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.

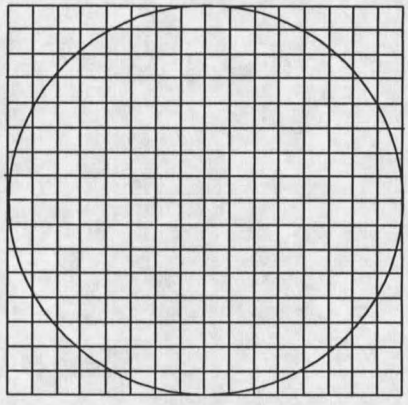

**Figure 1**

**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.

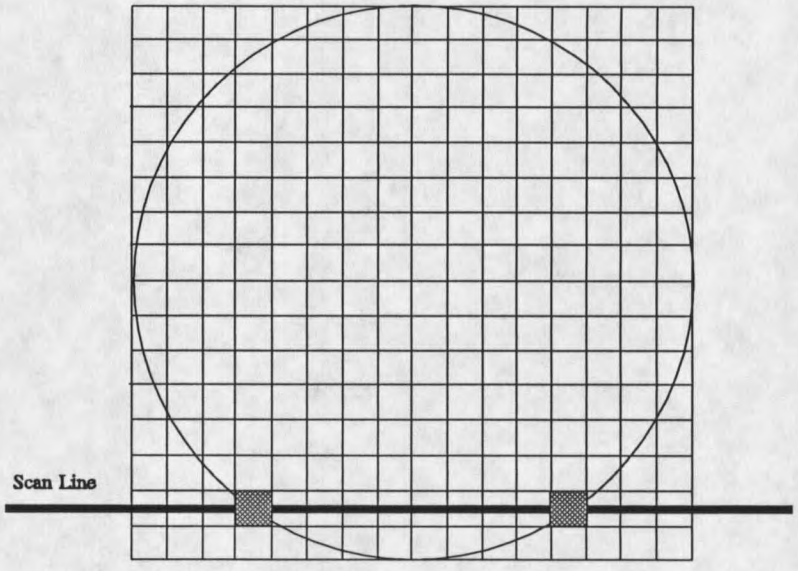


**Figure 2**

**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