CHAPTER

Physically Based Animation 7

Animators are often more concerned with the general quality of motion than with precisely controlling the position and orientation of each object in each frame. Such is the case with physically based animation. In *physically based animation*, forces are typically used to maintain relationships among geometric elements. Note that these forces may or may not be physically accurate. Animation is not necessarily concerned with accuracy, but it is concerned with believability, which is sometimes referred to as being *physically realistic*. Some forces may not relate to physics at all—they may model constraints that an animator may wish to enforce on the motion, such as directing a ball to go toward making contact with a bat. For the purposes of this chapter, the use of forces to control motion will be referred to as physically based animation whether or not the forces are actually trying to model some aspect of physics.

When modeling motion that is produced by an underlying mechanism, such as the principles of physics, a decision has to be made concerning the level at which to model the process. For example, wrinkles in a cloth can be modeled by trying to characterize common types of wrinkles and where they typically occur on a piece of cloth. On the other hand, the physics (stresses and strains) of the individual threads of cloth could be modeled in sufficient detail to give rise to naturally occurring wrinkles. Modeling the surface characteristics of the process, as in the former case, is usually computationally less expensive and easier to program, but it lacks flexibility. The only effects produced are the ones explicitly modeled. Modeling the physics of the underlying process usually incurs a higher computational expense but is more flexible—the inputs to the underlying process can usually produce a greater range of surface characteristics under a wider range of conditions. There are often several different levels at which a process can be modeled. In the case of the cloth example, a triangular mesh can be used to model the cloth as a sheet of material (which is what is usually done). The quest for the animation programmer is to provide a procedure that does as little computation as possible while still providing the desired effects and controls for the animator.

The advantage to the animator, when physical models are used, is that (s)he is relieved of low-level specifications of motions and only needs to be concerned with specifying high-level relationships or qualities of the motion. For our example of cloth, instead of specifying where wrinkles might go and the movement of individual vertices, the animator would merely be able to set parameters that indicate the type and thickness of the cloth material. For this, the animator trades off having specific control of the animation—wrinkles will occur wherever the process model produces them.

For the most part, this chapter is concerned with *dynamic control*—specifying the underlying forces that are then used to produce movement. First, a review of basic physics is given, followed by two common examples of physical simulation used in animation: spring meshes and particle systems. Rigid body dynamics and the use of constraints are then discussed.

7.1 Basic physics—a review

The physics most useful to animators is that found in high school text books. It is the physics based on Newton's laws of motion. The basics will be reviewed here; more in-depth discussion of specific animation techniques follows. More physics can be found in Appendix B.7 as well as any standard physics text (e.g., [6]).

The most basic (and most useful) equation relates force to the acceleration of a mass, Equation 7.1.

$$\boldsymbol{f} = m\boldsymbol{a} \tag{7.1}$$

When setting up a physically based animation, the animator will specify all of the possible forces acting in that environment. In calculating a frame of the animation, each object will be considered and all the forces acting on that object will be determined. Once that is done, the object's linear acceleration (ignoring rotational dynamics for now) can be calculated based on Equation 7.1 using Equation 7.2.

$$\boldsymbol{a} = \frac{\boldsymbol{f}}{m} \tag{7.2}$$

From the object's current velocity, v, and newly determined acceleration, a new velocity, v', can be determined as in Equation 7.3.

$$\boldsymbol{v}' = \boldsymbol{v} + \boldsymbol{a}\Delta t \tag{7.3}$$

The object's new position, p', can be updated from its old position, p, by the average of its old velocity and its new velocity.[1]

$$\boldsymbol{p}' = \boldsymbol{p} + \frac{1}{2}(\boldsymbol{v} + \boldsymbol{v}')\Delta t \tag{7.4}$$

Notice that these are vector-valued equations; force, acceleration, and velocity are all vectors (indicated in bold in these equations).

There are a variety of forces that might be considered in an animation. Typically, forces are applied to an object because of its velocity and/or position in space.

The gravitational force, **f**, between two objects at a given distance, d, can be calculated if their masses are known (G is the gravitation constant of $6.67\ 3\ 10^{-11}\mathrm{m}^3\mathrm{s}^{-2}\,\mathrm{kg}^{-1}$). This magnitude of the force, f, is applied along the direction between the two objects (Eq. 7.5).

$$f = \| \mathrm{f} \| = G\frac{m_1 m_2}{d^2} \tag{7.5}$$

If the earth is one of the objects, then Equations 7.1 and 7.5 can be simplified by estimating the distance to be the radius of the earth (r_e), using the mass of the earth (m_e), and canceling out the mass of the other object, m_o. This produces the magnitude of the gravitation acceleration, a_e, and is directed downward (see Eq. 7.6).

$$a_e = \frac{f}{m_o} = G\frac{m_e}{r_e^2} = 9.8\frac{meter}{sec^2} \tag{7.6}$$

[1]Better alternatives for updating these values will be discussed later in the chapter, but these equations should be familiar to the reader.

A spring is a common tool for modeling flexible objects, to keep two objects at a prescribed distance, or to insert temporary control forces into an environment. Notice that a spring may correspond to a geometric element, such as a thread of a cloth, or it may be defined simply to maintain a desired relationship between two points (sometimes called a *virtual spring*), such as a distance between the centers of two spheres. When attached to an object, a spring imparts a force on that object depending on the object's location relative to the other end of the spring. The magnitude of the spring force (f_s) is proportional to the difference between the spring's rest length (L_r) and its current length (L_c) and is in the direction of the spring. The constant of proportionality (k_s), also called the *spring constant*, determines how much the spring reacts to a change in length—its *stiffness* (see Eq. 7.7). The force is applied to both ends of the spring in opposite directions.

$$f_s = -k_s(L_c - L_r)\left(\frac{p_2 - p_1}{\|p_2 - p_1\|}\right) \tag{7.7}$$

The spring described so far is a *linear spring* because there is a linear relationship between how much the length of the spring deviates from the rest length and the resulting force. In some cases, it is useful to use something other than a linear spring. A *biphasic spring* is a spring that is essentially a linear spring that changes its spring constant at a certain length. Usually a biphasic spring is used to make the spring stiffer after it has stretched past some threshold and has found uses, for example, in facial animation and cloth animation. Other types of *nonlinear springs* have spring "constants" that are more complex functions of spring length.

A *damper*, like a spring, is attached to two points. However, the damper works against its relative velocity. The force of a damper (f_d) is negatively proportional to, and in the direction of, the velocity of the spring length (v_s). The constant of proportionality (k_d), or damper constant, determines how much resistance there is to a change in spring length. So the force at p_2 is computed as in Equation 7.8. The force on p_1 would be the negative of the force on p_2.

$$f_d = -k_d\left(\dot{p}_2 - \dot{p}_1\right)\cdot\left(\frac{p_2 - p_1}{\|p_2 - p_1\|}\right)\left(\frac{p_2 - p_1}{\|p_2 - p_1\|}\right) \tag{7.8}$$

Viscosity, similar to a damper, resists an object traveling at a velocity. However, viscosity is resisting movement through a medium, such as air. A force due to viscosity (f_v) is negatively proportional to the velocity of the object (v) (see Eq. 7.9).

$$f_v = -k_v v \tag{7.9}$$

Momentum is mass × velocity. In a closed system (i.e., a system in which energy is not changing), total momentum is conserved. This can be used to solve for unknowns in the system. For a system consisting of multiple objects in which there are no externally applied forces, the total momentum is constant (Eq. 7.10).

$$\sum m_i \mathrm{v}_i = \mathrm{c} \tag{7.10}$$

The rotational equivalent to force is *torque*. Think of torque, τ, as rotational force. Analogous to (linear) velocity and (linear) acceleration are *angular velocity*, ω, and *angular acceleration*, α. The mass of an object is a measure of its resistance to movement. Analogously, the *moment of inertia* is a measure of an object's resistance to change in orientation. The moment of inertia of an object is a 3 × 3 matrix of values, I, describing an object's distribution of mass about its center of mass. The relationship to torque, moment of inertia, and angular acceleration is similar to the familiar $f = ma$ (see Eq. 7.11).

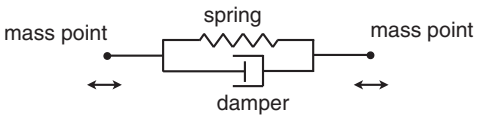


FIGURE 7.1

Spring-damper configuration.

$$\tau = I\alpha \tag{7.11}$$

More physics will be introduced as various topics are covered, but this should give the reader a starting point if physics is only a vague recollection.

7.1.1 Spring-damper pair

One of the most useful tools to incorporate some use of forces into an animation is the use of a spring-damper pair as in Figure 7.1. Consider a mass point, p_1, connected by a spring to a fixed point, p_2, with rest length L_r and current Length L_c. the resulting force on p_1 would be computed as in Equation 7.12. As mentioned before, the spring represents a force to maintain a relationship between two points or insert a control force into the system. The damper is used to restrict the motion and keep the system from reacting too violently.

$$\mathrm{f} = \left(k_s(\mathrm{L}_c - \mathrm{L}_r) - k_d\left(\dot{\mathrm{p}}_2 - \dot{\mathrm{p}}_1\right)\cdot\left(\frac{\mathrm{p}_2 - \mathrm{p}_1}{\|\mathrm{p}_2 - \mathrm{p}_1\|}\right)\right)\left(\frac{\mathrm{p}_2 - \mathrm{p}_1}{\|\mathrm{p}_2 - \mathrm{p}_1\|}\right) \tag{7.12}$$

In physical simulations, the challenge in using a spring-damper system is in selecting an appropriate time step, setting the spring and damper constants, and setting the mass of the object so that the resulting motion is desirable. Some combination of too small a time step, a large mass, a high damping constant, and a low spring constant might result in a system that is not very lively, where the object slowly moves toward the fixed point going slower and slower. On the other hand, a larger time step, a smaller mass, a lower damping constant, and a higher spring constant might result in a system that moves too much, resulting in a mass that oscillates around the fixed point, getting further and further away from it each time step. Balancing these parameters to get the desired effect can be tricky.

7.2 Spring animation examples

7.2.1 Flexible objects

In Chapter 4, kinematic approaches to animating flexible bodies are discussed in which the animator is responsible for defining the source and target shapes as well as implying how the shapes are to be interpolated. Various physically based approaches have been proposed that model elastic and inelastic behavior, viscoelasticity, plasticity, and fracture (e.g., [11] [13] [17] [23] [24] [25]). Here, the simplest approach is presented. Flexibility is modeled by a mass-spring-damper system simulating the reaction of the body to external forces.

Mass-spring-damper modeling of flexible objects

Probably the most common approach to modeling flexible shapes is the mass-spring-damper model. The most straightforward strategy is to model each vertex of an object as a point mass and each edge of the object as a spring. Each spring's rest length is set equal to the original length of the edge. A mass

can be arbitrarily assigned to an object by the animator and the mass evenly distributed among the object's vertices. If there is an uneven distribution of vertices throughout the object definition, then masses can be assigned to vertices in an attempt to more evenly distribute the mass. Spring constants are similarly arbitrary and are usually assigned uniformly throughout the object to some user-specified value. It is most common to pair a damper with each spring to better control the resulting motion at the expense of dealing with an extra set of constants for the dampers. To simplify the following discussion and diagrams, the spring-damper pairs will be referred to simply as springs with the understanding that there is usually an associated damper.

As external forces are applied to specific vertices of the object, either because of collisions, gravity, wind, or explicitly scripted forces, vertices will be displaced relative to other vertices of the object. This displacement will induce spring forces, which will impart forces to the adjacent vertices as well as reactive forces back to the initial vertex. These forces will result in further displacements, which will induce more spring forces throughout the object, resulting in more displacements, and so on. The result will be an object that is wriggling and jiggling as a result of the forces propagating along the edge springs and producing constant relative displacements of the vertices.

One of the drawbacks with using springs to model the effects of external forces on objects is that the effect has to propagate through the object, one time step at a time. This means that the number of vertices used to model an object and the length of edges used have an effect on the object's reaction to forces. Because the vertices carry the mass of the object, using a different distribution of vertices to describe the same object will result in a difference in the way the object reacts to external forces.

A simple example

In a simple two-dimensional example, an equilateral triangle composed of three vertices and three edges with uniform mass distribution and uniform spring constants will react to an initial external force applied to one of the vertices (Figure 7.2).

In the example, an external force is applied to vertex V_2, pushing it generally toward the other two vertices. Assume that the force is momentary and is applied only for one time step during the animation. At the application of the force, an acceleration ($a_2 = F/m_2$) is imparted to vertex V_2 by the force. The acceleration gives rise to a velocity at point V_2, which in turn creates a change in its position. At the next time step, the external force has already been removed, but, because vertex V_2 has been displaced, the lengths of edges E_{12} and E_{23} have been changed. As a result of this, a spring force is created along the two edges. The spring that models edge E_{12} imparts a restoring force to vertices V_1 and V_2, while the

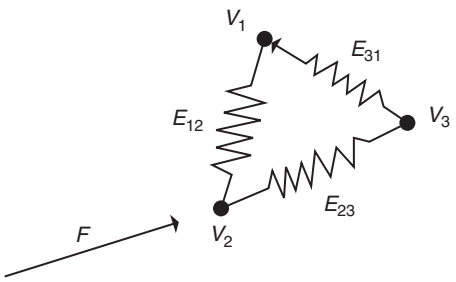


FIGURE 7.2

A simple spring-mass model of a flexible object.

spring that models edge E_{23} imparts a restoring force to vertices V_2 and V_3. Notice that the springs push back against the movement of V_2, trying to restore its position, while at the same time they are pushing the other two vertices away from V_2. In a polygonal mesh, the springs associated with mesh edges propagate the effect of the force throughout the object.

If the only springs (or spring-dampers) used to model an object are ones associated with the object edges, the model can have more than one stable configuration. For example, if a cube's edges are modeled with springs, during applications of extreme external forces, the cube can turn inside out. Additional spring dampers can help to stabilize the shape of an object. Springs can be added across the object's faces and across its volume. To help prevent such undesirable, but stable, configurations in a cube, one or more springs that stretch diagonally from one vertex across the interior of the cube to the opposite vertex can be included in the model. These springs will help to model the internal material of a solid object (Figure 7.3).

If specific angles between adjacent faces (dihedral angles) are desired, then angular springs (and dampers) can be applied. The spring resists deviation to the rest angle between faces and imparts a torque along the edge that attempts to restore the rest angle (Eq. 7.13 and Figure 7.4). The damper works to limit the resulting motion. The torque is applied to the vertex from which the dihedral angle is measured; a torque of equal magnitude but opposite direction is applied to the other vertex.

$$\hat{\tau} = k_s(\theta(t) - \theta_r) - k_d\dot{\theta}(t) \tag{7.13}$$

Alternatively, a linear spring could be defined between points A and B in Figure 7.4 in order to maintain the appropriate angle at the shared edge.

Depending on the size of forces applied to the spring's mass points, the size of the spring constant (and damper constant), and the size of the time step used to sample the system, a spring simulation may

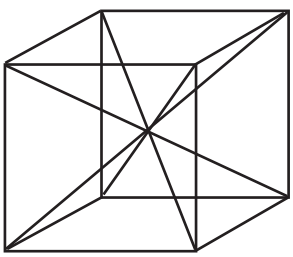


FIGURE 7.3

Interior springs to help stabilize the object's configuration.

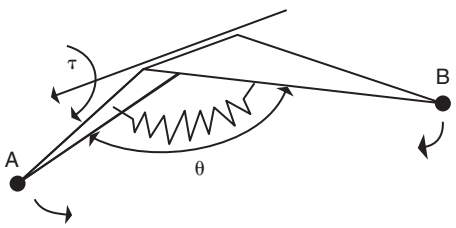


FIGURE 7.4

An angular spring imparting restoring torques.

numerically explode, resulting in computed behavior in which the motions get larger and larger over the course of time. This is because the force is (wrongly) assumed to be constant throughout the time step while, in fact, it is constantly changing as the spring length changes as it progresses from one time sample to the next. As mentioned before, modifying spring, damper, mass, and time step values can be adjusted to help control the simulation, but finding appropriate values can be tricky. Alternatively (or additionally) various simulation magnitudes, such as forces, accelerations, or velocities, can be clamped at maximum values to help control the motion.

7.2.2 Virtual springs

Virtual springs introduce forces into the system that do not directly model physical elements. These can be used to control the motion of objects in a variety of circumstances. In the previous section, it was suggested that virtual springs can be used to help maintain the shape of an object. Virtual springs with zero rest lengths can be used to constrain one object to lie on the surface of another, for example, or, with non-zero rest lengths, to maintain separation between moving objects.

Proportional derivative controllers (PDCs) are another type of virtual spring used to keep a control variable and its derivative within the neighborhood of desired values. For example, to maintain a joint angle and joint velocity close to desired values, the user can introduce a torque into a system. If the desired values are functions of time, PDCs are useful for biasing a model toward a given motion while still allowing it to react to system forces, as in Equation 7.14.

$$\tau = k_s(\theta(t) - \theta_d(t)) - k_d\left(\dot{\theta}(t) - \dot{\theta}_d(t)\right) \tag{7.14}$$

Springs, dampers, and PDCs are an easy and handy way to control the motion of objects by incorporating forces into a physically based modeling system. The springs can model physical elements in the system as well as represent arbitrary control forces. Dampers and PDCs are commonly used to keep system parameters close to desired values. They are not without problems, however. One drawback to using springs, dampers, and PDCs is that the user-supplied constants in the equations can be difficult to choose to obtain a desired effect. A drawback to using a spring mesh is that the effect of the forces must ripple through the mesh as the force propagates from one spring to the next. Still, they are often a useful and effective tool for the animator.

7.3 Particle systems

A particle system is a collection of a large number of point-like elements. Particle systems are often animated as a simple physical simulation. Because of the large number of elements typically found in a particle system, simplifying assumptions are used in both the rendering and the calculation of their motions. Various implementations make different simplifying assumptions. Some of the common assumptions made are the following:

Particles do not collide with other particles.
Particles do not cast shadows, except in an aggregate sense.
Particles only cast shadows on the rest of the environment, not on each other.
Particles do not reflect light—they are each modeled as point light sources.

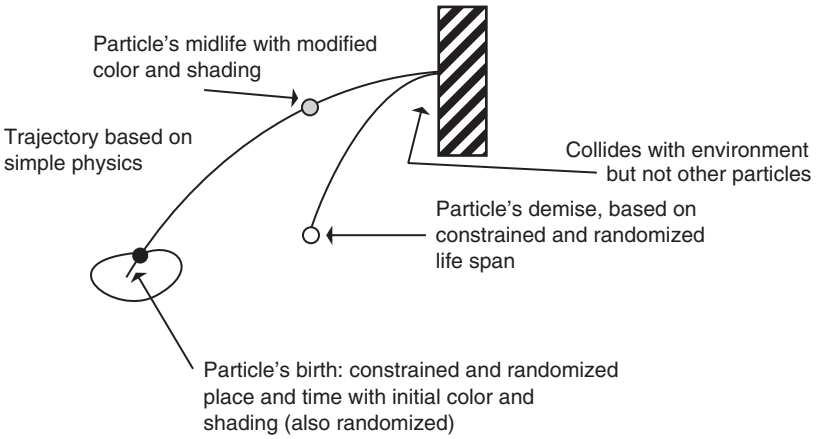


FIGURE 7.5

The life of a particle.

Particles are often modeled as having a finite life span, so that during an animation there may be hundreds of thousands of particles used but only thousands active at any one time. Randomness is introduced into most of the modeling and processing of particles to avoid excessive orderliness. In computing a frame of motion for a particle system, the following steps are taken (see Figure 7.5):

- Any new particles that are born during this frame are generated.
- Each new particle is assigned attributes.
- Any particles that have exceeded their allocated life span are terminated.
- The remaining particles are animated and their shading parameters changed according to the controlling processes.
- The particles are rendered.

The steps are then repeated for each frame of the animation. If the user can enforce the constraint that there are a maximum number of particles active at any one time, then the data structure can be static and particle data structures can be reused as one dies and another is created in its place.

7.3.1 Particle generation

Particles are typically generated according to a controlled stochastic process. For each frame, a random number of particles are generated using some user-specified distribution centered at the desired average number of particles per frame (Eq. 7.15.). The distribution could be uniform or Gaussian or anything else the animator wants. In Equation 7.15, *Rand*(_) returns a random number from -1.0 to $+1.0$ in the desired distribution, and r scales it into the desired range. If the particles are used to model a fuzzy object, then it is best to make the number of particles a function of the area, A, of the screen covered by the object to control the number of particles generated (Eq. 7.16). The features of the random number generator, such as average value and variance, can be a function of time to enable the animator to have more control over the particle system.

$$\#\ of\ particles = n + Rand() * r \tag{7.15}$$

$$\#\ of\ particles = n(A) + Rand() * r \tag{7.16}$$

7.3.2 Particle attributes

The attributes of a particle determine its motion status, its appearance, and its life in the particle system. Typical attributes include the following:

Position
Velocity
Shape parameters
Color
Transparency
Lifetime

Each of the attributes is initialized when the particle is created. Again, to avoid uniformity, the user typically randomizes values in some controlled way. The position and velocity are updated according to the particle's motion. The shape parameters, color, and transparency control the particle's appearance. The lifetime attribute is a count of how many frames the particle will exist in.

7.3.3 Particle termination

At each new frame, each particle's lifetime attribute is decremented by one. When the attribute reaches zero, the particle is removed from the system. This completes the particle's life cycle and can be used to keep the number of particles active at any one time within some desired range of values.

7.3.4 Particle animation

Typically, each active particle in a particle system is animated throughout its life. This activation includes not only its position and velocity but also its display attributes: shape, color, and transparency. To animate the particle's position in space, the user considers forces and computes the resultant particle acceleration. The velocity of the particle is updated from its acceleration, and then the average of its old velocity and newly updated velocity is computed and used to update the particle's position. Gravity, other global force fields (e.g., wind), local force fields (vortices), and collisions with objects in the environment are typical forces modeled in the environment.

The particle's color and transparency can be a function of global time, its own life span remaining, its height, and so on. The particle's shape can be a function of its velocity. For example, an ellipsoid can be used to represent a moving particle where the major axis of the ellipsoid is aligned with the direction of travel and the ratio of the ellipsoid's length to the radius of its circular cross section is related to its speed.

7.3.5 Particle rendering

To simplify rendering, model each particle as a point light source so that it adds color to the pixel(s) it covers but is not involved in the display pipeline (except to be hidden) or shadowing. In some applications, shadowing effects have been determined to be an important visual cue. The density of particles between a position in space and a light source can be used to estimate the amount of shadowing. See

Blinn [2], Ebert et al. [4], and Reeves [20] for some of these considerations. For more information on particle systems, see Reeves and Blau [21] and Sims [22].

7.3.6 Particle system representation

A particle is represented by a tuple $[x, v, f, m]$, which holds its position, velocity, force accumulator, and mass. It can also be used to hold any other attributes of the particle such as its age and color.

```
typedef struct particle_struct {
     vector3D         p;
     vector3D         v;
     vector3D         f;
     float            mass;
} particle;
```

The state of a particle, $[x, v]$, will be updated by $[v, f/m]$ by the ordinary differential equation solver. The solver can be considered to be a black box to the extent that the actual method used by the solver does not have to be known.

A particle system is an array of particles with a time variable $[*p, n, t]$.

```
typedef struct particleSystem_struct {
     particle    *p;    // array particle information
     int          n;    // number of particles
     float        t;    // current time (age) of the particle system
} particleSystem;
```

Updating particle system status

For each time step, the particle system is updated by the following steps:

a. For each particle, zero out the force vector.
b. For each particle, sum all forces affecting the particle.
c. Form the state array of all particles.
d. Form the update array of all particles.
e. Add the update array to the state array.
f. Save the new state array.
g. Update the time variable.

7.3.7 Forces on particles

Forces can be unary or binary and typically arise from the particle's relationship to the environment. Additional forces might be imposed by the animator in order to better control the animation. Unary forces include gravity and viscous drag. Particle pair forces, if desired, can be represented by springs and dampers. However, implementing particle pair forces for each possible pair of particles can be prohibitively expensive in terms of computational requirements. Auxiliary processing, such as the use of a spatial bucket sort, can be employed to consider only n-nearest-neighbor pairs and reduce the computation required, but typically particle–particle interaction is ignored. Environmental forces arise from a particle's relationship to the rest of the environment such as gravity, wind, air viscosity, and responding to collisions; most of these may be modeled using simple physics.

7.3.8 Particle life span

Typically, each particle will have a life span. The particle data structure itself can be reused in the system so that a particle system might have tens of thousands of particles over the life of the simulation but only, for example, one thousand in existence at any one time. Initial values are set pseudorandomly so that particles are spawned in a controlled manner but with some variability.

7.4 Rigid body simulation

A common objective in computer animation is to create realistic-looking motion. A major component of realistic motion is the physically based reaction of rigid bodies to commonly encountered forces such as gravity, viscosity, friction, and those resulting from collisions. Creating realistic motion with keyframe techniques can be a daunting task. However, the equations of motion can be incorporated into an animation system to automatically calculate these reactions. This can eliminate considerable tedium—if the animator is willing to relinquish precise control over the motion of some objects.

In rigid body simulation, various forces to be simulated are modeled in the system. These forces may arise due to relative positioning of objects (e.g., gravity, collisions), object velocity (e.g., viscosity), or the absolute position of objects in user-specified vector fields (e.g., wind). When applied to objects, these forces induce linear and angular accelerations based on the mass of the object (in the linear case) and mass distribution of the object (in the angular case). These accelerations, which are the time derivative of velocities, are integrated over a delta time step to produce changes in object velocities (linear and angular). These velocities, in turn integrated over a delta time step, produce changes in object positions and orientations (Figure 7.6). The new positions and velocities of the objects give rise to new forces, and the process repeats for the next time step.

The free flight of objects through space is a simple type of rigid body simulation. The simulation of rigid body physics becomes more complex as objects collide, roll and slide over one another, and come to rest in complex arrangements. In addition, a persistent issue in rigid body simulation, as with most animation, is the modeling of a continuous process (such as physics) with discrete time steps. The trade-off of accuracy for computational efficiency is an ever-present consideration.

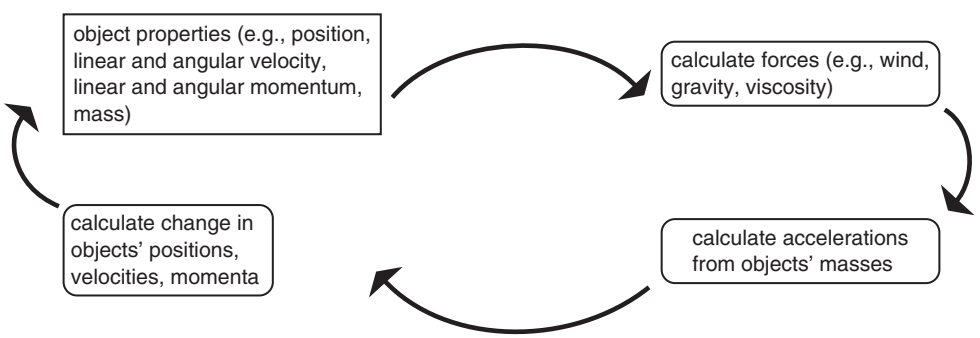


FIGURE 7.6

Rigid body simulation update cycle.

It should be noted that the difference between material commonly taught in standard physics texts and that used in computer animation is in how the equations of motion are used. In standard physics, the emphasis is usually in analyzing the equations of motion for times at which significant events happen, such as an object hitting the ground. In computer animation, the concern is with modeling the motion of objects at discrete time steps [12] as well as the significant events and their aftermath. While the fundamental principles and basic equations are the same, the discrete time sampling creates numerical issues that must be dealt with carefully. See Appendix B.7 for equations of motion from basic physics.

7.4.1 Bodies in free fall

To understand the basics in modeling physically based motion, the motion of a point in space will be considered first. The position of the point at discrete time steps is desired, where the interval between these time steps is some uniform Δt. To update the position of the point over time, its position, velocity, and acceleration are used, which are modeled as functions of time, $\boldsymbol{x}(t)$, $\boldsymbol{v}(t)$, $\boldsymbol{a}(t)$, respectively.

If there are no forces applied to a point, then a point's acceleration is zero and its velocity remains constant (possibly non-zero). In the absence of acceleration, the point's position, $\boldsymbol{x}(t)$, is updated by its velocity, $\boldsymbol{v}(t)$, as in Equation 7.17. A point's velocity, $\boldsymbol{v}(t)$, is updated by its acceleration, $\boldsymbol{a}(t)$. Acceleration arises from forces applied to an object over time. To simplify the computation, a point's acceleration is usually assumed to be constant over the time period Δt (see Eq. 7.18). A point's position is updated by the average velocity during the time period Δt. Under the constant-acceleration assumption, the average velocity during a time period is the average of its beginning velocity and ending velocity, as in Equation 7.19. By substituting Equation 7.18 into Equation 7.19, one defines the updated position in terms of the starting position, velocity, and acceleration (Eq. 7.20).

$$\boldsymbol{x}(t+\Delta t) = \boldsymbol{x}(t) + \boldsymbol{v}(t)\Delta t \tag{7.17}$$

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \boldsymbol{a}(t)\Delta t \tag{7.18}$$

$$\boldsymbol{x}(t+\Delta t) = \boldsymbol{x}(t) + \frac{\boldsymbol{v}(t) + \boldsymbol{v}(t+\Delta t)}{2}\Delta t \tag{7.19}$$

$$\begin{aligned} \boldsymbol{x}(t+\Delta t) &= \boldsymbol{x}(t) + \frac{\boldsymbol{v}(t) + \boldsymbol{v}(t) + \boldsymbol{a}(t) * \Delta t}{2}\Delta t \\ \boldsymbol{x}(t+\Delta t) &= \boldsymbol{x}(t) + \boldsymbol{v}(t) * \Delta t + \frac{1}{2}\boldsymbol{a}(t) * \Delta t^2 \end{aligned} \tag{7.20}$$

A simple example

Using a standard two-dimensional physics example, consider a point with an initial position of (0, 0), with an initial velocity of (100, 100) feet per second, and under the force of gravity resulting in a uniform acceleration of (0, 232) feet per second. Assume a delta time interval of 1/30 of a second (corresponding roughly to the frame interval in the National Television Standards Committee video). In this example, the acceleration is uniformly applied throughout the sequence, and the velocity is modified at each time step by the downward acceleration. For each time interval, the average of the beginning and ending velocities is used to update the position of the point. This process is repeated for each step in time (see Eq. 7.21 and Figures 7.7 and 7.8).

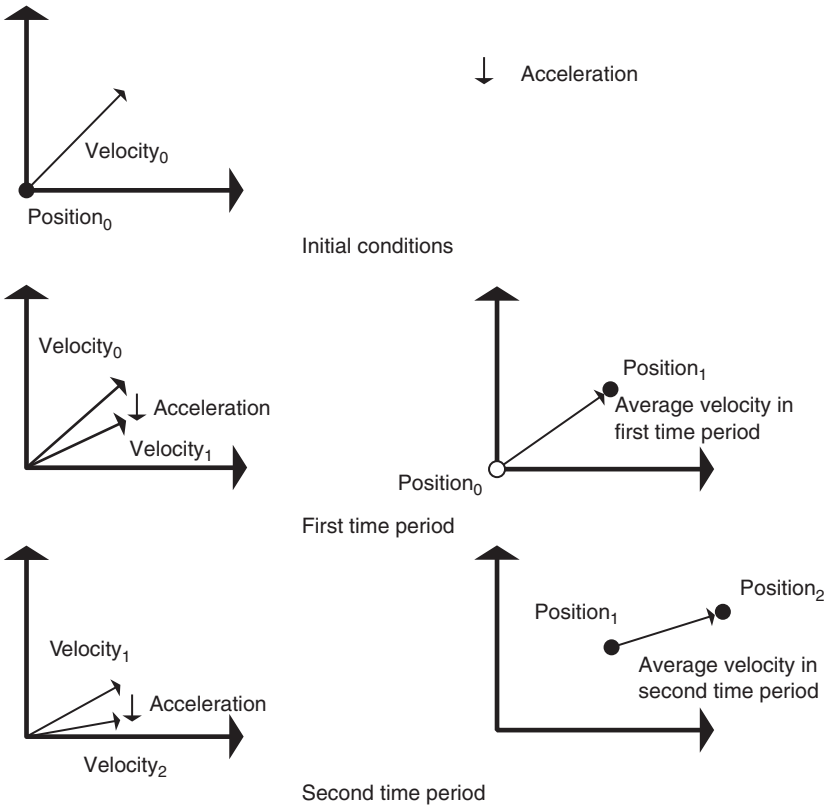


FIGURE 7.7

Modeling of a point's position at discrete time intervals (vector lengths are for illustrative purposes and are not accurate).

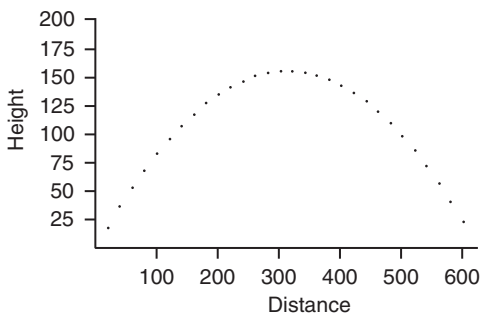


FIGURE 7.8

Path of a particle in the simple example from the text.

$$
\begin{aligned}
v(0) &= (100, 100) \\
x(0) &= (0, 0) \\
v\left(\frac{1}{30}\right) &= (100, 100) + (0, -32)\frac{1}{30} \\
x\left(\frac{1}{30}\right) &= (0, 0) + \frac{1}{2}\left(v(0) + v\left(\frac{1}{30}\right)\right) \\
v\left(\frac{2}{30}\right) &= v\left(\frac{1}{30}\right) + (0, -32)\frac{1}{30} \\
&\textit{etc.}
\end{aligned}
\tag{7.21}
$$

A note about numeric approximation

For an object at or near the earth's surface, the acceleration of the object due to the earth's gravity can be considered constant. However, in many rigid body simulations, the assumption that acceleration remains constant over the delta time step is incorrect. Many forces continually vary as the object changes its position and velocity over time, which means that the acceleration actually varies over the time step and it often varies in nonlinear ways. Sampling the force at the beginning of the time step and using that to determine the acceleration throughout the time step is not the best approach. The multiplication of acceleration at the beginning of the time step by Δt to step to the next function value (velocity) is an example of using the Euler integration method (Figure 7.9). It is important to understand the shortcomings of this approach and the available options for improving the accuracy of the simulation. Many books have been written about the subject; *Numerical Recipes: The Art of Scientific Computing*, by Press et al. [19], is a good place to start. This short section is intended merely to demonstrate that better options exist and that, at the very least, a Runge-Kutta method should be considered.

It is easy to see how the size of the time step affects accuracy (Figure 7.10). By taking time steps that are too large, the numerically approximated path deviates dramatically from the ideal continuous path. Accuracy can be increased by taking smaller steps, but this can prove to be computationally expensive.

Accuracy can also be increased by using better methods of integration. *Runge-Kutta* is a particularly useful one. Figure 7.11 shows the advantage of using the *second-order Runge-Kutta*, or *midpoint*,

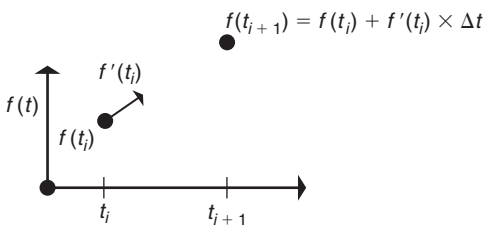


FIGURE 7.9

Euler integration.

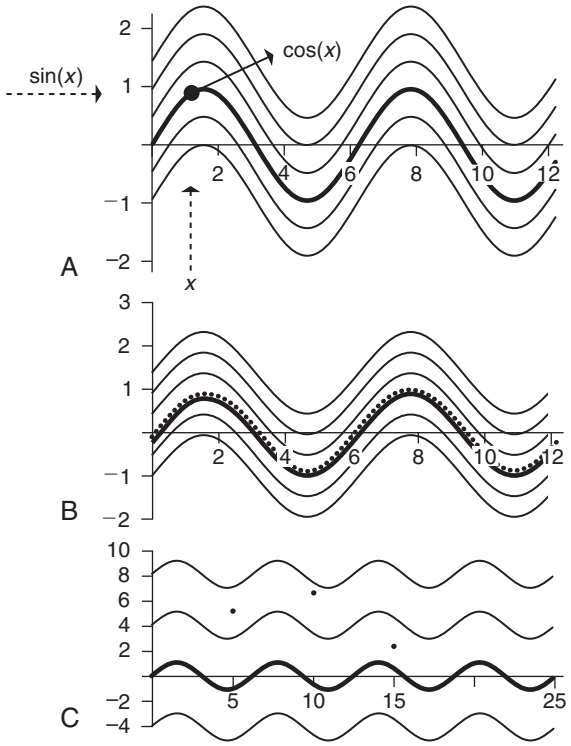


FIGURE 7.10

Approximating the sine curve by stepping in the direction of its derivative. (a) In this example, the sine function is the underlying (unknown) function. The objective is to reconstruct it based on an initial point and knowledge about the derivative function (cosine). Start out at any (x, y) location and, if the reconstruction is completely accurate, the sinusoidal curve that passes through that point should be followed. In (b) and (c), the initial point is (0, 0) so the thicker sine curve should be followed. (b) Updating the function values by taking small enough steps along the direction indicated by the derivative generates a good approximation to the function. In this example, $\Delta x = 0.2$. (c) However, if the step size becomes too large, then the function reconstructed from the sample points can deviate widely from the underlying function. In this example, $\Delta x = 5$.

method, which uses derivative information from the midpoint of the stepping interval. *Second-order* refers to the magnitude of the error term. Higher order Runge-Kutta methods are more accurate and therefore allow larger time steps to be taken, but they are more computationally expensive per time step. Generally, it is better to use a fourth- or fifth-order Runge-Kutta method.

While computer animation is concerned primarily with visual effects and not numeric accuracy, it is still important to keep an eye on the numerics that underlie any simulation responsible for producing the motion. Visual realism can be compromised if the numeric calculation is allowed to become too sloppy. One should have a basic understanding of the strengths and weaknesses of the numeric techniques used, and, in most cases, employing the Euler method should be done with caution. See Appendix B.8 for more information on numerical integration techniques.

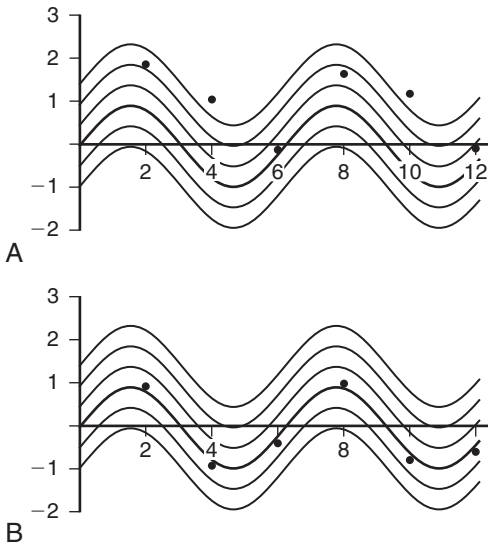


FIGURE 7.11

Euler method and midpoint method. (A) Using step sizes of 2 with the Euler method. (B) Using step sizes of 2 with the midpoint method.

Equations of motion for a rigid body

To develop the equations of motion for a rigid body, several concepts from physics are presented first [12]. The rotational equivalent of linear force, or *torque*, needs to be considered when a force is applied to an object not directly in line with its *center of mass*. To uniquely solve for the resulting motions of interacting objects, *linear momentum* and *angular momentum* have to be conserved. And, finally, to calculate the angular momentum, the distribution of an object's mass in space must be characterized by its *inertia tensor*. These concepts are discussed in the following sections and are followed by the equations of motion.

Orientation and rotational movement

Similar to linear attributes of position, velocity, and acceleration, three-dimensional objects have rotational attributes of orientation, angular velocity, and angular acceleration as functions of time. If an individual point in space is modeled, such as in a particle system, then its rotational information can be ignored. Otherwise, the physical extent of the mass of an object needs to be taken into consideration in realistic physical simulations.

For current purposes, consider an object's orientation to be represented by a rotation matrix, $R(t)$. Angular velocity is the rate at which the object is rotating irrespective of its linear velocity. It is represented by a vector, $\omega(t)$. The direction of the vector indicates the orientation of the axis about which the object is rotating; the magnitude of the angular velocity vector gives the speed of the rotation in revolutions per unit of time. For a given number of rotations in a given time period, the angular velocity of an object is the same whether the object is rotating about its own axis or rotating about an axis some distance away. Consider a point that is rotating about an axis that passes through the point and that the rate of rotation is two revolutions per minute. Now consider a second point that is rotating

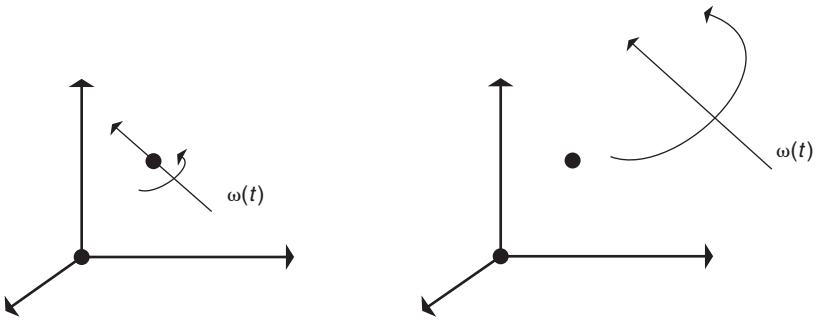


FIGURE 7.12

For a given number of rotations per unit of time, the angular velocity is the same whether the axis of rotation is near or far away.

about an axis that is parallel to the previous axis but is ten miles away, again at the rate of two revolutions per minute. The second point will have the same angular velocity as the first point. However, in the case of the second point, the rotation will also induce an instantaneous linear velocity (which constantly changes). But in both cases the object is still rotating at two revolutions per minute (Figure 7.12).

Consider a point, a, whose position in space is defined relative to a point, $b = x(t)$; a's position relative to b is defined by $r(t)$. The point a is rotating and the axis of rotation passes through the point b (Figure 7.13). The change in $r(t)$ is computed by taking the cross-product of $r(t)$ and $\omega(t)$ (Eq. 7.22). Notice that the change in $r(t)$ is perpendicular to the plane formed by $\omega(t)$ and $r(t)$ and that the magnitude of the change is dependent on the perpendicular distance between $\omega(t)$ and $r(t)$, $|r(t)|\sin\theta$, as well as the magnitude of $\omega(t)$.

$$\begin{aligned} \dot{r}(t) &= \omega(t) \times r(t) \\ |\dot{r}(t)| &= |\omega(t)||r(t)| \sin \end{aligned} \tag{7.22}$$

Now consider an object that has an extent (distribution of mass) in space. The orientation of an object, represented by a rotation matrix, can be viewed as a transformed version of the object's local unit coordinate system. As such, its columns can be viewed as vectors defining relative positions in the object. Thus, the change in the rotation matrix can be computed by taking the cross-product of $\omega(t)$ with

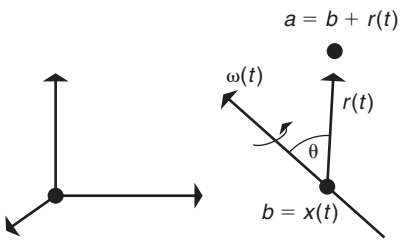


FIGURE 7.13

A point rotating about an axis.

each of the columns of $R(t)$ (Eq. 7.23). By defining a special matrix to represent cross-products (Eq.7.24), one can represent Equation 7.23 by matrix multiplication (Eq. 7.25).

$$\begin{aligned} R(t) &= [R_1(t) \quad R_2(t) \quad R_3(t)] \\ \dot{R}(t) &= [\omega \times R_1(t) \quad \omega \times R_2(t) \quad \omega \times R_3(t)] \end{aligned} \tag{7.23}$$

$$A \times B = \begin{bmatrix} A_yB_z - A_zB_y \\ A_zB_x - A_xB_z \\ A_xB_y - A_yB_x \end{bmatrix} = \begin{bmatrix} 0 & -A_z & A_y \\ A_z & 0 & -A_x \\ -A_y & A_x & 0 \end{bmatrix} \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} \equiv A^*B \tag{7.24}$$

$$\dot{R}(t) = \omega(t)^* R(t) \tag{7.25}$$

Consider a point, Q, on a rigid object. Its position in the local coordinate space of the object is q; q does not change. Its position in world space, $q(t)$, is given by Equation 7.26. The position of the body in space is given by $x(t)$, and the orientation of the body is given by $R(t)$. The velocity of the particle is given by differentiating Equation 7.26. The relative position of the particle in the rigid body is constant. The change in orientation is given by Equation 7.25, while the change in position is represented by a velocity vector. These are combined to produce Equation 7.27. The world space position of the point Q in the object, taking into consideration the object's orientation, is given by rearranging Equation 7.26 to give $R(t)q = q(t) - x(t)$. Substituting this into Equation 7.27 and distributing the cross-product produces Equation 7.28.

$$q(t) = R(t)q + x(t) \tag{7.26}$$

$$\dot{q}(t) = \omega(t)^* R(t)q + v(t) \tag{7.27}$$

$$\dot{q(t)} = \omega(t) \times (q(t) - x(t)) + v(t) \tag{7.28}$$

Center of mass

The center of mass of an object is defined by the integration of the differential mass times its position in the object. In computer graphics, the mass distribution of an object is typically modeled by individual points, which is usually implemented by assigning a mass value to each of the object's vertices. If the individual masses are given by m_i, then the total mass of the object is represented by Equation 7.29. Using an object coordinate system that is located at the center of mass is often useful when modeling the dynamics of a rigid object. For the current discussion, it is assumed that $\mathbf{x}(t)$ is the center of mass of an object. If the location of each mass point in world space is given by $q_i(t)$, then the center of mass is represented by Equation 7.30.

$$M = \Sigma m_i \tag{7.29}$$

$$\boldsymbol{x}(t) = \frac{\Sigma m_i \boldsymbol{q_i}(t)}{M} \tag{7.30}$$

Forces

A linear force (a force along a straight line), $\boldsymbol{f}$, applied to a mass, m, gives rise to a linear acceleration, $\mathbf{a}$, by means of the relationship shown in Equations 7.31 and 7.32. This fact provides a way to calculate acceleration from the application of forces. Examples of such forces are gravity, viscosity, friction, impulse forces due to collisions, and forces due to spring attachments. See Appendix B.7 for the basic equations from physics that give rise to such forces.

$$\boldsymbol{F} = m\boldsymbol{a} \tag{7.31}$$

$$\boldsymbol{a} = \boldsymbol{F}/m \tag{7.32}$$

The various forces acting on a point can be summed to form the total external force, $\boldsymbol{F}(t)$ (Eq. 7.33). Given the mass of the point, the acceleration due to the total external force can be calculated and then used to modify the velocity of the point. This can be done at each time step. If the point is assumed to be part of a rigid object, then the point's location on the object must be taken into consideration, and the effect of the force on the point has an impact on the object as a whole. The rotational equivalent of linear force is *torque*, $\tau(t)$. The torque that arises from the application of forces acting on a point of an object is given by Equation 7.34.

$$\mathrm{F} = \sum \mathrm{f}_i(t) \tag{7.33}$$

$$\begin{aligned} \tau_i &= (\mathrm{q}_i(t) - \mathrm{x}(t)) \times \mathrm{f}_i(t) \\ \tau &= \sum \tau_i(t) \end{aligned} \tag{7.34}$$

Momentum

As with force, the momentum (mass × velocity) of an object is decomposed into a linear component and an angular component. The object's local coordinate system is assumed to be located at its center of mass. The linear component acts on this center of mass, and the angular component is with respect to this center. *Linear momentum* and *angular momentum* need to be updated for interacting objects because these values are conserved in a closed system. Saying that the linear momentum is conserved in a closed system, for example, means that the sum of the linear momentum does not change if there are no outside influences on the system. The case is similar for angular momentum. That they are conserved means they can be used to solve for unknown values in the system, such as linear velocity and angular velocity.

Linear momentum is equal to velocity times mass (Eq. 7.35). The total linear momentum $\boldsymbol{P}(t)$ of a rigid body is the sum of the linear momentums of each particle (Eq. 36). For a coordinate system whose origin coincides with the center of mass, Equation 7.36 simplifies to the mass of the object times its velocity (Eq. 7.37). Further, since the mass of the object remains constant, taking the derivative of momentum with respect to time establishes a relationship between linear momentum and linear force (Eq. 7.38). This states that the force acting on a body is equal to the change in momentum. Interactions composed of equal but opposite forces result in no change in momentum (i.e., momentum is conserved).

$$\mathrm{P} = m\mathrm{v} \tag{7.35}$$

$$P(t) = \Sigma m_i \dot{q_i}(t) \tag{7.36}$$

$$\mathrm{P}(t) = M\mathrm{v}(t) \tag{7.37}$$

$$\dot{\mathrm{P}}(t) = M\dot{\mathrm{v}}(t) = \mathrm{F}(t) \tag{7.38}$$

Angular momentum, $\boldsymbol{L}$, is a measure of the rotating mass weighted by the distance of the mass from the axis of rotation. For a mass point in an object, the angular momentum is computed by taking the cross-product of a vector to the mass and a velocity vector of that mass point × the mass of the point. These vectors are relative to the center of mass of the object. The total angular momentum of a rigid body is computed by integrating this equation over the entire object. For the purposes of computer

animation, the computation is usually summed over mass points that make up the object (Eq. 7.39). Notice that angular momentum is not directly dependent on the linear components of the object's motion.

$$\begin{aligned} \mathrm{L}(t) &= \sum((\mathrm{q}_i(t) - \mathrm{x}(t)) \times m_i(\dot{\mathrm{q}}_i(t) - \mathrm{v}(t))) \\ &= \sum(R(t)\mathrm{q} \times m_i(\omega(t) \times (\mathrm{q}(t) - \mathrm{x}(t)))) \\ &= \sum(m_i(R(t)\mathrm{q} \times (\omega(t) \times R(t)\mathrm{q}))) \end{aligned} \tag{7.39}$$

In a manner similar to the relation between linear force and the change in linear momentum, torque equals the change in angular momentum (Eq. 7.40). If no torque is acting on an object, then angular momentum is constant. However, the angular velocity of an object does not necessarily remain constant even in the case of no torque. The angular velocity can change if the distribution of mass of an object changes, such as when an ice skater spinning on his skates pulls his arms in toward his body to spin faster. This action brings the mass of the skater closer to the center of mass. Angular momentum is a function of angular velocity, mass, and the distance the mass is from the center of mass. To maintain a constant angular momentum, the angular velocity must increase if the distance of the mass decreases.

$$\dot{\mathrm{L}}(t) = \tau(t) \tag{7.40}$$

Inertia tensor

Angular momentum is related to angular velocity in much the same way that linear momentum is related to linear velocity, $\boldsymbol{P}(t) = M\boldsymbol{v}(t)$ (see Eq. 7.41). However, in the case of angular momentum, a matrix is needed to describe the distribution of mass of the object in space, the *inertia tensor*, $I(t)$. The inertia tensor is a symmetric 3×3 matrix. The initial inertia tensor defined for the untransformed object is denoted as I_{object} (Eq. 7.42). Terms of the matrix are calculated by integrating over the object (e.g., Eq. 7.43). In Equation 7.43, the density of an object point, $q = (q_x, q_y, q_z)$, is $r(q)$. For the discrete case, Equation 7.44 is used. In a center-of-mass-centered object space, the inertia tensor for a transformed object depends on the orientation, $R(t)$, of the object but not on its position in space. Thus, it is dependent on time. It can be transformed according to the transformation of the object by Equation 7.45. Its inverse, which we will need later, is transformed in the same way.

$$L(t) = I(t)\omega(t) \tag{7.41}$$

$$I_{\text{object}} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \tag{7.42}$$

$$I_{xx} = \iiint(\rho(q)({q_y}^2 + {q_z}^2))dxdydz \tag{7.43}$$

$$\begin{aligned} I_{xx} &= \sum m_i({y_i}^2 + {z_i}^2) & I_{xy} &= \sum m_i x_i y_i \\ I_{yy} &= \sum m_i({x_i}^2 + {z_i}^2) & I_{xz} &= \sum m_i x_i z_i \\ I_{zz} &= \sum m_i({x_i}^2 + {y_i}^2) & I_{yz} &= \sum m_i x_i z_i \end{aligned} \tag{7.44}$$

$$I(t) = R(t)I_{\text{object}}R(t)^T \tag{7.45}$$

The equations

The state of an object can be kept in a heterogeneous structure called the *state vector*, $S(t)$, consisting of its position, orientation, linear momentum, and angular momentum (Eq. 7.46). Object attributes, which do not change over time, include its mass, M, and its object-space inertia tensor, I_{object}. At any time, an object's time-varying inertia tensor, angular velocity, and linear velocity can be computed (Eqs. 7.47–7.49). The time derivative of the object's state vector can now be formed (Eq. 7.50).

$$S(t) = \begin{bmatrix} \mathrm{x}(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} \tag{7.46}$$

$$I(t) = R(t) I_{\text{object}} R(t)^T \tag{7.47}$$

$$\omega(t) = I(t)^{-1} L(t) \tag{7.48}$$

$$v(t) = \frac{P(t)}{M} \tag{7.49}$$

$$\frac{d}{dt} S(t) = \frac{d}{dt} \begin{bmatrix} \mathrm{x}(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} = \begin{bmatrix} \mathrm{v}(t) \\ \omega(t)^* R(t) \\ \mathrm{F}(t) \\ \tau(t) \end{bmatrix} \tag{7.50}$$

This is enough information to run a simulation. Once the ability to compute the derivative information is available, then a differential equation solver can be used to update the state vector. In the simplest implementation, Euler's method can be used to update the values of the state array. The values of the state vector are updated by multiplying their time derivatives by the length of the time step. In practice, Runge-Kutta methods (especially fourth-order) have found popularity because of their trade-off in speed, accuracy, and ease of implementation.

Care must be taken in updating the orientation of an object. Because the derivative information from earlier is only valid instantaneously, if it is used to update the orientation rotation matrix, then the columns of this matrix can quickly become nonorthogonal and not of unit length. At the very least, the column lengths should be renormalized after being updated. A better method is to update the orientation matrix by applying to its columns the axis-angle rotation implied by the angular velocity vector, $\omega(t)$. The magnitude of the angular velocity vector represents the angle of rotation about the axis along the angular velocity vector (see Appendix B.3.3 for the axis-angle calculation). Alternatively, quaternions can be used to represent both the object's orientation and the orientation derivative, and its use here is functionally equivalent to the axis-angle approach.

7.4.2 Bodies in collision

When an object starts to move in any kind of environment other than a complete void, chances are that sooner or later it will bump into something. If nothing is done about this in a computer animation, the object will penetrate and then pass through other objects. Other types of contact include objects sliding against and resting on each other. All of these types of contact require the calculation of forces in order to accurately simulate the reaction of one object to another.

Colliding bodies

As objects move relative to one another, there are two issues that must be addressed: (1) detecting the occurrence of collision and (2) computing the appropriate response to those collisions. The former is strictly a kinematic issue in that it has to do with the positions and orientations of objects and how they change over time. The latter is usually a dynamic issue in that forces that are a result of the collision are computed and used to produce new motions for the objects involved.

Collision detection considers the movement of one object relative to another. In its most basic form, testing for a collision amounts to determining whether there is intersection in the static position of two objects at a specific instance in time. In a more sophisticated form, the movement of one object relative to the other object during a finite time interval is tested for overlap. These computations can become quite involved when dealing with complex geometries.

Collision response is a consideration in physically based simulation. The geometric extent of the object is not of concern but rather the distribution of its mass. Localized forces at specific points on the object impart linear and rotational forces onto the other objects involved.

In dealing with the time of collision, there are two options. The first is to proceed as best as one can from this point in time by calculating an appropriate reaction to the current situation by the particle involved in the collision (the penalty method). This option allows penetration of the particle before the collision reaction takes place. Of course, if the particle is moving rapidly, this penetration might be visually significant. If multiple collisions occur in a time interval, they are treated as occurring simultaneously even though handling them in their correct sequential order may have produced different results. While more inaccurate than the second option, this is simpler to implement and often gives acceptable results.

The second option is to back up time t_i to the first instant that a collision occurred and determine the appropriate response at the time of collision. If multiple collisions occur in a time interval, then time is backed up to the point at which the first collision took place. In complex environments in which collisions happen at a high rate, this constant backing up of time and recomputing the motion of objects can become quite time-consuming.

There are three common options for collision response: a strictly kinematic response, the penalty method, and the calculation of an impulse force. The kinematic response is quick and easy. It produces good visual results for particles and spherically shaped objects. The penalty method introduces a temporary, nonphysically based force in order to restore nonpenetration. It is typically used when response to the collision occurs at the time step when penetration is detected (as opposed to backing up time). The advantage of this technique is that it is easy to compute and the force is easily incorporated into the computational mechanism that simulates rigid body movement. Calculating the impulse force is a more precise way of introducing a force into the system and is typically used when time is backed up to the point of first contact. Detecting collisions and reacting to them are discussed next.

Particle-plane collision and kinematic response

One of the simplest illustrative situations to consider for collision detection and response is that of a particle traveling at some velocity toward a stationary plane at an arbitrary angle (see Figure 7.14). The task is to detect when the particle collides with the plane and have it bounce off the plane. Because a simple plane is involved, its planar equation can be used (Eq. 7.51). $E(p)$ is the planar

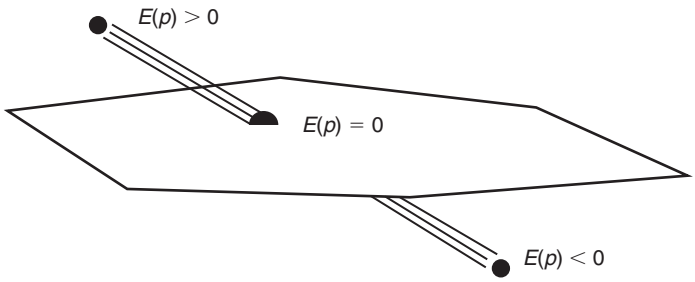


FIGURE 7.14

Point-plane collision.

function, a, b, c, d are coefficients of the planar equation, and p is, for example, a particle's position in space that has no physical extent of its own. Points on the plane satisfy the planar equation, that is, $E(p) = 0$. Assuming for now that the plane represents the ground, the planar equation can be formed so that for points above the plane the planar equation evaluates to a positive value, $E(p) > 0$; for points below the plane, $E(p) < 0$.

$$E(\mathrm{p}) = ap_x + bp_y + cp_z + d = 0 \tag{7.51}$$

The particle travels toward the plane as its position is updated according to its average velocity over the time interval, as in Equation 7.52. At each time step t_i, the particle is tested to see if it is still above the plane, $E(p(t_i)) > 0$. As long as this evaluates to a positive value, there is no collision. The first time t at which $E(p(t_i)) < 0$ indicates that the particle has collided with the plane at some time between t_{i-21} and t_i. What to do now? The collision has already occurred and something has to be done about it.

$$\mathrm{p}(t_i) = \mathrm{p}(t_{i+1}) + \mathrm{v}_{\mathrm{ave}}(t)\Delta t \tag{7.52}$$

In simple kinematic response, when penetration is detected, the velocity of the particle is decomposed, using the normal to the plane (N), into the normal component and the perpendicular-to-the-normal component (Figure 7.15). The normal component of the velocity vector is negated by subtracting it out of the original velocity vector and then subtracting it out again. To reduce the height of each successive bounce, a damping factor, $0 < k < 1$, can be applied when subtracting it out the second time (Eq. 7.53). This bounces the particle off a surface at a reduced velocity This approach

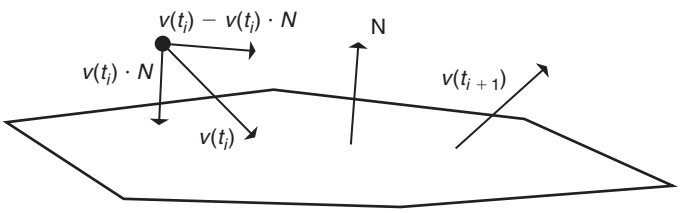


FIGURE 7.15

Kinematic solution for collision reaction.

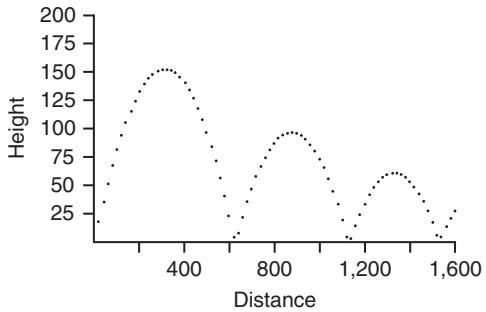


FIGURE 7.16

Kinematic response to collisions with ground using 0.8 as the damping.

is not physically based, but it produces reasonable visuals, especially for particles and spherically shaped objects. Incorporating kinematic response to collisions with the ground produces Figure 7.16.

$$\begin{aligned} \mathrm{v}(t_{t+1}) &= \mathrm{v}(t_i) - (\mathrm{v}(t_i) \cdot \mathrm{N})\mathrm{N} - k(\mathrm{v}(t_i) \cdot \mathrm{N})\mathrm{N} \\ &= \mathrm{v}(t_i) - (1+k)(\mathrm{v}(t_i) \cdot \mathrm{N})\mathrm{N} \end{aligned} \tag{7.53}$$

The penalty method

When objects penetrate due to temporal sampling, a simple method of constructing a reaction to the implied collision is the *penalty method*. As the name suggests, a point is penalized for penetrating another object. In this case, a spring, with a zero rest length, is momentarily attached from the offending point to the surface it penetrated in such a way that it imparts a restoring force on the offending point. For now, it is assumed that the surface it penetrates is immovable and thus does not have to be considered as far as collision response is concerned. The closest point on the penetrated surface to the penetrating point is used as the point of attachment (see Figure 7.17). The spring, therefore, imparts a force on the point in the direction of the penetrated surface normal and with a magnitude according to Hooke's law ($F = -kd$). A mass assigned to the point is used to compute a resultant acceleration ($a = F/m$), which contributes an upward velocity to the point. When this upward velocity is combined with the point's original motion, the point's downward motion will be stopped and reversed by the spring, while any component of its motion tangent to the surface will be unaffected. While easy to implement, this approach is not ideal. An arbitrary mass (m) must be assigned to the point, and an arbitrary constant (k) must be determined for the spring. It is difficult to control because if the spring

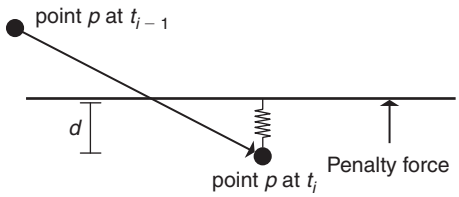


FIGURE 7.17

Penalty spring.

constant is too weak or if the mass is too large, then the collision will not be corrected immediately. If the spring constant is too strong or the mass is too small, then the colliding surfaces will be thrown apart in an unrealistic manner. If the point is moving fast relative to the surface it penetrated, then it may take a few time steps for the spring to take effect and restore a state of nonpenetration. If the desired velocity after the next time step is known, then this can be used help determine the spring constants.

Using the previous example and implementing the penalty method produces the motion traced in Figure 7.18. In this implementation, a temporary spring is introduced into the system whenever both the y component of the position and the y component of the velocity vector are negative; a spring constant of 250 and a point mass of 10 are used. While the spring force is easy to incorporate with other system forces such as gravity, the penalty method requires the user to specify control parameters that typically require a trial-and-error process to determine appropriate values because of other forces that may be present in the system.

When used with polyhedra objects, the penalty force can give rise to torque when not acting in line with the center of mass of an object. When two polyhedra collide, the spring is attached to both objects and imparts an equal but opposite force on the two to restore nonpenetration. Detecting collisions among polyhedra is discussed next, followed by a more accurate method of computing the impulse forces due to such collisions.

Testing polyhedra

In environments in which objects are modeled as polyhedra, at each time step each polyhedron is tested for possible penetration against every other polyhedron. A collision is implied whenever the environment transitions from a nonpenetration state to a state in which a penetration is detected. A test for penetration at each time step can miss some collisions because of the discrete temporal sampling, but it is sufficient for many applications.

Various tests can be used to determine whether an overlap condition exists between polyhedra, and the tests should be chosen according to computational efficiency and generality. Bounding box tests, also known as min-max tests, can be used to quickly determine if there is any chance for an intersection. A bounding box can easily be constructed by searching for the minimum and maximum values in x, y, and z. Bounding boxes can be tested for overlap. If there is no overlap of the bounding boxes, then there can be no overlap of the objects. If the object's shape does not match a rectangle well, then a bounding

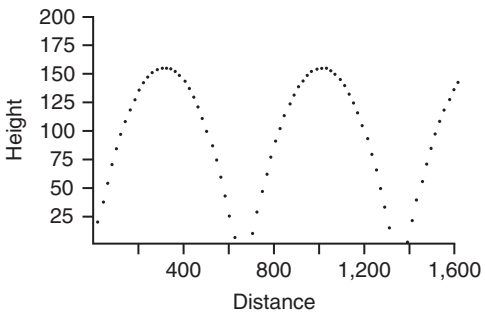


FIGURE 7.18

Penalty method with a spring constant of 250 and a point mass of 10, for example, from Section 7.4.2.

sphere or bounding slabs (pairs of parallel planes at various orientations that bound the object) can be used. A hierarchy of bounding shapes can be used on object groups, individual objects, and individual faces. If these bounding tests fail to conclusively establish that a nonoverlap situation exists, then more elaborate tests must be conducted.

Most collisions can be detected by testing each of the vertices of one object to see if any are inside another polyhedron and then testing each of the vertices of the other object to see if any are inside the first. To test whether a point is inside a convex polyhedron, each of the planar equations for the faces of the object is evaluated using the point's coordinates. If the planes are oriented so that a point to the outside of the object evaluates to a positive value and a point to the inside of the object evaluates to a negative value, then the point under consideration has to evaluate to a negative value in all of the planar equations in order to be declared inside the polyhedron.

Testing a point to determine if it is inside a concave polyhedron is more difficult. One way to do it is to construct a semi-infinite ray emanating from the point under consideration in any particular direction. For example, $y = P_y$, $z = P_z$, $x > P_x$ defines a line parallel to the x-axis going in the positive direction from the point P. This semi-infinite ray is used to test for intersection with all of the faces of the object, and the intersections are counted; an odd number of intersections means that the point is inside the object, and an even number means that the point is outside. To intersect a ray with a face, the ray is intersected with the planar equation of the face and then the point of intersection is tested to see if it is inside the polygonal face.

Care must be taken in correctly counting intersections if this semi-infinite ray intersects the object exactly at an edge or vertex or if the ray is colinear with an edge of the object. (These cases are similar to the situations that occur when scan converting concave polygons.) While this does not occur very often, it should be dealt with robustly. Because the number of intersections can be difficult to resolve in such situations, it can be computationally more efficient to simply choose a semi-infinite ray in a different direction and start over with the test.

One can miss some intersection situations if performing only these tests. The edges of each object must be tested for intersection with the faces of the other object and vice versa to capture other overlap situations. The edge–face intersection test consists of two steps. In the first step, the edge is tested for intersection with the plane of the face (i.e., the edge vertices are on opposite sides of the plane of the face) and the point of intersection is computed if it exists. In the second step, the point of intersection is tested for two-dimensional containment inside the face. The edge–face intersection tests capture almost all of the penetration situations,[2] but because it is the more expensive of the two tests, it is usually better to perform the vertex tests first (see Figure 7.19).

In any of these tests, vertices of one object that lie exactly in the plane of a face from the other object are a special case that must be handled carefully. Usually it is sufficient to logically and consistently push such a vertex to one side of the plane.

Often, a normal that defines the plane of intersection is used in the collision response calculations. For collisions that are detected by the penetration tests, one of two collision situations can usually be established by looking at the relative motion of the objects. Either a vertex has just penetrated a face, or an edge from one object has just intersected an edge from the other object. In either case, the plane of

[2]The edge–face intersection test will miss the case when one object is entirely inside of the other; this case is handled by testing a single vertex from each object to see if it is inside the other object.

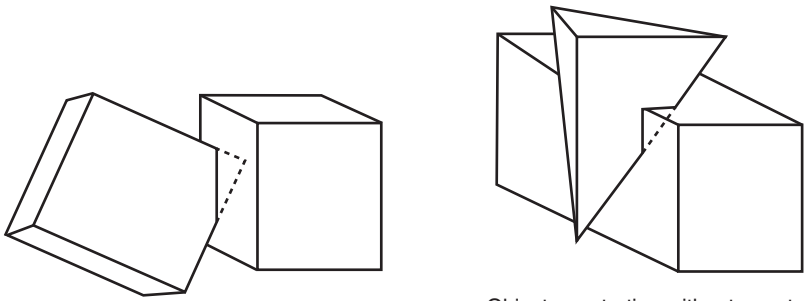


FIGURE 7.19

Detecting polyhedra intersections.

intersection can be established. When a vertex has penetrated a face, a normal to the face is used. When there is an edge–edge intersection, the normal is defined as the cross-product of the two edges.

In certain limited cases, a more precise determination of collision can be performed. For example, if the movement of one object with respect to another is a linear path, then the volume swept by one object can be determined and intersected with the other object whose position is assumed to be static relative to the moving object. If the direction of travel is indicated by a vector, V, then front faces and back faces with respect to the direction of travel can be determined. The *boundary edges* that share a front face and a back face can be identified; *boundary vertices* are the vertices of boundary edges. The volume swept by the object is defined by the original back faces, the translated positions of the front faces, and new faces and edges that are extruded in the direction of travel from boundary edges and boundary vertices (Figure 7.20). This volume is intersected with the static object.

Impulse force of collision

To allow for more accurate handling of the moment of collision, time can be "backed up" to the point of impact, the reaction can be computed, and the time can be moved forward again. While more accurate in its reconstruction of events, this approach can become very computationally intense in complex

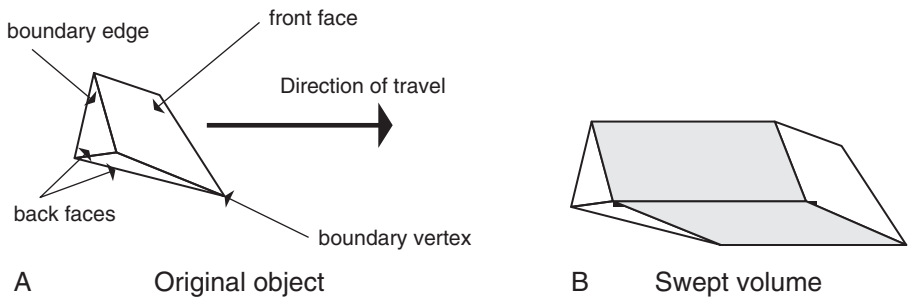


FIGURE 7.20

Sweep volume by translating front-facing faces and extruding new edges from original boundary vertices and extruding new faces from original boundary edges.

environments if many closely spaced collisions occur. However, for a point-plane collision, the impulse can be simplified because the angular velocity can be ignored.

The actual time of collision within a time interval can be searched for by using a binary search strategy. If it is known that a collision occurred between t_{i-1} and t_i, these times are used to initialize the lower ($L = t_{i-1}$) and upper ($U = t_i$) bounds on the time interval to be tested. At each iteration of the search, the point's position is tested at the middle of the bounded interval. If the test indicates that the point has not collided yet, then the lower bound is replaced with the middle of the time interval. Otherwise, the upper bound is replaced with the middle of the time interval. This test repeats with the middle of the new time interval and continues until some desired tolerance is attained. The tolerance can be based either on the length of the time interval or on the range of distances the point could be on either side of the surface.

Alternatively, a linear-path constant-velocity approximation can be used over the time interval to simplify the calculations. The position of the point before penetration and the position after penetration can be used to define a linear path. The point of intersection along the line with a planar surface can be calculated analytically. The relative position of the intersection point along this line can be used to estimate the precise moment of impact (Figure 7.21).

As previously discussed, at the time of impact, the normal component of the point's velocity can be modified as the response to the collision. A scalar, the coefficient of restitution, between zero and one can be applied to the resulting velocity to model the degree of elasticity of the collision (Figure 7.22).

Computing impulse forces

Once a collision is detected and the simulation has been backed up to the point of intersection, the reaction to the collision can be calculated. By working back from the desired change in velocity due to a collision, an equation that computes the required change in momentum can be formed. This equation uses a new term, called *impulse*, expressed in units of momentum. The impulse, $\boldsymbol{J}$, can be viewed as a large force acting over a short time interval (Eq. 7.54). The change in momentum $\boldsymbol{P}$, and therefore the change in velocity, can be computed once the impulse is known (Eq. 7.55).

$$\mathbf{J} = \mathbf{F}\Delta t \tag{7.54}$$

$$\mathbf{J} = \mathbf{F}\Delta t = M\mathbf{a}\Delta t = M\Delta\mathbf{a} = \Delta(M\mathbf{v}) = \Delta\mathbf{P} \tag{7.55}$$

To characterize the elasticity of the collision response, the user selects the coefficient of restitution, in the range $0 \leq k \leq 1$, which relates the relative velocity before the collision, $v_{\text{rel}}{}^{+}$ to the relative

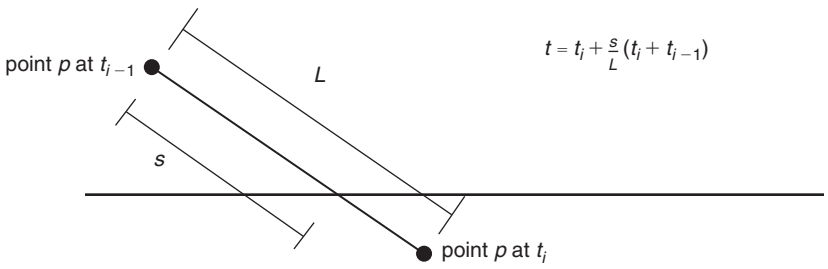
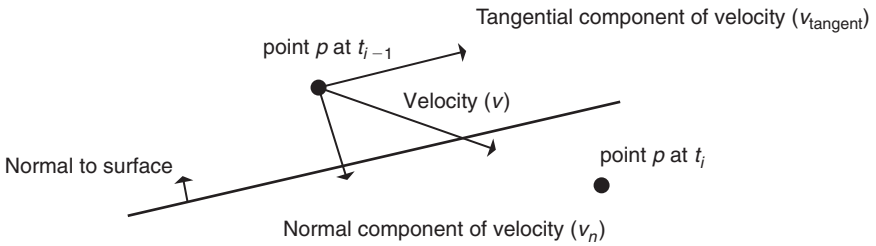
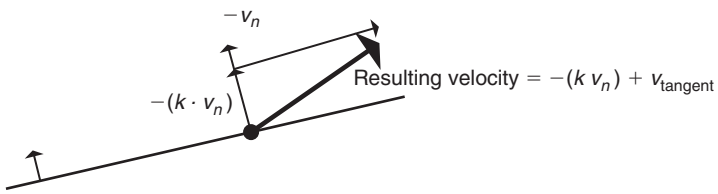


FIGURE 7.21

Linearly estimating time of impact, *t*.



Components of a particle's velocity colliding with a plane



Computing velocity resulting from a collision (k is the coefficient of restitution)

FIGURE 7.22

Impact response of a point with a plane.

velocity after the collision in the direction normal to the surface of contact, $\boldsymbol{N}$ (Eq. 7.56). Impulse is a vector quantity in the direction of the normal to the surface of contact, $\boldsymbol{J} = j\boldsymbol{N}$. To compute $\boldsymbol{J}$, the equations of how the velocities must change based on the equations of motion before and after the collision are used. These equations use the magnitude of the impulse, j, and they are used to solve for its value.

$$\mathrm{v}_{rel}^{+}(t) = -k\mathrm{v}_{rel}^{-}(t) \tag{7.56}$$

Assume that the collision of two objects, A and B, has been detected at time t. Each object has a position for its center of mass ($x_A(t)$, $x_B(t)$), linear velocity ($v_A(t)$, $v_B(t)$), and angular velocity ($v_A(t)$, $v_B(t)$). The points of collision (p_A, p_B) have been identified on each object (see Figure 7.23).

At the point of intersection, the normal to the surface of contact, $\boldsymbol{N}$, is determined depending on whether it is a vertex–face contact or an edge–edge contact. The relative positions of the contact points with respect to the center of masses are $\boldsymbol{r_A}$ and $\boldsymbol{r_B}$, respectively (Eq. 7.57). The relative velocity of the contact points of the two objects in the direction of the normal to the surface is computed by Equation 7.58. The velocities of the points of contact are computed as in Equation 7.59.

$$\begin{aligned} \mathrm{r}_A(t) &= \mathrm{p}_A(t) - \mathrm{x}_A(t) \\ \mathrm{r}_B(t) &= \mathrm{p}_B(t) - \mathrm{x}_B(t) \end{aligned} \tag{7.57}$$

$$\mathrm{v}_{rel}(t) = ((\dot{\mathrm{p}}_A(t) - \dot{\mathrm{p}}_B(t)) \cdot \mathrm{N})\mathrm{N} \tag{7.58}$$

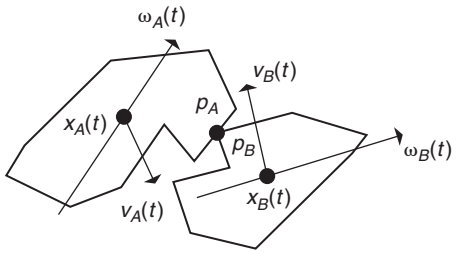


FIGURE 7.23

Configuration of colliding objects.

$$\begin{aligned} \dot{\mathrm{p}}_A(t) &= \mathrm{v}_A(t) + \omega_A(t) \times \mathrm{r}_A(t) \\ \dot{\mathrm{p}}_B(t) &= \mathrm{v}_B(t) + \omega_B(t) \times \mathrm{r}_B(t) \end{aligned} \tag{7.59}$$

The linear and angular velocities of the objects before the collision ($\boldsymbol{v_A}^-, \boldsymbol{w_B}^-$) are updated by the impulse to form the linear and angular velocities after the collision ($\boldsymbol{v_A}^+, \boldsymbol{w_B}^+$). The linear velocities are updated by adding in the effect of the impulse scaled by one over the mass (Eq. 7.60). The angular velocities are updated by computing the effect of the impulse on the angular velocity of the objects (Eq. 7.61).

$$\begin{aligned} \mathrm{v}_A^+(t) &= \mathrm{v}_A^-(t) + \frac{j\mathrm{N}}{M_A} \\ \mathrm{v}_B^+(t) &= \mathrm{v}_B^-(t) + \frac{j\mathrm{N}}{M_B} \end{aligned} \tag{7.60}$$

$$\begin{aligned} \omega_A^+(t) &= \omega_A^-(t) + I_A^-(r_A(t) \times jN) \\ \omega_B^+(t) &= \omega_B^-(t) + I_B^-(r_B(t) \times jN) \end{aligned} \tag{7.61}$$

To solve for the impulse, form the difference between the velocities of the contact points after collision in the direction of the normal to the surface of contact (Eq. 7.62). The version of Equation 7.59 for velocities after collision is substituted into Equation 7.62. Equations 7.60 and 7.61 are then substituted into that to produce Equation 7.63. Finally, substituting into Equation 7.56 and solving for j produces Equation 7.64.

$$\mathrm{v}_A^+(t) = ((\dot{\mathrm{p}}_A^+(t) - \dot{\mathrm{p}}_B^+(t)) \cdot \mathrm{N})\mathrm{N} \tag{7.62}$$

$$\mathrm{v}_{rel}^+(t) = \mathrm{N}(\mathrm{v}_A^+(t) + \omega_A^+(t) \times \mathrm{r}_A(t) - (\mathrm{v}_B^+(t) + \omega_B^+ \times \mathrm{r}_B(t))) \tag{7.63}$$

$$j = \frac{-((1+k)v_{rel}^-}{\frac{1}{M_A} + \frac{1}{M_B} + \mathrm{N}(I_A^{-1}(t)(\mathrm{r}_A(t) \times \mathrm{N}) \times \mathrm{r}_A(t) - I_B^{-1}(t)(\mathrm{r}_B(t) \times \mathrm{N}) \times \mathrm{r}_B(t))} \tag{7.64}$$

Contact between two objects is defined by the point on each object involved in the contact and the normal to the surface of contact. A point of contact is tested to see if an actual collision is taking place. When collision is tested for, the velocity of the contact point on each object is calculated. A collision is detected if the component of the relative velocity of the two points in the direction of the normal indicates the contact points are approaching each other.

```
Compute dpA, dpB                    ;Eq. 7.60
Vrelative <- dot(N,( dpA - dpB)     ;Eq. 7.63
if Vrelative > threshold
        compute j                   ;Eq. 7.65
        J <- j*N
        PA <- PA + J                ;update object A linear momentum
        PB <- PB - J                ;update object B linear momentum
        LA <- LA + rA x J           ;update object A angular momentum
        LB <- LB - rB x J           ;update object B angular momentum
else if Vrelative > -threshold
        resting contact
else
        objects are moving away from each other
```

FIGURE 7.24

Computing the impulse force of collision.

If there is a collision, then Equation 7.64 is used to compute the magnitude of the impulse (Figure 7.24). The impulse is used to scale the contact normal, which can then be used to update the linear and angular momenta of each object.

If there is more than one point of contact between two objects, then each must be tested for collision. Each time a collision point is identified, it is processed for updating the momentum as above. If any collision is processed in the list of contact points, then after the momenta have been updated, the contact list must be traversed again to see if there exist any collision points with the new object momenta. Each time one or more collisions are detected and processed in the list, the list must be traversed again; this is repeated until it can be traversed and no collisions are detected.

Friction

An object resting on the surface of another object (the *supporting object*) is referred to as being in a state of *resting contact* with respect to that supporting object. In such a case, any force acting on the first object is decomposed into a component parallel to the contact surface and a component in the direction of the normal of the contact surface, called the *normal force*, $\boldsymbol{F_N}$ (Figure 7.25). If the normal force is

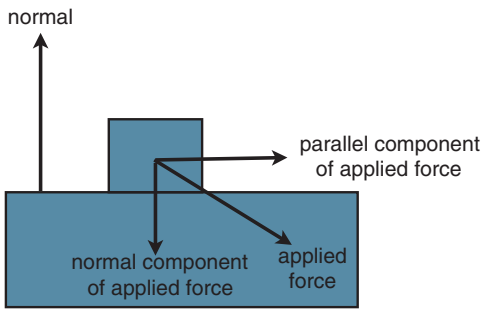


FIGURE 7.25

Normal and parallel components of applied force.

toward the supporting object, and if the supporting object is considered immovable, such as the ground or a table, the normal force is immediately and completely canceled by a force equal and opposite in direction that is supplied by the supporting object. If the supporting object is movable, then the normal force is applied transitively to the supporting object and added to the forces being applied to the supporting object. If the normal force is directed away from the supporting object, then it is simply used to move the object up and away from the supporting object.

The component of the force applied to the object that is parallel to the surface is responsible for sliding (or rolling) the object along the surface of the supporting object. If the object is initially stationary with respect to the supporting object, then there is typically some threshold force, due to *static friction*, that must be exceeded before the object will start to move. This static friction force, F_s, is a function of the normal force, F_N (Eq. 7.65). The constant μ_s is the coefficient of static friction and is a function of the two surfaces in contact.

$$F_{\mathrm{s}} = \mu_{\mathrm{s}} F_{\mathrm{N}} \tag{7.65}$$

Once the object is moving along the surface of the supporting object, there is a *kinetic friction* that works opposite to the direction of travel. This friction creates a force, opposite to the direction of travel, which is a linear function of the normal force, $\boldsymbol{F}_k$, on the object (Eq. 7.66). The constant μ_k is the coefficient of kinetic friction and is a function of the two surfaces in contact.

$$F_{\mathrm{k}} = \mu_{\mathrm{k}} F_{\mathrm{N}} \tag{7.66}$$

Resting contact

Computing the forces involved in resting contact is one of the more difficult dynamics problems for computer animation. The exposition here follows that found in Baraff's work [1]. The solution requires access to quadratic programming, the implementation of which is beyond the scope of this book. An example situation in which several objects are resting on one another is shown in Figure 7.26.

For each contact point, there is a force normal to the surface of contact, just as in colliding contact. The objective is to find the magnitude of that force for a given configuration of objects. These forces (1)

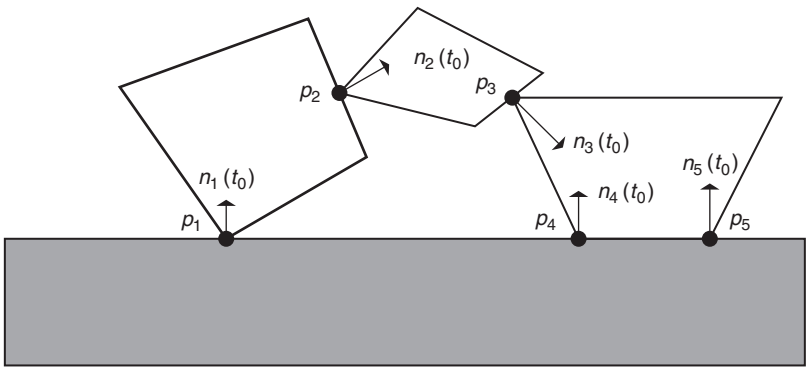


FIGURE 7.26

Multiple-object resting contacts.

have to be strong enough to prevent interpenetration; (2) must only push objects apart, not together; and (3) have to go to zero at the point of contact at the moment that objects begin to separate.

To analyze what is happening at a point of contact, use a distance function, $d_i(t)$, which evaluates to the distance between the objects at the ith point of contact. Assuming objects A and B are involved in the ith contact and the points involved in the ith contact are p_A and p_B from the respective objects, then the distance function is given by Equation 7.67.

$$d_i(t) = (\mathrm{p}_A(t) - \mathrm{p}_B(t)) \cdot \mathrm{N}_i \tag{7.67}$$

If $d_i(t)$ is zero, then the objects involved are still in contact. Whenever $d_i(t) > 0$, the objects are separated. One of the objectives is to avoid $d_i(t) < 0$, which would indicate penetration.

Assume at some time t_0, the distance between objects is zero. To prevent object penetration from time t_0 onward, the relative velocity of the two objects must be greater than or equal to zero, $\dot{d}_i(t_0) \geq 0$ for $t > t_0$. The equation for relative velocity is produced by differentiating Equation 7.67 and is shown in Equation 7.68. At time t_0, the objects are touching, so $p_A(t_0) = p_B(t_0)$. This results in Equation 7.69. In addition, for resting contact, $\dot{d}_i(t_0) = 0$.

$$\dot{d}_i(t) = \dot{\mathrm{N}}_i(t) \cdot (\mathrm{p}_A(t) - \mathrm{p}_B(t)) + \mathrm{N}_i \cdot (\dot{\mathrm{p}}_A(t) - \dot{\mathrm{p}}_B(t)) \tag{7.68}$$

Since $d_i(t_0) = \dot{d}_i(t_0) = 0$, penetration will be avoided if the second derivative is greater than or equal to zero, $\ddot{d}_i(t) \geq 0$. The second derivative is produced by differentiating Equation 7.68 as shown in Equation 7.69. At t_0, remembering that $p_A(t_0) = p_B(t_0)$, one finds that the second derivative simplifies as shown in Equation 7.70. Notice that Equation 7.70 further simplifies if the normal to the surface of contact does not change ($\dot{n}_i(t_0) = 0$).

$$\ddot{d}_i(t) = (\mathrm{p}_A(t) - \mathrm{p}_B(t)) \cdot \ddot{\mathrm{N}}_i + (\dot{\mathrm{p}}_A(t) - \dot{\mathrm{p}}_B(t)) \cdot \dot{\mathrm{N}}_i + (\ddot{\mathrm{p}}_A(t) - \ddot{\mathrm{p}}_B(t)) \cdot \mathrm{N}_i \tag{7.69}$$

$$\ddot{d}_i(t) = 2(\dot{\mathrm{p}}_A(t_0) - \dot{\mathrm{p}}_B(t_0)) \cdot \dot{\mathrm{N}}_i + (\ddot{\mathrm{p}}_A(t_0) - \ddot{\mathrm{p}}_B(t_0)) \cdot \mathrm{N}_i \tag{7.70}$$

The constraints on the forces as itemized at the beginning of this section can now be written as equations: the forces must prevent penetration (Eq. 7.71); the forces must push objects apart, not together (Eq. 7.72); and either the objects are not separating or, if the objects are separating, then the contact force is zero (Eq. 7.73).

$$d_i(t) \geq 0 \tag{7.71}$$

$$f_i \geq 0 \tag{7.72}$$

$$\ddot{d}_i(t) f_i = 0 \tag{7.73}$$

The relative acceleration of the two objects at the ith contact point, $\ddot{d}_i(t_0)$, is written as a linear combination of all of the unknown f_is (Eq. 7.74). For a given contact, j, its effect on the relative acceleration of the bodies involved in the ith contact point needs to be determined. Referring to Equation 7.70, the component of the relative acceleration that is dependent on the velocities of the points, $2 \cdot \dot{n}_i(t_0)(\dot{p}_A(t_0) - \dot{p}_B(t_0))$, is not dependent on the force f_j and is therefore part of the b_i term. The component of the relative acceleration dependent on the accelerations of the points involved in the contact, $n_i(t_0)(\ddot{p}_A(t_0) - \ddot{p}_B(t_0))$, is dependent on f_j if object A or object B is involved in the jth contact. The acceleration at a point on an object arising from a force can be determined by differentiating Equation 7.59, producing Equation 7.75 where $r_A(t) = p_A(t) - x_A(t)$.

$$\ddot{d}_i(t) = b_i + \sum_{j=1}^{n}(a_{ij}f_j) \tag{7.74}$$

$$\ddot{\mathrm{p}}_A(t) = \dot{\mathrm{v}}_A + \dot{\omega}_A(t) \times \mathrm{r}_A(t) + \omega_A(t) \times (\omega_A(t) \times \mathrm{r}_A(t)) \tag{7.75}$$

In Equation 7.75, $\dot{v}_A(t)$ is the linear acceleration as a result of the total force acting on object A divided by its mass. A force f_j acting in direction $n_j(t_0)$ produces $f_j / m_A \cdot n_j(t_0)$. Angular acceleration, $\dot{\omega}_A(t)$, is formed by differentiating Equation 7.48 and produces Equation 7.76, in which the first term contains torque and therefore relates the force f_j to $\ddot{p}_A(t)$ while the second term is independent of the force f_j and so is incorporated into a constant term, b_i.

$$\ddot{\omega}_A(t) = I_A^{-1}(t)\tau_A(t) + I^{-1}(t)(L_A(t) \times \omega_A(t)) \tag{7.76}$$

The torque from force f_j is $(p_j - x_A(t_0)) \times f_j\, n_j(t_0)$. The total dependence of $\ddot{p}_A(t)$ on f_j is shown in Equation 7.77. Similarly, the dependence of $\ddot{p}_B(t)$ on f_j can be computed. The results are combined as indicated by Equation 7.70 to form the term a_{ij} as it appears in Equation 7.74.

$$\mathrm{f}_j\left(\frac{\mathrm{N}_j(t_0)}{m_A} + (I_A^{-1}(t_0)(\mathrm{p}_j - \mathrm{x}_A(t_0) \times \mathrm{N}_j(t_0)) \times \mathrm{r}_A\right) \tag{7.77}$$

Collecting the terms not dependent on an f_j and incorporating a term based on the net external force, $F_A(t_0)$, and net external torque, $\tau_A(t_0)$ acting on A, the part of $\ddot{p}_A(t)$ independent of the f_j is shown in Equation 7.78. A similar expression is generated for $\ddot{p}_B(t)$. To compute b_i in Equation 7.74, the constant parts of $\ddot{p}_A(t)$ and $\ddot{p}_B(t)$ are combined and dotted with $n_i(t_0)$. To this, the term $2\dot{n}_i(t_0)(\dot{p}_A(t_0) - \dot{p}_B(t_0))$ is added.

$$\frac{\mathrm{F}_A(t_0)}{m_A} + I_A^{-1}(t)\tau_A(t) + \omega_A(t) \times \left(\omega_A(t) \times \mathrm{r}_A\right) + (I_A^{-1}(t)(L_A(t) \times \omega_A(t))) \times \mathrm{r}_A \tag{7.78}$$

Equation 7.74 must be solved subject to the constraints in Equations 7.72 and 7.73. These systems of equations are solved by *quadratic programming*. It is nontrivial to implement. Baraff [1] uses a package from Stanford [7] [8] [9] [10]. Baraff notes that the quadratic programming code can easily handle $\ddot{d}_i(t) = 0$ instead of $\ddot{d}_i(t) \geq 0$ in order to constrain two bodies to never separate. This enables the modeling of hinges and pin-joints.

7.4.3 Dynamics of linked hierarchies

Applying forces to a linked figure, such as when falling under the force of gravity or taking a blow to the chest from a heavy object, results in complex reactions as the various links pull and push on each other. There are various ways to deal with the resulting motion depending on the quality of motion desired and the amount of computation tolerated. We consider two approaches here: constrained dynamics and the Featherstone equations.

Constrained dynamics

Geometric point constraints can be used as an alternative to rigid body dynamics [18]. In a simple case, particles are connected together with distance constraints. A mass particle (e.g., representing the torso of a figure) reacts to an applied force resulting in a spatial displacement. The hierarchy is traversed starting at the initially displaced particle. During traversal, each particle is repositioned in order to

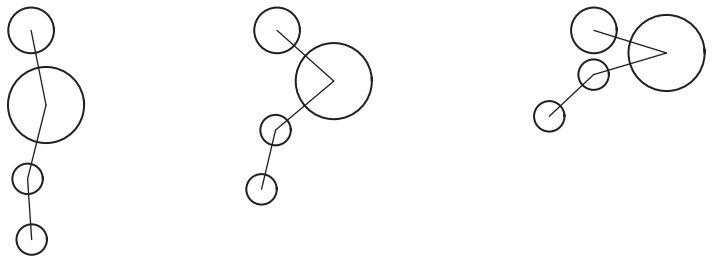


FIGURE 7.27

Frames from a sequence of constrained dynamics. The large sphere's motion is driven by outside forces; the small spheres are repositioned to satisfy distance constraints.

reestablish the distance constraint with respect to a particle already processed. Figure 7.27 shows frames from a simple example.

When an impulse force is applied to the torso of a figure, the torso reacts to the force as an independent rigid body. The appendages react to the force by enforcing distance constraints one link at a time traversing the hierarch from the root outwards as shown in Figure 7.28.

In reality, however, as the torso reacts to the applied force, the appendages exert forces and torques on the torso as it tries to move away from them. The full equations of motion must account for the dynamic interactions among all the links. The *Featherstone equations* do this.

The Featherstone equations

The actual forces of one link acting on adjacent links can be explicitly calculated using basic principles of physics. The effect of one link on any attached links must be computed. The equations presented here are referred to as the Featherstone equations of motion [5] [14] [15]. In a nutshell, the algorithm uses four hierarchy-traversal loops: initializing link velocities (from root outward), initializing link values (from root outward), updating values (from end effector inward), and computing accelerations (from root outward).

The notation to relate one link to the next is shown in Figure 7.29.

A *spatial notation*, which combines the angular and linear components, has been developed to make the computations more concise [5]. The spatial velocity vector is a six-element vector of angular velocity and the linear velocity (Eq. 7.79).

$$\hat{\mathrm{v}} = \begin{bmatrix} \omega \\ \mathrm{v} \end{bmatrix} \tag{7.79}$$

Similarly, the spatial acceleration vector is made of angular and linear acceleration terms (Eq. 7.80).

FIGURE 7.28

Sequence of impulse force applied to linked figure. Motion is generated by constrained dynamics.

(Image courtesy of Domin Lee.)

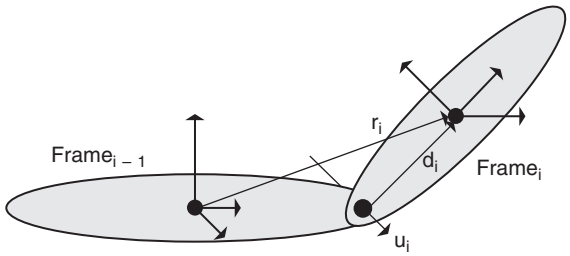


FIGURE 7.29

Vectors relating one coordinate frame to the next: u_i is the axis of revolution associated with *Frame*$_i$, r_i is the displacement vector from the center of *Frame*$_{i-1}$ to the center of *Frame*$_i$; d_i is the displacement vector from the axis of revolution to the center of *Frame*$_i$.

$$\hat{\mathrm{a}} = \begin{bmatrix} \alpha \\ \mathrm{a} \end{bmatrix} \tag{7.80}$$

The force vector (Eq. 7.81) and mass matrix (Eq. 7.82) are defined so that $\hat{f} = \hat{M}\hat{a}$ holds in the spatial notation.

$$\hat{\mathrm{f}} = \begin{bmatrix} f \\ \tau \end{bmatrix} \tag{7.81}$$

$$\hat{M} = \begin{bmatrix} 0 & M \\ I & 0 \end{bmatrix} \tag{7.82}$$

The spatial axis is given by combining the axis of rotation of a revolute joint and the cross-product of that axis with the displacement vector from the axis to the center of the frame (Eq. 7.83).

$$\hat{S}_i = \begin{bmatrix} u_i \\ u_i \times d_i \end{bmatrix} \tag{7.83}$$

Using Equation 7.84 to represent the matrix cross-product operator, if r is the translation vector and R is the rotation matrix between coordinate frames, then Equation 7.85 transforms a value from frame F to frame G.

$$r^* = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} \tag{7.84}$$

$$G\hat{\times}F = \begin{bmatrix} R & 0 \\ r^*R & R \end{bmatrix} \tag{7.85}$$

Armed with this notation, a procedure that implements the Featherstone equations can be written succinctly [5] [15]. The interested reader is encouraged to refer to Mirtich's Ph.D. dissertation [15] for a more complete discussion.

Computing accurate dynamics is a nontrivial thing to do. However, because of the developed literature from the robotics community and the interest from the computer animation community, understandable presentations of the techniques necessary are available to the mathematically inclined. For real-time applications, approximate methods that produce visually reasonable results are very accessible.

7.5 Cloth

Cloth is a type of object that is commonly encountered in everyday living in a variety of forms— most notably in the form of clothes, but also as curtains, tablecloths, flags, and bags. Because of its ubiquity, it is a type of object worth devoting special attention to modeling and animating. Cloth is any man-made fabric made out of fibers. It can be woven, knotted, or pressed (e.g, felt). The distinguishing attributes of cloth are that it is thin, flexible, and has some limited ability to be stretched. In addition, there are other cloth-like objects, such as paper and plastic bags, that have similar attributes that can be modeled with similar techniques. However, for purposes of this discussion, woven cloth, a frequently encountered type of cloth-like object, will be considered.

Woven cloth, as well as some knotted cloth, is formed by a rectilinear pattern of threads. The two directions of the threads are typically referred to as *warp*, in the longitudinal direction, and *weft*, in the side-to-side direction and orthogonal to the warp direction. The thread patterns of cloth vary by the over-under structure of the threads. For example, in a simple weave, each thread of the warp goes over one weft thread and then under the next. The over-under of adjacent warp rows are offset by one thread. This same over-under pattern holds for the weft threads relative to the warp threads. Other patterns can be used, such as over-over-under and over-over-under-under (Figure 7.30). For more variety, different over-under patterns can be used in the weft and warp directions. Such differences in the weave pattern make the material more or less subject to shear and bend.

Because cloth is thin, it is usually modeled as a single sheet of geometric elements, that is, it is not modeled as having an interior or having any depth. Because it is flexible and distorts easily, cloth of any significant size is usually modeled by a large number of geometric elements typically on the order of hundreds, thousands, or even tens of thousands of elements. The number of elements depends on the desired quality of the resulting imagery and the amount of computation that can be tolerated. Because of the large number of elements, it is difficult for the animator to realistically animate cloth using simple positioning tools, so more automatic, and computationally intensive, approaches are employed. Major cloth features, such as simple draping, can be modeled directly or the underlying processes that give rise to those features can be modeled using physically based methods.

The main visual feature of cloth is simple draping behavior. This draping behavior can be modeled kinematically— just by modeling the fold that is generated between fixed points of the cloth (see the following section). This is possible in simple static cases where the cloth is supported at a few specific points. However, in environments where the cloth is subject to external forces, a more physically based approach is more appropriate. Because cloth can be stretched to some limited degree, a common approach to modeling cloth is the use a mass-spring-damper system. Modifying the mass-spring-damper parameters provides for some degree of modeling various types of cloth material and various types of cloth weave. The spring model, being physically based, also allows the cloth mesh to react to external forces from the environment, such as gravity, wind, and collisions. Real-time systems, such as computer games or interactive garment systems, place a heavy demand on computational efficiency not required of off-line applications and techniques have to be selected with an appropriate quality-cost

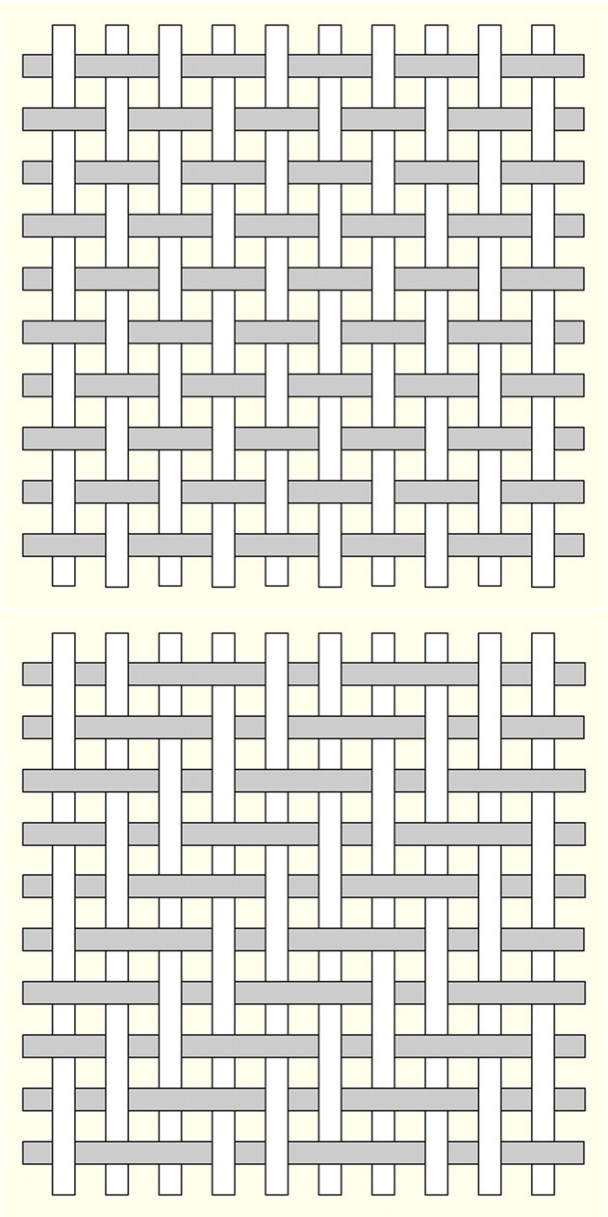


FIGURE 7.30

Example warp and weft patterns. Top: over-under. Bottom: over-over-under-under.

(Image courtesy of Di Cao.)

trade-off. At the other extreme, the approach of using finite element methods for modeling can provide very accurate simulation results, but it will not be discussed here.

7.5.1 Direct modeling of folds

With cloth, as with a lot of phenomena, the important superficial features can be modeled directly as opposed to using a more computationally expensive physically based modeling approach. In the case of cloth an important feature is the cloth fold that forms between two supported points. In the special case of a cloth supported at a fixed number of constrained points, the cloth will drape along well-defined lines. Weil [28] presents a geometric method for hanging cloth from a fixed number of points. The cloth is represented as a two-dimensional grid of points located in three-space with a certain number of the grid points constrained to fixed positions.

The procedure takes place in two stages. In the first stage, an approximation is made to the draped surface within the convex hull of the constrained points. The second stage is a relaxation process that continues until the maximum displacement during a given pass falls below a user-defined tolerance.

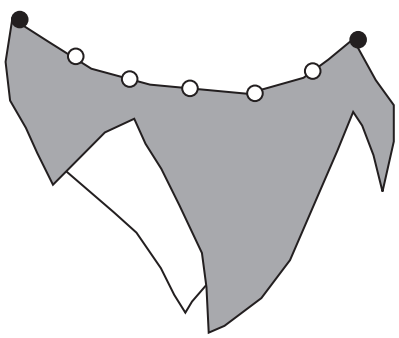
Vertices on the cloth grid are classified as either *interior* or *exterior*, depending on whether they are inside or outside the convex hull of the constrained points. This inside/outside determination is based on two-dimensional grid coordinates. The first step is to position the vertices that lie along the line between constrained vertices. The curve of a thread hanging between two vertices is called a *catenary curve* and has the form shown in Equation 7.86.

$$y = c + \left(a \cdot \cos \mathrm{h}\left(\frac{x - b}{a} \right) \right) \tag{7.86}$$

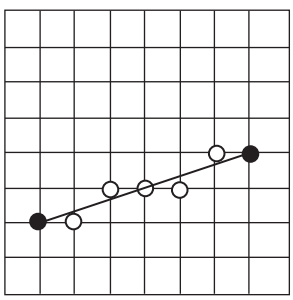
Catenary curves traced between paired constrained points are generated using vertices of the cloth that lie along the line between constrained vertices. The vertices between the constrained vertices are identified in the same way that a line is drawn between points on a raster display (Figure 7.31).

If two catenary curves cross each other in grid space (Figure 7.32), then the lower of the two curves is simply removed. The reason for this is that the catenary curves essentially support the vertices along the curve. If a vertex is supported by two curves, but one curve supports it at a higher position, then the higher curve takes precedence.

A catenary curve is traced between each pair of constrained vertices. After the lower of two crossing catenary curves is removed, a triangulation is produced in the grid space of the constrained vertices



Cloth supported at two constrained points



Constrained points in grid space

FIGURE 7.31

Constrained cloth and grid coordinate space.

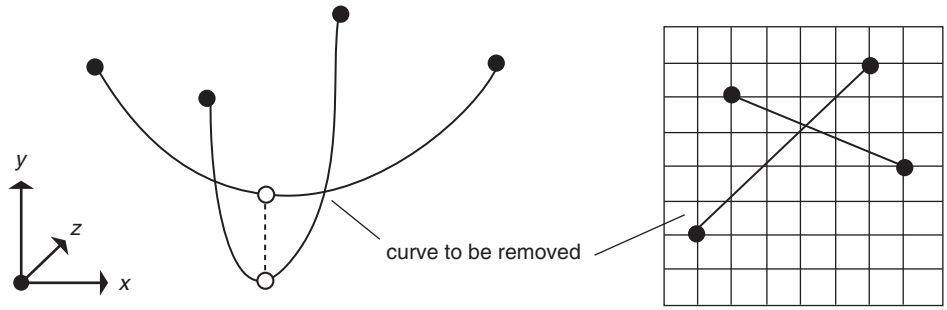


FIGURE 7.32

Two catenary curves supporting the same point.

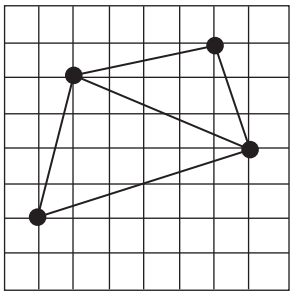


FIGURE 7.33

Triangulation of constrained points in grid coordinates.

(Figure 7.33). The vertices of the grid that fall on the lines of triangulation are positioned in three-space according to the catenary equations. To find the catenary equation between two vertices (x_1, y_1), (x_2, y_2) of length L, see Equation 7.87. Each of these triangles is repeatedly subdivided by constructing a catenary from one of the vertices to the midpoint of the opposite edge on the triangle. This is done for all three vertices of the triangle. The highest of the three catenaries so formed is kept and the others are discarded. This breaks the triangle into two new triangles (Figure 7.34). This continues until all interior vertices have been positioned.

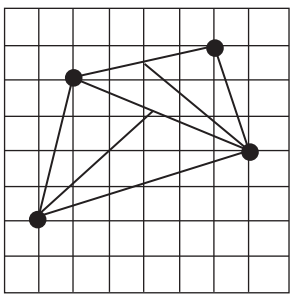


FIGURE 7.34

Subdividing triangles.

$$
\begin{aligned}
y_1 &= c + a \cdot \cosh\left[\frac{(x_1 - b)}{a}\right] \\
y_2 &= c + a \cdot \cosh\left[\frac{(x_2 - b)}{a}\right] \\
L &= a \cdot \sinh\left[\frac{(x_2 - b)}{a}\right] - a \cdot \sinh\left[\frac{(x_1 - b)}{a}\right] \\
\sqrt{L^2 - (y_2 - y_1)^2} &= 2 \cdot a \cdot \sinh\left[\frac{(x_2 - x_1)}{2 \cdot a}\right] (a \text{ can be solved for numerically at this point}) \\
M &= \sinh\left(\frac{x_2}{a}\right) - \sinh\left(\frac{x_1}{a}\right) \\
N &= \cosh\left(\frac{x_2}{a}\right) - \cosh\left(\frac{x_1}{a}\right) \\
\text{if } N > M \quad \mu &= \tanh^{-1}\left(\frac{M}{N}\right) \\
Q &= \frac{M}{\sinh(\mu)} = \frac{N}{\cosh(\mu)} \\
b &= a \cdot \left[\mu - \sinh^{-1}\left(\frac{L}{Q \cdot a}\right)\right] \\
\text{if } M > N \quad \mu &= \tanh^{-1}\left(\frac{N}{M}\right) \\
Q &= \frac{N}{\sinh(\mu)} = \frac{M}{\cosh(\mu)} \\
b &= a \cdot \left[\mu - \cosh^{-1}\left(\frac{L}{Q \cdot a}\right)\right]
\end{aligned}
\tag{7.87}
$$

A relaxation process is used as the second and final step in positioning the vertices. The effect of gravity is ignored; it has been used implicitly in the formation of the catenary curves using the interior vertices. The exterior vertices are initially positioned to affect a downward pull. The relaxation procedure repositions each vertex to satisfy unit distance from each of its neighbors. For a given vertex, displacement vectors are formed to each of its neighbors. The vectors are added to determine the direction of the repositioning. The magnitude of the repositioning is determined by taking the square root of

the average of the squares of the distance to each of the neighbors. Self-intersection of the cloth is ignored. To model material stiffness, the user can add dihedral angle springs to bias the shape of the material.

Animation of the cloth is produced by animating the existence and/or position of constrained points. In this case, the relaxation process can be made more efficient by updating the position of points based on positions in the last time step. Modeling cloth this way, however, is not appropriate for many situations, such as clothing, in which there are a large number of closely packed contact points. In these situations, the formation of catenary curves is not an important aspect, and there is no provision for the wrinkling of material around contact points. Therefore, a more physically based approach is often needed.

7.5.2 Physically based modeling

In many cases, a more useful approach for modeling and animating cloth is to develop a physically based model of it. For example, see Figure 7.35. This allows the cloth to interact and respond to the environment in fairly realistic ways. The cloth weave is naturally modeled as a quadrilateral mesh that mimics the warp and weft of the threads. The characteristics of cloth are its ability to stretch, bend, and skew. These tendencies can be modeled as springs that impart forces or as energy functions that contribute to the overall energy of a given configuration. Forces are used to induce accelerations on system masses. The energy functions can be minimized to find optimal configurations or differentiated to find local force gradients. Whether forces or energies are used, the functional forms are similar and are usually based on the amount some metric deviates from a rest value.

Stretching is usually modeled by simply measuring the amount that the length of an edge deviates from its rest length. Equation 7.88 shows a commonly used form for the force equation of the corresponding spring. $v1^*$ and $v2^*$ are the rest positions of the vertices defining the edge; $v1$ and $v2$ are the current positions of the edge vertices. Notice that the force is a vector equation, whereas the energy equation is a scalar. The analogous energy function is given in Equation 7.89. The metric has been normalized by the rest length ($|v1^* - v2^*|$).

FIGURE 7.35

Cloth example.

(Image courtesy of Di Cao).

$$F_s = \left(\frac{k_s|v1 - v2| - |v1^* - v2^*|}{|v1^* - v2^*|}\right)\frac{v1 - v2}{|v1 - v2|} \tag{7.88}$$

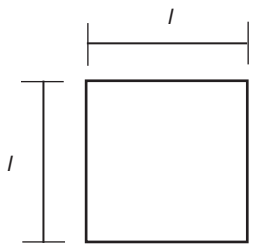
$$E_s = k_s\frac{1}{2}\left(\frac{|v1 - v2| - |v1^* - v2^*|}{|v1^* - v2^*|}\right)^2 \tag{7.89}$$

Restricting the stretching of edges only controls changes to the surface area of the mesh. Skew is in-plane distortion of the mesh, which still maintains the length of the original edges (Figure 7.36a,b). To control such distortion (when using forces), one may employ diagonal springs (Figure 7.36c). The energy function suggested by DeRose, Kass, and Truong [29] to control skew distortion is given in Equation 7.90.

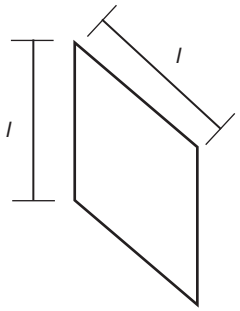
$$S(v1\,,\,v2) = \left(\frac{1}{2}\right)\left(\frac{|v1 - v2| - |v1^* - v2^*|}{|v1^* - v2^*|}\right)^2 \tag{7.90}$$

$$E_w = k_w \cdot S(v1, v3)\; S(v2, v4)$$

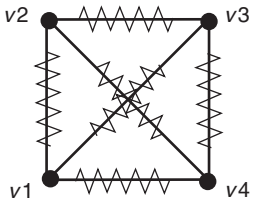
Edge and diagonal springs (energy functions) control in-plane distortions, but out-of-plane distortions are still possible. These include the bending and folding of the mesh along an edge that does not



A Original quadrilateral of mesh



B Skew of original quadrilateral without changing the length of edges



C Diagonal springs to control skew

FIGURE 7.36

Original quadrilateral, skewed quadrilateral, and controlling skew with diagonal springs.

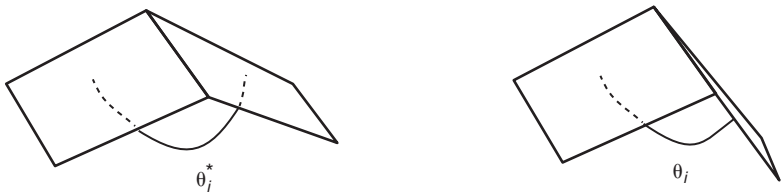


FIGURE 7.37

Control of bending by dihedral angle.

change the length of the edges or diagonal measurements. Bending can be controlled by either restricting the dihedral angles (the angle between adjacent quadrilaterals) or controlling the separation among a number of adjacent vertices. A spring-type equation based on deviation from the rest angle can be used to control the dihedral angle: $F_b = k_b?(\theta_i - \theta_i^*)$ (Figure 7.37). Bending can also be controlled by considering the separation of adjacent vertices in the direction of the warp and weft of the material (Figure 7.38). See Equation 7.91 for an example.

The mass-spring-damper system can then be updated according to basic principles of physics and the cloth model can respond when subjected to external forces. Whether spring-like forces or energy functions are used, the constants are globally and locally manipulated to impart characteristics to the mesh. However, choosing appropriate constants can be an issue in trying to get a desired amount of stretching, bending, and shearing.

For an effective cloth model, there are several issues that deserve special attention. First, the integration method used to update the physical model has an effect on the accuracy of the model as well as on the computational efficiency. The computational efficiency is especially important in real-time systems. Second, a simple spring model can allow for unrealistic stretching of the cloth, referred to as *super-elasticity*. Cloth models typically use something other than the basic linear spring. Third, collision detection can be a large part of the computational effort in animating cloth. This is especially true for clothes where the cloth is in near-constant contact with the underlying figure geometry. Fourth, collision response has to be handled appropriately for the inelastic properties of cloth. Cloth does not bounce away from collisions as much as it distorts, often taking the shape of the underlying surface. Fifth, unlike a spring, cloth does not really compress. Instead, cloth bends, forming folds and wrinkles.

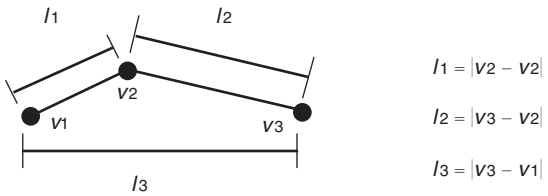


FIGURE 7.38

Control of bending by separation of adjacent vertices.

Special attention to bending can make a cloth simulation more efficient and more realistic. These issues are discussed in the following paragraphs.

Typical simulations of cloth use explicit, or forward, Euler integration, especially for real-time applications because it is fast to execute and easy to implement. But because of the stiffness encountered in cloth simulations, which explicit Euler integration does not handle well, other integration schemes have been used, especially for off-line applications. Implicit Euler is used but, because it is iterative, it tends to be too computationally expensive for real-time or interactive applications. However, it does provide a more stable simulation for the stiffer models. Semi-implicit Euler integration has been used with success. This integration technique is easy to implement, is fast to execute, and handles stiffness fairly well. See Appendix B.8 for details.

In modeling cloth, controlling stretching is a primary concern. When cloth is pulled, there is typically some initial stretching that occurs as the weave straightens out, but the cloth soon becomes very stiff and resistant to further stretching. If the cloth is modeled using a basic mass-spring-damper system, the pulled cloth will not become stiff and will continue to stretch in unrealistic ways because of the linear response inherent in the spring-damper equations. This is referred to as super-elasticity. This pulling can simply be the result of gravity as it pulls down on all the mass points of the cloth model when the cloth is suspended along one edge. To prevent this super-elasticity of the cloth model, either very stiff springs must be used, requiring very small time steps and limiting the amount of initial stretching, or some nonlinear effects must be captured by the model. Approaches to prevent super-elasticity include biphasic springs and velocity clamping. Biphasic springs are springs that respond differently to large displacements. Under small displacements, they allow the initial stretching, but at a certain displacement, they become very stiff. This still requires small time steps at large displacements. Velocity clamping limits the velocity of the mass points due to spring response and thus is a kinematic solution to controlling vertex displacement.

One of the major activities of cloth animation is to detect and respond to collisions. Especially in real-time environments, the efficient detection and response of collisions is a primary concern where computational shortcuts are often worth sacrificing accuracy. There are two types of collision: self-collision and collision with the environment. These two can be handled separately because knowledge about the cloth configuration can help in processing self-collisions. Collisions with cloth are handled much like collisions with elements of any complex environment; that is, they must be handled efficiently. One of the most common techniques for handling collisions in a complex environment is to organize data in a hierarchy of bounding boxes. As suggested by DeRose, Kass, and Truong [29], such a hierarchy can be built from the bottom up by first forming bounding boxes for each of the faces of the cloth. Each face is then logically merged with an adjacent face, forming the combined bounding box and establishing a node one level up in the hierarchy. This can be done repeatedly to form a complete hierarchy of bounding boxes for the cloth faces. Care should be taken to balance the resulting hierarchy by making sure all faces are used when forming nodes at each level of the hierarchy. Because the cloth is not a rigid structure, additional information can be included in the data structure to facilitate updating the bounding boxes each time a vertex is moved. Hierarchies can be formed for rigid and nonrigid structures in the environment. To test a vertex for possible collision with geometric elements, compare it with all the object-bounding box hierarchies, including the hierarchy of the object to which it belongs. Another approach takes advantage of the graphics process unit's on-board z-buffer to detect an underlying surface that shows through the cloth and, therefore, indicates a collision [30].

Cloth response to collisions does not follow the rigid body paradigm. The collision can be viewed as elastic in that the cloth will change shape as a result. It is really a damped local inelastic collision, but rigid body computations are usually too expensive, especially for real-time applications, and even for off-line simulations. The response to collision is typically to constrain the position and/or velocity of the colliding vertices. This will constrain the motion of the colliding cloth to be on the surface of the body. This avoids treating periods of extended contact with a collision detection–collision response cycle.

Folds are a particular visual feature of cloth that are important to handle in a believable way. In the case of spring-damper models of cloth, there is typically no compression because the cloth easily bends, producing folds and wrinkles. It is typical to incorporate some kind of bending spring in the cloth model. Choi and Koh [31] have produced impressive cloth animation by special handling of what they call the post-buckling instability together with the use of implicit numerical integration. A key feature of their approach is the use of a different energy function for stretching than for compression.

$$F_{\mathrm{b}} = k_{\mathrm{b}}\left(\frac{l3}{(l1 + l2)} - \frac{l3^*}{(l1^* + l2^*}\right) \tag{7.91}$$

7.6 Enforcing soft and hard constraints

One of the main problems with using physically based animation is for the animator to get the object to do what he or she wants while having it react to the forces modeled in the environment. One way to solve this is to place constraints on the object that restrict some subset of the degrees of freedom (DOF) of the object's motion. The remaining DOF are subject to the physically based animation system. Constraints are simply requirements placed on the object. For animation, constraints can take the form of colocating points, maintaining a minimum distance between objects, or requiring an object to have a certain orientation in space. The problem for the animation system is in enforcing the constraints while the object's motion is controlled by some other method.

If the constraints are to be strictly enforced, then they are referred to as *hard constraints*. If the constraints are relations the system should only attempt to satisfy, then they are referred to as *soft constraints*. Satisfying hard constraints requires more sophisticated numerical approaches than satisfying soft constraints. To satisfy hard constraints, computations are made that search for a motion that reacts to forces in the system while satisfying all of the constraints. As more constraints are added to the system, this becomes an increasingly difficult problem. Soft constraints are typically incorporated into a system as additional forces that influence the final motion. One way to model flexible objects is to create soft distance constraints between the vertices of an object. These constraints, modeled by a mesh of interconnected springs and dampers, react to other forces in the system such as gravity and collisions to create a dynamic structure. Soft constraint satisfaction can also be phrased as an energy minimization problem in which deviation from the constraints increases the system's energy. Forces are introduced into the system that decrease the system's energy. These approaches are discussed in the following section.

7.6.1 Energy minimization

The concept of a system's energy can be used in a variety of ways to control the motion of objects. Used in this sense, energy is not defined solely as physically realizable, but it is free to be defined in whatever form serves the animator. Energy functions can be used to pin objects together, to restore the shape of objects, or to minimize the curvature in a spline as it interpolates points in space.

As presented by Witkin, Fleischer, and Barr [26], energy constraints induce restoring forces based on arbitrary functions of a model's parameters. The current state of a model can be expressed as a set of parameter values. These are external parameters such as position and orientation as well as internal parameters such as joint angles, the radius of a cylinder, or the threshold of an implicit function. They are any value of the object model subject to change. The set of state parameters will be referred to as ψ.

A constraint is phrased in terms of a non-negative smooth function, $E(\psi)$, of these parameters, and a local minimum of the constraint function is searched for in the space of all parametric values. The local minimum can be searched for by evaluating the gradient of the energy function and stepping in the direction of the negative of the gradient, $-\nabla E$. The gradient of a function is the parameter space vector in the direction of greatest increase of that function. Usually the parametric state set will have quite a few elements, but for illustrative purposes suppose there are only two state parameters. Taking the energy function as the surface of a height function of those two parameters, one can consider the gradient as pointing uphill from any point on that surface (see Figure 7.39).

Given an initial set of parameters, ψ_0, the parameter function at time zero is defined as $F(0) = \psi_0$. From this it follows that $(d/dt)\, F(t) = -\nabla E$. The force vector in parameter space is the negative of the gradient of the energy function, $-\nabla E$. Any of a number of numerical techniques can be used to solve the equation, but a particularly simple one is Euler's method, as shown in Equation 7.92 (see Appendix B.8 or [19] for better methods).

$$F(t_{i+1}) = F(t_i) - h\nabla E \tag{7.92}$$

Determining the gradient, ∇E, is usually done numerically by stepping along each dimension in parameter space, evaluating the energy function, and taking differences between the new evaluations and the original value.

Three useful functions

As presented by Witkin, Fleischer, and Barr [26], three useful functions in defining an energy function are as follows:

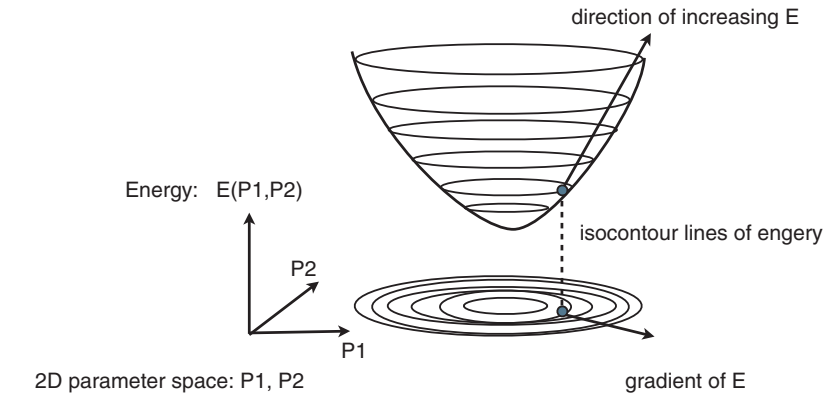


FIGURE 7.39

Sample simple energy function.

1. The parametric position function, $P(u, v)$
2. The surface normal function, $N(u, v)$
3. The implicit function, $I(x)$

The parametric position function and surface normal function are expressed as functions of surface parameters u and v (although for a given model, the surface parameters may be more complex, incorporating identifiers of object parts, for example). Given a value for u and v, the position of that point in space and the normal to the surface at that point are given by the functions. The implicit function takes as its parameter a position in space and evaluates to an approximation of the signed distance to the surface where a point on the surface returns a zero, a point outside the object returns a positive distance to the closest point on the surface, and a point inside the object returns a negative distance to the closest point on the surface. These functions will, of course, also be functions of the current state of the model, ψ.

Useful constraints

The functions above can be used to define several useful constraint functions. The methods of Witkin, Fleischer, and Barr [26] can also lead to several intuitive constraint functions. The functions typically are associated with one or more user-specified weights; these are not shown.

Point-to-fixed-point

The point Q in space is fixed and is given in terms of absolute coordinate values. The point P is a specific point on the surface of the model and is specified by the u, v parameters. The energy function will be zero when these two points coincide.

$$E = |\mathrm{P}(u, v) - \mathrm{Q}|^2$$

Point-to-point

A point on the surface of one object is given and the point on the surface of a second object is given. The energy function will be zero when they coincide. Notice that within a constant, this is a zero-length spring. Also notice that the orientations of the objects are left unspecified. If this is the only constraint given, then the objects could completely overlap or partially penetrate each other to satisfy this constraint.

$$E = |\mathrm{P}^a(u_a, v_a) - \mathrm{P}^b(u_b, v_b)|^2$$

Point-to-point locally abutting

The energy function is zero when the points coincide, and the dot product of the normals at those points is equal to -1 (i.e., they are pointing away from each other).

$$E = |\mathrm{P}^a(u_a, v_a) - \mathrm{P}^b(u_b, v_b)|^2 + \mathrm{N}^a(u_a, v_a) \cdot \mathrm{N}^b(u_b, v_b) + 1.0$$

Floating attachment

With the use of the implicit function of object b, a specific point on object a is made to lie on the surface of object b.

$$E = (\mathrm{I}^b(\mathrm{P}^a(u_a, v_a)))^2$$

Floating attachment locally abutting

A point of object a is constrained to lie on the surface of b, using the implicit function of object b as previously mentioned. In addition, the normal of object a and the point must be colinear to and in the opposite direction of the normal of object b at the point of contact. The normal of object b is computed as the gradient of its implicit function.

Other constraints are possible. Witkin, Fleischer, and Barr [26] present several others as well as some examples of animations using this technique.

$$E = (I^b(\mathrm{P}^a(u_a, v_a)))^2 + \mathrm{N}^a(u_a, v_a) \cdot \frac{\nabla I^b(\mathrm{P}^a(u_a, v_a))}{|\nabla I^b(\mathrm{P}^a(u_a, v_a))|} + 1.0$$

Energy constraints are not hard constraints

While such energy functions can be implemented quite easily and can be effectively used to assemble a structure defined by relationships such as the ones discussed previously, a drawback to this approach is that the constraints used are not *hard constraints* in the sense that the constraint imposed on the model is not always met. For example, if a point-to-point constraint is specified and one of the points is moved rapidly away, the other point, as moved by the constraint satisfaction system, will chase around the moving point. If the moving point comes to a halt, then the second point will, after a time, come to rest on top of the first point, thus satisfying the point-to-point constraint.

7.6.2 Space-time constraints

Space-time constraints view motion as a solution to a constrained optimization problem that takes place over time in space. Hard constraints, which include equations of motion as well as nonpenetration constraints and locating the object at certain points in time and space, are placed on the object. An objective function that is to be optimized is stated, for example, to minimize the amount of force required to produce the motion over some time interval.

The material here is taken from Witkin and Kass [27]. Their introductory example will be used to illustrate the basic points of the method. Interested readers are urged to refer to that article as well as follow-up articles on space-time constraints (e.g., [3] [16]).

Space-time particle

Consider a particle's position to be a function of time, $x(t)$. A time-varying force function, $f(t)$, is responsible for moving the particle. Its equation of motion is given in Equation 7.93.

$$m\ddot{\mathrm{x}}(t) - \mathrm{f}(t) - m\mathrm{g} = 0 \tag{7.93}$$

Given the function $f(t)$ and values for the particle's position and velocity, (t_0) and $\dot{x}(t_0)$, at some initial time, the position function, $x(t)$, can be obtained by integrating Equation 7.93 to solve the initial value problem.

However, the objective here is to determine the force function, $f(t)$. Initial and final positions for the particle are given as well as the times the particle must be at these positions (Eq. 7.94). These equations along with Equation 7.93 are the constraints on the motion.

$$\begin{aligned} \mathrm{x}(t_0) &= \mathrm{a} \\ \mathrm{x}(t_1) &= \mathrm{b} \end{aligned} \tag{7.94}$$

The function to be minimized is the fuel consumption, which here, for simplicity, is given as $|f|^2$. For a given time period $t_0 < t < t_1$, this results in Equation 7.95 as the function to be minimized subject to the time-space constraints and the motion equation constraint.

$$R = \int_{t_0}^{t_1} |\mathrm{f}|^2 \, dt \tag{7.95}$$

In solving this, discrete representations of the functions $x(t)$ and $f(t)$ are considered. Time derivatives of x are approximated by finite differences (Eqs. 7.96 and 7.97) and substituted into Equation 7.93 to form n physics constraints (Eq. 7.98) and the two boundary constraints (Eq. 7.99).

$$\dot{\mathrm{x}}_i = \frac{x_i - x_{i-1}}{h} \tag{7.96}$$

$$\dot{\mathrm{x}}_i = \frac{\mathrm{x}_{i+1} - 2\mathrm{x}_i + \mathrm{x}_{i-1}}{h^2} \tag{7.97}$$

$$\mathrm{p}_i = m\frac{\mathrm{x}_{i+1} - 2\mathrm{x}_i + \mathrm{x}_{i-1}}{h^2} - \mathrm{f} - m\mathrm{g} = 0 \tag{7.98}$$

$$\begin{aligned} c_a &= |\mathrm{x}_1 - \mathrm{a}| = 0 \\ c_b &= |\mathrm{x}_n - \mathrm{b}| = 0 \end{aligned} \tag{7.99}$$

If one assumes that the force function is constant between samples, the object function, R, becomes a sum of the discrete values of f. The discrete function R is to be minimized subject to the discrete constraint functions, which are expressed in terms of the sample values, x_i and f_i, that are to be solved for.

Numerical solution

The problem as stated fits into the generic form of constrained optimization problems, which is to "find the S_j values that minimize R subject to $C_i(S_j) = 0$." The S_j values are the x_i and f_i. Solution methods can be considered black boxes that request current values for the S_j values, R, and the C_i values as well as the derivatives of R and the C_i with respect to the S_j as the solver iterates toward a solution.

The solution method used by Witkin and Kass [27] is a variant of sequential quadratic programming (SQP) [10]. This method computes a second-order Newton-Raphson step in R, which is taken irrespective of any constraints on the system. A first-order Newton-Raphson step is computed in the C_i to reduce the constraint functions. The step to minimize the objective function, R, is projected onto the null space of the constraint step, that subspace in which the constraints are constant to a first-order approximation. Therefore, as steps are taken to minimize the constraints, a particular direction for the step is chosen that does not affect the constraint minimization and that reduces the objective function.

Because it is first-order in the constraint functions, the first derivative matrix (the Jacobian) of the constraint function must be computed (Eq. 7.100). Because it is second-order in the objective function, the second derivative matrix (the Hessian) of the objective function must be computed (Eq. 7.101). The first derivative vector of the objective function must also be computed, $^{\partial}R/^{\partial}S_j$.

$$\mathrm{J}_{ij} = \frac{\partial C_i}{\partial S_j} \tag{7.100}$$

$$\mathrm{H}_{ij} = \frac{\partial^2 R}{\partial S_i \partial S_j} \tag{7.101}$$

The second-order step to minimize R, irrespective of the constraints, is taken by solving the linear system of equations shown in Equation 7.102. A first-order step to drive the C_js to zero is taken by solving the linear system of equations shown in Equation 7.103. The final update is $\Delta S_j = \hat{S}_j + \tilde{S}_j$. The algorithm reaches a fixed point when the constraints are satisfied, and any further step that minimizes R would violate one or more of the constraints.

$$-\frac{\partial}{\partial S_j}(R) = \sum_j H_{ij}\hat{S}_j \tag{7.102}$$

$$-C_i = \sum_j J_{ij}(\hat{S}_j + \tilde{S}_j) \tag{7.103}$$

Although one of the more complex methods presented here, space-time constraints are a powerful and useful tool for producing realistic motion while maintaining given constraints. They are particularly effective for creating the illusion of self-propelled objects whose movement is subject to user-supplied time constraints.

7.7 Chapter summary

If complex realistic motion is needed, the models used to control the motion can become complex and mathematically expensive. However, the resulting motion is oftentimes more natural looking and believable than that produced using kinematic (e.g. interpolation) techniques. While modeling physics is a nontrivial task, oftentimes it is worth the investment. It should be noted that there has been much work in this area that has not been included in this chapter. Some other physically based techniques are covered in later chapters that address specific animation tasks such as the animation of clothes and water.

References

[1] Baraff D. Rigid Body Simulation. In: Course Notes, SIGGRAPH 92. Chicago, Ill.; July 1992.
[2] Blinn J. Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. In: Computer Graphics. Proceedings of SIGGRAPH 82, vol. 16(3). Boston, Mass.; July 1982. p. 21–9.
[3] Cohen M. Interactive Spacetime Control for Animation. In: Catmull EE, editor. Computer Graphics. Proceedings of SIGGRAPH 92, vol. 26(2). Chicago, Ill.; July 1992. p. 293–302. ISBN 0-201-51585-7.
[4] Ebert D, Carlson W, Parent R. Solid Spaces and Inverse Particle Systems for Controlling the Animation of Gases and Fluids. Visual Computer March 1994;10(4):179–90.
[5] Featherstone R, Orin D. Robot Dynamics: Equations and Algorithms.
[6] Frautschi S, Olenick R, Apostol T, Goodstein D. The Mechanical Universe: Mechanics and Heat, Advanced Edition. Cambridge: Cambridge University Press; 1986.
[7] Gill P, Hammarling S, Murray W, Saunders M, Wright M. User's Guide for LSSOL: A Fortran Package for Constrained Linear Least-Squares and Convex Quadratic Programming, Technical Report Sol 84-2. Systems Optimization Laboratory, Department of Operations Research, Stanford University; 1986.
[8] Gill P, Murray W, Saunders M, Wright M. User's Guide for NPSOL: A Fortran Package for Nonlinear Programming, Technical Report Sol 84-2. Systems Optimization Laboratory. Department of Operations Research, Stanford University; 1986.
[9] Gill P, Murray W, Saunders M, Wright M. User's Guide for QPSOL: A Fortran Package for Quadratic Programming, Technical Report Sol 84-6. Systems Optimization Laboratory, Department of Operations Research, Stanford University; 1984.

[10] Gill P, Murray W, Wright M. Practical Optimization. New York: Academic Press; 1981.
[11] Gourret J, Magnenat-Thalmann N, Thalmann D. Simulation of Object and Human Skin Deformations in a Grasping Task. In: Lane J, editor. Computer Graphics. Proceedings of SIGGRAPH 89, vol. 23(3). Boston, Mass.; July 1989. p. 21–30.
[12] Hahn J. Introduction to Issues in Motion Control. In: Tutorial 3, SIGGRAPH 88. Atlanta, Ga.; August 1988.
[13] Haumann D. Modeling the Physical Behavior of Flexible Objects. In: SIGGRAPH 87 Course Notes: Topics in Physically Based Modeling. Anaheim, Calif.; July 1987.
[14] Kokkevis E. Practical Physics for Articulated Characters. In: Game Developers Conference. 2004.
[15] Mirtich B. Impulse-Based Dynamic Simulation of Rigid Body Systems. Ph.D. dissertation. University of California at Berkeley, Fall; 1996.
[16] Ngo J, Marks J. Spacetime Constraints Revisited. In: Kajiya JT, editor. Computer Graphics. Proceedings of SIGGRAPH 93, Annual Conference Series. Anaheim, Calif.; August 1993. p. 343–50. ISBN 0-201-58889-7.
[17] O'Brien J, Hodges J. Graphical Modeling and Animation of Brittle Fracture. In: Rockwood A, editor. Computer Graphics. Proceedings of SIGGRAPH 99, Annual Conference Series. Los Angeles, Calif.: Addison-Wesley Longman; August 1999. p. 137–46. ISBN 0-20148-560-5.
[18] van Overveld C, Barenburg B. All You Need is Force: A Constraint-based Approach for Rigid Body Dynamics in Computer Animation. In: Terzopoulos D, Thalmann D, editors. Proceedings of Computer Animation and Simulation '95. Springer Computer Science. 1995. p. 80–94.
[19] Press W, Flannery B, Teukolsky S, Vetterling W. Numerical Recipes: The Art of Scientific Computing. Cambridge: Cambridge University Press; 1986.
[20] Reeves W. Particle Systems: A Technique for Modeling a Class of Fuzzy Objects. In: Computer Graphics. Proceedings of SIGGRAPH 83, vol. 17(3). Detroit, Mich.; July 1983. p. 359–76.
[21] Reeves W, Blau R. Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. In: Barsky BA, editor. Computer Graphics. Proceedings of SIGGRAPH 85, vol. 19(3). San Francisco, Calif.; August 1985. p. 31–40.
[22] Sims K. Particle Animation and Rendering Using Data Parallel Computation. In: Baskett F, editor. Computer Graphics. Proceedings of SIGGRAPH 90, vol. 24(4). Dallas, Tex.; August 1990. p. 405–14. ISBN 0-201-50933-4.
[23] Terzopoulos D, Fleischer K. Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture. In: Dill J, editor. Computer Graphics. Proceedings of SIGGRAPH 88, vol. 22(4). Atlanta, Ga.; August 1988. p. 269–78.
[24] Terzopoulos D, Platt J, Barr A, Fleischer K. Elastically Deformable Models. In: Stone MC, editor. Computer Graphics. Proceedings of SIGGRAPH 87, vol. 21(4). Anaheim, Calif.; July 1987. p. 205–14.
[25] Terzopoulos D, Witkin A. Physically Based Models with Rigid and Deformable Components. IEEE Comput Graph Appl November 1988;41–51.
[26] Witkin A, Fleischer K, Barr A. Energy Constraints on Parameterized Models. In: Stone MC, editor. Computer Graphics. Proceedings of SIGGRAPH 87, vol. 21(4). Anaheim, Calif.; July 1987. p. 225–32.
[27] Witkin A, Kass M. Spacetime Constraints. In: Dill J, editor. Computer Graphics. Proceedings of SIGGRAPH 88, vol. 22(4). Atlanta, Ga.; August 1988. p. 159–68.
[28] Weil J. The Synthesis of Cloth Objects. In: Evans DC, Athay RJ, editors. Computer Graphics. Proceedings of SIGGRAPH 86, vol. 20(4). Dallas, Tex.; August 1986. p. 49–54.
[29] DeRose T, Kass M, Truong T. Subdivision Surfaces for Character Animation. In: Cohen M, editor. Computer Graphics. Proceedings of SIGGRAPH 98, Annual Conference Series Orlando, Fla.: Addison-Wesley; July 1998. p. 85–94. ISBN 0-89791-999-8.
[30] Vassilev T, Spanlang B, Chrysanthou Y. Fast Cloth Animation on Walking Avatars. Computer Graphics Forum 2001;20(3):1–8.
[31] Choi K-J, Ko H-S. Stable but Responsive Cloth. Transactions on Graphics 2002;21(3):604–11, SIGGRAPH.