have been placed on the queue, the pointer is again moved to the right and the process is repeated. The algorithm will continue in this manner until all neighbors have been inserted.

A running count of all particles placed on the queue is maintained. In the case of Figure 6, particles in Piece 2 will never be placed on the queue because they are not neighbors of any particles that are on the queue. Therefore, the running count will be less than the total number of peripheral particles for the object. Whenever the running count is less than the total, the object is broken into at least two pieces, and must be redefined. Had there been a connection anywhere between the two pieces in Figure 6, all peripheral particles would eventually have been placed on the queue and counted. The running count would then have been equal to the total number of peripheral particles; thus guaranteeing that object *A* would still be in one piece.

Redefining An Object

An object needs to be redefined whenever it has broken into separate pieces. Basically, redefinition consists of deleting the original object from the object list and inserting its pieces as whole objects. New peripheral particle lists must be added to all of the voxels and the old lists must be deleted. Particle lists must also be created for the new objects themselves.

The redefinition algorithm is similar to the one used to test for separation, but interior particles are included. Every time all neighbors are exhausted, the queue represents a new object. Each particle in the queue is then placed in all the appropriate lists for the new object. A new queue is started by using another particle from the old object. This process is repeated until all particles are accounted for.

CHAPTER 6

ANIMATION

The goal of ACME is to provide a means of learning about collisions. Without displaying the effects of the simulation, no insight into the behavior of colliding objects can be gained. Animation is therefore a very important aspect of the system.

Because of the enormous number of particles that comprise a scene, real time animation is not used. Instead, results from the simulation are stored in data files and animation is a post-simulation function based on the data files. A frame by frame approach is taken where a frame is defined to be the positions of all particles in the scene at the end of a time interval. One data file represents one frame of animation.

The user specifies the number of frames to be generated and the interval at which to store frames. An interval greater than one is often desired because the difference from frame to frame can be nearly negligible. A count of the number of frames that has been generated is maintained. After an update for a time interval has been completed, the count is checked with the user-supplied frame interval to see if the frame should be saved. If so, all the appropriate scene data is stored in a file. The name of the file is appended with a number that indicates which frame it represents.

The data comprising a frame consists of the extents of the scene so that the boundaries of the environment can be drawn. The number of objects in the scene is also provided, as well as the color of each object, the number of particles comprising an object, and the positions of all particles. No other simulation data is needed because the animation program simply draws all particles in their given positions using the appropriate object color.

After the simulation is done and all frames have been generated, the frame interval is supplied to an animation program; and frames are read from file and displayed on the screen consecutively. Once again, due to the number of particles being displayed, the animation is not real time. However, the frame buffer can be sent to a recorder and each frame can be saved on film. When the film is viewed, realistic animation of the collision environment is achieved.

CHAPTER 7

CONCLUSION AND FUTURE DIRECTIONS

Conclusion

The goal of ACME is to find a simple and general model to simulate the behavior of colliding objects. This model is conceptually one of the simplest collision models. Once the objects are created, only the forces through the springs determine the shapes of the colliding objects. Newton's Laws of motion and Hooke's Law for springs provide all the necessary information. ACME is also much more general than other models. Objects of arbitrary shape can be created, and after creation an object is allowed to deform arbitrarily. Dents, stretches, fractures, and complete separation of objects can be effectively modeled. Objects can also take on rotational velocity without disrupting the model.

Most of the algorithms are generally linked list manipulators. Objects, particle systems, peripheral particles, colliding pairs of particles, and collision information are all stored in linked list form. Based on the mathematical results of the model, the appropriate linked list is added to or deleted from.

Simulations have been carried out using different object shapes and parameters. Figure 7 shows two circles in collision. Deformation can be clearly seen in both circles. Note also that most of the deformation occurs near the point of impact and is dissipated further away. The aliasing effect seen in the objects was not eliminated in order to preserve the pattern of deformation that took place.

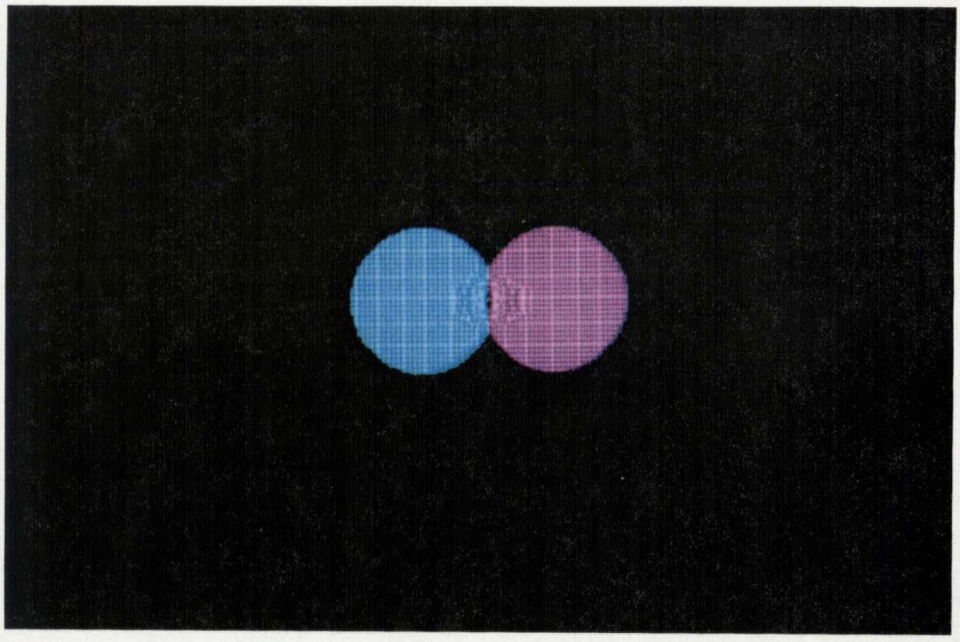

Figure 7

Colliding Circles

Simulations were done on a Silicon Graphics Personal Iris, an IBM RS 6000, an Apollo DN10000 and an HP350-SRX workstation. On all of these machines, time was a serious constraint. It took approximately nine days on the Personal Iris to complete the simulation of the two spheres which have a grid size of 60x60x60.

The most immediate drawback is the amount of time it takes to do a

simulation. However, the ACME system is highly parallel in nature. The task of updating a single particle is a completely independent event. The acceleration of every particle could be found simultaneously. Updating the positions and velocities could be done immediately afterwards, also simultaneously. The testing for a broken object could also be done in parallel as well as the redefinition of broken objects. Parallelizing ACME would result in very significant increases in performance.

Another area where work is needed is in the method of obtaining thresholds for dents and fractures. It would be more useful to derive the thresholds from some material property rather than forcing the user to provide them. This would eliminate undesirable results due to incorrect thresholds.

The user should also be relieved of supplying the grid size for an object. The grid size is indirectly related to the density of the object, but devising a method for computing the grid size based on this has not been attempted. User-defined grid sizes could result in particle systems that are too sparse to model the objects accurately. This problem would be alleviated by relating the grid size to a material property.

Future Directions

Because the focus of ACME has been to find a simple, general model, no work has yet been done to make it an easy system to use. All user input is read entirely from a data file before anything else is done. This means that the user has to know in advance everything about the environment. An interface should be developed so that the user can create objects, edit them, move them, and delete them interactively before the simulation begins. This would make ACME much easier to use.

The ACME system could be extended to incorporate other phenomena. For

not modeled in the current system. A module that computes frictional forces due to objects sliding across one another would be useful.

Although work remains to be done, ACME is still a useful visual tool for studying colliding objects. Mathematical complexity has been eliminated by using simple laws of physics, such as Hooke's Law and Newton's Laws. ACME's ability to simulate arbitrarily shaped objects in both two and three dimensions makes it a general solution to the collision problem. By visualizing impact, deformation, dents, stretches, fractures, and broken objects, the versatility of ACME promises to be a great resource in any study of collision modeling.

REFERENCES CITED

1. Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., Computer Graphics: Principles and Practice, Second Edition, Addison-Wesley, 1990, pp491-501.

2. Glassner, A.S., Space Subdivision for Fast Ray Tracing, IEEE Computer Graphics and Applications, Vol. 4, No. 10, October 1984, pp15-22.

3. Glassner, A.S., editor, Graphics Gems, Academic Press, 1990, pp124-125.

4. Hahn, J.K., Realistic Animation of Rigid Bodies, SIGGRAPH 88, pp299-308.

5. Halliday, D., Resnick, R., Fundamentals of Physics, Third Edition, John Wiley & Sons, Inc. 1988, pp293-297.

6. Hill, F.S., Jr., Computer Graphics, Macmillan Publishing Company, 1990, pp483-507.

7. Hunt, K.H., Crossley, F.R.E., Coefficient of Restitution Interpreted as Damping in Vibroimpact, Transactions of the ASME, June, 1975, p440.

8. Koti, S., Simulation of Deformation of Colliding Objects Using Particle Systems, Master's Thesis, 1988.

9. Perry, Robert H., Green, D.W., Maloney, J.O., editors, Perry's Chemical Engineering Handbook, Sixth Edition, McGraw-Hill Book Company, 1984, p6-98

10. Reeves, W.T., Particle Systems - A Technique for Modeling a Class of Fuzzy Objects, SIGGRAPH 83, pp359-376.

11. Reeves, W.T., Blau, R., Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems, SIGGRAPH 85, pp313-322

12. Rogers, David F., Procedural Elements for Computer Graphics, McGraw-Hill Book Company, 1985, pp85-88.

13. Terzopoulos, D., Fleischer, K., Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture, SIGGRAPH 88, pp269-278.

780042