Rigid Body Simulation

Jeremy Ulrich
Advised by David Mount
Fall 2013

Overview

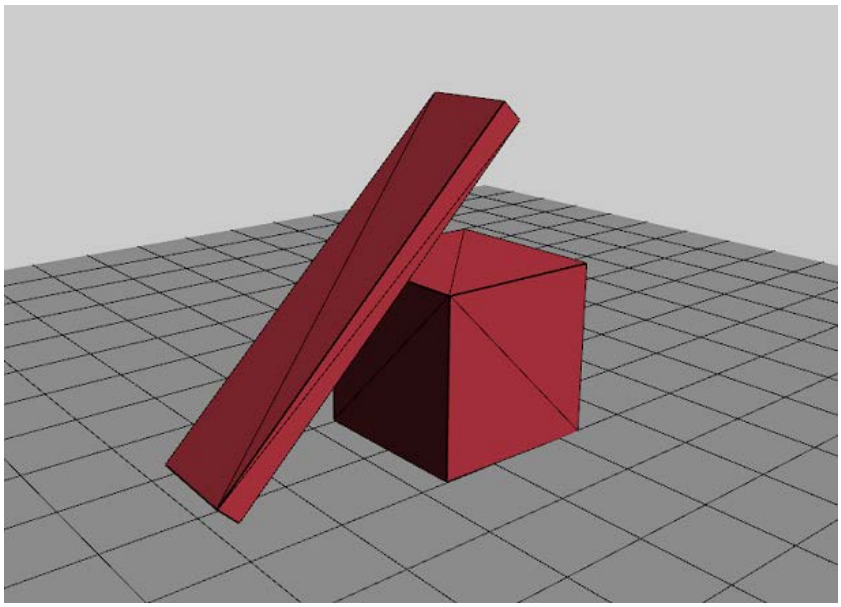
The project presented here is a real-time 3D rigid body physics engine, and is the result of an Independent Study on collision detection and response in the context of rigid body dynamics. The simulation produces realistic motion, collisions, friction and resting contact between multiple bodies.

This paper briefly discusses the spectrum of ideas and algorithms that were needed to implement the engine.

- The engine works mostly with convex polyhedra, and employs several tools in order to efficiently perform convexity-based queries and calculations. A few of these, such as the Quickhull algorithm, are examined in the first section.
- The collision detection system consists of a broad phase and narrow phase. The broad phase uses a uniform grid and bounding spheres, and the narrow phase includes the Separating Axis Test for determining intersection between convex polyhedra. The second section deals with collision detection.
- Finally, the dynamics system employs a velocity-based constraint model for contact and friction constraints, resolves the constraints using an iterative Sequential Impulse solver, and calculates the motion of the rigid bodies. These topics are covered in the third section.

About the Engine

The project was implemented in C++ with Visual Studio 2010. The graphics uses OpenGL and FreeGLUT, an open-source alternative to the GL Utility Toolkit. No other outside code or programs were used. All algorithms and functionality in this paper are implemented as described, barring details or efficiency-related improvements not mentioned. The engine is intended to be used as a practical 3D physics engine in upcoming projects, and will be extended to support more shapes and constraint types in the near future.



1. Convex Polyhedra

A polyhedron is convex if every line segment between two points on its surface is entirely contained in the polyhedron. This simple property makes convex polyhedra extremely useful in many areas of real-time simulation. They are able to approximate any convex shape arbitrarily well, but still allow for relatively efficient geometric queries and computations. For instance, finding a vertex that is farthest in some given direction can be done in logarithmic time, rather than the expected linear time from iterating over each vertex.

Every rigid body in the physics engine is a convex polyhedron. Each polyhedron is stored as a Doubly Connected Edge List (DCEL), and is constructed from a point set using a 3D implementation of Quickhull. The engine also implements several typical convex polyhedra-related algorithms, such as testing for point containment, finding supporting vertices, and testing for intersection with primitives such as rays and planes, although these topics are not discussed further here.

Doubly Connected Edge Lists

Efficiently solving convexity-based problems requires a data structure that exploits the connected nature of the polyhedron. The DCEL can represent general polygonal meshes, and is well-suited for traversing the features of the mesh.

A general DCEL consists of three sets of structures – vertices, half-edges, and faces. Each edge on the polyhedron is represented as two directed half-edges in the DCEL. A vertex structure contains its coordinates, and a reference to some half-edge that is incident to it. A face structure contains a reference to a half-edge bordering it, where the face is on the half-edge's left. Finally, a half-edge structure contains a reference to its origin vertex, its twin half-edge, the face to its left, and the next and previous edges on that face's border.

Using a DCEL, many convexity-based algorithms can be improved. The farthest vertex query mentioned earlier, for example, can be found with a simple hill climbing algorithm, avoiding the need to check many of the polyhedron's vertices.

The engine's implementation sacrificed a bit of space for clarity of code by including a few more references in the structures. The polyhedra always have triangular faces, primarily to simplify calculating their inertia tensor. Therefore, each face holds a reference to all three of its edges and vertices. Each face also stores its normal vector to reduce run-time computations.

Quickhull

In order to construct a convex polyhedron from an arbitrary set of points, the engine computes the convex hull of the set using the Quickhull algorithm. The central idea of the algorithm is that if a point is the farthest point in some direction (it is an extreme point) then it must be on the convex hull. Conversely, any point inside the current hull is certainly not on it, and can therefore be discarded.

The algorithm starts by selecting four extreme vertices from the input set, initializing the hull to the tetrahedron defined by those points, and discarding any points inside. It then repeatedly extends the polyhedron by finding an extreme point in some direction, adding it to the hull, and discarding any points now contained in the hull. Once there are no points left outside the hull, it is returned.

Extra care was needed when extending the 2D version of Quickhull into 3D. In the 2D version, extreme points are found by iterating over the edges of the current hull. If an extreme point V is found for some edge E, the hull is easily updated by replacing E with two edges from V to each of E's vertices. The same approach does not work in 3D – when an extreme point is added, multiple adjacent faces may need to be deleted, and only the outer edges should be connected to the extreme point to form new faces. The DCEL structure comes in handy here. It provides an easy way to both find all the faces that need to be discarded, and construct a loop of the edges that need to be connected to the new extreme point.

One iteration of Quickhull is shown in Figure 1. The blue faces and red points make up the current hull, and the white points are those still in the input list. The green point was found as the next extreme vertex. The two faces in gray will be discarded, and the green triangles will become new faces. Input points that were found to be inside the new hull and can be discarded are black.

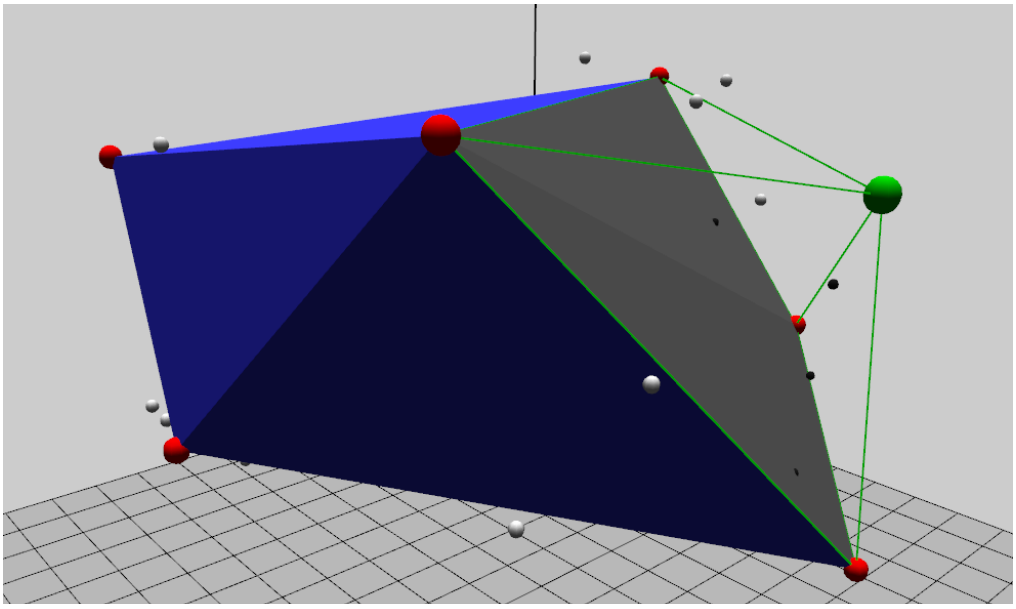


Figure 1 - One iteration of Quickhull

2. Collision Detection

The central task of a collision detection system is to find all intersecting pairs of objects in a system. In order to avoid testing $O(n^2)$ object-pairs for intersection in a system of n bodies, most collision systems split the process into a broad phase and narrow phase. The broad phase is responsible for pruning the number of object-pairs that need to be tested, and makes use of various tools such as spatial partitioning structures and bounding volumes in order to discard as many pairs as quickly as possible. The narrow phase then tests each object pair, and generates contact information for any pair of objects that are actually intersecting.

Broad phase – Bounding Spheres and Uniform Grids

To efficiently find all intersecting object-pairs, the broad phase must minimize the number of pairs that get tested in the narrow phase. There are two general strategies for this task. The first is to construct simple bounding volumes for each object. A bounding volume can be any simple shape – popular choices include boxes, spheres, cylinders, and capsules – that encloses a more complicated object, and provides a very cheap approximate intersection test. Only if the bounding volumes of a potential pair of objects are intersecting would the pair be passed on to the narrow phase for more expensive tests.

For this project, the convex polyhedra are enclosed in spheres, which are in many ways the simplest bounding volume. They are defined only by a center and a radius. The spheres are somewhat trivially calculated by using the centroid of the polyhedra as the center of the sphere, and the maximal distance from any point on the polyhedra to its centroid as the radius. While spheres sometimes provide poor fits for certain object shapes (thereby reducing the effectiveness of the bounding volume) they have the cheapest intersection test. Two spheres intersect simply if their centers are closer together than the sum of their radii.

The second general broad phase strategy is to keep all the objects in a spatial partitioning data structure that allows the system to quickly find all other objects that are near a given object. Again, there are several popular structures, each with their pros and cons. Here, the collision system uses a uniform grid. World space is split into a regular grid of 2D cells, and each cell contains a reference to objects that overlap it. Adding an object to the grid requires adding it to each cell that it overlaps, and a query for nearby objects returns the objects contained in all of those cells. The efficiency of these operations is maximized by using a cell size at least as large as the largest object in the grid, but not much larger. The lower bound ensures that an object only overlaps at most four cells, while the upper bound minimizes the number of objects per cell.

Uniform grids have extremely low time cost. Given a point in space, its corresponding grid cell can be found in constant time with respect to the number of objects in the grid. Given a radius, the

neighboring cells that are overlapped can also be found in constant time. Because of this, insertion, deletion, and queries for nearby objects can all be done in constant time. The drawback (at least in this straightforward implementation) is the memory cost – a grid that splits space into n cells by n cells stores n^2 cells, even if many of them are empty. Very parse systems therefore would benefit from a more dynamic partitioning, such as a quad-tree.

Figure 2 illustrates the effectiveness of the broad phase. The 15 bodies in free fall would require 210 expensive convex-convex intersection tests with a brute force approach. Using bounding spheres, the 210 approximate sphere-sphere tests lowered the number of expensive tests needed this frame to only 22. Storing the bodies in a uniform grid, only 38 object pairs were close enough to require a sphere-sphere test in the first place. A more spread out system of bodies would result in even greater savings.

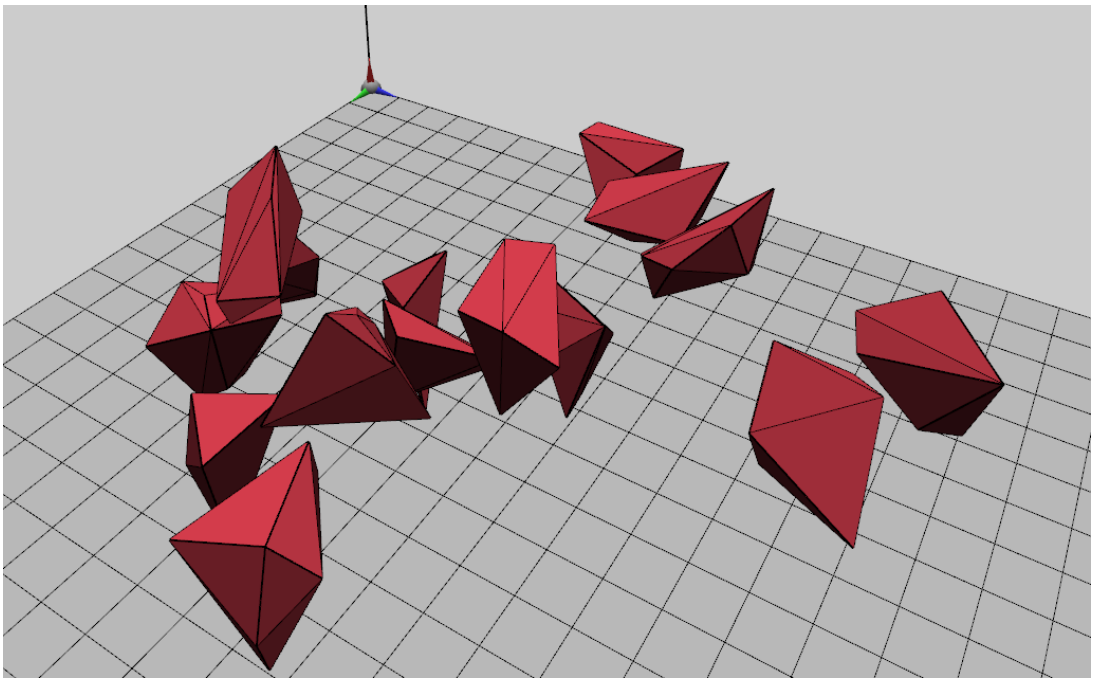


Figure 2 - The broad phase reduces the number of convex tests needed here by almost 90%

Narrow phase – The Separating Axis Test

Once two objects have been determined to be sufficiently close to each other, the collision system must check if they actually intersect. The exact test used for intersection depends on the combination of shapes being tested, resulting in a large number of possible tests for even a moderate number of shapes. In fact, the wide variety of shapes supported by a commercial physics engine makes collision dispatch, the task of finding the correct algorithm for a given object pair, a nontrivial problem.

The intersection test of interest here is the one for two convex polyhedra. The algorithm implemented for this project is the Separating Axis Test (SAT). The SAT relies on another important

theorem regarding convex shapes – two convex shapes do not intersect if and only if there is a plane that separates them. If a separating plane exists, its normal vector is called a separating axis, and so finding whether two convex shapes intersect is equivalent to finding whether a separating axis exists between the two shapes.

A given axis can easily be tested to see if it is a separating axis. The projection of a convex polyhedron onto an axis is a line segment, the endpoints of which are determined by the polyhedron's extreme vertices in each direction of the axis. If the projected segments of the two polyhedra do not overlap, then the axis is a separating axis, and the test can return no intersection.

The problem that remains is to find such an axis, if it exists. It turns out that there are only a limited set of possible axes that need to be tested. If a separating axis exists, then either the normal vector of one of the faces of one of the polyhedra will provide one, or the cross product of an edge vector from one polyhedron with an edge vector of the other will. The SAT iterates through each possible axis, and returns no intersection if a separating axis is found. If none of the axes work, then the two polyhedra intersect.

This algorithm is not very efficient for large polyhedra – it needs to check $2F + E^2$ axes. It is possible to improve the running time by using the temporal coherence usually present in real-time simulations. If a particular axis serves as a separating axis between two objects in one frame, it will very likely still be a separating axis next frame. By caching the result of each SAT, and checking the cache before testing a pair of objects from scratch, the test becomes much more efficient. This optimization was not implemented, but the algorithm runs well at least for moderately small polyhedra.

Contact Manifold Generation

Accurately handling the collision of two objects requires more than just the simple fact that they are indeed colliding. The collision response system needs information about the contact between the objects, including the points or areas of contact, and the penetration depth. The collision detection system is responsible for producing this information – the contact manifold – for each pair of intersecting objects.

The implemented engine produces somewhat simplified contact manifolds. They contain a reference to the two colliding objects, a normal vector for the contact, a list of contact points, and the approximate penetration depth of the intersection. The SAT from the previous section is extended to produce the manifold for intersecting objects.

If two intersecting convex polyhedra are projected onto an axis, the amount of overlap of the projections is the distance the two polyhedra would need to be pushed away from each other along that axis to not be intersecting. As the SAT tests each axis for separation, it keeps track of the axis that has the smallest overlap. If no separating axis is found, then the direction of the axis producing the minimal

overlap – the “contact axis” – is used as the contact normal. By convention, the contact normal in an intersection between objects A and B is chosen to point towards A and away from B. The minimal overlap is used as the penetration depth. The method for finding contact points depends on the type of the contact axis. If it was formed from the cross product of two edges, the intersection of those edges is used as the single contact point. Otherwise, the axis corresponds to a face on one of the polyhedra, and the points are found by examining the “contact plane” defined by that face. Any vertex in one polyhedron that is both within a small tolerance of the contact plane and a small tolerance of being contained in the other polyhedron is included as a contact point. The tolerances serve to improve performance in the physical simulation, by making the contact manifold more consistent from frame to frame. Requiring exact intersection can cause points to count as intersecting one frame, not intersecting the next, and so on, which leads to jitter.

The current scheme for generating manifolds is possibly the weak link of the current collision detection implementation. No matter how good an engine’s constraint solver is, inaccurate contacts have a tendency to cause weird behavior. One possible improvement would be to give each contact point its own penetration depth. Alternatively, rather than a set of points, a contact region could be computed – a point, line segment, or polygon, depending on the configuration of the intersecting polyhedra.

Figures 3 and 4 depict the two types of contact points, shown as black points. On the left, the two tetrahedrons were found to intersect along two edges, and the intersection of those edges is used as the contact point. The figure on the right shows two manifolds resulting from face intersections – one between the box and tetrahedron, and one between the box and the ground.

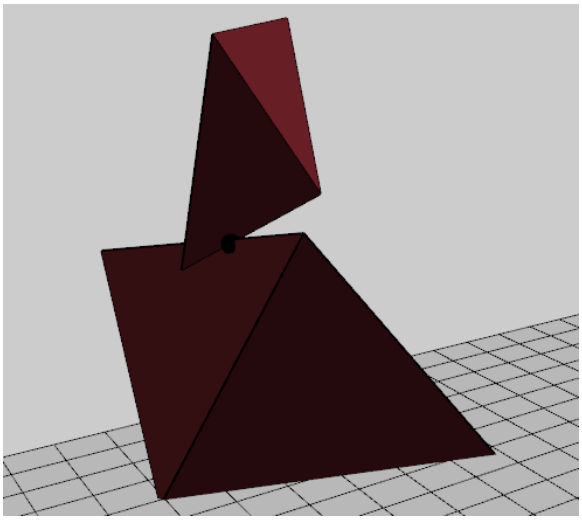


Figure 4 - Single edge-edge contact point

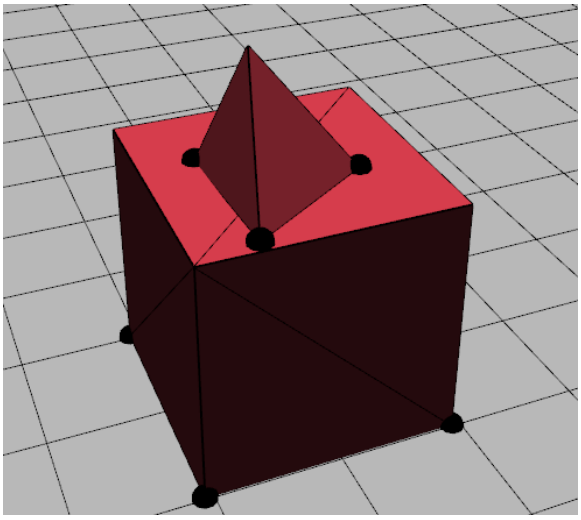


Figure 3 - Multiple face contact points

3. Rigid Body Simulation

The other half of the physics engine is concerned with the motion of the simulated bodies. Rigid body dynamics can also be broken into two core components. First, the dynamics component encapsulates the physical state of a rigid body, and governs how that state changes over time. At its center is a set of equations of motion, relating the forces on an object to its change in position and velocity.

Second, the collision response component works to ensure that the current state of the system is physically valid. The foremost restriction on rigid bodies is that they cannot penetrate each other. Many other restrictions can be used to create various behaviors as well, for instance keeping two objects a certain distance from each other, or only allowing a lever to rotate a certain amount. A popular way to implement these restrictions in rigid body simulations is with a constraint model – each restriction of the physical state of the system is enforced with a constraint, an equation on the variables of the state. For example, if the function $dist(p, q)$ returns the distance between points p and q, then the equation $dist(p, q) = d$ is a position constraint on p and q. Enforcing the constraint results in keeping p and q at a distance d from each other. Constraints are enforced (at a conceptual level at least) by applying internal forces, called constraint forces, to the constrained bodies. Non-penetration constraints, for instance, produce the normal force between two objects. The engine currently supports contact and friction constraints, and hopefully more types of constraints will be implemented soon.

Given a set of constraints, the simulation would then use a constraint solver to figure out the constraint forces necessary to satisfy them. There are two main approaches to this problem – global solvers and iterative solvers. Global solvers solve the entire system simultaneously, and while they produce optimal answers, they are very slow. Iterative solvers adjust each constraint force locally, looping over all the constraints repeatedly until some stopping criteria is met. Good iterative methods do converge to a global solution, and therefore provide a fast approximation for the system of constraint forces. The solver that this project uses, the Sequential Impulses method, is an iterative solver.

Rigid Body Dynamics

The physical state of the rigid bodies in the engine consists of several variables. Each body has a constant scalar mass m, and a constant (in body-space) 3x3 inertia tensor I. Its time dependent state is described by its position x, linear velocity v, rotation R, and angular velocity ω, where x, v, and ω are 3D vectors and R is a 3x3 matrix. The change of these time-dependent variables can be expressed as functions of each other, the applied force F and torque τ on the object, and time, to form a set of differential equations that drive the motion of the system:

$\frac{dv}{dt}(t) = F(t)/m, \quad \frac{dx}{dt}(t) = v(t)$ for linear motion, and

$\frac{d\omega}{dt}(t) = I^{-1}\tau(t)$, $\frac{dR}{dt}(t) = (\omega(t)^*)R(t)$ for angular motion, where the star operator is defined as

$$a^* = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & x & 0 \end{bmatrix}.$$

These equations are integrated using the symplectic Euler method. Given the current linear motion state, applied force, and time step length h, the following two equations produce the new linear state:

(1) $v_{new} = v_{old} + hF(t)/m$
(2) $x_{new} = x_{old} + hv_{new}$

The equations for angular motion are similar, although extra work needs to be done to address the fact that the body's inertia tensor is stored in body space.

Given these equations, the system of bodies are updated each frame in the following manner. First, external forces and torques (such as gravity) are applied to each body. Next, using (1), the velocities of each body are integrated. The constraint solver then corrects the velocities in order to enforce all constraints (as described in the next two sections). Finally, (2) is used to integrate the new positions.

Detailed explanations and derivations for the equations in this section can be found in Baraff[1].

Constraints

For this project, contact and frictional constraints were implemented. A constraint of each type is created for each contact point found in the collision detection step. The constraints are velocity-based, in that they are constraints on the velocities of the two contacting objects.

The contact constraint requires that the relative velocities of the two objects parallel to the contact normal must be greater than or equal to zero – that is, they cannot be moving towards each other. The contact constraint force, or normal force (technically impulse, see next section), is applied along the contact normal and in opposite directions in order to push the objects apart. Its magnitude is limited to [0, ∞), so that it can only push the two objects away from each other, and will push as hard as it needs to in order to enforce the constraint.

The frictional constraint requires that the relative velocities of the two objects perpendicular to the contact normal – i.e., along the contact plane – be zero. However, frictional forces are not unlimited. In fact, they are limited by the normal force at that contact point: $F_f \leq \mu F_n$, where μ is the friction constant for the pair of contacting objects. Because of this, each time an impulse is applied to a certain contact point's normal constraint, the limit for its friction constraint must be updated.

Sequential Impulses

The constraint solver chosen for this engine is a Sequential Impulse solver. Impulse is simply change in momentum: $J = \Delta P = m\Delta v$. Rather than find a set of constraint forces, which technically would need to be infinite in magnitude in order to produce an instantaneous change in velocity, the solver attempts to find a set of impulses that will result in the desired post-collision relative velocities.

The solver starts by building the constraints out of the given contact manifolds, each of which keeps track of its current applied impulse (initialized to zero). It then runs for a fixed number of iterations. In a given iteration, the solver loops over each constraint, calculates the additional impulse it needs to apply given the constraint's current applied impulse and velocities, and clamps the resulting total impulse according to the limits in the previous section. Finally, the velocities of the rigid bodies are updated based on the constraint impulses, and the simulation can safely continue, penetration-free.

The precise relation between a velocity constraint and the impulse needed depends on the type of constraint. See Catto[2] for details on the necessary equations for normal and frictional constraints. It turns out though that solving the set of constraints comes down to solving a system of linear equations. Because of this, applying the Sequential Impulses method is actually equivalent to solving the system of equations with the Projected Gauss Seidel algorithm. Again, see Catto[2] for details.

Because the simulation updates bodies in discrete time steps, small penetrations between objects cannot be avoided, even with a good constraint solver. To fix this, the engine uses Baumgarte stabilization – if the penetration depth at a contact point reaches a certain threshold, the velocity constraint at that point is increased by an amount proportional to the depth. This serves to push intersecting bodies away from each other a little harder to correct the penetration.

Sources

[1] David Baraff. Physically based modeling. *SIGGRAPH 2001 Course Notes,* 2001.

[2] Erin Catto. Iterative dynamics with temporal coherence. Technical Report, Crystal Dynamics, Menlo Park, CA, 2005.

[3] Christopher Ericson. *Real Time Collision Detection*. Morgan Kaufmann, 2005.

[4] David Mount. Geometric Data Structures for Games: Meshes and Manifolds. *CMSC425 Course Notes*, 2013.

[4] F. Tonon. Explicit exact formulas for the 3-D tetrahedron inertia tensor in terms of its vertex coordinates. *Journal of Mathematics and Statistics*, 1:8-11, 2004.

[5] Advanced Micro Devices. Bullet Collision Detection and Physics Library [Computer program]. http://bulletphysics.org (Accessed 2014).