Building a 2D Physics Engine for MASON

Christian Thompson
GMU CS 798

Introduction

For my Master's Project, I developed a 2-dimensional physics engine as a plug-in to the GMU Evolutionary Computation Lab's Multi-Agent Simulator (MASON). The main goals of this simulator are to provide a generic framework for physically realistic simulation and to be small, easy to use, and fast. The framework provides users the ability to easily build MASON simulations that include collisions, joints, and forces. Two MASON simulations that demonstrate the features of the system are "collisions," and "robots."

Collisions

The "collisions" simulation demonstrates the collision detection and response features of the system. Rectangles and circles move around on a frictionless surface and collide with each other and the bordering walls. Since the surface is frictionless and there is no momentum loss during collisions, the circles and rectangles should theoretically continue to bounce around forever. By changing the "coefficient of friction" and "coefficient of restitution" parameters, however, the simulator can simulate friction between the objects and the surface and momentum loss during collisions. Figure 1 shows a screen shot of the simulation.

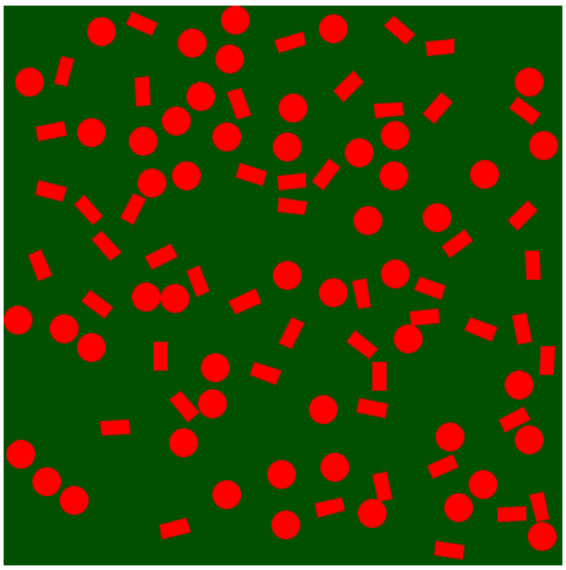


Figure 1: Collisions simulation

Robots

In the "robots" simulation, gray robots attempt to pick up blue cans and move them to the top of the screen. Once the robots have collected all the cans they return to their initial positions.

The "robots" simulation demonstrates all features of the system. Each robot is comprised of a circular body and an effector with which to pick up the cans. The effector is simulated with two small circles constrained to the robot's body using pin joints. Additionally, the robots "pick up" the cans by constraining them to themselves with a temporary pin joint. Both the robots and cans experience friction with the surface on which they are moving. The robots experience very little friction to simulate their wheeled locomotion, while the cans experience much more friction to simulate their relatively large mass and lack of wheels. While moving around, the robots collide with cans, each other, and the walls demonstrating collision detection and response. Figure 2 shows a screen shot of the simulation.

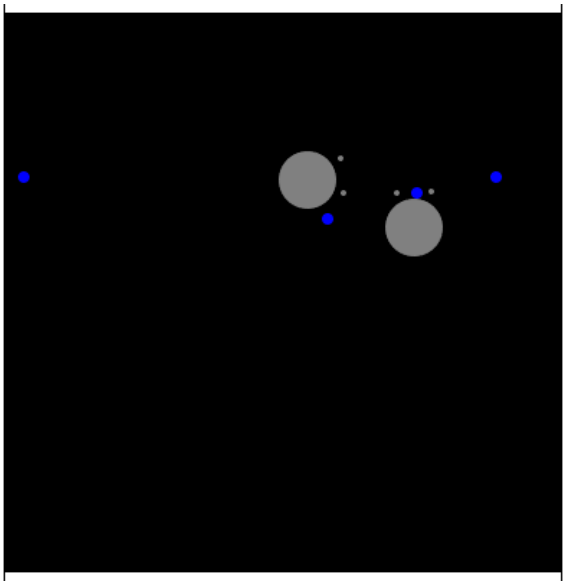


Figure 2: Robots simulation

Motion

The fundamental functionality of the physics engine is to move the simulated objects in a physically realistic way based on their positions, orientations, linear and angular velocities, their masses, and the forces and torques being applied to them. The physics engine calculates these quantities by using a numerical integrator to integrate the kinematic and kinetic equations that describe the objects' motions. By storing the state of all objects in matrices and vectors, it is able to calculate these quantities for all objects in one step.

Kinematics

Kinematics describes how bodies move in the absence of force [3]. The simulator uses kinematic equations to determine an object's current position, orientation, velocity, and angular velocity based on its previous position, orientation, linear and angular velocities, and linear and angular accelerations.

An object's velocity is its rate of change in position ($\Delta x / \Delta t$) and its angular velocity is its rate of change in orientation ($\Delta \theta / \Delta t$). Multiplying an object's velocity and angular velocity by an amount of time gives the object's changes in position and

orientation over that period. The following equations calculate an object's position and orientation after an infinitesimally small period of time (dt)

$$x(t+dt) = x(t) + \dot{x}(t)dt \quad (1)$$

$$\theta(t+dt) = \theta(t) + \dot{\theta}(t)dt \quad (2)$$

Likewise, an object's linear acceleration is the rate of change in velocity ($\Delta\dot{x}/\Delta t$) and angular acceleration is the rate of change in angular velocity ($\Delta\dot{\theta}/\Delta t$). The following equations calculate an object's linear and angular velocities after the time period dt .

$$\dot{x}(t+dt) = \dot{x}(t) + \ddot{x}(t)dt \quad (3)$$

$$\dot{\theta}(t+dt) = \dot{\theta}(t) + \ddot{\theta}(t)dt \quad (4)$$

The simulator can not directly evaluate equations (3) and (4) because it does not track objects' linear and angular accelerations. It must calculate the accelerations based on the forces and torques that are applied to the objects.

Kinetics

Kinetics extends the subject of kinematics to include the effects of forces (f) and torques (τ) on objects [3]. When forces and torques are applied to objects the objects experience linear and angular accelerations. The physics engine calculates linear and angular accelerations using the following equations, also known as the equations of motion [3]:

$$f = m\ddot{x} \quad (5)$$

$$\tau = I\ddot{\theta} \quad (6)$$

Rearranging (5) and (6) and plugging into (3) and (4) provides equations to calculate an object's linear and angular velocities using only quantities tracked by the simulator

$$\dot{x}(t+1) = \dot{x}(t) + \frac{f}{m}dt \quad (7)$$

$$\dot{\theta}(t+1) = \dot{\theta}(t) + \frac{\tau}{I}dt \quad (8)$$

Numerical Integration

The smallest unit of time in the simulator is one time step. This quantity is much larger than the infinitesimally small dt so the simulator must integrate the equations over each time step. Because the forces and torques applied to objects may not be defined by simple functions, the simulator must approximate the answers to the kinematic equations using numerical integration.

The kinematic equations can be expanded using the Taylor series into the following forms (in which Δt is a finite number representing the step size) [3]:

$$\dot{x}(t+\Delta t) = \dot{x}(t) + \ddot{x}(t)\Delta t + \frac{\dddot{x}(t)\Delta t^2}{2!} + \ldots \quad (9)$$

$$x(t+\Delta t) = x(t) + \dot{x}(t)\Delta t + \frac{\ddot{x}(t)\Delta t^2}{2!} + \ldots \quad (10)$$

The Euler method of numerical integration is to use the first two terms of the expansions and throw out the rest [3]:

$$\dot{x}(t+\Delta t) = \dot{x}(t) + \ddot{x}(t)\Delta t \quad (11)$$

$$x(t+\Delta t) = x(t) + \dot{x}(t)\Delta t \quad (12)$$

Although easy to implement, the Euler method doesn't work well with some systems because it is a rough approximation and can allow errors to build up quickly. For example, figure 3 shows the result of simulating a pendulum with an Euler integrator.

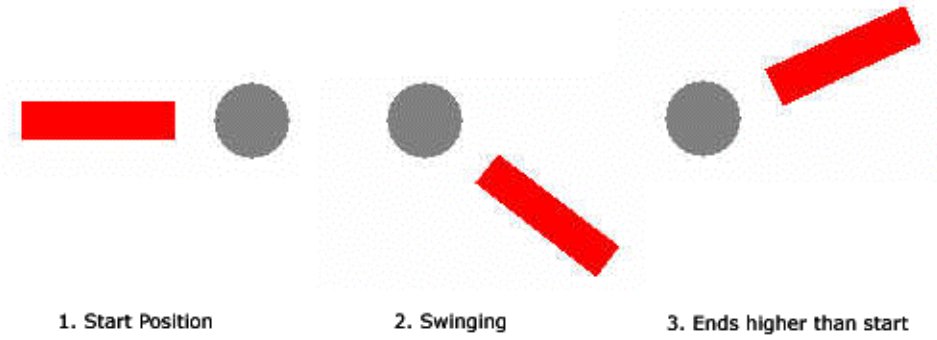


Figure 3: Euler Pendulum

Since the pendulum starts in a horizontal position on the left side, it should never get above the horizontal position on the right side. As figure 3 shows, however, the pendulum quickly gains energy when the Euler integrator is used.

Another, more accurate method, of integration is the Runge-Kutta method, which approximates the first four terms of the Taylor series expansion. This function is calculated using the following equations [3]:

$$\begin{aligned} k1 &= \Delta t\dot{x}(x(t)) \\ k2 &= \Delta t\dot{x}\left(x(t) + \frac{k1}{2}\right) \\ k3 &= \Delta t\dot{x}\left(x(t) + \frac{k2}{2}\right) \\ k4 &= \Delta t\dot{x}(x(t) + k3) \end{aligned} \quad (13)$$

$$x(t+\Delta t) = x(t) + \frac{(k1 + 2k2 + 2k3 + k4)}{6}$$

Although it requires more calculations, the Runge-Kutta method is necessary in simulations like the pendulum simulation to more accurately simulate reality. When the pendulum simulation uses the Runge-Kutta integrator, the pendulum does not gain or lose energy.

State Representation

The simulator stores the states of all objects in a single set of vectors and matrices. The position vector ($\mathbf{x}$) holds the current positions and orientation of all objects. Each object has a block of three variables in the position vector (x , y , and θ) stored sequentially by the object's index as shown by the following example position vector for two objects:

$$\begin{bmatrix} object\ 1 \begin{cases} x_1 \\ y_1 \\ \theta_1 \end{cases} \\ object\ 2 \begin{cases} x_2 \\ y_2 \\ \theta_2 \end{cases} \end{bmatrix}$$

The velocity ($\dot{\mathbf{x}}$) and force ($\mathbf{F}$) vectors are organized in the same way. An object's block in the velocity vector holds its $\dot{x}$, $\dot{y}$, and $\dot{\theta}$ variables. Its block in the force vector holds the sum of the x force components, y force components, and torques being applied to the object.

Finally, the simulator stores the "mass inverse matrix" ($\mathbf{W}$) containing the mass inverses ($1/m$) and mass moment of inertia inverses ($1/I$) of all objects (the mass moment of inertia I is the rotational equivalent of mass). The values are stored along the diagonal of the matrix with the rows corresponding to the rows in the position, velocity, and force vectors as shown by the following example mass inverse matrix:

$$\begin{matrix} \frac{1}{m_1} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{m_1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{I_1} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{m_2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{m_2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{I_2} \end{matrix}$$

The advantage of this representation is that the numerical integrator can calculate the positions and velocities of all objects in one step using matrix arithmetic and the numerical integration equations describe above. When matrices are used with the Euler method of integration, for example, the equations used to update the positions and velocities of all objects in one step become

$$\mathbf{x} = \mathbf{x} + \dot{\mathbf{x}} \quad (14)$$

$$\dot{\mathbf{x}} = \dot{\mathbf{x}} + \mathbf{W}\mathbf{F} . \quad (15)$$

Collision Detection

As the physics engine moves objects around in simulations, it needs to determine at each time step if any of the objects are colliding. The naïve approach to collision detection is to check for collisions between all pairs of objects at each time step. As the number of objects in the system grows large, this method becomes unacceptably slow because the physics engine must perform expensive exact collision detection for n^2 pairs of objects. To improve efficiency many collision detection libraries divide collision detection into two phases: a rough, but fast, "broad phase" and an accurate, but slower, "narrow phase" [6]. Several methods exist for performing both broad and narrow phase collision detection. For broad phase collision detection, I chose to implement the dimension reduction strategy described in [4].

Broad Phase

The dimension reduction strategy considers each dimension one at a time. The endpoints of each object are projected onto each dimension and a check is done to see if any of the objects' endpoints are overlapping. If a pair of objects has overlapping endpoints in all dimensions the pair is marked as possibly colliding. Figure 4 illustrates this for 1 dimension. In the illustration, the pair containing objects A and B is a good candidate for more accurate collision detection because their endpoints are overlapping. On the other hand, neither object should be tested for collision with object C.

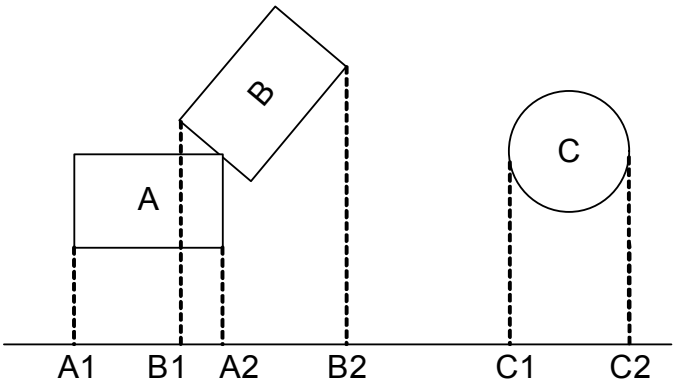


Figure 4: dimension reduction strategy

The algorithm described in [4] keeps a sorted list of object end points for each dimension. At the beginning of each time step the lists are sorted using an insertion sort. As the sorting algorithm moves each end point into its place in the sorted list, the algorithm tracks any overlaps created by the change in order. Because the objects don't move very much during one time step [4], the lists remain almost sorted, and the insertion sort can be completed in an expected O(n) time.

After sorting both dimensions, the broad phase collision detection logic creates a list of the pairs of objects whose end points overlap in both dimensions and passes this list to the narrow phase collision detection logic for more accurate testing.

Narrow Phase

Once the broad phase collision detection logic identifies pairs of objects that might be colliding, the next step is to determine if they are indeed colliding using narrow phase collision detection. I chose to use an algorithm based on the Lin-Canny closest feature tracking algorithm described in [6].

The Lin-Canny algorithm searches for the closest pair of features between two objects (a feature is either a vertex or an edge). It determines if a given pair is the closest using Voronoi regions. Figure 5 shows the Voronoi regions for some of the features for a pair of rectangles. In the figure, it is clear that the object A's top right vertex and object B's and bottom edge are the closest features. The Voronoi region for A's vertex is the region between the two rays emanating from the two edges that meet to form the vertex. The Voronoi region of B's edge is the one between the two rays that extend perpendicularly from the vertices at both ends of the edge.

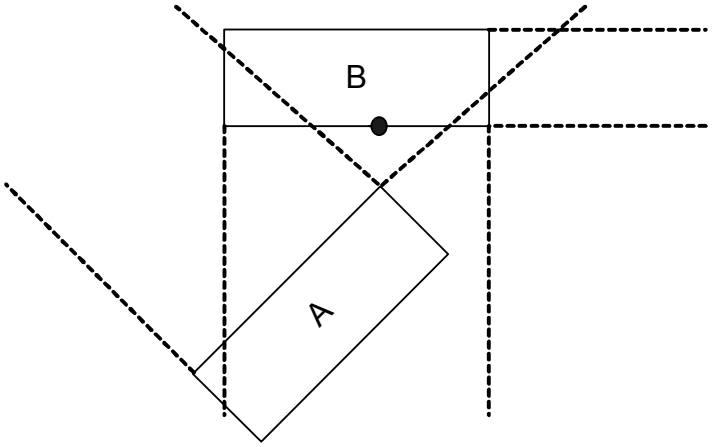


Figure 5: Voronoi Regions

The closest pair of features between two non-intersecting convex polygons is the only pair for which the closest point on each feature falls within the Voronoi region of the other feature [6]. In figure 5, the proof that A's top right vertex and B's bottom edge form the closest pair is that A's vertex falls in the Voronoi region of B's edge and that the point (the black dot) on B's edge that is closest to A's vertex is within A's vertex's Voronoi region.

The narrow phase algorithm finds the closest features by searching through the pairs of features until it finds a pair for which this property holds. For each pair, it must check if the closest point on each feature falls within the other's Voronoi region. Since vertices and edges have different Voronoi regions, the algorithm to check this is slightly different for each feature type.

For a point to lie in a vertex's Voronoi region, it must fall to the right of the region's left ray and to the left of the regions right ray. To check if this is true, the physics engine first finds the vector $\mathbf{v}$ that points from the vertex to the point (see figure 6).

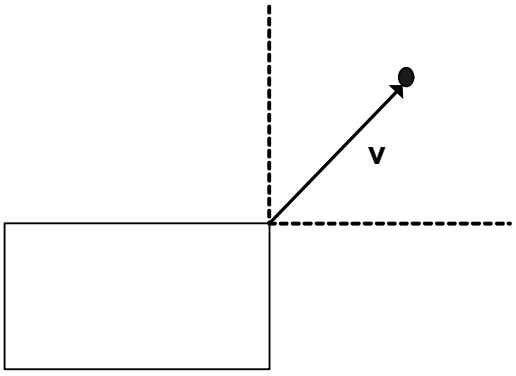


Figure 6: Vector **v** from vertex to point

It then determines if the point lies to the right of the left ray by taking the dot product of the vector $\mathbf{v}$ and the *right* ray (see projection vector $\mathbf{p}$ in figure 7). If the dot product is positive ($\mathbf{p}$ points to the right), the point falls to the right of the left ray.

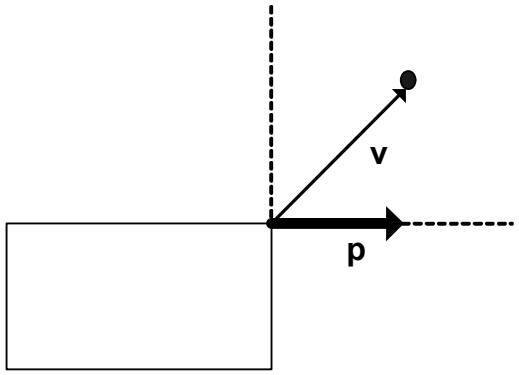


Figure 7: Projection **p** of vector **v** onto vertex's Voronoi region's right ray

If the point lies to right of the left ray, the physics engine then checks if the point lies to the left of the right ray by projecting the vector $\mathbf{v}$ onto the left ray. If both checks return true, the point lies within the vertex's Voronoi region.

For a point to lie within an edge's Voronoi region, the physics engine must perform two similar checks. It first checks that the point lies within the region formed by the vector extending perpendicularly from the left vertex and the vector pointing from the left vertex to the right vertex. It then checks that the point lies in the opposite region on the right side. Figure 8 shows the vectors and projections required to determine if a point lies within an edge's Voronoi region.

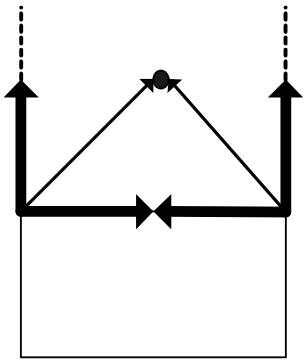


Figure 8: Projections for an edge's Voronoi region

The Lin-Canny algorithm searches through the feature pairs intelligently using the result of the current feature pair test to determine which feature pair to test next. Unfortunately the algorithm is complex and sometimes has problems converging [6]. Because the objects in this simulator are two-dimensional and have relatively fewer features than those in a three-dimensional simulation, I chose to simply loop through the pairs of features until the closest is found.

After the narrow phase logic finds the closest feature pair between two objects, it determines if those features are close enough to be considered colliding. If so, it reports the collision. If not, there is a reasonable chance that the objects will collide soon. Since objects don't move much in one time step, it is likely that the current closest feature pair will still be the closest pair in the next time step [6]. Therefore, the simulator saves time by recording the current feature pair to use as a starting point for the closest feature pair search in the next time step.

Finally, if no pairs of features fall into each other's Voronoi region, the objects have penetrated. The narrow phase collision detector finds the collision in this case by performing a binary search back in time over the last time step to find the point at which they penetrated.

Collision Response

Once the collision detector reports a collision, the simulator needs to instantaneously change the velocities of the colliding objects to prevent them from penetrating. It does this by applying the equivalent of an infinite force over an infinitesimally short period of time. This quantity is called an "impulse" [5].

Point Masses

The simulator uses three main concepts to calculate the impulses needed for objects to respond to collisions realistically. The first concept is Newton's Law of Restitution, which relates the pre and post-collision relative velocities at the collision point. The second is that an impulse causes a change in an object's momentum. Finally, the third concept is that impulses should only cause the objects' momentums to change along the collision normal [5].

Newton's Law of Restitution relates the pre-collision relative velocity at the collision point to the post collision relative velocity. The relative velocity ($\mathbf{v}^{AB}$) between object A and object B is calculated by subtracting the velocity of the collision point on object A ($\mathbf{v}^{AP}$) from the velocity of the collision point on object B ($\mathbf{v}^{BP}$).

$$\mathbf{v}^{AB} = \mathbf{v}^{AP} - \mathbf{v}^{BP} \tag{16}$$

For point masses, the collision point velocities $\mathbf{v}^{AP}$ and $\mathbf{v}^{BP}$ are simply the velocities of the objects. Newton's Law relates the pre and post-collision relative velocities using the following equation [5]:

$$\mathbf{v}_{post}^{AB} \cdot \mathbf{n} = -e\mathbf{v}_{pre}^{AB} \cdot \mathbf{n} \tag{17}$$

Equation (17) says that the post-collision relative velocity along the collision normal is a percentage of the pre-collision relative velocity along the collision normal. The percentage depends on the elasticity of the collision, represented by the variable e which is known as the coefficient of restitution [5]. In a perfectly elastic collision ($e = 1$), the incoming relative velocity is equal to minus the outgoing relative velocity. A super ball bouncing on the floor is a good approximation of a perfectly elastic collision. If a super ball is moving downward toward the floor at a speed of -10, this equation says that it will be moving upward at a speed of 10 after its collision with the floor.

The next set of equations expresses the fact that impulses change the momentums of both objects as a result of a collision. Newton's Third law of equal and opposite forces says that the impulses applied to both objects are of equal magnitude, but opposite directions [5]. Therefore, the equations for both objects include the same

impulse $\mathbf{R}$, but with opposite signs. The equations for the changes in momentums of the objects are

$$m^A \dot{\mathbf{x}}^A_{post} = m^A \dot{\mathbf{x}}^A_{pre} + \mathbf{R} \quad (18)$$

$$m^B \dot{\mathbf{x}}^B_{post} = m^A \dot{\mathbf{x}}^B_{pre} - \mathbf{R} \quad (19)$$

The final concept needed to solve for collision impulses is that they must only occur along the collision normal [7]. In the following equation $\mathbf{R}_\perp$ represents the impulse $\mathbf{R}$ rotated by 90 degrees:

$$\mathbf{R}_\perp \cdot \mathbf{n} = 0 \quad (20)$$

After writing this equation, the system has six equations and six unknowns (the x and y components of the post collision velocities and of the impulse) and can be solved using standard matrix techniques.

Rigid Bodies

Point masses provide an easy way to demonstrate the process of solving for collision responses because they do not require consideration of rotational effects. Rigid bodies use the same general process but require rotational effects to be considered when calculating the relative velocities at the collision point and when updating objects' momentums in response to an impulse.

The rotation of rigid bodies affects the velocities of points on the bodies, so the simulator must consider the colliding bodies' rotations when calculating the relative velocities for Newton's Law of Restitution. Calculating the pre-collision relative velocities of two rotating objects requires knowledge of the objects' linear and angular velocities. For example, the velocity of point $\mathbf{P}_A$ in figure 9 depends on object A's down and right linear velocity and its clockwise angular velocity.

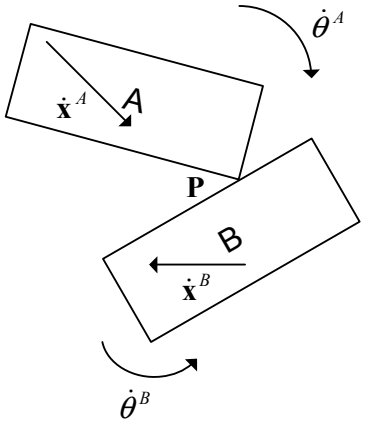


Figure 9: Relative velocity at collision point

The simulator calculates the velocity of a point due to the rotation of an object by multiplying the angular velocity of the object with the vector $\mathbf{r}_\perp$ formed by rotating by 90 degrees the vector from the center of the object to the point [5]

$$\dot{\mathbf{x}}^{\text{rotation}} = \dot{\theta}\mathbf{r}_{\perp}. \tag{21}$$

Figure 10 shows the vector $\mathbf{r}_{\perp}$ for the rectangle's top right vertex.

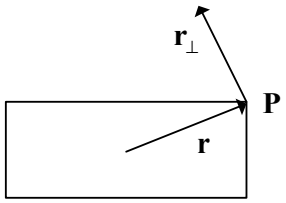


Figure 10: Velocity due to rotation

The total velocity of a point, therefore is

$$\dot{\mathbf{x}}^{\mathbf{P}} = \dot{\mathbf{x}} + \dot{\theta}\mathbf{r}_{\perp} \tag{22}$$

Plugging equation (22) into Newton's Law of Restitution (equation (17)) gives:

$$\left((\dot{\mathbf{x}}^{A}_{post} + \dot{\theta}^{A}_{post}\mathbf{r}^{A}_{\perp}) - (\dot{\mathbf{x}}^{B}_{post} + \dot{\theta}^{B}_{post}\mathbf{r}^{B}_{\perp})\right)\cdot\mathbf{n} = -e\left((\dot{\mathbf{x}}^{A}_{pre} + \dot{\theta}^{A}_{pre}\mathbf{r}^{A}_{\perp}) - (\dot{\mathbf{x}}^{B}_{pre} + \dot{\theta}^{B}_{pre}\mathbf{r}^{B}_{\perp})\right)\cdot\mathbf{n} \tag{23}$$

Because rigid bodies can rotate, impulses change not only their linear momentums, but also their angular momentums. Equations (18) and (19) express the changes in linear momentum experienced by colliding objects. The equation that expresses the change in an object's angular momentum is

$$I\dot{\theta}_{post} = I\dot{\theta}_{pre} + \mathbf{r}_{\perp}\cdot\mathbf{R}. \tag{24}$$

In this equation, the impulse application point $\mathbf{r}$ is rotated 90 degrees and dotted with the impulse. This operation is analogous to taking the three-dimensional cross product in two dimensions [5].

The simulator now has the eight equations needed to solve for the collision impulses for a pair of colliding rigid bodies. The full system of equations is:

$$\begin{aligned}
m^{A}\dot{x}^{A}_{post} &= m^{A}\dot{x}^{A}_{pre} + \mathbf{R}_{x}\\
m^{A}\dot{y}^{A}_{post} &= m^{A}\dot{y}^{A}_{pre} + \mathbf{R}_{y}\\
I\dot{\theta}^{A}_{post} &= I\dot{\theta}^{A}_{pre} + \mathbf{r}_{\perp}\cdot\mathbf{R}\\
m^{B}\dot{x}^{B}_{post} &= m^{B}\dot{x}^{B}_{pre} - \mathbf{R}_{x}\\
m^{B}\dot{y}^{B}_{post} &= m^{B}\dot{y}^{B}_{pre} - \mathbf{R}_{y}\\
I\dot{\theta}^{B}_{post} &= I\dot{\theta}^{B}_{pre} - \mathbf{r}_{\perp}\cdot\mathbf{R}\\
\left((\dot{\mathbf{x}}^{A}_{post} + \dot{\theta}^{A}_{post}\mathbf{r}^{A}_{\perp}) - (\dot{\mathbf{x}}^{B}_{post} + \dot{\theta}^{B}_{post}\mathbf{r}^{B}_{\perp})\right)\cdot\mathbf{n} &= -e\left((\dot{\mathbf{x}}^{A}_{pre} + \dot{\theta}^{A}_{pre}\mathbf{r}^{A}_{\perp}) - (\dot{\mathbf{x}}^{B}_{pre} + \dot{\theta}^{B}_{pre}\mathbf{r}^{B}_{\perp})\right)\cdot\mathbf{n}\\
\mathbf{R}_{\perp}\cdot\mathbf{n} &= 0
\end{aligned}$$

Constrained Dynamics

"Constrained dynamics" provides a method to implement joints and articulated bodies. In the physical world, a joint restricts the motions of the objects it connects. A set of door hinges, for example, restricts a door to rotate around the axis created by the side of the door frame to which the hinges are connected (see figure 11).

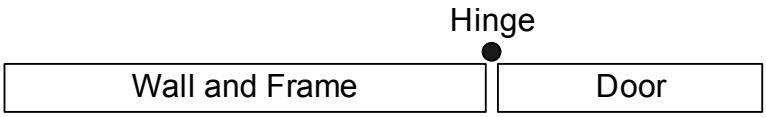


Figure 11: Hinge joint connecting a door to its frame

The physics engine simulates such joints using constraints which, like joints, restrict the ways in which objects are allowed to move. Constraints restrict objects' motions by applying "constraint forces" that offset accelerations in illegal directions.

Bead on a Wire

A simple example of a constrained system is a bead sliding along a two dimensional wire (see figure 12). The derivation of this example follows the derivation of a similar (but slightly more complicated) example in [8].

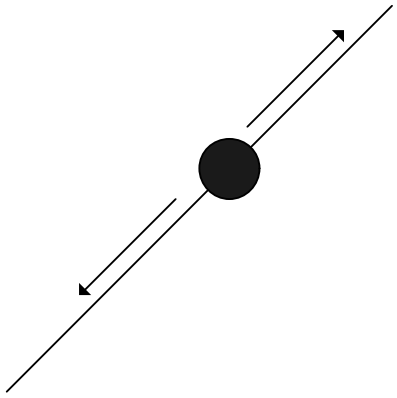


Figure 12: Bead sliding on a two dimensional wire

In the physical world, no matter what forces are applied to the bead, it always stays on the wire (assuming the wire doesn't break). If the wire lies along the line $y = x$, the bead's position always satisfies the equation $y - x = 0$. The physics engine can simulate this system by calculating and applying forces that prevent the bead from leaving the line of the simulated wire.

In the bead-on-a-wire simulation, the bead's position must satisfy the equation $y - x = 0$. This constraint can be written as:

$$C(\mathbf{x}) = y - x = 0 \tag{25}$$

Taking the derivative of the constraint function yields another that constrains the bead's velocity.

$$\dot{C}(\mathbf{x}) = \dot{y} - \dot{x} = 0 \tag{26}$$

Taking a final derivative provides a function that constrains the bead's acceleration.

$$\ddot{C}(\mathbf{x}) = \ddot{y} - \ddot{x} = 0 \tag{27}$$

This function can be re-written in terms of forces using Newton's Second Law ($f = m\ddot{x}$):

$$\ddot{C}(\mathbf{x}) = (f_y - f_x)/m = 0 \tag{28}$$

The forces applied to the bead include the external forces (i.e. someone pushing the bead) and the constraint forces that keep the bead on the wire. Therefore, the previous equation can be re-written again as

$$\ddot{C}(\mathbf{x}) = \left((f_y^{external} + f_y^{constra\,\mathrm{int}}) - (f_x^{external} + f_x^{constra\,\mathrm{int}})\right)/m = 0 \tag{29}$$

Because this one equation has two unknowns ($f_x^{constra\,\mathrm{int}}$ and $f_y^{constra\,\mathrm{int}}$), the physics engine needs one more equation to solve for the constraint forces. The principal of virtual work provides the extra equation. This principal says that constraint forces must do no work in the system. Since work is the amount of displacement caused by a force, this can be restated by saying that constraint forces can not cause displacements; they can only serve to cancel out other forces. The final equation in the system ensures that the constraint force does not cause displacements by requiring it to be orthogonal to all legal displacements. Since the bead in the above example can only move along the line $y = x$, the constraint forces must therefore be along the orthogonal line $y = -x$ (see heavy arrows representing possible directions for constraint forces in figure 13).

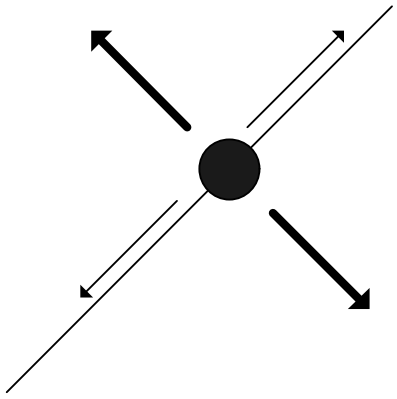


Figure 13: Constraint forces (heavy arrows) must be orthogonal to all legal displacements

Because the bead can only move in the directions of legal displacements, the constraint forces must also be orthogonal to the legal velocities. The following equations restrict the direction of the constraint force to be orthogonal to the bead's velocity:

$$f_x^{constra\,\mathrm{int}} = -\lambda\dot{y} \tag{30}$$

$$f_y^{constra\,\mathrm{int}} = \lambda\dot{x} \tag{31}$$

Plugging these equations into equation (29) gives

$$\left((f_y^{external} + \lambda\dot{x})\right) - (f_x^{external} - \lambda\dot{y})/m = 0 \tag{32}$$

or, after solving for lambda,

$$\lambda = (f_x^{external} - f_y^{external})/(\dot{y} + \dot{x})\,. \tag{33}$$

Plugging λ into equations 30 and 31 then provides $f_x^{external}$ and $f_y^{external}$.

General Constraint Equations

While the method described above is useful for understanding the process of solving for constraint forces, the physics engine needs a way to allow simulation writers to add any number and combination of constraints. [8] derives a more general method in which vectors and matrices are used to calculate constraint forces for any number and combination of constraints and objects. This section follows that derivation.

The first step is to define the constraint vector, in which each row holds a constraint equation:

$$\mathbf{C} = 0 \tag{34}$$

Like the bead-on-a-wire example, this equation needs to be differentiated twice with respect to time to give the legal acceleration equations. The first derivative uses the chain rule, multiplying the objects' velocities times the partial derivative of the constraint vector with respect to the objects' positions. The partial derivative vector $\frac{\partial \mathbf{C}}{\partial \mathbf{x}}$ is also known as the "Jacobian" ($\mathbf{J}$) of $\mathbf{C}$.

$$\dot{\mathbf{C}} = \mathbf{J}\dot{\mathbf{x}} = 0 \tag{35}$$

Differentiating again gives the legal acceleration equation

$$\ddot{\mathbf{C}} = \mathbf{J}\ddot{\mathbf{x}} + \dot{\mathbf{J}}\dot{\mathbf{x}} = 0\,. \tag{36}$$

The equations of motion can be used to substitute masses, mass moments of inertia, and forces for accelerations.

$$\ddot{\mathbf{x}} = \mathbf{W}(\mathbf{F} + \mathbf{F}^{constra\,\mathrm{int}}) \tag{37}$$

Substituting this equation for $\ddot{\mathbf{x}}$ in equation 36 gives

$$\ddot{\mathbf{C}} = \mathbf{J}\mathbf{W}(\mathbf{F} + \mathbf{F}^{constra\,\mathrm{int}}) + \dot{\mathbf{J}}\dot{\mathbf{x}} = 0 \tag{38}$$

or

$$\mathbf{JWF}^{constraint} = -\mathbf{JWF} - \dot{\mathbf{J}}\dot{\mathbf{x}} \tag{39}$$

The final step is to ensure that the constraints do no work. As in the bead-on-a-wire example, this restriction means that the constraint forces must be orthogonal to the legal displacements. Since in legal displacements, $\mathbf{C}$ does not change, $\partial\mathbf{C}/\partial x^T$ ($\mathbf{J}^T$) contains the set of vectors that are orthogonal to the legal displacements. The following equation, therefore, ensures that the constraint force vector falls into the space spanned by the set of the illegal displacement vectors [8]:

$$\mathbf{F}^{constraint} = \mathbf{J}^T\boldsymbol{\lambda} \tag{40}$$

Plugging this equation into equation (39) gives

$$\mathbf{JWJ}^T\boldsymbol{\lambda} = -\mathbf{JWF} - \dot{\mathbf{J}}\dot{\mathbf{x}} \tag{41}$$

which can be solved for $\boldsymbol{\lambda}$ and, thus, $\mathbf{F}^{constraint}$.

To make this method more concrete, the following example calculates the variables needed to solve equation (41) for a pin joint constraint. Solving that equation for the constraint forces requires $\mathbf{J}$ and $\dot{\mathbf{J}}$ (since $\mathbf{W}$, $\mathbf{F}$, and $\dot{\mathbf{x}}$ are already known).

Calculating $\mathbf{J}$ requires first defining $\mathbf{C}$. The pin joint has two constraint equations (one for each dimension x and y). It requires each object's local coordinate of the pin joint to convert to the same global coordinate. In the case of the door simulation, it requires the door's and frame's local coordinates of the hinge to convert to the same global coordinate (see figure 14).

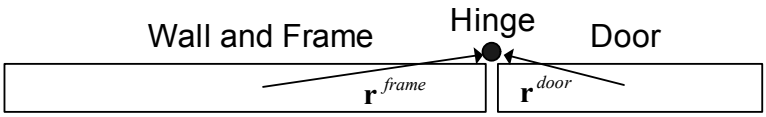


Figure 14: Local coordinates of hinge

The pin joint is located in the door frame's local coordinate at $\mathbf{r}^{frame}$ and in the door's local coordinate frame at $\mathbf{r}^{door}$. The door frame's local coordinate for the hinge is converted to the global coordinate frame by

$$\mathbf{x}^{hinge} = \mathbf{R}_{\theta^{frame}}\mathbf{r}^{frame} + \mathbf{x}^{frame} \tag{42}$$

in which $\mathbf{R}_{\theta^{frame}}$ is the rotation matrix for angle θ^{frane}.

The door's local coordinate for the hinge is converted to the global coordinate frame by

$$\mathbf{x}^{hinge} = \mathbf{R}_{\theta^{door}}\mathbf{r}^{door} + \mathbf{x}^{door}. \tag{43}$$

Because the door's hinge and the door frame's hinge must have the same position, the constraint therefore is:

$$\mathbf{R}_{\theta^{frame}}\mathbf{r}^{frame} + \mathbf{x}^{frame} - (\mathbf{R}_{\theta^{door}}\mathbf{r}^{door} + \mathbf{x}^{door}) = 0 \tag{44}$$

Multiplying by the rotation matrix gives the constraint equations for the x and y dimensions

$$\cos\theta^{frame}\mathbf{r}_x^{frame} - \sin\theta^{frame}\mathbf{r}_y^{frame} + \mathbf{x}^{frame} - \left(\cos\theta^{door}\mathbf{r}_x^{door} - \sin\theta^{door}\mathbf{r}_y^{door} + \mathbf{x}^{door}\right) = 0 \tag{45}$$

$$\sin\theta^{frame}\mathbf{r}_x^{frame} + \cos\theta^{frame}\mathbf{r}_y^{frame} + \mathbf{y}^{frame} - \left(\sin\theta^{door}\mathbf{r}_x^{door} + \cos\theta^{door}\mathbf{r}_y^{door} + \mathbf{y}^{door}\right) = 0 \tag{46}$$

Therefore, the constraint vector C is

$$\begin{bmatrix} \cos\theta^{frame}\mathbf{r}_x^{frame} - \sin\theta^{frame}\mathbf{r}_y^{frame} + \mathbf{x}^{frame} - \left(\cos\theta^{door}\mathbf{r}_x^{door} - \sin\theta^{door}\mathbf{r}_y^{door} + \mathbf{x}^{door}\right) \\ \sin\theta^{frame}\mathbf{r}_x^{frame} + \cos\theta^{frame}\mathbf{r}_y^{frame} + \mathbf{y}^{frame} - \left(\sin\theta^{door}\mathbf{r}_x^{door} + \cos\theta^{door}\mathbf{r}_y^{door} + \mathbf{y}^{door}\right) \end{bmatrix}$$

$\mathbf{J}$ is the partial derivative of the constraint vector with respect to the vector containing the positions of all objects. The resulting matrix's columns relate to the rows of the position vector and its rows relate to the rows of $\mathbf{C}$. The door and frame simulation has two objects and one constraint (the pin joint), so $\mathbf{J}$ will have 6 columns and 2 rows.

$$\mathbf{J} = \begin{bmatrix} 1 & 0 & -\mathbf{r}_x^{frame}\sin\theta^{frame} - \mathbf{r}_y^{frame}\cos\theta^{frame} & -1 & 0 & \mathbf{r}_x^{door}\sin\theta^{door} + \mathbf{r}_y^{door}\cos\theta^{door} \\ 0 & 1 & \mathbf{r}_x^{frame}\cos\theta^{frame} - \mathbf{r}_y^{frame}\sin\theta^{frame} & 0 & -1 & -\mathbf{r}_x^{door}\cos\theta^{door} + \mathbf{r}_y^{door}\sin\theta^{door} \end{bmatrix}$$

Multiplying $\mathbf{J}$ by the velocity vector $\dot{\mathbf{x}}$ gives $\dot{\mathbf{C}}$

$$\dot{\mathbf{C}} = \begin{bmatrix} 1 & 0 & -\mathbf{r}_x^{frame}\sin\theta^{frame} - \mathbf{r}_y^{frame}\cos\theta^{frame} & -1 & 0 & \mathbf{r}_x^{door}\sin\theta^{door} + \mathbf{r}_y^{door}\cos\theta^{door} \\ 0 & 1 & \mathbf{r}_x^{frame}\cos\theta^{frame} - \mathbf{r}_y^{frame}\sin\theta^{frame} & 0 & -1 & -\mathbf{r}_x^{door}\cos\theta^{door} + \mathbf{r}_y^{door}\sin\theta^{door} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{x}}^{frame} \\ \dot{\mathbf{y}}^{frame} \\ \dot{\theta}^{frame} \\ \dot{\mathbf{x}}^{door} \\ \dot{\mathbf{y}}^{door} \\ \dot{\theta}^{door} \end{bmatrix}$$

Finally, taking the partial derivative of $\dot{\mathbf{C}}$ with respect to $\mathbf{x}$ gives $\dot{\mathbf{J}}$

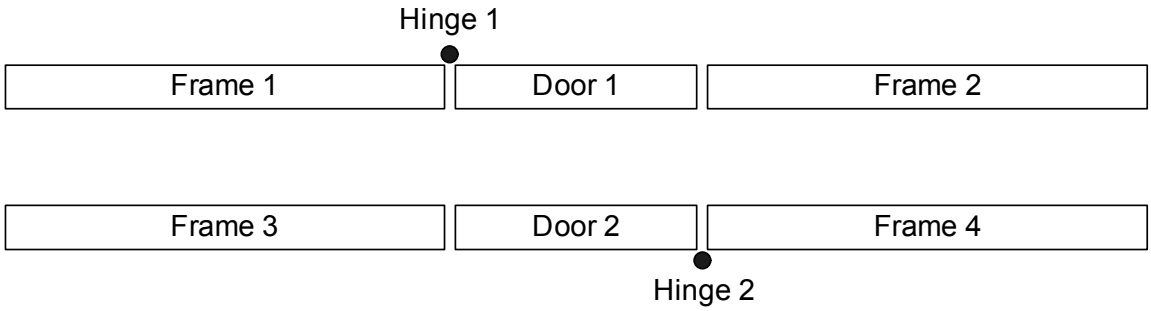
$$\dot{\mathbf{J}} = \begin{bmatrix} 0 & 0 & -\mathbf{r}_x^{frame}\cos\theta^{frame}\dot{\theta}^{frame} + \mathbf{r}_y^{frame}\sin\theta^{frame}\dot{\theta}^{frame} & 0 & 0 & \mathbf{r}_x^{door}\cos\theta^{door}\dot{\theta}^{door} - \mathbf{r}_y^{door}\sin\theta^{door}\dot{\theta}^{door} \\ 0 & 0 & -\mathbf{r}_x^{frame}\sin\theta^{frame}\dot{\theta}^{frame} - \mathbf{r}_y^{frame}\cos\theta^{frame}\dot{\theta}^{frame} & 0 & 0 & \mathbf{r}_x^{door}\sin\theta^{door}\dot{\theta}^{door} + \mathbf{r}_y^{door}\cos\theta^{door}\dot{\theta}^{door} \end{bmatrix}$$

The simulator can now solve for the constraint forces for the door simulation given $\dot{\mathbf{x}}$, $\mathbf{W}$, and $\mathbf{F}$.

Implementation

In the example above, the $\mathbf{J}$ and $\dot{\mathbf{J}}$ matrices only included two objects and two constraint equations (representing one pin joint). In a real simulation, however, there can be many objects and many constraints. In this case, the $\mathbf{J}$ and $\dot{\mathbf{J}}$ matrices have columns for the pose variables of all objects in the system and one row for each constraint equation in the system. The individual constraints represent blocks in the global $\mathbf{J}$ and $\dot{\mathbf{J}}$ matrices at the intersection of the constraint's row numbers in the global constraint vector and the column numbers of the pose variables of the objects they constrain.

The following simulation, for example, has 6 objects - four walls and two doors - and two pin joints (for a total of four constraint equations).



The $\mathbf{J}$ matrix has columns representing the pose variables of the 6 objects and rows for each of the four constraint equations. The following global $\mathbf{J}$ matrix shows the individual $\mathbf{J}$ blocks at the intersections of the pose variables and constraint equations in the global $\mathbf{J}$ matrix.

$$\begin{bmatrix} & \overbrace{x\ x\ x}^{frame\ 1} & \overbrace{x\ x\ x}^{door\ 1} & \overbrace{0\ 0\ 0}^{frame\ 2} & \overbrace{0\ 0\ 0}^{frame\ 3} & \overbrace{0\ 0\ 0}^{door\ 2} & \overbrace{0\ 0\ 0}^{frame\ 4} \\ hinge\ 1 & x\ x\ x\ x\ x\ x & 0\ 0\ 0\ 0\ 0\ 0 & 0\ 0\ 0\ 0\ 0\ 0 & 0\ 0\ 0\ 0\ 0\ 0 \\ hinge\ 2 & 0\ 0\ 0\ 0\ 0\ 0 & 0\ 0\ 0\ 0\ 0\ 0 & 0\ 0\ 0\ 0\ 0\ 0 & x\ x\ x\ x\ x\ x \end{bmatrix}$$

The physics engine is implemented such that each constraint is represented with a "Constraint" object. Each constraint object knows its positions in the global constraint matrices and is responsible for updating these matrices at each time step based on the states of and forces applied to the objects it constrains. After each constraint updates its block in the global matrices, the physics engine solves for the current constraint forces for all constraints in one step.

Constrained Collision Response

Constraints work under the assumption that the velocity and positions of the objects they are constraining are valid at the current time step [8]. They keep this assumption true by enforcing legal accelerations. Unfortunately, the impulses used to simulate collisions violate this assumption because they instantaneously change the velocities of objects. Therefore, collision response for constrained objects must account for the constraints. The simulator does this by not only solving for the impulses required at collision points, but also the impulses required at constraint connection points.

The "Collision Response" section details the system of equations needed to solve for the collision impulses for a simple collision involving two objects. When constraints are involved, the simulator also needs to solve for the constraint impulses. Each additional constraint and object pair introduces five new unknowns: the x and y components of the constraint impulse, the x and y components of the new object's post collision velocity, and the new object's post collision angular velocity. For example, if object B from figure 9 is connected with another object using a pin joint (object C in figure 15), the system of equations developed in the collision response section needs to be updated with information about the additional impulse and to include five more equations for the five new unknown variables.

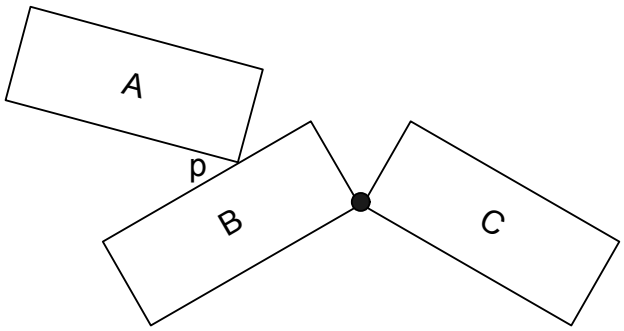


Figure 15: Collision with a constraint

Three equations of the original eight need to be updated to include the new impulse. Since body B is constrained by the pin joint, it will receive an impulse from it. Therefore, the equations relating its momentum before and after the collision need to include terms for the constraint impulse ($\mathbf{R}^{joint}$) [7]:

$$m^B \dot{x}^B_{post} = m^B \dot{x}^B_{pre} - R^{collision}_x + R^{joint}_x \quad (47)$$

$$m^B \dot{y}^B_{post} = m^B \dot{y}^B_{pre} - R^{collision}_y + R^{joint}_y \quad (48)$$

$$I^B \dot{\theta}^B_{post} = I^B \dot{\theta}^B_{pre} - \mathbf{r}^{collision}_{\perp} \cdot \mathbf{R}^{collision} + \mathbf{r}^{joint}_{\perp} \cdot \mathbf{R}^{joint} \quad (49)$$

Five new equations are needed to solve for the five new unknowns. The first three relate the momentum of object C before and after the collision (note that object C does not directly receive a collision impulse, only an impulse from the pin joint).

$$m^C \dot{x}^C_{post} = m^C \dot{x}^C_{pre} - R^{joint}_x \quad (50)$$

$$m^C \dot{y}^C_{post} = m^C \dot{y}^C_{pre} - R^{joint}_y \quad (51)$$

$$I^C \dot{\theta}^C_{post} = I^C \dot{\theta}^C_{pre} - \mathbf{r}^{jo\,\mathrm{int}}_{\perp} \cdot \mathbf{R}^{joint} \quad (52)$$

The last two equations express the property of the pin joint that the points connected by the pin joint must have the same velocity after the collision [7].

$$\dot{x}^B + x(\dot{\theta}^B \mathbf{r}^B_{\perp}) = \dot{x}^C + x(\dot{\theta}^C \mathbf{r}^C_{\perp}) \quad (53)$$

$$\dot{y}^B + y(\dot{\theta}^B \mathbf{r}^B_{\perp}) = \dot{y}^C + y(\dot{\theta}^C \mathbf{r}^C_{\perp}) \quad (54)$$

After solving the system of equations, the simulator updates the velocities of all the involved objects by applying the appropriate impulses. This maintains the assumption of legal velocities and allows the constraint engine to continue.

Matrix Optimization

After getting the collision response and constraint logic working, the simulator was very slow. According to the profiler, the biggest bottleneck was in manipulating the matrices used to calculate collision responses and constraint forces. These matrices get very large as objects are added to the system. The matrix used for collision response, for example, is a square matrix of dimension 3 times the number of objects in the system (x, y, and θ) plus 2 times the number of constraints in the system plus 2 (for the one collision impulse being solved for). When the "collisions" simulation is run with 1000 squares, therefore, a 3002x3002 matrix must be inverted to resolve each collision. Needless to say, this caused the simulation to ground to a halt.

Sparse Matrices

Because the collision response matrix has enough room for every object in the system, but is only used to resolve a collision involving two objects (in collisions that don't involve constraints), the matrix encodes a very small amount of useful information relative to its size.

The following shows a collision response matrix for the "collisions" simulation with 3 rectangles

1	0	0	0	0	0	0	0	0	0.03	0.00
0	1	0	0	0	0	0	0	0	0.00	0.03
0	0	1	0	0	0	0	0	0	-0.02	0.01
0	0	0	1	0	0	0	0	0	-0.03	0.00
0	0	0	0	1	0	0	0	0	0.00	-0.03
0	0	0	0	0	1	0	0	0	-0.02	0.01
0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	1	0	0
-0.46	-0.89	-0.05	0.46	0.89	0.05	0	0	0	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	-0.89	0.46

This matrix has three features that are common to all collision response matrices. The first is the identity matrix formed by removing rows at the bottom and columns on the right. This identity matrix represents the equations relating the objects' pre and post-collision momentums. The second feature is the rows at the bottom of the matrix that only have non-zero values for the blocks representing the colliding objects. The third is the columns on the right that only have non-zero blocks for the constraints involved in the collision.

Because this structure is always present, it can be exploited to dramatically increase the efficiency of matrix operations on these matrices. The only parts of the matrix that need to be stored and manipulated are the small blocks of information contained within the borders. In the matrix shown above, for example, the non-redundant information is stored in the outlined blocks - two 2x3 sub-matrices, two 3x2 sub-matrices, and one 2x2 sub-matrix. These blocks contain 28 values vs. the 121 contained in the entire matrix. The savings become much more dramatic as objects are added to the system.

The constraint matrices are also very big and very sparse [8]. In the constraint equation

$$\mathbf{JWJ}^T \lambda = -\mathbf{JWF} - \dot{\mathbf{J}}\dot{\mathbf{x}},$$

the $\mathbf{J}$ and $\dot{\mathbf{J}}$ matrices both have (3 * number of objects) columns and number of rows proportional to the number of constraints in the system. The $\mathbf{W}$ matrix is a square matrix of dimension (3 * number of objects). Because constraints generally only relate two objects and the $\mathbf{J}$ matrices only have values at the intersection of the constraint's rows in the matrix (2 for a pin joint) and the constrained objects' columns (3 for each object), this matrix has relatively few non-zero blocks of data. For example, the following is a $\mathbf{J}$ matrix for the robots simulation with 1 robot and 1 can (for a total of 4 objects and two constraints).

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	6	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-12	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	-6	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-12	0	0	0	0	1	0

Additionally, the $\mathbf{W}$ matrix only contains non-zero values along its diagonal. Using standard matrix techniques on these matrices is very wasteful since the amount of information contained within them is very small in comparison to their size.

Efficient Sparse Matrix Methods

Fortunately, there are methods to efficiently manipulate such "sparse" matrices. In [8], Witken suggests using a conjugate gradient algorithm to solve matrix equations of the form $\mathbf{Mx} = \mathbf{b}$ for sparse matrices. This algorithm solves these equations iteratively by choosing a value for $\mathbf{x}$ and minimizing the resulting error. A significant benefit of the algorithm is that it can work on any matrix as long as the matrix provides functionality to multiply itself by a vector and to multiply the transpose of itself by a vector. The physics engine includes a matrix package that implements these methods for all three sparse matrix types found in the system – diagonal and

block sparse for constrained dynamics and bordered diagonal identity for collision response.

Diagonal Matrix
The "times" and "transpose times" methods for diagonal matrices are very simple. The diagonal matrix is stored as a one dimensional array with the value at array position "n" representing the value at matrix position "n" x "n." Multiplying by a vector simply requires an element by element multiplication. Transposing a diagonal matrix has no effect, so the "transpose times" operation is the same.

Block Sparse Matrix
Block sparse matrices are stored as a collection of sub-matrices, each representing a block of non-zero data in the matrix. The sub-matrices are stored with their row and column offsets to identify where they fall in the matrix. To multiply the entire matrix with a vector, an "answer" vector is first initialized to 0. Each sub-matrix is then multiplied by the section of the vector starting at the block's row index and ending at its row index plus its number of columns using standard matrix multiplication. Finally, the result of this operation is added to the appropriate section of the answer matrix.

The following example multiplies a small sparse matrix by a vector. The block sparse matrix contains one 2 x 2 block at row 3 and column 2:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 \\ 0 & 5 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$$

The algorithm first creates a 5 element answer vector and initializes it to zero. It then multiplies the sub-matrix represented by the block with the corresponding vector section

$$\begin{bmatrix} 3 & 4 \\ 5 & 2 \end{bmatrix} * \begin{bmatrix} 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 32 \\ 30 \end{bmatrix}$$

Finally, it adds the result of that operation to the corresponding answer vector section

$$\begin{bmatrix} 0 \\ 0 \\ 32 \\ 30 \\ 0 \end{bmatrix}$$

The "transpose times" operation for block sparse matrices is similar to the "times" operation except each sub-matrix's row and column offsets are reversed and the

sub-matrices are transposed before being multiplied by their corresponding vector sections.

Bordered Diagonal Identity Matrix
The operations for bordered diagonal identity matrices are similar to that of the block sparse matrix. Since the upper left sub-matrix is the identity matrix, the corresponding section of the vector with which the matrix is being multiplied is first copied into the answer vector. The blocks in the two borders are then multiplied with their corresponding sections of the vector using the same steps used by the block sparse matrix algorithms.

BiConjugate Gradient Algorithm
The biconjugate gradient algorithm uses the times and transpose times methods to solve the matrix equation $\mathbf{Mx} = \mathbf{b}$. See [9] for details of the algorithm. Because the matrix package implements very efficient versions of the times and transpose times operations for all sparse matrices found in the collision response and constrained dynamics equations, the simulator can efficiently solve for impulses and constraint forces using the biconjugate gradient algorithm.

Future Work

The physics engine implements many features that MASON users can use to build complex, efficient, and physically realistic simulations. There is still much work to be done, however, for it to compete with more advanced physics engines. The most obvious feature is to extend the simulator to three dimensions. Other features that would shorten the gap with the more advanced engines are resting contact and frictional collision response.

Resting contact is the term giving to objects that are colliding, but have zero relative velocity. A real-world example of this is an apple sitting on a table. The apple exerts force on the table equal to its mass times gravity. The apple doesn't fall through the table because the table exerts an equal and opposite force to the apple. As the simulator is currently implemented, it will try to resolve such contact with an impulse at each time step. This appears to work for a while, but the simulated apple will eventually creep through the table. A resting contact feature of the physics simulator would analytically solve for the forces needed to prevent resting objects from penetrating like this. See [1] and [2] for further details.

Collision responses in the physics engine currently ignore the effects of friction during collisions. In the physical world if a billiard ball hits the edge of a table at a 45 degree angle, the collision will impart some spin to the ball because of the friction between the ball and the edge of the table. A simulated billiard ball, however, would not spin as a result of such a collision. A frictional collision feature would solve for both the impulse along the collision normal (as the physics engine currently does) and the impulse along the collision tangent due to friction.

References

1. D. Baraff, "Fast Contact Force Computation for Nonpenetrating Rigid Bodies," *Computer Graphics (Proc. SIGGRAPH)*, 1994
2. D. Baraff, "An Introduction to Physically Based Modeling: Rigid Body Simulation II – Nonpenetration Constraints," *SIGGRAPH '97 Course Notes*
3. D. Bourg, "Physics for Game Developers," O'Reilly (1/2002)
4. J. Cohen, M. Lin, D. Manocha, M. Ponamgi, "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments," *Department of Computer Science, University of North Carolina*, Chapel Hill, NC
5. C. Hecker, "Physics, Part 3: Collision Response," Game Developer (March 1997), pp. 11 – 18
6. B. Mirtich, "Efficient Algorithms for Two-Phase Collision Detection," *MERL – A Mitsubishi Electric Research Laboratory*, TR-97-23 (12/1997)
7. M. Moore, J. Wilhelms, "Collision detection and response for computer animation," Computer (Proc. SIGGRAPH), vol. 22, pp. 289-298, 1998
8. A. Witkin, "Constrained Dynamics," *SIGGRAPH '97 Course Notes*
9. E. Weisstein et al. "Biconjugate Gradient Method," *MathWorld—A Wolfram Web Resource,* http://mathworld.wolfram.com/biconjugategradientmethod.html