Collision Detection and Response for Computer Animation

Matthew Moore and *Jane Wilhelms*

Computer Graphics & Imaging Laboratory
Computer & Information Sciences Board
University of California at Santa Cruz
Santa Cruz, California 95064

Abstract

When several objects are moved about by computer animation, there is the chance that they will interpenetrate. This is often an undesired state, particularly if the animation is seeking to model a realistic world. Two issues are involved: *detecting* that a collision has occurred, and *responding* to it. The former is fundamentally a kinematic problem, involving the positional relationship of objects in the world. The latter is a dynamic problem, in that it involves predicting behavior according to physical laws. This paper discusses collision detection and response in general, presents two collision detection algorithms, describes modeling collisions of arbitrary bodies using springs, and presents an analytical collision response algorithm for articulated rigid bodies that conserves linear and angular momentum.

CR Categories and Subject Descriptors: I.3.5: [Computer Graphics]: Computational Geometry and Object Modeling - Geometric algorithms; I.3.7: [Computer Graphics]: Three Dimensional Graphics and Realism - Animation.

Key Words and Phrases: computer animation, collision detection, collision response, analytical solution, dynamical simulation.

1. OVERVIEW

Computer animation provides a number of methods for controlling object motion.[28] The object's positions and orientations as functions of time may be interpolated from keyframes or parameter specification,[27] or may be the output of special computer programs written by the user,[23] or may be produced by physical simulation of the effect of internal, model-derived, and user-specified forces and torques.[1, 12, 16, 29, 32] In any such scheme, the main questions when animating a single object are how to achieve realistic motion and how to economize on the human animator's time. When several objects are animated at once, the additional problem of detecting and controlling object interactions is encountered. When no special attention is paid to object interactions, the objects will sail majestically through each other, which is usually not physically reasonable and produces a disconcerting visual effect. Whenever two objects attempt to interpenetrate each other, we call it a collision.

The most general requirement that arises from this is an ability to *detect* collisions. Most animation systems at present do not provide even minimal collision detection, but require the animator to visually inspect the scene for object interaction and respond accordingly. This is time-consuming and difficult even for keyframe or parameter systems where the user explicitly defines the motion; it is even worse for procedural or dynamical animation systems where the motion is generated by subroutines and laws defining their behavior. Though automatic collision detection is somewhat expensive both to code and to run, it is a considerable convenience for animators, particularly when more automated methods of motion control, such as dynamics or behavioral control, are used.[24, 31] This paper describes two collision detection algorithms. One algorithm deals with triangulated surface representations of objects, and is appropriate for flexible or rigid surfaces. The other algorithm applies to objects modeled as rigid polyhedra. Both algorithms are simple, robust, and not dreadfully expensive.

The related issue is *response* to collisions once they are detected. Even keyframe systems could benefit from automatic suggestions about the motion of objects immediately following a collision; animation systems using dynamical simulation inherently must respond to collisions automatically and realistically. Linear and angular momentum must be preserved, and surface friction and elasticity must be reasonable. This article presents two methods that satisfy these criteria. One is the obvious method, based on temporary springs introduced at collision points. The other method is an analytical linear system solution. The former method is more general, working equally well for flexible, rigid, and articulated bodies. The latter, limited to rigid and articulated objects, is typically faster. Furthermore, while the spring solution assumes the ability to use the dynamics equations of motion to predict the motion immediately after impact, the analytical solution could be used within a kinematic animation system.

2. COLLISION DETECTION

Collision detection involves determining when one object penetrates another. It is clearly an expensive proposition, particularly when large numbers of objects are involved and the objects have complex shapes. Collision detection has been extensively pursued in the fields of CAD/CAM and robotics,[2, 3, 6, 7, 11, 30] and it is with some diffidence that we offer any more algorithms. Some published algorithms[2, 3] solve the problem in more generality (and at higher cost) than we have found to be necessary for computer animation. Others[6] do not easily produce the collision points and normal directions necessary if collision response is to be calculated. Voxel-based methods have also been used,[30] but would not be appropriate for all applications. Finally, many collision detection algorithms are quite intricate and must deal with many special cases, which we wished to avoid for software engineering reasons. Two collision algorithms are discussed here: the first is designed to test the interpenetration of surfaces modeling flexible objects; the latter is designed to test for interpenetration of convex polyhedra.

2.1. Collision Detection for Flexible Surfaces

Surfaces are modeled as a grid of points connected to form triangles.[19] Collisions between surfaces are detected by testing for penetration of each vertex point through the planes of any triangle not including that vertex (thus, self-intersection of surfaces is detected). The surfaces are assumed to be initially separate. For each time step of animation, the positions of points at the beginning and the end of the time step must be compared to see if any point went through a triangle during that time step. If so, a collision has occurred. The algorithm is $O(nm)$ for n triangles and m points.

A correct test must consider edges and triangles, as polyhedral objects can collide edge-on without any vertices being directly involved. However, in many cases merely testing points versus triangles produces acceptable results. This algorithm only tests points versus triangles. It is worth noting that the mathematics for testing intersection of a moving point with a fixed triangle is the same as for testing a fixed edge versus a fixed triangle. Thus the fully general edge versus triangle tests could be done at fixed instants in time, with the same advantages and disadvantages that will be discussed for the second collision detection algorithm.

The question of whether a moving point has intersected a surface can be divided into two cases. The easy case requires the surface to be fixed in space, whereas the hard case allows the surface to be moving also. When the surface triangle is fixed, the parametric vector equation

$$P + (P' - P)\,t = P_0 + (P_1 - P_0)\,u + (P_2 - P_0)\,v$$

where P and P' are the beginning and ending positions of the point and the P_i's define the triangle, is set up and solved for the variables u, v, t. u and v are parametric variables for the plane defined by the triangle, whereas t is a time variable which is 0 at the beginning of the simulation step in question, and 1 at the end. The left hand side is the parametric equation for the path of the point, and the right hand side is the parametric equation for any point on the plane. This vector equation represents three scalar equations in three unknowns and is solved by matrix inversion. If $0 \le t \le 1$ and $u \ge 0$ and $v \ge 0$ and $u + v \le 1$, then the point has intersected the triangle during the time step.

The hard case is solved by setting up the parametric vector equation

$$P + V\,t = P_0 + V_0\,t + ((P_1 - P_0) + (V_1 - V_0)\,t)\,u + ((P_2 - P_0) + (V_2 - V_0)\,t)\,v$$

where P is the point (with velocity V per time step), the P_is define the triangle vertices (with velocity V_i per time step), and t, u, v are the parametric variables. Rearranging, we can write this as three linear equations in three unknowns.

$$\begin{aligned} a\,u + b\,v + c\,t &= d \\ e\,u + f\,v + g\,t &= h \\ i\,u + j\,v + k\,t &= l \end{aligned}$$

where

$$\begin{aligned}
a &= (P_{1x} - P_{0x}) + t(V_{1x} - V_{0x}) \\
b &= (P_{2x} - P_{0x}) + t(V_{2x} - V_{0x}) \\
c &= -V_x \\
d &= P_x - (P_{0x} + tV_{0x}) \\
e &= (P_{1y} - P_{0y}) + t(V_{1y} - V_{0y}) \\
f &= (P_{2y} - P_{0y}) + t(V_{2y} - V_{0y}) \\
g &= -V_y \\
h &= P_y - (P_{0y} + tV_{0y}) \\
i &= (P_{1z} - P_{0z}) + t(V_{1z} - V_{0z}) \\
j &= (P_{2z} - P_{0z}) + t(V_{2z} - V_{0z}) \\
k &= -V_z \\
l &= P_z - (P_{0z} + tV_{0z})
\end{aligned}$$

The P_ws and V_ws are the position and velocity components of the point, and the P_{iw}s and V_{iw}s are the position and velocity components of the triangle vertices. The velocities are per time step.

The linear system above can be solved for t and expanded to

$$0 = a(ja - ib)(ha - ed) - a(ja - ib)(ga - ec)\,t + (fa - eb)(ka - ic)\,t - (la - id)(fa - eb)$$

Substitution of the actual expressions for a through l gives a 5'th order polynomial in t. If further substitutions were made, the equations could be written in the form

$$c_5 t^5 + c_4 t^4 + c_3 t^3 + c_2 t^2 + c_1 t^1 + c_0 = 0$$

Polynomials of order 5 and above cannot be solved analytically,[10] so a binary search technique is used to find approximate values for t.[5] Binary search is used because it is guaranteed to converge, and because, using economizing techniques described below, this algorithm is not used often enough to warrant large efforts at optimization. The interval from t=0 to t=1 is subdivided into a number of sub-intervals, and the left-hand side is evaluated at each dividing point. If the sign of that value is different for the two endpoints of some subinterval, then some t for which the equation is true must lie within that interval. A binary search of values of t within that interval brings the brackets around that value of t closer together, until a limit is reached (after 10 iterations, in our system) and an approximate value of t is found. Each value of t thus arrived at is used to get values for u and v by back substitution, and then the standard $0 \le t \le 1$ and $u \ge 0$ and $v \ge 0$ and $u + v \le 1$ test is used to determine whether a collision has occurred.

To minimize the cost of executing the above calculations, a preliminary step is used. Every point is compared to every triangle. The perpendicular distance of the point from the plane defined by the triangle is first derived, by substituting into the plane equation,[25] for the beginning and the end of the time step. If the sign of the perpendicular distance has not changed intersection is assumed not to have occurred. If the sign has changed, then the more expensive tests outlined above must be done, but in practice this test eliminates most point-triangle pairs.

A special kind of bounding box can also be used to minimize computation. This bounding box includes the beginning and ending position of the triangle. This box is then grown by the distance between the beginning and ending positions of the point being tested. (This is necessary to avoid the point passing unnoticed completely through the box during the time step. A similar growth technique is used in the Lozano-Perez path planning algorithm.)[15]

The basic algorithm is $O(nm)$, for n triangles and m points. Use of an octree[19] and bounding boxes can reduce the time to $O(m \log m)$ to construct the octree, and $O(n \log m)$ to search it (assuming that the tree is almost balanced and that the bounding boxes are small compared to the space covered by the tree).

The search finds all point - bounding box pairs that must be examined more closely for possible intersection. All of the points in the model are inserted into an octree, which is created anew for each round of collision detection. This octree is based on the points themselves, with each point P having up to 8 subtrees containing points in each of the octants of space defined by the P's position. This is an obvious generalization of the well known binary search tree.[13, 14] A pseudo-random number generator is used to scramble the order of insertion; in this way, Knuth assures us,[13] the tree will be almost balanced, i.e. the height of the octree will be $O(\log m)$ almost always.

Each triangle's bounding box is grown by the distance between the starting and ending positions of the fastest point being tested. Each bounding box is then recursively compared against the octree to find the points inside it. If a point is inside the box, all of its subtrees must be searched recursively. If a point is outside the box, at least half of its subtrees do not need to be searched. If a point is found to be inside a box, then the algorithm above must be run to determine if the point intersected the associated triangle during the time step.

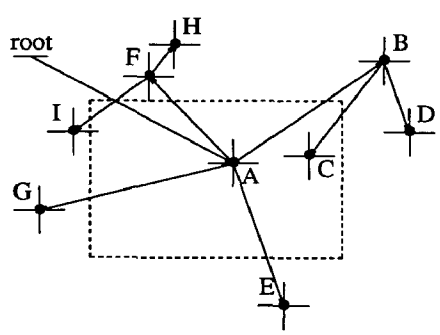


Figure 1
Searching a Quadtree

Figure 1 illustrates a two-dimensional version of this procedure. The points A through I were inserted into an initially empty quadtree in alphabetical order, so that A is the root element of the tree. The tree is to be searched for all points inside the dotted box. A is inside, so all of it's subtrees must be searched. B is above and to the right of the dotted box, so only its lower left subtree must be searched. This finds C, which is inside the box. If C had subtrees, they would all have to be searched. The next subtree of A starts with F. F is above the bounding box, so both of its lower subtrees must be searched. One is empty, and the other contains only I, which is also outside the box. A's third subtree contains only E, which is below the box. If E had subtrees, only the upper ones would need to be searched. A's fourth and last subtree contains only G, which is outside the box. If G had subtrees, only the two right hand ones would be searched. A large, bushy quadtree would be very fast to search (if the dotted box were small relative to the area covered by the tree) because the unsearched subtrees would often contain large numbers of points.

2.2. Collision Detection for Convex Polyhedra

The detection of collisions between solids (or closed surfaces) can be treated somewhat differently, for the objects have a distinguishable *inside* and *outside*. The problem is somewhat more complex than might be initially thought. Edges as well as vertex points may be involved in collisions.

This method for detecting collisions is based on the Cyrus - Beck clipping algorithm.[25] Collisions of articulated objects can be detected by applying this algorithm to all pairs of the polyhedra making up the two objects. The two polyhedra are assumed to be convex; concave polyhedra can be decomposed into collections of convex ones. The basic algorithm is $O(n^2m^2)$ for n polyhedra and m vertices per polyhedron. Methods for reducing these exponents are discussed below.

The two-dimensional Cyrus - Beck algorithm[25] tells whether a point is inside a convex polygon. It takes the dot product of each side's outward normal vector (**n**) with a vector from some point (**v**) on the side to the point in question (**p**). If that dot product is negative for all edges of the polygon, then the point is inside; if not, it is outside (see Figure 2).

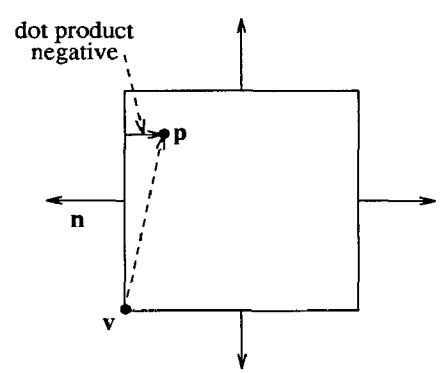


Figure 2
Cyrus - Beck Clipping

The collision detection algorithm is developed as a three-dimensional analogy to Cyrus - Beck clipping. The algorithm works by testing whether representative points of one polyhedron are inside the other polyhedron. First points from polyhedron B are tested against polyhedron A, and then the process is reversed and points from A are tested for inclusion in B. These two steps combine to cover all special cases and give a reliable answer. The algorithm given below terminates when a single point of interpenetration is found, which is sufficient for collision detection. If collision response is also required, the algorithm below should be modified to find all points of interpenetration. The rest of this section describes the test of points from B against A.

Let A consist of a set of planar polygonal faces (p_i). Each polygon contains a set of vertices (u_{ij}) and an outward pointing normal vector n_i. Let B consist of a set of vertices (v_i), a set of edges (e_i), and a set of planar polygonal faces (f_i). All coordinates of B have been transformed into the reference frame of A.

The first step tests for the presence of vertices of B inside of A. Each vertex of B is compared to every face of A; if any vertex is on the inward side of all such faces, it is inside A and the algorithm terminates having detected a collision. For each vertex i of B and for each face j of A, form the dot product $(v_i - u_{j1}) \cdot n_j$. If this dot product is negative the vertex is on the inward side of the face.

The second step tests for penetration of the edges of B through the faces of A. Each edge of B is divided into a number of smaller line segments by intersecting it with the infinite planes corresponding to every face of A. See Figure 3. This subdivision is done as follows. Let some edge of B connect the vertices v_i and v_j, and let us compare it against some face of A that has an outward pointing normal n_k and a vertex point u_{k1}. First the perpendicular distance of each vertex from the plane defining the face is calculated, by substitution into the plane equation.[25] If the perpendicular distances differ in sign, then the edge intersects the plane, and the intersection point P can be calculated.

$$d_i = (v_i - u_{k1}) \cdot n_k$$

$$d_j = (v_j - u_{k1}) \cdot n_k$$

$$t = \frac{|d_i|}{|d_i| + |d_j|}$$

$$P = v_i + t\,(v_j - v_i)$$

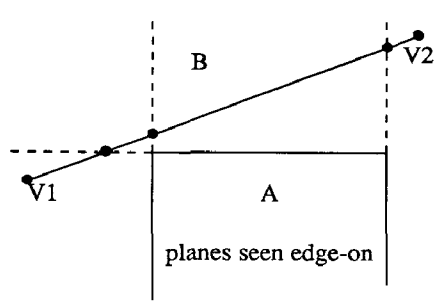


Figure 3
Edge Subdivision

This will result in a collection of intersection points P lying along the edge. Intersection points with $t < 0$ or $t > 1$ do not lie on the actual edge and are discarded. The remaining intersections are sorted into order according to their t values, forming a sequence of points from one vertex to the other along the edge. Each adjacent pair of points in this sequence, including those made by the vertices and the first and last subdivision points, defines a sub-segment of the edge. The midpoint of each resulting line segment is checked for being inside A by the same method that was used for vertices, above. Again, if any of these midpoints is inside A the algorithm terminates with a detected collision.

The third step tests for the infrequent case where two identical polyhedra are moving through each other with faces perfectly aligned. Here, the centroid point of each face of B is tested against A by the method used for vertices, above. If any of these centroids is inside A the algorithm terminates with a detected collision.

If the algorithm survives the above three steps without detecting a collision, and also does not detect one when reversing and comparing A against B, then the two polyhedra do not interpenetrate.

The above algorithm can be speeded up by a variety of tricks. A bounding box or bounding sphere test can be applied to every pair of polyhedra, yielding an immediate "no collision" result in most cases. Many of these bounding box tests can even be eliminated by octree or voxel methods. [4,9] When a point is to be tested against a polyhedron, it can first be compared to the polyhedron's bounding box, which will probably eliminate the need to compare it against all of the faces. The bounding box can be aligned with the coordinate axes of the polyhedron's local frame to make this point elimination test particularly fast.

It should be noted that this algorithm, or indeed any algorithm which point samples the positions of objects over time, could fail if one object moved entirely through another during a single time step. This is a rather unusual occurence in procedural or dynamic animation because simulation time steps are normally small relative to the velocities of the objects. The correct solution to this problem is to generalize to four dimensions;[3] the starting and ending positions of the polyhedra define 4-D hyper-polyhedra which are checked for interpenetration by higher-dimensional analogues to the algorithm given above. The more practical approach is either to ignore the problem (as we do) or to restrict the animation step size so that the change in any object's position in any step is small relative to the object's size.

3. COLLISION RESPONSE

In keyframed and procedural animation systems, collision detection is the main requirement; collision response usually consists of informing the animator or the motion control program that a collision has occurred, and trusting them to handle it. In animation systems using dynamics to generate motion, the system itself must respond to a collision by determining new linear and angular velocities for the colliding objects. These new velocities must conserve linear and angular momentum, or else the resulting "funny bounce" will be very obvious to viewers of the animation. The elasticity of the surfaces must also be taken into account, as this determines how much kinetic energy is lost in the collision; no-one will be-

lieve that a bean bag should bounce off of a hard surface as if it were a golf ball.

3.1. Collision Response Using Springs

The most intuitive way to handle collisions is with springs. Dynamic simulation systems must already have a method for applying external forces to objects. Thus, when a collision is detected, a very stiff spring is temporarily inserted between the points of closest approach (or deepest interpenetration) of the two objects. The spring law is usually K/d, or some other functional form that goes to infinity as the separation d of the two objects approaches 0 (or the interpenetration depth approaches some small value). K is a spring constant controlling the stiffness of the spring. The spring force is applied equally and in opposite directions to the two colliding objects. The direction of the force is such as to push the two objects apart (or to reduce their depth of interpenetration).

Our particular implementation handles variable elasticity by making a distinction between collisions where the objects are approaching each other and collisions where the objects are receding from each other. For $\varepsilon = 1$, i.e. perfectly elastic (hard) collisions, the spring constant K will be the same whether the objects are approaching or receding. For $\varepsilon = 0$, i.e. totally inelastic (soft) collisions, the spring will act as noted above as long as the objects are approaching each other, but as soon as they start to move apart the spring force will decrease to 0. For elasticities between 0 and 1, the two spring constants will be related by $K_{recede} = \varepsilon K_{approach}$.

The spring method is easy to understand and easy to program. It applies equally well to rigid bodies (articulated or not) and to flexible bodies, whether modeled as point masses connected by springs, or by energy of deformation techniques.[29] The main problem with this method is that it is computationally expensive; stiffer springs mean stiffer equations, which require smaller time steps for accurate numerical integration.[8] The numerical effort required goes up with the violence of the collision; as the springs are compressed more and more, the equations become stiffer and stiffer, and smaller and smaller time steps are needed. This was the motivation for seeking a better method of collision response.

3.2. Collision Response Using an Analytical Solution

An analytical solution for the collision of two arbitrary articulated rigid objects is available. The analytical solution depends upon the conservation of momentum during a collision, and results in a new angular and linear velocity for each body. Thus, the solution bypasses the question of collision forces and can be used independent of dynamic simulation, assuming information concerning the bodies mass and mass distribution can be provided.

Some combination of spring and analytical collision response may be desirable for a dynamic animation system. Analytical solutions are typically faster for strong collisions, because the solution need only be found once. However, for gentle collisions, such as a body resting quietly atop another body, springs may be desirable.[26] In such a case, gravity may consistently cause the two objects to interpenetrate and, thus, the analytical solution would have to be applied time and time again. A simple spring that counteracts gravity will be faster and more stable in this case.

This section develops the solution in stages. First, an analytical solution for the collision of two rigid bodies is presented; this result is due to MacMillan.[17] MacMillan's solution is extended to tree-like articulated rigid objects with revolute joints. Then, the restriction to tree-like objects is removed, and finally the method is extended to encompass joints with one or two sliding degrees of freedom.

3.2.1. Single Rigid Bodies

MacMillan gives a general solution for the collision of two arbitrary rigid objects. Each object has a linear velocity vector v_i, an angular velocity vector ω_i, a mass m_i, a center of mass vector c_i, and an inertial tensor matrix I_i which is relative to the center of mass. All of these quantities, for both objects, are expressed in the same inertial reference frame. In addition, each object has a vector ρ_i which points from its center of mass to the collision point. The solution also requires three orthogonal unit vectors i, j, k that define the "collision frame". k will be perpendicular to the plane of collision and i and j will be in that plane. The definition of the plane of collision is somewhat arbitrary; for convenience we will define it as follows. If a vertex of one object is colliding with a face of the other, then that face defines the plane of collision. If an edge of one object is colliding with an edge of the other, these two edges define the plane of collision. If two vertices are colliding, k is directed along the line joining them. See Figure 4.

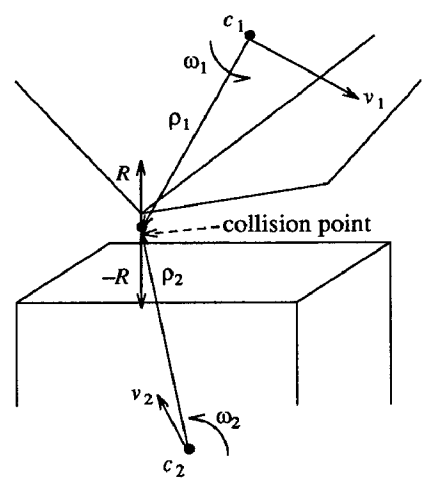


Figure 4.
Collision Problem - Two Rigid Objects

It is desirable to assume that there is only one collision point in any given collision; this restriction is not totally necessary, but it simplifies the formulations given below. It is reasonable to say that whenever two objects collide in the real world, there is one point at which they collide first (other collisions may follow within microseconds). Thus, the collision detection algorithm must furnish a single collision point between two objects. Because of the time-stepped nature of dynamics simulations, this will only be an approximate collision point; a good heuristic is to take the point of greatest interpenetration of any two objects in the simulation, provided that the relative velocities of the two objects at that common point are such that the interpenetration depth is increasing. If adaptive step size control is available, this heuristic can be

refined by stating that interpenetration to greater than some threshold depth is unacceptable, and causes backtracking and reduction of the step size. This allows the simulation to close in on a collision point very close to the surfaces of the objects by a process similar to binary search. Multiple collision points can be handled by a straightforward extension to the algorithms given below, by inventing multiple collision impulses and incorporating them into the matrix.

The solution involves solving a set of 15 linear equations in 15 unknowns. The fifteen unknowns are: the new linear velocity vector for each object ($\bar{v}_1$, $\bar{v}_2$); the new angular velocity vector for each object ($\bar{\omega}_1$, $\bar{\omega}_2$); and the impulse vector R. An impulse has units of momentum and can be thought of as a huge force applied for a tiny time. Because the collision is assumed to occur in a negligible time (approximately instantaneous), only the collision impulse itself matters; any other forces being applied to the objects will be too small to have an effect. By convention, the impulse is directed from object 2 to object 1.

Twelve linear equations can be written down immediately, expressing the change in linear and angular momentum that each object experiences as a result of the collision impulse R.

$$m_1\bar{v}_1 = m_1 v_1 + R$$
$$m_2\bar{v}_2 = m_2 v_2 - R$$
$$I_1\bar{\omega}_1 = I_1\omega_1 + \rho_1 \times R$$
$$I_2\bar{\omega}_2 = I_2\omega_2 - \rho_2 \times R$$

The last three linear equations come from some assumptions about the collision conditions; the assumptions that we will use are that the elasticity, ε, is zero (so that the two colliding objects come to rest relative to each other, at least at the collision point) and that the surfaces are frictionless (so that the impulse must be perpendicular to the collision plane). Other assumptions are possible and are discussed below. Our assumptions require the dot products of R with the collision frame unit vectors i and j to be zero, and the difference in the velocity of the collision point, as seen from each of the two objects, to be zero in the k direction. We can write:

$$R \cdot i = 0$$
$$R \cdot j = 0$$
$$(\bar{v}_2 + \bar{\omega}_2 \times \rho_2 - \bar{v}_1 - \bar{\omega}_1 \times \rho_1) \cdot k = 0$$

These equations can be solved by standard Gauss-Jordan elimination with maximal pivoting,[5] LU-decomposition,[22] or by more advanced sparse matrix methods.[20, 21] It is possible at this point in the algorithm to find the solution for an elastic collision. The actual elasticity of the collision can be taken as the lower of the elasticities of the two colliding surfaces. A new collision impulse R_{actual} can then be calculated as $R_{actual} = (1 + \varepsilon_{actual})R$. This new collision impulse is then plugged back into the defining equations above, to solve for the $\bar{v}_i$ and $\bar{\omega}_i$ vectors that are required. The $\bar{v}_i$ vectors come out easily; the $\bar{\omega}_i$ vectors require inverting the I_i inertial tensor matrices.

Next consider including friction. If the objects are infinitely rough and $\varepsilon = 0$, the collision condition requires that the objects come to rest (relative to each other) at the collision point. This corresponds to the vector equation:

$$\bar{v}_2 + \bar{\omega}_2 \times \rho_2 - \bar{v}_1 - \bar{\omega}_1 \times \rho_1 = 0$$

In between perfectly smooth and perfectly rough collisions lies the great middle ground of partially rough friction. Modeling partial (i.e. realistic) friction can become quite complex; the simple treatment given here is from MacMillan[17] and McLean[18] and is sufficient to produce visually reasonable results.

The coefficient of friction, γ, is the maximum allowed ratio of force parallel to the collision plane versus force perpendicular to that plane. Although properly speaking, γ is a property of pairs of surfaces, we assign a γ value to each surface, and then use the larger of the γ values of the colliding objects. When the two objects have finite γ and $\varepsilon = 0$, the collision can be solved in two steps. First the collision is solved as if it were infinitely rough. Then the resulting collision impulse, R, is examined. If the allowed ratio, γ, of the components of R parallel and perpendicular to the collision plane is not exceeded (i.e. if $\gamma R \cdot k \geq | R - k(R \cdot k) |$), all is well and the solution stands, because the objects should stick.

Otherwise, the objects should slide. The system of equations must be set up and solved again with different collision conditions. These new conditions will give a smaller restraining parallel force, because only a limited amount of friction can act against sliding motion. Two constants α and β are calculated, such that the collision impulse will exactly fulfill $\gamma R \cdot k = | R - k\,(R \cdot k) |$, or in other words such that the ratio of the parallel and perpendicular components of R is exactly γ, and the direction of the parallel component of R is the same as before. This gives the maximum parallel force allowed by the necessary perpendicular force and the coefficient of friction. The collision conditions are then:

$$R \cdot i = \alpha R \cdot k$$
$$R \cdot j = \beta R \cdot k$$
$$(\bar{v}_2 + \bar{\omega}_2 \times \rho_2 - \bar{v}_1 - \bar{\omega}_1 \times \rho_1) \cdot k = 0$$

α and β are calculated as follows, with Q the component of R perpendicular to the collision plane, and P the unit direction vector of the component of R parallel to the collision plane:

$$Q = k\,(R \cdot k)$$
$$P = \frac{R - Q}{| R - Q |}$$
$$\alpha = \gamma(P \cdot i)$$
$$\beta = \gamma(P \cdot j)$$

To reiterate, the full algorithm for solving a general collision of two rigid objects is to transform the required quantities (incoming velocities, tensor matrices, ρ_i, etc) from the objects' local coordinate frames to a common inertial frame, define the collision frame's orthogonal unit vectors i, j, and k, choose appropriate collision conditions, set up and solve the system of equations as outlined above, and transform the new linear and angular velocities back to the objects' local frames. This may seem like a drastic amount of work when compared with inserting a simple spring between the two objects, and it does require more lines of computer code, but this method is usually applied only once for each collision, whereas springs

generally must be applied over a large number of very small time steps. This analytical method is cheaper computationally unless the collision is very gentle indeed, and the cost of this collision solution does not depend upon the violence of the collision, certainly a desirable property.

3.2.2. Articulated Rigid Bodies - Tree-Structured, Revolute Joints

Now we extend MacMillan's solution to tree-like articulated rigid objects with revolute joints. The various rigid objects that make up the tree-like linkages will be numbered from 1 to n. Objects 1 and 2 will be the colliding objects, and the rest will be linked to one or both of them, either directly or through some number of intermediaries. Note that this solution allows the links of an articulated object to collide with other links of the same object or with another object entirely. Each rigid object will again have a linear velocity vector v_i, an angular velocity vector ω_i, an inertial tensor matrix I_i, a mass m_i, and a center of mass c_i, all expressed in a common inertial reference frame.

The revolute joints connecting the various rigid objects will be assumed to be ideal, that is, perfectly elastic and with no mechanical tolerance. The single joint that connects object i to object j will be described by the vector ρ_{ij} that points from c_i to the joint, and the vector ρ_{ji} that points from c_j to the joint. As well as the collision impulse R, this solution will calculate an attachment impulse R_{ij} for each joint. Unlike the collision impulse, the attachment impulses R_{ij} are unconstrained as to direction. By convention, the attachment impulse R_{ij} points from object j to object i, and $R_{ij} = -R_{ji}$. R_{ij} will be (0,0,0) if objects i and j are not connected by a joint. See Figure 5.

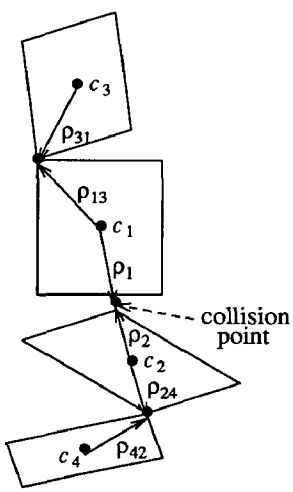


Figure 5.
Articulated Collision Problem

For a collision involving n rigid objects there are $6n$ unknowns corresponding to the resulting linear and angular velocities of the objects, 3 unknowns for the collision impulse, and either $3(n-1)$ unknowns corresponding to the attachment impulses if the objects are all part of one articulated linkage, or $3(n-2)$ unknowns if two different articulated objects are colliding. Thus, the total size of the linear system to be solved is approximately $9n$ for n rigid objects involved in the collision. The sparsity of the matrix increases as n increases, so that if sparse matrix methods are used the solution should be around $O(n)$.[20, 21]

Once again, the unknowns to be solved for are $\bar{v}_i$ and $\bar{\omega}_i$ for $i = 1...n$, R, and R_{ij} for all pairs of objects i and j connected by a joint. The equations for objects 1 and 2, and for the collision impulse, still look familiar. The extra summation terms reflect the change in linear and angular momentum resulting from any attachment impulses felt by those objects.

$$m_1\bar{v}_1 = m_1 v_1 + R + \sum_{i=1}^{n} R_{1i}$$

$$m_2\bar{v}_2 = m_2 v_2 - R + \sum_{i=1}^{n} R_{2i}$$

$$I_1\bar{\omega}_1 = I_1\omega_1 + \rho_1 \times R + \sum_{i=1}^{n} \rho_{1i} \times R_{1i}$$

$$I_2\bar{\omega}_2 = I_2\omega_2 - \rho_2 \times R + \sum_{i=1}^{n} \rho_{2i} \times R_{2i}$$

The conditions on the collision impulse R are still the same.

$$R \cdot i = 0$$

$$R \cdot j = 0$$

$$(\bar{v}_2 + \bar{\omega}_2 \times \rho_2 - \bar{v}_1 - \bar{\omega}_1 \times \rho_1) \cdot k = 0$$

For the objects that are not directly colliding (for object $i = 3..n$), the momentum conservation equations are

$$m_i\bar{v}_i = m_i v_i + \sum_{j=1}^{n} R_{ij}$$

$$I_i\bar{\omega}_i = I_i\omega_i + \sum_{j=1}^{n} \rho_{ij} \times R_{ij}$$

Each joint requires three more linear equations to make the system of equations complete and solvable. These are derived from the basic requirement of a revolute joint: the velocity of the joining point, when seen as part of either of the rigid objects which it connects, must be the same. Otherwise, the joint would tend to pull apart. For each joint connecting objects i and j, three more equations can be written.

$$\bar{v}_i + \bar{\omega}_i \times \rho_{ij} = \bar{v}_j + \bar{\omega}_j \times \rho_{ji}$$

Once again, the algorithm requires that the necessary information about all of the objects be transformed from their local reference frames to a common inertial reference frame. The collision frame orthogonal unit vectors i, j, and k must be determined. The (potentially rather large) linear system is set up and solved for the variables $\bar{v}_i$, $\bar{\omega}_i$, R, and R_{ij}, by standard methods.[5, 22, 20, 21] The actual elasticity of the collision is determined as above, and actual impulses are determined by multiplying R and the R_{ij}'s by $(1 + \varepsilon_{actual})$. The actual impulses are then put back into the equations above to get the final solution for linear and angular velocities. The last step is to transform the solution back to the object's local frames.

3.2.3. Articulated Rigid Bodies - Revolute Joints

The above solution for tree-like articulated rigid objects can be extended by removing the requirement for tree-like linkage. Since the two articulated objects that are colliding are defined to be connected objects, some subset of the attachment points will define tree-like linkages. The first step is to set up the problem as above for the objects and for those joints. Then the extra joints are added; each contributes another attachment impulse R_{ij}, and thus adds three variables to the problem. Each joint also allows three more linear equations to be written down, the familiar velocity matching condition.

$$\bar{v}_i + \bar{\omega}_i \times \rho_{ij} = \bar{v}_j + \bar{\omega}_j \times \rho_{ji}$$

This larger system once again contains as many variables as equations, and so can be solved by standard techniques. The actual collision and attachment impulses are calculated and applied in the same way as above, and the solution values are transformed back to the objects' local coordinate systems. It is even possible to have more than one joint connecting two objects i and j, although in that case the notation used above would have to be expanded slightly. If two joints connect objects i and j, the objects will have only one degree of freedom of motion relative to each other; they will be able to twist around the line connecting the two joints. If a third joint is added connecting i and j, which is not colinear with the other two, the two objects will be locked in position relative to each other, and will form in effect a single rigid object. This is probably not a desirable state of affairs, but the solution algorithm does permit it.

3.2.4. Articulated Bodies - Sliding Joints

A sliding joint is one in which the joining points on the two objects are allowed to move freely with respect to each other in one or two dimensions, but are at the same time constrained to a fixed relationship in the other dimensions. A linear sliding joint allows sliding motion in one degree of freedom, while controlling two others; a planar sliding joint controls one degree of freedom, allowing sliding motion in the other two. For now, assume that these joints allow three revolute degrees of freedom as well.

For a linear sliding joint, assume that objects i and j are connected, and that a joint coordinate system is defined by three orthogonal unit vectors d_i, d_j, and d_k. These vectors are stated in the local coordinate system of object i and rotate with it. Further assume that the attachment point on object j is allowed to move freely in the d_i direction, but must maintain some particular value for its position along d_j and d_k as seen from object i. Note that the "attachment point" on object i is in fact a line; this requires us to calculate ρ_{ij}, the actual attachment point on object i, separately for each collision. Note also that the joint coordinate system orthogonal unit vectors must be rotated into the common inertial frame.

Notice that the attachment impulse R_{ij} is required to lie in the d_j, d_k plane. As it is possible to write down three linear constraining equations about this joint, such a joint can be treated within the linear systems described above. One equation expresses the constraint that R_{ij} must lie in a plane; the other two equations constrain the attachment point velocities to match in two of the three joint coordinate system directions.

$$R_{ij} \cdot d_i = 0$$
$$(\bar{v}_j + \bar{\omega}_j \times \rho_{ji} - \bar{v}_i - \bar{\omega}_j \times \rho_{ij}) \cdot d_j = 0$$
$$(\bar{v}_j + \bar{\omega}_j \times \rho_{ji} - \bar{v}_i - \bar{\omega}_j \times \rho_{ij}) \cdot d_k = 0$$

The argument for a planar sliding joint is similar. In this case the attachment point on object j is allowed to move freely in the d_i and d_j directions, constraining the collision impulse to exactly the d_k direction, but leaving its magnitude unknown. The attachment point velocities must still match in the d_k direction, but are allowed to vary in the other two directions. The equations are as follows.

$$R_{ij} \cdot d_i = 0$$
$$R_{ij} \cdot d_j = 0$$
$$(\bar{v}_j + \bar{\omega}_j \times \rho_{ji} - \bar{v}_i - \bar{\omega}_j \times \rho_{ij}) \cdot d_k = 0$$

The sliding joints described above allow the two objects that they connect either 4 (linear) or 5 (planar) degrees of freedom of movement relative to each other. Sliding joints that provide fewer degrees of freedom can be constructed by adding one or more extra joints of the above types. For instance, suppose that a planar sliding joint is desired such that one object can slide relative to the other, but the objects cannot rotate relative to each other. This can be accomplished by defining three planar sliding joints (all using the same plane) with the sliding points not colinear. A piston type joint (one degree of translational freedom and one degree of rotational freedom) can be described by two linear sliding joints of the above type, with the connection points constrained to slide along the same line.

3.3. Collisions of Dynamic Objects with Non-Dynamic Objects

A complication seems to arise when dynamically controlled objects collide with objects that are controlled in other ways (such as keyframe interpolation). In these cases the velocities of the non-dynamic objects involved in the collision cannot change. Thus, $\bar{v}_i = v_i$ and $\bar{\omega}_i = \omega_i$ for the objects that are not under dynamic control, and $\bar{v}_i$ and $\bar{\omega}_i$ are not variables in the linear system formulations above. The systems can be reformulated with fewer rows and columns, and the solution proceeds just as before. The result is collisions that do not conserve linear or angular momentum. The keyframe or procedurally controlled objects move along their assigned paths with lordly disdain, brushing aside the dynamically controlled objects as if they had negligible mass.

More complex responses from the non-dynamic objects are possible. Programs that control objects could be written to take the results of dynamic collisions into account. In effect, the procedural object could become dynamic for the duration of the collision, and its velocity could change. The program would have to be alert to this possibility. This would be fairly simple to implement using the analytic solution, which does not require setting up and solving the complete dynamics equations of motion. Alternatively, collisions between dynamic and keyframed objects could be defined as exceptional events that require that the human animator be notified.

4. CONCLUSIONS

Collision detection is important for any animation system. The coding requirements are not excessive, and, while a naive approach to collision detection can consume large amounts of computer time, several tricks are available to keep the cost reasonable.

Dynamical simulation systems must resolve collisions after detecting them. The obvious method of inserting temporary springs is general and easy to program, but exacts a severe execution time penalty, particularly for violent collisions. This makes an analytical collision resolution algorithm attractive. On the other hand, for objects resting against each other but encouraged by forces to interpenetrate, the spring solution is more appropriate. A dynamical simulation system should have a combination of both methods available.

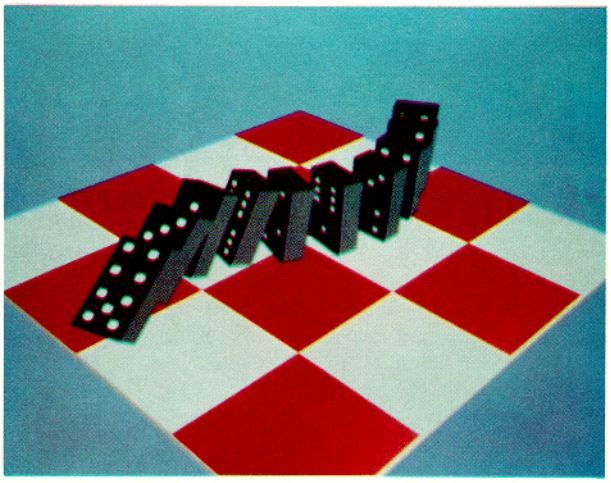
ACKNOWLEDGEMENTS

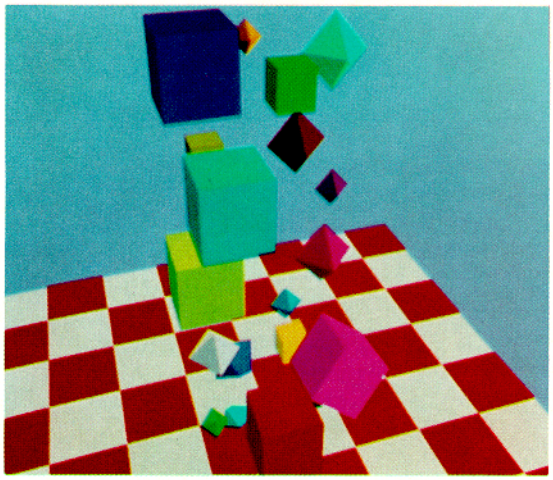
This work was supported by National Science Foundation grant number CCR-8606519. We wish to thank Robert Skinner, David Forsey, and Peter Valtin for contributing to the dynamical animation software that we used to implement these algorithms. We would also like to thank our reviewers for their incisive and helpful comments and references.

References

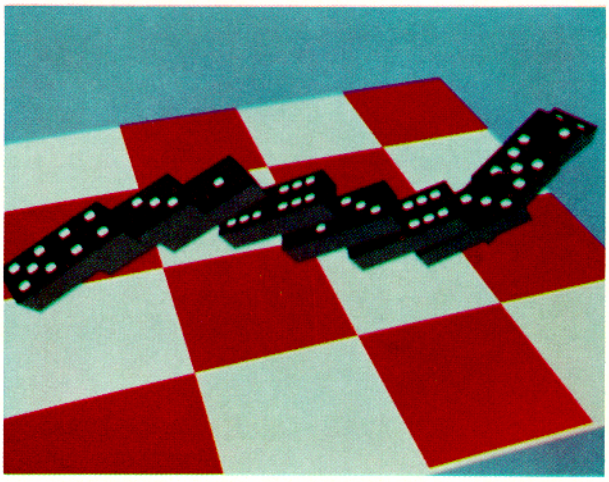
1. William W. Armstrong and Mark W. Green, "The Dynamics of Articulated Rigid Bodies for Purposes of Animation," *Proceedings of Graphics Interface '85*, pp. 407-415, Canadian Information Processing Society, Toronto, Ontario, Canada, May 1985.
2. John W. Boyse, "Interference Detection Among Solids and Surfaces," *Communications of the ACM*, vol. 22:1, pp. 3-9, January, 1979.
3. John Canny, "Collision Detection for Moving Polyhedra," *MIT A.I. Lab Memo 806*, October, 1984.
4. Ingrid Carlbom, "An Algorithm for Geometric Set Operations Using Cellular Subdivision Techniques," *IEEE Computer Graphics and Applications*, vol. 7, pp. 44-55, Computer Society of the IEEE, Los Alamitos, CA, May 1987.
5. Brice Carnahan and James O. Wilkes, *Digital Computing and Numerical Methods*, John Wiley and Sons, Inc., New York, 1973.
6. Scott E. Fahlman, "A Planning System for Robot Construction Tasks," *Artificial Intelligence*, vol. 5, pp. 1-49, 1974.
7. Wm. Randolph Franklin, "Efficient Polyhedron Intersection and Union," *Proceedings of Graphics Interface 1982*, pp. 73-80, 1982.
8. C. William Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1971.
9. Jeffrey Goldsmith and John Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, vol. 7, pp. 14-20, Computer Society of the IEEE, Los Alamitos, CA, May 1987.
10. I. N. Herstein, *Topics in Algebra*, Xerox College Publishing, Lexington, MA, 1964.
11. J.E. Hopcroft, J.T. Schwartz, and M. Sharir, "Efficient Detection of Intersections among Spheres," *The International Journal of Robotics Research*, vol. 2:4, pp. 77-80, Winter 1983.
12. Paul M. Isaacs and Michael F. Cohen, "Controlling Dynamic Simulation with Kinematic Constraints," *Computer Graphics*, vol. 21, no. 4. Proceedings of SIGGRAPH'87 (Anaheim, CA, July 27-31, 1987)
13. Donald Knuth, *Fundamental Algorithms*, Addison-Wesley Publishing Co., Reading, MA, 1975.
14. Donald Knuth, *Searching and Sorting*, Addison-Wesley Publishing Co., Reading, MA, 1975.
15. Tomas Lozano-Perez and Michael A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Communications of ACM*, vol. 22, no. 10, pp. 560-570, October, 1979.
16. Richard V. Lundin, "Motion Simulation," *Proceedings of Nicograph 1984*, pp. 2-10, November, 1984.
17. William D. MacMillan, *Dynamics of Rigid Bodies*, Dover Publications, Inc, New York, 1936.
18. W. G. McLean and E. W. Nelson, *Engineering Mechanics: Statics and Dynamics*, Shaum's Outline Series, McGraw-Hill Book Co., New York, 1978.
19. Matthew Moore, "A Flexible Object Animation System," Masters Thesis, University of California, Santa Cruz, Computer & Information Sciences, Santa Cruz, California, March, 1988.
20. Ole Osterby and Zahari Zlatev, *Direct Methods for Sparse Matrices*, Springer-Verlag, Berlin, 1983.
21. Sergio Pissanetsky, *Sparse Matrix Technology*, Academic Press, London, 1984.
22. William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical Recipes*, Cambridge University Press, Cambridge, England, 1986.
23. Craig W. Reynolds, "Computer Animation with Scripts and Actors," *Computer Graphics*, vol. 16, no. 4, pp. 289-296, Association for Computing Machinery, July, 1982. Proceedings of SIGGRAPH'82
24. Craig W. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics*, vol. 21, no. 4, pp. 25-34, Association for Computing Machinery. Proceedings of SIGGRAPH'87 (Anaheim, CA, July 27-31, 1987)
25. David F. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, New York, 1985.
26. Robert Skinner, U Cal. Santa Cruz, CIS Dept. personal communication.
27. Scott N. Steketee and Norman I. Badler, "Parametric Keyframe Interpolation Incorporating Kinetic Adjustment and Phrasing Control," *Proceedings of SIGGRAPH '85*, vol. 19, no. 4, pp. 255-262, July, 1985.
28. David Sturman, *A Discussion on the Development of Motion Control Systems*, Association for Computing Machinery, July 1987. SIGgraph '87 Course 10 Notes: Computer Animation: 3-D Motion Specification and Control.

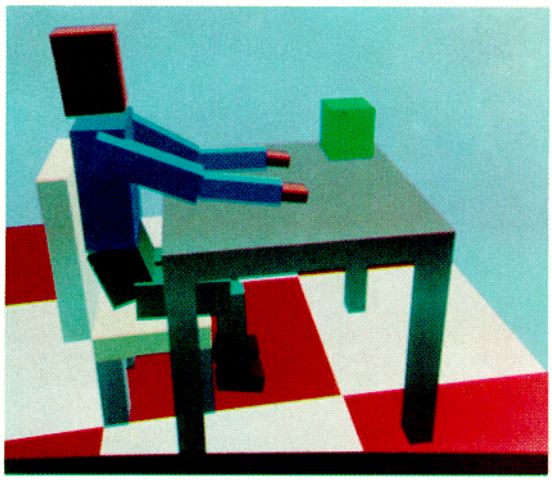
29. Demetri Terzopoulous, John Platt, Alan H. Barr, and Kurt Fleischer, "Elastically Deformable Models," *Computer Graphics*, vol. 21, no. 4. Proceedings of SIGGRAPH'87 (Anaheim, CA, July 27-31, 1987)
30. Tetsuya Uchiki, Toshiaki Ohashi, and Mario Tokoro, "Collision Detection in Motion Simulation," *Computers & Graphics*, vol. 7:3-4, pp. 285-293, 1983.
31. Jane Wilhelms, "Towards Automatic Motion Control," *IEEE Computer Graphics and Animation April, 1987*, vol. 7, no. 4, pp. 11-22, April, 1987.
32. Jane Wilhelms, "Using Dynamic Analysis for Animation of Articulated Bodies," *IEEE Computer Graphics and Applications*, vol. 7, no. 6, June, 1987.



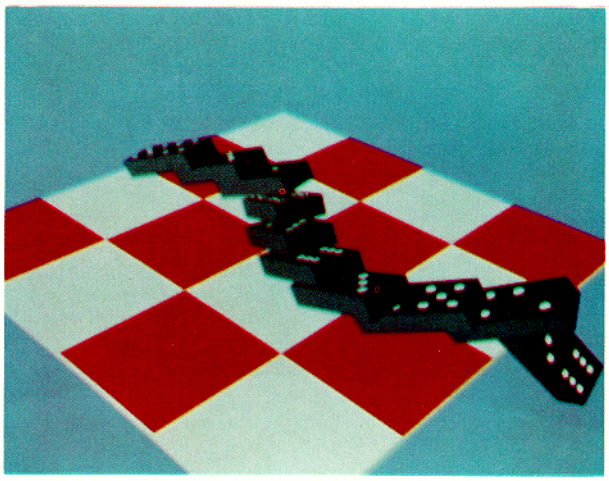


Example 1: early stage rockpile





Example 2: man sitting
positioned with collision detection



Examples 3-5: falling domino movie