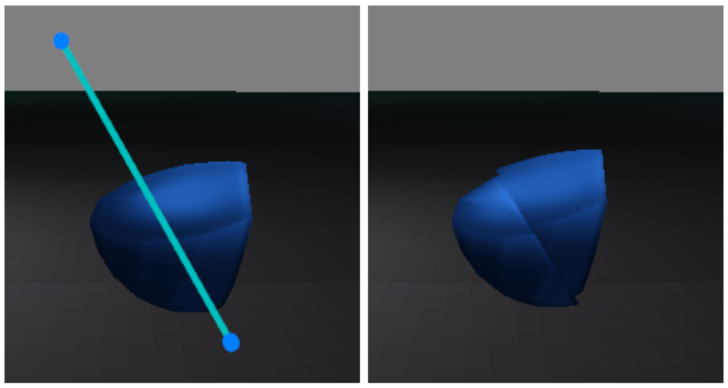
Constraint-Based Physics With Real-Time Object Slicing

Tommy A. Brosman IV
DigiPen Institute of Technology
tommy.brosman@gmail.com

April 2011



Abstract

This paper presents a method for slicing meshes suitable for real-time simulation. The physics engine itself is also briefly covered, along with some implementation details. A derivation of constraint-based dynamics from D'Alembert's principle of virtual work and variational principles is also presented.

1 Previous Work

The physics engine itself is most similar to the one described in [Cat05], with an iterative, local constraint solver. The additional fracturing code borrows concepts from [Mil04] in that avoids relying on FEM methods for breaking and re-meshing objects.

2 Constrained Dynamics

2.1 Concepts

The physical animation is accomplished using constraint-based methods. These methods can be derived from classical mechanics, starting from Newtons equations and DAlemberts principle of virtual work.

According to Newtons equations, for a system in equilibrium, the total force on each particle will disappear [Gol02]:

$$F_i = 0$$

Here, F_i is the total force for the i^{th} particle in the system. The total work in the system is written:

$$W = \int \sum_i F_i \cdot dr_i$$

Taking the variation of the work gives the virtual work:

$$\delta W = \sum_i F_i \cdot \delta r_i$$

Which is 0. Splitting F up into external (applied) and constraint forces:

$$\delta W = \sum_i F_i^{(c)} \cdot \delta r_i + \sum_i F_i^{(ext)} \cdot \delta r_i = 0$$

For rigid bodies, it is a necessary condition that the net virtual work is always 0. As a result, when a system is in equilibrium, the virtual work of applied forces also vanishes.

A simple positional velocity constraint can be derived using this concept. Given some particle with position $r(t)$, and a plane with normal n, the velocity of the particle should be orthogonal to the normal (in other words, all motion is within the plane):

$$\hat{n} \cdot v(t) = 0$$

This is equivalent to constraining a particle to the plane with a virtual force:

$$\begin{aligned} F^{(c)} \cdot \delta r &= 0 \\ F^{(c)} \cdot \frac{dr}{dt} &= 0 \\ F^{(c)} \cdot v &= 0 \end{aligned}$$

Which illustrates that no work over time is equivalent to no power. Where the constraint force is equal to some undetermined multiplier times the normal to the plane:

$$F^{(c)} = \lambda \hat{n}$$

2.2 Variational Principles

The variation of a variable is different than the variable itself (the displacement over an infinitesimal amount of time vs. the actual position). Under general conditions, it is also different from the total differential, (which by itself has no direct physical interpretation, unlike a variation). Therefore,

$$F \cdot r \neq F \cdot \delta r$$

The equality between the variation of displacement and actual displacement can be made because an infinitesimal amount of time is passing [Lan70].

$$\delta r = \frac{dr}{dt} dt = v \ dt$$

In the formulation of virtual work, both sides are divided by the infinitesimal scalar quantity dt ($\frac{0}{dt} = 0$). This changes the equation from work (Joules) to power (Watts). Note that this is different from taking the time derivative directly, as the equation is already expressed in terms of a variation.

$$\frac{1}{dt}(F^{(c)} \cdot \delta r) = F^{(c)} \cdot \frac{dr}{dt} = F^{(c)} \cdot v$$

2.3 Geometric Constraints

Given some particle with position $r(t)$, and a plane with normal $\hat{n}$, an inequality constraint can be formulated to keep the particle to one side of the plane. Geometrically, this is the implicit equation:

$$\hat{n} \cdot (r - p) \geq 0$$

Where p is some point in the plane. This is different from the earlier constraint equation as the motion is now restricted to an affine plane (a plane that does not necessarily pass through the origin) and expressed as an inequality. This is a geometric constraint, specifically a contact constraint, and is the basis for formulating contact constraint forces.

The scalar quantity $\hat{n} \cdot v(t)$ associated with the previous constraint is called the closing velocity and can be used to classify whether a constraint is violated. $\hat{n} \cdot v(t) < 0$ corresponds to a potential collision (closing), $\hat{n} \cdot v(t) = 0$ is a resting contact (when the spatial constraint is also 0), and $\hat{n} \cdot v(t) > 0$ is separating. This quantity is the time derivative of the contact constraint:

$$\begin{aligned} \frac{d}{dt}(\hat{n} \cdot (r - p)) &\geq \frac{d}{dt} 0 \\ \hat{n} \cdot v &\geq 0 \end{aligned}$$

So for a contact constraint to be satisfied, it is a necessary (but not sufficient) condition that the particle is either at rest or separating.

2.4 Generalized Scleronomic Constraints

Given some vector of linearly independent generalized coordinates q in a configuration space [Sha01], an implicit inequality constraint (also called a ”one-sided” constraint, [Sha05]) can be specified as:

$$\begin{aligned} f : \mathbb{R}^n \to \mathbb{R},\ a \in \mathbb{R} \\ C(q) : f(q) \geq a \end{aligned}$$

Similarly, an equality constraint (two-sided) can be specified as:

$$C(q) : f(q) = a$$

Where a is a scalar. Both the above cases have only positional dependencies and an implicit time dependence, making them scleronomic constraints [TM04, p. 238]. In both cases, to obtain the corresponding necessary velocity constraint, a time derivative of both sides is computed:]

$$\dot{C}(q) : \frac{d}{dt} f(q) \geq \frac{d}{dt} a$$

Then using the chain rule, and the fact that the right-hand side is constant:

$$\dot{C}(q) : \frac{df(q)}{dq} \frac{dq}{dt} \geq 0$$

Which is equivalent to:

$$\dot{C}(q) : \nabla f \cdot \frac{dq}{dt} \geq 0$$

Which can be further generalized to:

$$\dot{C}(q) : (\nabla f)^T \frac{dq}{dt} \geq 0$$

For vector valued constraints and systems with multiple scalar valued constraints, the initial constraint equation becomes:

$$\begin{aligned} F : \mathbb{R}^n \to \mathbb{R}^m,\ a \in \mathbb{R}^m \\ C(q) : F(q) \geq a \end{aligned}$$

This can be solved the same way as the scalar field equation by replacing $(\nabla f)^T$ with the Jacobian matrix J_F:

$$\dot{C}(q) : J_F \frac{dq}{dt} \geq 0$$

In the case of the plane constraint, the normal vector is simply the transpose of the Jacobian matrix [Cat09].

2.5 Lagrange's Equations For Constrained Dynamics

Lagranges equations for systems containing holonomic (time-dependent positional) equality constraints [TM04, p. 249]:

$$f_k(q_j, t) = 0$$
$$\mathscr{L} = T - U$$

Where $\mathscr{L}$ is the Lagrangian, consisting of the difference in kinetic energy T and potential energy U, and each f_k is a holonomic constraint equation. The equation of motion is then written as:

$$\frac{\partial \mathscr{L}}{\partial q_i} - \frac{d}{dt}\frac{\partial \mathscr{L}}{\partial \dot{q}_i} + \sum_k \lambda_k(t)\frac{df_k}{dq_j} = 0$$

Which turns into the vector equation:

$$-\nabla U - M\ddot{q} + \sum_k \lambda_k(t)\frac{df_k}{dq} = 0$$

Where the first two terms should sum to a constant under the same assumptions used by virtual work (conservative forces). Using the same technique as used for the velocity constraint equations, the equations of motion for the system can be expressed using the Jacobians of the constraint functions. First substitute in the generalized force Q, then re-write the last term using the gradient of the constraint function:

$$Q - M\ddot{q} + \sum_k \lambda_k(t)\nabla f_k = 0$$

Remove the sum by changing the last term to a matrix product:

$$Q - M\ddot{q} + J_f^T \lambda = 0$$

Then re-arrange the equation to make it explicit:

$$M\ddot{q} = Q + J_f^T \lambda$$

It can be seen that this equation is simply D'Alembert's equation in generalized coordinates. Finally, if the constraint does no work (as in the principle of virtual work), and therefore its instantaneous power vanishes:

$$F^{(c)} \cdot v = (J_f^T \lambda)^T \dot{q} = \lambda^T J_f \dot{q} = 0$$

So to simulate conservative geometric constraints, the following equations can be solved using a suitable numerical method:

$$M\ddot{q} = Q + J_f^T \lambda J_f \dot{q} = \zeta$$

Where ζ is some positional error term compensating for the fact that the constraints only correct velocities [Cat05]:

$$\zeta = -\beta C$$

Where C is the current spatial deviation from the constraint. This is Baumgartes stabalization scheme for non-holonomic constraints [BL07]:

$$\begin{aligned} J_f\dot{q} + \beta C &= 0 \\ \dot{C} + \beta C &= 0 \end{aligned}$$

Which when integrated causes the error (the value of C) to decay exponentially.

2.6 LCP Formulation

For inequality constraints, the signed magnitude of the constraint force is subject to minimum and maximum bounds:

$$\lambda^- \leq \lambda \leq \lambda^+$$

When $\lambda^- = 0$ and $\lambda^+ = \infty$ (as in the case of contact forces), the constraint can be stated as the following Linear Complementarity Problem (LCP) [AP97, eq. 10], taking into account the Baumgarte term:

$$J\dot{q} \geq \zeta \;\; \text{compl. to} \;\; \lambda \geq 0$$

For non-trivial bounds on the Lagrangian multiplier (such as the bounds for friction constraints), the problem becomes a Mixed Linear Complementarity Problem (MLCP).

For example, a contact constraint should never push an object through the ground plane (see Generalized Sclerenomic Constraints). The generalized acceleration vector for an object can be approximated using Euler integration:

$$\ddot{q} \approx \frac{\dot{q}_2 - \dot{q}_1}{\Delta t}$$

Which leads to the following equation [Cat05] for equality constraints:

$$\begin{aligned} A &= JM^{-1}1J^T \\ b &= \frac{1}{\Delta t}\zeta - J\left(M^{-1}Q + \frac{1}{\Delta t}\dot{q}_1\right) \\ A\lambda &= b \end{aligned}$$

And the equivalent MLCP problem for general inequality constraints [Erl05, p. 53]:

$$\begin{aligned} w &= A\lambda - b \\ \lambda^- \leq \lambda &\leq \lambda^+ \end{aligned}$$

Where for each element of λ and the equation itself, one of the three conditions holds:

$$\begin{aligned} \lambda_i = \lambda^- \quad &, \quad w_i \geq 0 \\ \lambda_i = \lambda^+ \quad &, \quad w_i \leq 0 \\ \lambda^- < \lambda < \lambda^+ \quad &, \quad w_i = 0 \end{aligned}$$

2.7 Contact Caching

To create a more numerically stable simulation, contacts are cached after they are solved. Using an implicit equispatial KD-tree in the model-space of each rigid body, previous multiplier values can be stored and retrieved. For each point in model-space, KD code can be calculated as a 32-bit int using the following algorithm:

```
int GetKDCode(AABB box, Vector point)
  int code = 0
  Vector N[3] = { Vector(1, 0, 0), Vector(0, 1, 0), Vector(0, 0, 1) }
  Vector C(box.center)

  for(int bit = 0; bit < 32; bit++)
    float div = float(1 << (bit / 3))
    if(N[bit % 3] * (point - C) >= 0.0f)
      code = code | (1 << bit)
      C[bit % 3] += box.extents[bit % 3] / div;
    else
      C[bit % 3] -= box.extents[bit % 3] / div;
    end if
  end for

  return code;
end
```

For each of 32 planes, the test point is classified as being in the positive or negative halfspace. Depending on the result at each point, the current bit of the KD code is either a 0 or a 1, and the center point (the point through which the test plane passes) is updated. The result corresponds to a discrete element within the objects model-space. This result is non-unique, but since contacts are being cached (and all objects are assumed to be convex), adjacent force multipliers will be similar (continuity).

The smallest representable volume can be given in terms of the number of bits. First, calculate how many times the d-th dimension divides (where b is the total number of bits and d is a number from 1 to 3):

$$f(b,d) = \sum_{i=0}^{b-1} \delta_{i \ mod \ 3, \ d-1}$$

Where δ is the Kronecker delta and f accumulates 1 every time $imod3 = d-1$. Since the dimensions are cut in half each time this condition is true, for the d^{th} dimension originally of length L_d, the new length is:

$$L_d^{'} = \frac{L_d}{2^{f(b,d)}}$$

Which makes the total volume in three dimensions:

$$\begin{aligned} V^{'} &= \prod_{d=1}^{3} L_d^{'} \\ &= \prod_{d=1}^{3} \frac{L_d}{2^{f(b,d)}} \\ &= V \prod_{d=1}^{3} \frac{1}{2^{f(b,d)}} \\ &= 2^{(-\sum_{i=0}^{b-1} f(b,d))} V \\ &= 2^{-b} V \end{aligned}$$

Since:

$$\begin{aligned} \sum_{d=1}^{3} f(b,d) &= \sum_{d=1}^{3} \sum_{i=0}^{b-1} \delta_{i \ mod \ 3, \ d-1} \\ &= \sum_{i=0}^{b-1} \sum_{d=1}^{3} \delta_{i \ mod \ 3, \ d-1} \\ &= \sum_{i=0}^{b-1} 1 \\ &= b \end{aligned}$$

For 32 bits, the dimensions of the region are:

$$L_1^{'} \times L_2^{'} \times L_3^{'} = \frac{L_1}{2^{11}} \times \frac{L_2}{2^{11}} \times \frac{L_3}{2^{10}}$$

Each cached constraint is indexed by a key containing four values: the index of object A, the KD code for object As contact point, the index of object B, and the KD code for object Bs contact point. Each entry consists of the last calculated λ value and an age value. The age is incremented at the end of the solver update, and set to 0 when a new value is calculated. Then, all cached contacts older than some "stale" value (in this case, 10 iterations) are removed. This prevents the contact cache from growing too large as well as deleting potentially useless values (for example, if an object bounces, then makes contact again 15 frames later).

2.8 Inertia and Mass Calculation

Since all objects in the simulation are convex, mass calculation is relatively straightforward, and done using the fact that the object can be decomposed into pyramidal sections and the geometric barycenter is inside the objects hull:

$$\begin{aligned} O &= \frac{1}{N}\sum_j X_j \\ A_i &= \frac{1}{2}\|(Q_i - P_i) \times (R_i - P_i)\| \\ h_i &= \|\frac{1}{3}(P_i + Q_i + R_i) - O\| \\ m &= \rho \sum_i \frac{1}{3} A_i h_i \end{aligned}$$

Where ρ is the constant density and the height h_i of each right tetrahedron is the distance from the centroids base to the geometric barycenter of the object. Note that since the mass is distributed evenly the geometric barycenter is also the center of mass.

The moment of inertia is calculated using the standard inertial tensor formulation [TM04, p. 417] with the mass moved outside because of the assumed uniform mass:

$$I_{ij} = m \sum_\alpha \left(\delta_{ij} \sum_k x_{\alpha,k}^2 - x_{\alpha,i} x_{\alpha,j} \right)$$

Where $\delta_{ij} = 1$ when $i = j$ (0 otherwise), indexes each particle in the system, and x is relative to the center of mass.

To make inversion of the inertial tensor efficient, it must be in a diagonal form. There are two commonly used approaches: to diagonalize the inertial tensor by finding the eigenvalues, or to calculate the inertia in a coordinate frame approximately aligned with the inertial reference frame and simply discard the off-diagonal elements (which has been observed to simulate sufficiently regular objects realistically). This implementation uses the second approach, as real-time recalculation is desired for object fragments after fracturing occurs.

3 Plastic Fracturing

3.1 Convexity Limitations

For the purpose of this simulation, all bodies considered are convex. In the case of a simulation where this is not the case, the mesh may be embedded in higher-resolution objects with different topologies, similar to [CGC$^+$02] with the difference being that the embedding mesh is a convex hull, not a control lattice.

3.2 Algorithm Overview

The splitting algorithm can be broken into the following stages:

- Split Stage – The mesh is split along a plane using standard triangle clipping.
- Repair Stage – The holes on the two resulting meshes are capped.
- Build Objects Stage – Determine which meshes form valid objects, generate physics/collision components, and add the objects to the simulation.

3.3 Mesh Format

All meshes are stored in a simple indexed format with per-vertex attributes. The basic structures are:

```
Mesh
  Array<Triangle> triangles
  Array<Vertex> verts

Triangle
  int index[3] // indexes into the verts array

Vertex
  Vector position
  Vector normal
  Vector color
```

3.4 Splitting Stage

For all fractures considered, the problem is reduced to clipping a convex object against a plane. This typically employs standard triangle clipping, as described by [Eri05].

All triangles are passed through the clipping algorithm, resulting in two meshes being created. These meshes are referred to as the front mesh (in the positive half-space of the clipping plane) and the back mesh (in the negative half-space). To avoid vertex aliasing (multiple vertices with the same location), connectivity information is re-constructed for both meshes during the clipping phase.

3.4.1 Connectivity Table Method

To build the mesh, a table is used to look up (or create) the vertex index using the index from the old mesh. Two cases are handled: original mesh vertices and newly-created vertices resulting from a triangle being clipped against a plane (i-verts).

For regular vertices, the original vertex array (M, corresponding to the old mesh) is mapped to the vertex arrays (A and B, corresponding to the new meshes) through two tables. For i-verts, an undirected edge is constructed from the edge being clipped (the vertex indices are swapped if necessary to make the first index the smaller of the two) and used as a key to store/retrieve the indices for mesh A and mesh B.

```
// If V (at index Mv) is being inserted into a single mesh
// V is the old vert
// Mv is the index of the old vert
// A is the new vertex array
// MtoA is the map between vert indices
int InsertRegularVert(Vertex V, int Mv, Array<Vertex>& A,
                      Map<int, int>& MtoA)
 int Av
 if(Mv in keys of MtoA)
   Av = MtoA[Mv]
 else
   Av = A.Size()
   A.Add(V)
   MtoA[Mv] = Av
 end if
 return Av
end

// If V (newly-created) is to be inserted into A and B
// V is the new vert
// A and B are the new vertex arrays
// e is the undirected edge from the edge being clipped
// edgeToIndex maps undirected edges to indices for A and B
(int, int) InsertNewVert(Vertex V, Array<Vertex>& A,
                         Array<Vertex>& B, Edge e,
                         Map<Edge, (int, int)>& edgeToIndex)
 int Av, Bv
 if(e in keys of edgeToIndex)
   Av = edgeToIndex[e].first
   Bv = edgeToIndex[e].second
 else
   Av = A.Size()
   Bv = B.Size()
   A.Add(V)
   B.Add(V)
   edgeToIndex[e] = (Av, Bv)
 end if
 return (Av, Bv)
end
```

3.5 Repair Stage

In the repair stage, the meshes resulting from the split stage are capped with a triangle fan. If all the vertices classified as inside the clipping plane are recorded during the splitting stage, determining the fan becomes much more efficient.

3.5.1 Hole-Finding Algorithm

The hole-finding algorithm takes a mesh and returns a set of boundaries, which are in the form of a list of counterclockwise-ordered vertex indices. Each boundary is assumed to be closed (cyclic). The algorithm presented finds a list of borders using the simple fact that for any edge in a closed, non-degenerate mesh, there will be two neighboring triangles. In its simplest form (no limitations on vertices considered):

```
Array<Boundary> FindHoleBorders(Mesh mesh)
  Map<int, Array<int>> vertsToVerts = BuildVertsToVerts(mesh)
  Array<Edge> directedEdges = BuildDirectedEdges(mesh, vertsToVerts)
  Array<Boundary> boundaryList = CreateBoundaryList(directedEdges)
  return boundaryList
end
```

The first step builds the vertsToVerts map, which is an efficient representation of the meshs graph. Note that a Dictionary is not required for the vertsToVerts map in this simple case; it is used here to remain consistent with the vertex-limited form of the algorithm.

```
Map<int, Array<int>> BuildVertsToVerts(Mesh mesh)
  foreach(Triangle t in mesh)
    foreach(Edge (a, b) in t)
      vertsToVerts[a].Add(b)
    end for
  end for
end
```

After the vertsToVerts map is built, a list of missing directed edges is built. For each known edge, if the opposite edge does not exist, add it to the list.

```
Array<Edge> BuildDirectedEdges(Mesh mesh,
              Map<int, Array<int>> vertsToVerts)
  Array<Edge> directedEdges

  foreach(Edge edge in each tree in vertsToVerts)
    if(vertsToVerts does not contain the opposite edge)
      Add the opposite edge to directedEdges
  end for

 return directedEdges
end
```

Finally, the directed edges are turned into a list of closed boundaries. For non-degenerate convex meshes, dividing the mesh against a splitting plane will always produce non-degenerate boundaries.

```
Array<Boundary> CreateBoundaryList(Array<Edge> directedEdges)
  // Set to true for each closed boundary
  Array<bool> isClosed

  foreach(Edge edge in directedEdges)
    int frontMatch = −1
    int backMatch = −1

    for(int i = 0; i < boundaryList.Size(); i++)
      // Do not append/prepend to closed boundaries
      if(!isClosed[j])
        if(boundaryList[i].First() == edge.first)
          frontMatch = i
        if(boundaryList[i].Last() == edge.second)
          backMatch = i
      end if
    end for

    AddEdgeToBoundaries(frontMatch, backMatch, boundaryList,
                        isClosed, edge)
  end for

 // For valid boundaries to be formed, all boundaries must
 // be marked closed at this point
end
```

AddEdgeToBoundaries is the merge step, creating and joining boundaries based on index matches. The flow of the function is based on the connectivity between the current edge and the boundaries:

```
void AddEdgeToBoundaries(int frontMatch,
                         int backMatch,
                         Array<Boundary>& boundaryList,
                         Array<bool>& isClosed,
                         Edge edge)
 if(there were matches for both front and back)
   if(they are unique)
     The specified edge joins to existing boundaries,
     Merge those boundaries and the edge
   else if(they are not unique)
     The specified edge closes an existing boundary,
     Mark the boundary as closed
   end if
 else if(there is only one match)
   Append the edge to the matching boundary
 else there are no matches
   Create a new boundary containing only the edge
 end if
end
```

A simple optimization to the algorithm is limiting the vertices considered when building the vertToVerts map.

For each mesh, after the ordered boundaries are generated, a triangle fan is created from the boundary. The resulting meshes are convex and closed.

3.6 Build Objects Stage

To maintain a numerically stable simulation, only non-degenerate meshes (meshes with four or more vertices) are kept. Other criteria can be used to determine what constitutes a valid mesh, such as culling small volumes/etc.

Once the resulting meshes have been determined, game objects can be constructed. Physics data and collision components are generated using the methods covered in this paper. The resulting objects are translated to match the pre-split object, then new vertex normals are generated for lighting.

References

[AP97] M. Anitescu and F.A. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementary problems. 1997.

[BL07] O. Bauchau and A. Laulusa. Review of contemporary approaches for constraint enforcement in multibody systems. page 3, 2007.

[Cat05] E. Catto. Iterative dynamics with temporal coherence. June 5, 2005.

[Cat09] E. Catto. Modelling and solving constraints. page 13, 2009.

[CGC$^+$02] S. Capell, S. Green, B. Curless, T. Duchamp, and Z. Popovic. Interactive skeleton-driven dynamic deformations. page 1, 2002.

[Eri05] C. Ericson. Real time collision detection. pages 367–373, 2005.

[Erl05] K. Erleben. Stable, robust, and versatile multibody dynamics animation. April 2005.

[Gol02] H. Goldstein. Classical mechanics, 3rd ed. pages 17–19, 2002.

[Lan70] C. Lanczos. The variational principles of mechanics. page 120, 1970.

[Mil04] A. Miller. A cracking algorithm for exploding objects. May 2004.

[Sha01] A. A. Shabana. Computational dynamics, 2nd ed. page 133, 2001.

[Sha05] A. A. Shabana. Dynamics of multibody systems, 3rd ed. pages 91–92, 2005.

[TM04] S. Thornton and J. Marion. Classical dynamics of particles and systems, 5th ed. 2004.