Iterative Dynamics with Temporal Coherence

Erin Catto
Crystal Dynamics
Menlo Park, California
ecatto@crystald.com

June 5, 2005

Abstract

This article introduces an iterative constraint solver for rigid body dynamics with contact. Our algorithm requires linear time and space and is easily expressed in vector form for fast execution on vector processors. The use of an iterative algorithm opens up the possibility for exploiting temporal coherence. A method for caching contact forces is presented that allows contact points to move from step to step and to appear and disappear. Examples are provided to illustrate the effectiveness of the algorithm.

1 Introduction

Recently, there has been much interest in using rigid body physics to enhance video games. Examples of rigid bodies in games are vehicles, rag dolls, cranes, barrels, crates, and even whole buildings. As more objects in the game world become physically based, realism and immersion are increased and the consumer has a more satisfying experience. Rigid body physics has moved from being a novelty to become a checklist feature in video game development.

Developing a physics engine for games is a tremendous challenge. Many developers do not have the know-how and/or resources to develop general purpose physics engines, as witnessed by the recent growth in the physics middle-ware industry. A physics engine must have exceptional performance, far exceeding the levels

needed for off-line animation and research. The stability of physics simulation is vital because without stability the game play may become frustrating, ruining the player's experience. Finally, the memory specifications of the current generation of consoles demands serious consideration of the physics engine memory footprint and cache usage.

Fortunately, the performance and space requirements can be balanced by low accuracy requirements. As long as the motion is visually plausible, the physics programmer is free to modify the equations of motion and to approximate the contact geometry. Furthermore, the chosen model does not need to be solved to high precision. As long as stability is maintained, numerical accuracy has secondary importance.

Designing a physics engine for games is an exercise in cost-benefit analysis when considering collision detection methods, the form of the equations of motion, and the solution techniques. In a high performance engine, these design aspects are usually interdependent. For example, it may advantageous to form the equations of motion such that their solution may be obtained rapidly. Thus, standard models of inertia and friction can be placed on the workbench to be retooled for performance and stability.

Stability can also be enhanced by using authoring practices that recognize the limitations of the physics engine. For example, a high performance engine typically has trouble dealing with interacting bodies that have a large discrepancy in mass. Typically, mass ratios must be less than an order of magnitude. This limitation can often be incorporated into design practices without sacrificing game play quality.

In this article we describe algorithms to implement rigid body physics with contact and friction using only linear time and space. Compared to previous approaches, these algorithms are relatively simple to understand and implement.

2 Previous Work

Modeling choices have a profound effect on the ability to solve the contact problem efficiently and robustly. We wish to avoid the NP-hard problems faced by Baraff [2]. Our model has more in common with Anitescu [1] because it is a time-stepping scheme. The formulation follows Smith [13]; it is velocity based and uses an approximate friction model.

Other simulation systems in recent years have approached the contact and constraint problem from a different angle. Guendelman [7] uses impulses to prevent penetration. First a tentative integration step is taken and then overlap is measured. Impulses are then applied sequentially at the original configuration until the contacts are separating. Jakobsen [9] uses a particle model with distance constraints. At each time step the particles are allowed to fall under the influence of gravity. After the time step, the algorithm loops over all the constraints. For each distance constraint, the positions of the two particles are adjusted to satisfy the constraint. Since this breaks other constraints, several iterations are used.

In contrast, the system we present in this paper is based on constraint forces and rigid bodies. Some advantages of this approach are:

- Penetration is handled, improving robustness.
- Various joints types, such as revolute and prismatic joints, are straight forward to model and incorporate.
- Joint reactions are available for game logic, such as breakable joints and triggers.
- Authoring is relatively easy due to the use of rigid bodies.
- Joints and contacts are handled in the same way, simplifying the code.

3 Constrained Dynamics Model

3.1 Kinematics

Consider a three-dimensional rigid body with position x and quaternion q. We assume the center of mass is located at x. We often use the rotation matrix R which can be computed from q as needed [5]. The linear velocity is v and the angular velocity is ω. The position terms are related to the velocity terms by kinematic differential equations [5].

$$\dot{x} = v \tag{1}$$

$$\dot{q} = \frac{1}{2}\omega * q \tag{2}$$

The overhead dot denotes differentiation with respect to time and $*$ denotes quaternion multiplication (ω is treated as a quaternion with a zero scalar part).

For a system of n bodies, the linear and angular velocities are stacked in a $6n$-by-1 column vector V.

$$V = \begin{pmatrix} v_1 \\ \omega_1 \\ \vdots \\ v_n \\ \omega_n \end{pmatrix} \tag{3}$$

3.2 Constraints

Our system allows for pairwise constraints between rigid bodies. There are cases where constraints involve more than two bodies, such as gears on moving bodies, but these situations are uncommon. Furthermore, some important performance and memory optimizations become possible by restricting constraints to be pairwise.

A single position constraint C_k is represented abstractly as a scalar function of x and q for two bodies i and j.

$$C_k(x_i, q_i, x_j, q_j) = 0 \tag{4}$$

The constraints for a system of rigid bodies are collected in an s-by-1 column vector C, where s is the number of constraints.

The time derivative of C yields the velocity constraint vector. By the chain rule of differentiation, the velocity constraint is guaranteed to be linear in velocity.[1] Therefore,

$$\dot{C} = JV = 0 \tag{5}$$

where J is the s-by-$6n$ Jacobian.

[1] This is true because $\dot{q}$ is a linear function of ω, as seen in equation (2).

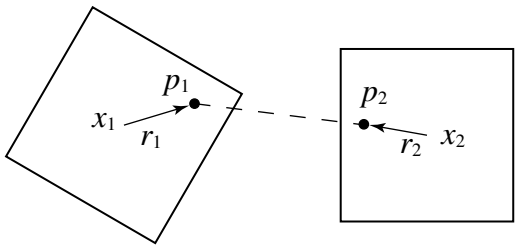

Figure 1: Distance constraint between points p_1 and p_2, which are fixed bodies 1 and 2, respectively.

3.3 Constraint Forces

Each constraint has an internal reaction force f_c and reaction torque τ_c. For a system of n bodies these are collected in the $6n$-by-1 column vector F_c:

$$F_c = \begin{pmatrix} f_{c_1} \\ \tau_{c_1} \\ \vdots \\ f_{c_n} \\ \tau_{c_n} \end{pmatrix} \tag{6}$$

Notice that the velocity constraint (5) states that the *admissible* velocity V is orthogonal to the rows of J. Constraint forces do no work, so they must also be orthogonal to V. Therefore,

$$F_c = J^T \lambda \tag{7}$$

where λ is a vector of s undetermined multipliers. These multipliers represent the signed magnitudes of the constraint forces.

3.4 Computing the Jacobian

The constraint vector C is an unordered column array of all the position constraints in the model, including contacts, joints, etc. Each element C_k is a single constraint equation governing some aspect of the motion. Usually C_k can be determined by an analysis of the constraint geometry.

For example, the distance constraint shown in Figure 1 acts between two rigid bodies. The constraint equation C_{dist} measures the difference between the actual distance and the desired distance L.

$$C_{dist} = \frac{1}{2}\left[(p_2 - p_1)^2 - L^2\right] \tag{8}$$

When $C_{dist} = 0$, the distance between p_1 and p_2 is equal to L. To determine the Jacobian, we differentiate C_{dist} with respect to time.

$$\dot{C}_{dist} = d \cdot (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) \tag{9}$$

where $d = p_2 - p_1$. Through algebraic manipulation and use of the vector identity $A \cdot B \times C = B \cdot C \times A$, the velocities can be isolated.

$$\dot{C}_{dist} = \begin{pmatrix} -d^T & -(r_1 \times d)^T & d^T & (r_2 \times d)^T \end{pmatrix} \begin{pmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{pmatrix} \tag{10}$$

Now we can simply write the Jacobian by inspection.

$$J = \begin{pmatrix} -d^T & -(r_1 \times d)^T & d^T & (r_2 \times d)^T \end{pmatrix} \tag{11}$$

The preceding example gives a recipe for computing the Jacobian.

1. Determine each constraint equation as a function of body positions and rotations.
2. Differentiate the constraint equation with respect to time.
3. Identify the coefficient matrix of V. This matrix is J.

This process is performed off-line for each type of constraint. The resulting system of constraint equations can be assembled in an object-oriented fashion. See Smith [12] and Shabana [11] for more details.

Often steps 1 and 2 can be skipped by considering the constraint directly at the velocity level. It is still useful to form constraints at the position level and differentiate them to ensure the velocity constraints are correct. Furthermore, it is useful to use the position constraint for dealing with constraint drift (discussed below).

Algorithm 1 Compute $\dot{C} = JV$

for $i = 1$ to s **do**
  $b_1 = J_{map}(i,1)$
  $b_2 = J_{map}(i,2)$
  $sum = 0$
  if $b_1 > 0$ **then**
    $sum = sum + J_{sp}(i,1)V(b_1)$
  end if
  $sum = sum + J_{sp}(i,2)V(b_2)$
  $\dot{C}(i) = sum$
end for

In general, the Jacobian is s-by-$6n$ for s constraints and n bodies. Since we are considering only pairwise constraints, each row of J has at most, two nonzero blocks of length six (three scalars for position and three scalars for rotation). Thus, for s constraints, we can store J as an s-by-12 array J_{sp}.

$$J_{sp} = \begin{pmatrix} J_{11} & J_{12} \\ \vdots & \vdots \\ J_{s1} & J_{s2} \end{pmatrix} \tag{12}$$

Each block J_{ij} is a row vector of length 6. The row index i denotes the constraint number and column index j is either 1 or 2, denoting the first and second body involved in the constraint.

The sparse representation J_{sp} is combined with the s-by-2 body map J_{map}.

$$J_{map} = \begin{pmatrix} b_{11} & b_{12} \\ \vdots & \vdots \\ b_{s1} & b_{s2} \end{pmatrix} \tag{13}$$

Each b_{ij} is the index of a rigid body. By convention, if a constraint is between a single rigid body and ground, then $b_{i1} = 0$ and the corresponding J_{i1} is zero. Using J_{sp} and J_{map}, the original Jacobian J can be assembled.

Using J_{sp} and J_{map}, we can compute the products JV and $J^T\lambda$ in $O(s+n)$ time. Algorithms 1 and 2 show how this is done. In the pseudo-code some indexing refers to blocks. For example, $V(b_1)$ refers to the block 6-vector $(v_{b1}^T, \omega_{b1}^T)^T$. Note that these algorithms involve simple dot products and sums of 6-vectors. These operations can easily be decomposed to take advantage of SIMD hardware.

Algorithm 2 Compute $F_c = J^T \lambda$

```
for i = 1 to n do
    F_c(i) = 0
end for
for i = 1 to s do
    b_1 = J_map(i, 1)
    b_2 = J_map(i, 2)
    if b_1 > 0 then
        F_c(b_1) = F_c(b_1) + J_sp(i, 1) λ(i)
    end if
    F_c(b_2) = F_c(b_2) + J_sp(i, 2) λ(i)
end for
```

3.5 Handling Inequality Constraints

Usually constraint equations are partitioned into *equality* and *inequality* constraints. Examples of equality constraints are distance constraints, revolute joints, prismatic joints, and many other joint types. Examples of inequality constraints are contact constraints and joint angle limits.

For our purposes, all constraint equations are treated in a similar manner. For each constraint, a lower and upper bound on λ is specified as part of the constraint model.

$$\lambda_i^- \leq \lambda_i \leq \lambda_i^+, \ \forall i \in [1, s] \tag{14}$$

An equality constraint would specify that $(\lambda^-, \lambda^+) = (-\infty, \infty)$, while an inequality constraint might specify that $(\lambda^-, \lambda^+) = (0, \infty)$. This approach eliminates a lot of bookkeeping and allows for interesting effects, such as motors with bounded torques.

3.6 Constraint Bias

Constraint forces can be made to do work by adding a bias vector ζ to equation (5).

$$JV = \zeta \tag{15}$$

Usually ζ is specified as a function of position and time. This allows us to support active constraints like motors and provides a simple method for position stabiliza-

tion, as discussed below.

4 Contact Model

We use a discrete model of contact that identifies a point of overlap and a normal vector. The model allows for penetration and uses a simple scheme to resolve any residual penetration. The friction model makes some fairly radical changes to the Coulomb model, as we will see.

4.1 Normal Constraint

contact constraint says that the contact points on the touching bodies don't move relative to each other along the contact normal vector n

The normal constraint is formed using the vectors shown in Figure 2. The position constraint C_n measures the object separation, so it is *negative* when the bodies overlap.

$$C_n = (x_2 + r_2 - x_1 - r_1) \cdot n_1 \tag{16}$$

Differentiating C_n with respect to time leads to the velocity constraint.

$$\dot{C}_n = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) \cdot n_1 + (x_2 + r_2 - x_1 - r_1) \cdot \omega_1 \times n_1 \tag{17}$$

We make the usual approximation that the penetration is small, so the second term may be ignored. In this case, it is unimportant which body is associated with the normal vector, so we drop the subscript from n_1. However, our convention is that the normal vector always points from body 1 to body 2.

The Jacobian is determined by separating the velocities from the other terms.

$$J_n V = \begin{pmatrix} -n^T & -(r_1 \times n)^T & n^T & (r_2 \times n)^T \end{pmatrix} \begin{pmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{pmatrix} \tag{18}$$

Note that the Jacobian is a row vector of length 12.

The multiplier for the normal constraint is bounded so that the normal force can push the bodies apart, but not pull them together.

$$0 \leq \lambda_n < \infty \tag{19}$$

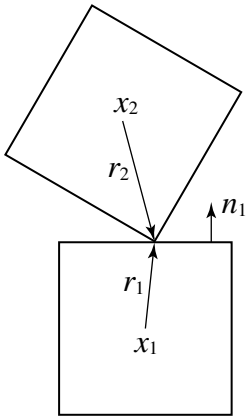

Figure 2: Normal constraint definition. The object origins are x_1 and x_2. The vectors r_1 and r_2 locate the contact points on bodies 1 and 2.

4.2 Handling Penetration

Overlap can occur for a couple reasons. First, using discrete collision detection means that contact is not recognized until the bodies suddenly overlap. Second, numerical integration of the equations of motion is usually not accurate enough to prevent the bodies from drifting into each other.

A Baumgarte [3] scheme is used to push the bodies apart when they overlap. The velocity constraint is augmented with a feedback term proportional to the penetration depth.

$$J_n V = -\beta C_n \tag{20}$$

The scalar β is tunable and governs the speed of penetration resolution. The constraint error C_n is computed using equation (16).

The intuition for equation (20) can be established by rearranging terms. Recalling that $\dot{C}_n = J_n V$, we have

$$\dot{C}_n + \beta C_n = 0$$

This is a first order differential equation in C_n. The solution is $C_n(t) = C_n(0)e^{-\beta t}$, a decaying exponential if $\beta > 0$. Therefore the position error C_n decays more rapidly as β is increased. Alas, we are confined to approximate integration techniques and so β cannot be made too large.

We now present a simplified analysis to determine the stable range of β. Consider

a rigid body which has its center of mass constrained to the point p, but the body may initially violate the constraint. The constraint equation is $C = x - p$. The Jacobian is the 3-by-3 identity matrix, i.e. $J = E_{3\times 3}$. Since the center of mass is fully constrained, we may ignore inertia and external forces. Therefore, the body's velocity is precisely $v = -\beta(x-p)$. Now consider integration of x over the time step Δt using the Euler rule:

$$\begin{aligned} x(t+\Delta t) &= x(t) + \Delta t\, v(t+\Delta t) \\ &= x(t) - \Delta t \beta (x(t) - p) \end{aligned}$$

Rearranging terms leads to:

$$x(t+\Delta t) - p = (1 - \Delta t\beta)(x(t) - p)$$

This is a recurrence relation on the position error and the solution after k time steps is:

$$C(k\Delta t) = (1 - \Delta t\beta)^k C(0)$$

Clearly x will converge to p if and only if $0 < \beta < 2/\Delta t$. If $0 < \beta \leq 1/\Delta t$ then the position error will decay smoothly to zero. If $1/\Delta t < \beta < 2/\Delta t$ then the position error will decay with an oscillation.

The stability analysis provides an upper bound of $\beta \leq 1/\Delta t$ for smooth decay. In practice, smaller values may be necessary for stability. Therefore β should be tuned experimentally, starting with a small value and increasing it until overlap is resolved at a sufficient speed.

4.3 Friction Constraint

Static and dynamic friction are modeled together with two constraints. Given the contact normal vector, two tangent unit vectors are computed, u_1 and u_2, such that $u_1 \times u_2 = n$. The friction force tries to prevent motion in the two tangent directions independently. We write the velocity form of the constraint directly.

$$\dot{C}_{u1} = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) \cdot u_1 \tag{21}$$

$$\dot{C}_{u2} = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) \cdot u_2 \tag{22}$$

The Jacobian is found by inspection.

$$J_u V = \begin{pmatrix} -u_1^T & -(r_1 \times u_1)^T & u_1^T & (r_2 \times u_1)^T \\ -u_2^T & -(r_1 \times u_2)^T & u_2^T & (r_2 \times u_2)^T \end{pmatrix} \begin{pmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{pmatrix} \tag{23}$$

Our experience is that no position stabilization is necessary for friction constraints, even when an object is at rest on an incline.

In Coulomb's friction law, the static and dynamic friction forces have a magnitude that is bounded by the normal force magnitude. This creates an awkward coupling between the constraint multipliers. Using this relationship would complicate our solver and decrease robustness. Therefore, we use a simpler friction model where the friction force is bounded by a constant value.

$$-\mu m_c g \leq \lambda_{u_1} \leq \mu m_c g \tag{24}$$

$$-\mu m_c g \leq \lambda_{u_2} \leq \mu m_c g \tag{25}$$

The usual friction coefficient is represented by μ (static and dynamic coefficients are equal). Each contact point is assigned a certain amount of mass m_c. Typically a body's mass is divided uniformly between the current contact points. The acceleration of gravity is g.

Our experience is that this friction model is sufficient for video games. Static friction works well. Boxes rest on slopes. Dynamic friction is also realistic. However, box stacking friction is unrealistic because lower boxes slide just as easily as upper boxes. The normal force does not affect the strength of the friction.

5 Equations of Motion

Consider a rigid body with a scalar mass m and a 3-by-3 rotational inertia I. The rotational inertia is in world coordinates and is computed from the body version using $I = RI_bR^T$. Similarly, the inverse inertia is computed from the body version using $I^{-1} = RI_b^{-1}R^T$.

An external force f_{ext} and torque τ_{ext} act on the body and are functions of time, position, and velocity. The constraint force f_c and torque τ_c represent the internal

reaction forces of joints and contacts. The Newton-Euler equations of motion are:

$$m\dot{v} = f_c + f_{ext} \tag{26}$$
$$I\dot{\omega} = \tau_c + \tau_{ext} \tag{27}$$

Notice that we are neglecting the inertial torque $\omega \times I\omega$. This term is often small, and when it is not small, it can lead to numerical instabilities. The absence of the inertial torque causes angular velocity—instead of angular momentum—to be conserved in bodies that are free from external and constraint torques.

For a system of n rigid bodies, we collect the masses and rotational inertias along the diagonal of the mass matrix M.

$$M = \begin{pmatrix} m_1 E_{3\times 3} & 0 & \dots & 0 & 0 \\ 0 & I_1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & m_n E_{3\times 3} & 0 \\ 0 & 0 & \dots & 0 & I_n \end{pmatrix} \tag{28}$$

Here $E_{3\times 3}$ is the 3-by-3 identity matrix. Similarly, the external forces and torques are also collected into the vector F_{ext}.

$$F_{ext} = \begin{pmatrix} f_{ext_1} \\ \tau_{ext_1} \\ \vdots \\ f_{ext_n} \\ \tau_{ext_n} \end{pmatrix} \tag{29}$$

As discussed above, the constraint forces for the system can be expressed as $J^T\lambda$. This leads to the constrained equations of motion for n bodies.

$$M\dot{V} = J^T\lambda + F_{ext} \tag{30}$$
$$JV = \zeta \tag{31}$$

Notice that there are $6n+s$ equations and $12n+s$ unknowns: $\dot{V}$, V, and λ.

6 Time Stepping

Our time stepping method supports variable time steps within the region of stability. However, precise repeatability requires a fixed time step. Repeatability may or may not be important, depending on the application.

Consider a time step Δt where the system velocity evolves from V^1 to V^2 (the superscript denotes the time step). The acceleration is approximated to first order.

$$\dot{V} \approx \frac{V^2 - V^1}{\Delta t} \tag{32}$$

This expression is substituted into equation (30).

$$M(V^2 - V^1) = \Delta t(J^T \lambda + F_{ext}) \tag{33}$$

We solve for V^2 in terms of λ, then the constraint equation $JV^2 = \zeta$ is used to eliminate V^2, reducing the problem to a linear equation in λ.

$$JB\lambda = \eta \tag{34}$$

where $B = M^{-1}J^T$ and

$$\eta = \frac{1}{\Delta t}\zeta - J(\frac{1}{\Delta t}V^1 + M^{-1}F_{ext}) \tag{35}$$

The reason for the curious factorization of the coefficient matrix into JB will become apparent in the next section. It is easy to show that B has the same sparsity pattern as J^T, therefore we store B as B_{sp} and use J_{map} for indexing.

We must also consider the bound conditions on λ from equation (14). Obviously, the bounds on λ prevent strict equality from being achieved in equation (34). When a contact constraint does not achieve equality, it means that the objects are separating. When a friction constraint does not achieve equality, it means that it is sliding. Thus, each λ does as much as it can to satisfy its constraint while satisfying its bound conditions.

The matrix JB is positive semi-definite. The reason for any rank deficiency is due to redundant constraints. If the constraints are consistent then there is a non-unique solution for λ. For example, a table with six legs has redundant but consistent support forces. Therefore, an infinite number of force combinations is possible.

Once λ is computed, equation (33) is used to compute V^2. Then the position variables for all bodies are updated independently using first order approximations of the kinematic equations (1) and (2).

$$x^2 = x^1 + \Delta t v^2 \tag{36}$$

$$q^2 = q^1 + \frac{\Delta t}{2} q^1 * \omega^2 \tag{37}$$

Here the superscripts represent the time step.

Since the new velocities are used to compute the position update, our integrator is semi-implicit Euler. This is also known as *symplectic* Euler and is known to have stability and energy conservation properties that rival the Verlet integration scheme [8].

7 Iterative Solution

We obtain the solution of equation (34) using the Projected Gauss-Seidel (PGS) algorithm. This is an iterative algorithm based on matrix splitting [6]. The cost of each iteration is $O(s)$, where s is the number of constraints. An iteration involves simple vector operations on $O(s+n)$ data. The performance of the algorithm is dominated by the number of constraints and the number of iterations used.

7.1 Gauss-Seidel

Algorithm 3 shows the basic Gauss-Seidel method applied to an n-dimensional generic linear equation $Ax = b$. The algorithm requires an initial guess x^0 for the unknowns. The algorithm proceeds for a number of iterations. During an iteration, each row of A is solved by adjusting the element of x corresponding to the diagonal element of A on the current row.

Iterations can be terminated using several different criteria, such as:

- Terminate after a fixed number of iterations.
- Terminate when $||Ax - b||$ falls below a tolerance.
- Terminate when the maximum $|\Delta x_i|$ falls below a tolerance.

Algorithm 3 Approximately solve $Ax = b$ given x^0

$x = x^0$
for $iter = 1$ to iteration limit **do**
  for $i = 1$ to n **do**
    $\Delta x_i = \left[b_i - \sum_{j=1}^{n} A_{ij} x_j \right] / A_{ii}$
    $x_i = x_i + \Delta x_i$
  end for
end for

- Terminate when each $|\Delta x_i|$ is less than some fraction of its value in the previous iteration.

For simplicity, we use a fixed number of iterations.

7.2 Projected Gauss-Seidel

The PGS algorithm extends the basic Gauss-Seidel algorithm to handle bounds on the unknowns. In our case, these are the bounds on λ from equation (14). Bounds are handled by simple clamping, which is surprisingly effective.

We also modify the linear algebra of Algorithm 3 to exploit the sparsity of J and B. Thus, we avoid forming the s-by-s matrix JB. This is crucial for improving performance and reducing memory requirements.

Algorithm 4 shows the Projected Gauss-Seidel method. For simplicity, we have ignored the case where $b_1 = 0$ (indicating ground). First the algorithm creates the temporary vectors a and d. The vector a is initialized with the product $B\lambda$, which can be computed in a fashion similar to Algorithm 2. The vector d is initialized with the diagonal elements of JB. Note that the elements of J, B, η, and a are stored as 6-vectors. Each time an increment to λ_i is computed, λ_i is clamped to its bounds and the actual increment is determined. With the actual increment of λ_i, the vector a is updated so that it remains equal to $B\lambda$.

The PGS algorithm has $O(s)$ running time and $O(s+n)$ storage requirements. According to Cottle [4], convergence is guaranteed if JB is positive definite. In many typical simulations JB is only positive semi-definite. Nevertheless, we have seen good visual and numerical results in practice.

Algorithm 4 Approximately solve $JB\lambda = \eta$ given λ^0

Work variables: a ($6n$-by-1), d (s-by-1)
$\lambda = \lambda^0$
$a = B\lambda$
for $i = 1$ to s **do**
  $d_i = J_{sp}(i,1) \cdot B_{sp}(1,i) + J_{sp}(i,2) \cdot B_{sp}(2,i)$
end for
for $iter = 1$ to iteration limit **do**
  for $i = 1$ to s **do**
    $b_1 = J_{map}(i,1)$
    $b_2 = J_{map}(i,2)$
    $\Delta\lambda_i = [\eta_i - J_{sp}(i,1) \cdot a(b_1) - J_{sp}(i,2) \cdot a(b_2)] / d_i$
    $\lambda_i^0 = \lambda_i$
    $\lambda_i = \max(\lambda_i^-, \min(\lambda_i^0 + \Delta\lambda_i, \lambda_i^+))$
    $\Delta\lambda_i = \lambda_i - \lambda_i^0$
    $a(b_1) = a(b_1) + \Delta\lambda_i B_{sp}(1,i)$
    $a(b_2) = a(b_2) + \Delta\lambda_i B_{sp}(2,i)$
  end for
end for

7.3 Complementarity

Technically speaking, we are solving a Mixed Linear Complementarity Problem (MLCP). Our constrained dynamics problem may be written as the MLCP:

$$\begin{aligned} w &= JB\lambda - \eta \\ \lambda^- &\le \lambda \le \lambda^+ \\ w_i = 0 \;&\leftrightarrow\; \lambda_i^- \le \lambda_i \le \lambda_i^+, \;\; \forall i \\ \lambda_i = \lambda^- \;&\leftrightarrow\; w_i \ge 0, \;\; \forall i \\ \lambda_i = \lambda^+ \;&\leftrightarrow\; w_i \le 0, \;\; \forall i \end{aligned}$$

where w is the *constraint velocity*. The first two lines are a restating of equations (34) and (14). The last three lines are the complementarity conditions. The first condition states that the constraint can be satisfied as long as λ is within its bounds. The last two conditions state that λ will only reach its bound if the constraint is violated (the constraint velocity is nonzero). There are many ways of analyzing and solving MLCPs. See Cottle [4] for details.

8 Contact Caching

The PGS algorithm is iterative by design. The convergence rate can be slow, depending on the eigenvalues of JB. However, by caching the constraint force multiplier λ, we can improve the solution over several time steps, especially if there is temporal coherence in our simulation. Contact caching is discussed in the larger context of contact reduction in Moravanszky [10].

8.1 Caching Scheme

In a typical simulation, contact points come and go. We use a discrete approach to collision detection, i.e., contact points are generated without using the previous position or velocity. Only the current position data is used to determine overlaps and generate contact points. Thus we need an efficient method for storing all contact points and their λ values in a contact cache after each time step. And we need an efficient method to query the cache as new contact points are generated in the next time step.

We now describe our caching scheme as shown in Algorithm 5. After each time step a new contact cache is created from scratch. Each collision primitive—sphere, box, convex polyhedron—has cache space reserved to keep track of interactions with other primitives. These *interaction pairs* are symmetric: primitive A knows that it is interacting with primitive B and primitive B knows it is interacting with primitive A. For each primitive pair, we keep a list of contact point entries. Each contact point entry stores λ and an identifier.

Cache queries are performed as contact points are generated, before the points are passed to the PGS solver. First, the cache is queried for the interaction pair. If the pair is found, then each contact point entry corresponding to the pair is scanned and its identifier is matched to the query identifier. If a match is found, then the λ values are retrieved from the cache and passed to the solver. If there is a cache miss, then the λ values are initialized to zero.

8.2 Contact Point Identifiers

The contact point identifier can take several forms. Some possibilities are:

Algorithm 5 Contact caching and queries.

```
while simulate = true do
  generate contact points
  for all contact points do
    query the cache for point i
    if cache hit then
      λ_i^0 = λ_cache
    else
      λ_i^0 = 0
    end if
    initialize contact constraint using λ_i^0
  end for
  solve for V^2 and λ using Algorithm 4
  destroy old contact cache
  create new contact cache
  for all contact constraints do
    store λ_i and the contact identifier in the cache
  end for
end while
```

1. position in global coordinates
2. position in local coordinates
3. incident edge labels

Position identifiers are easy to implement, but they require a tolerance and may lead to aliasing. In other words, a contact point may *borrow* λ from a neighboring contact point.

Incident edge labels may be preferred over position identifiers because they prevent aliasing. Consider the case of box-box collision as shown in Figure 3. Each box has its edges numbered. If the collision is edge-edge then the contact identifier is combination of the two edge numbers. If the collision is face versus vertex, edge, or face, then the polygon for the reference face is identified. Then an incident face on the other box is chosen to be the face that is most anti-parallel to the reference face. The incident face is clipped against the reference face to generate a set of potential contact points, see Figure 4. The separation distance is measured for each clipping point. Any clipping point with a non-positive separation be-

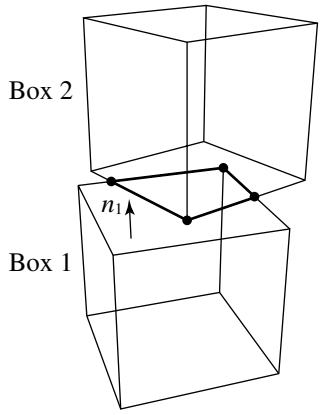

Figure 3: Two boxes in contact. The incident face of box 2 is clipped against the reference face of box 1.

comes a contact point. During the clipping process, we keep track of which edges created the point. The combination of the indices of these two edges becomes the contact identifier.

9 Results

9.1 Box Stacking

Box stacking is one of the most common tests performed on physics engines. The stacking problem requires the physics engine to have several capabilities that all perform well.

- The collision system needs the ability to compute a contact manifold between two boxes. The manifold can be computed from scratch each frame, but it should not change much if the relative position of the boxes does not change significantly.
- The engine must accurately solve a large coupled system of constraints. If the constraint force solution is not accurate enough, the stack will fall.
- The constraint solver needs to handle *redundant* constraints. For example, in our engine, each box is often supported by four points.

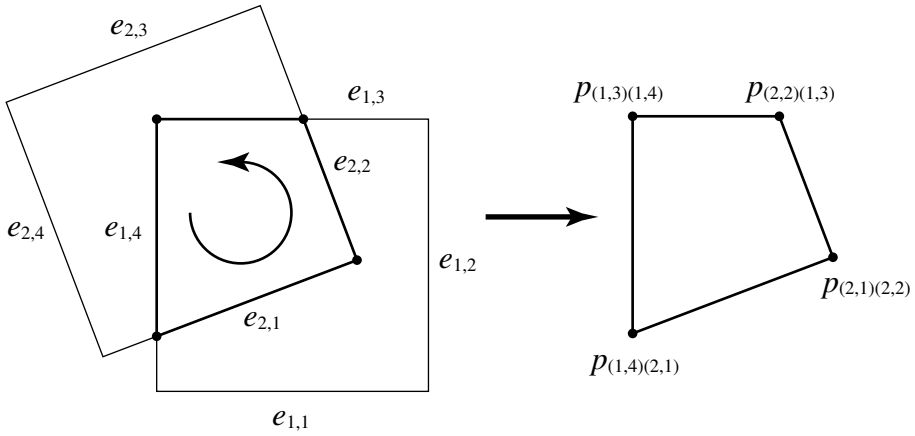

Figure 4: Polygon clipping. Each resulting vertex is identified by the labels of the two edges that created it, using counter-clockwise order.

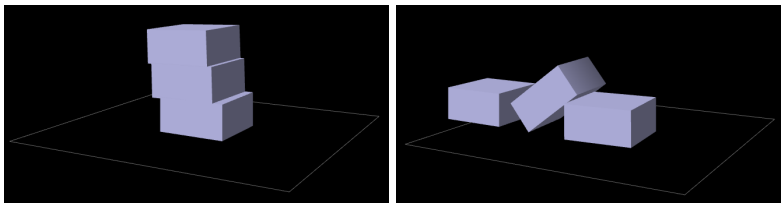

Figure 5: Without contact caching, a stack of three boxes is unstable.

- The engine needs to compute friction forces, especially static friction. This is needed to prevent the boxes from sliding off of one another.
- If the stack does fall, due to an unstable configuration or external forces, the boxes should come apart and slide naturally without sticking together.

We performed a comparison of box stacking with and without contact caching. Otherwise, all simulation parameters were identical. Table 1 gives some details. Note that these parameters correspond to our internal game units, but roughly speaking, the box is about 70cm $\times$ 70cm $\times$ 30cm and has a mass of about 110kg, gravity is $9.8\text{m}/\text{s}^2$, and the time step corresponds to 60 frames per second. All collisions were treated as inelastic and no separate algorithms were used for collision impulses.

Figure 5 shows a stack of three boxes simulated without contact caching. The boxes were initially separated vertically by 10 units. As the simulation progressed,

Parameter	Value
density	0.0000275
dimensions	$200 \times 200 \times 100$
gravity	3
β	0.1
μ	0.2
time step	0.5
PGS iterations	10
initial separation	10

Table 1: Box stacking test parameters.

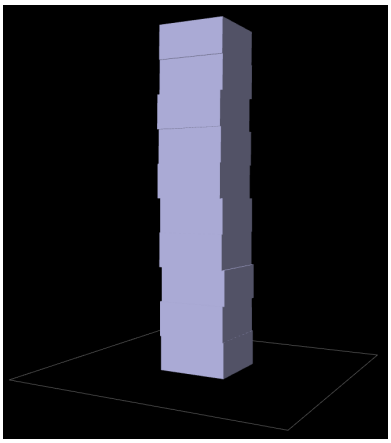

Figure 6: With contact caching, a stack of ten boxes is stable.

the boxes slid and fell. Figure 6 shows a stack of ten boxes simulated with contact caching. The stack teetered a bit when first formed, but stabilizes and becomes motionless within a few seconds. No body sleeping (deactivation) or damping was used.

9.2 Extreme Mass Ratios

The PGS algorithm does have some limitations, particularly when dealing with extreme mass ratios. In box stacking this leads to stability problems when a heavy box is placed on top of a light box. We tested a stack of two boxes using the same parameters as before and with contact caching enabled. We steadily decreased the

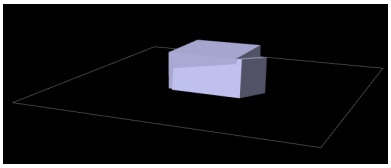

Figure 7: Without contact caching and a mass ratio of 1%, the top box sinks into the lighter bottom box.

density of the bottom box until the top box would fall. The stack was stable until the density ratio was lowered to 3%. With lower mass ratios the top box would bounce and slide off the bottom box.

We tried the test with a 1% mass ratio. With contact caching, the top box slides off the bottom box. Without contact caching, the top box sinks into the bottom box! See Figure 7. On the other hand, the stack is completely stable when the light box is on top, with or without contact caching. This leads the the design principle: *heavy objects can support light objects, but not vice-versa*.

10 Conclusion

We have presented a system for modeling and solving rigid body dynamics for games, including contact and friction. The system requires linear time and space. Using contact caching, we are able to amortize the cost of computing accurate contact forces over several frames. The result is a system that performs well enough to be used on the current generation of console hardware.

11 Acknowledgments

The author wishes to thank Gary Snethen for his helpful feedback and stimulating conversations. The author would also like to thank the wonderful team at Crystal Dynamics for their support.

References

[1] Mihai Anitescu and Florian A. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems. *Nonlinear Dynamics*, 14:231–247, 1997.

[2] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of SIGGRAPH 1994*, pages 23–34, 1994.

[3] J. Baumgarte. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering*, 1:1–16, 1972.

[4] Richard W. Cottle, Jong-Shi Pang, and Richard E. Stone. *The Linear Complementarity Problem*. Academic Press, 1992.

[5] David H. Eberly. *Game Physics*. Morgan Kaufmann, 2003.

[6] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.

[7] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex rigid bodies with stacking. In *Proceedings of SIGGRAPH 2003*, pages 871–878, 2003.

[8] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration*. Springer-Verlag, 2002.

[9] Thomas Jakobsen. Advanced character physics. In *Proceedings of Game Developers Conference 2001*, 2001.

[10] Adam Moravanszky and Pierre Terdiman. Fast contact reduction for dynamics simulation. In *Game Programming Gems 4*, pages 253–263. Charles River Media, 2004.

[11] Ahmed A. Shabana. *Computational Dynamics*. John Wiley and Sons, 2nd edition, 2001.

[12] Russell Smith. Constraints in rigid body dynamics. In *Game Programming Gems 4*, pages 241–251. Charles River Media, 2004.

[13] Russell Smith. Open dynamics engine. http://www.ode.org, 2004.