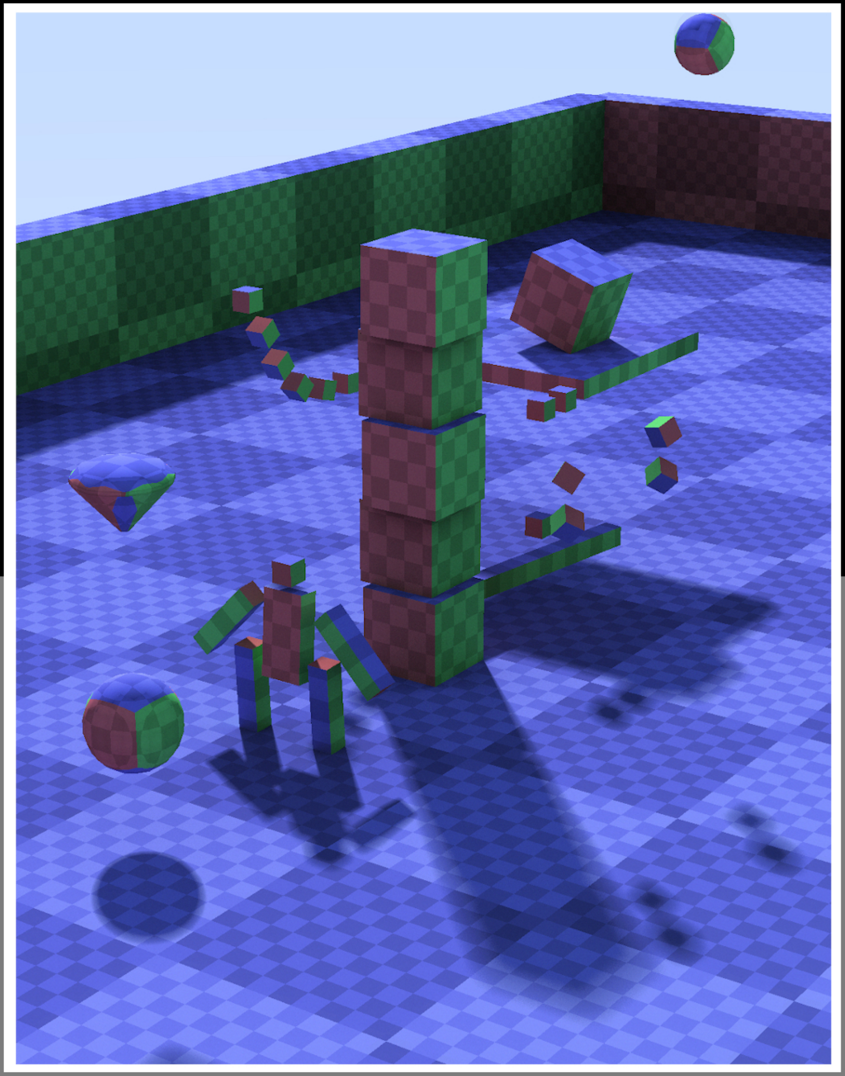
# Game Physics

The Rest Of Your Life



Gregory Hodges

# Game Physics: The Rest of Your Life

Gregory Hodges
Version 1.08

## Contents

## 1 Overview

This is the last planned book in the series. And it is meant to cover the last core features required for typical rigid body physics systems.

So far we've covered ballistic impulse response, continuous collision detection, and GJK. However, we do not have a means of doing ragdolls, nor stable stacking. In order to handle these final tasks, we're going to use constraints.

This book will either be the most involved or the easiest book depending on your background. If you've studied Lagrangian mechanics, then this is likely to be the simplest book in the series for you. However, even if you haven't studied this topic before, I assure you the math is straightforward. It may just take a little longer to properly digest.

Before we can get into constraints though. I'd like to give an overview of the VecN and MatN class that we'll need for solving algebraic systems of equations. They'll be necessary for solving constraints, and I'd rather get these utilities out of the way now, so that we don't have to derail ourselves by interrupting the explanation of constraints.

A quick note on notation. In this book you'll occassionally see a little dot above a variable, like so $\dot{C}$. This should be interpreted as a derivative with respect to time. So it's no different than writing $\frac{dC}{dt}$. It's just a common shorthand, nothing more. Likewise, seeing two dots over a variable, $\ddot{C}$, is shorthand for the second time derivative, $\frac{d^2C}{dt^2}$.

$$\dot{C} \equiv \frac{dC}{dt}$$
$$\ddot{C} \equiv \frac{d^2C}{dt^2}$$

## 2 VecN class

The VecN class is very similar to the Vec3 class. But since it's meant to handle an N-tuple instead of a 3-tuple, we will allocate the memory dynamically:

```
class VecN {
public:
  VecN() : m_num( 0 ), m_data( NULL ) {}
  VecN( int N );
  VecN( const VecN & rhs );
  VecN & operator = ( const VecN & rhs );
  ~VecN() { delete[] m_data; }

  float operator[] ( const int idx ) const { return m_data[ idx ]; }
  float & operator[] ( const int idx ) { return m_data[ idx ]; }
  const VecN & operator *= ( float rhs );
  VecN operator * ( float rhs ) const;
  VecN operator + ( const VecN & rhs ) const;
  VecN operator - ( const VecN & rhs ) const;
  const VecN & operator -= ( const VecN & rhs );

  float Dot( const VecN & rhs ) const;
  void Zero();

public:
  int    m_num;
  float * m_data;
};

inline VecN::VecN( int N ) {
  m_num = N;
  m_data = new float[ N ];
}

inline VecN::VecN( const VecN & rhs ) {
  m_num = rhs.m_num;
  m_data = new float[ m_num ];
  for ( int i = 0; i < m_num; i++ ) {
    m_data[ i ] = rhs.m_data[ i ];
  }
}

inline VecN & VecN::operator = ( const VecN & rhs ) {
  delete[] m_data;

  m_num = rhs.m_num;
  m_data = new float[ m_num ];
  for ( int i = 0; i < m_num; i++ ) {
    m_data[ i ] = rhs.m_data[ i ];
  }
  return *this;
}

inline const VecN & VecN::operator *= ( float rhs ) {
  for ( int i = 0; i < m_num; i++ ) {
    m_data[ i ] *= rhs;
  }
  return *this;
}

inline VecN VecN::operator * ( float rhs ) const {
  VecN tmp = *this;
  tmp *= rhs;
  return tmp;
}

inline VecN VecN::operator + ( const VecN & rhs ) const {
  VecN tmp = *this;
  for ( int i = 0; i < m_num; i++ ) {
    tmp.m_data[ i ] += rhs.m_data[ i ];
  }
  return tmp;
}

inline VecN VecN::operator - ( const VecN & rhs ) const {
  VecN tmp = *this;
  for ( int i = 0; i < m_num; i++ ) {
    tmp.m_data[ i ] -= rhs.m_data[ i ];
  }
  return tmp;
}

inline const VecN & VecN::operator -= ( const VecN & rhs ) {
  for ( int i = 0; i < m_num; i++ ) {
    m_data[ i ] -= rhs.m_data[ i ];
  }
  return *this;
}

inline float VecN::Dot( const VecN & rhs ) const {
  float sum = 0;
  for ( int i = 0; i < m_num; i++ ) {
    sum += m_data[ i ] * rhs.m_data[ i ];
  }
  return sum;
}

inline void VecN::Zero() {
  for ( int i = 0; i < m_num; i++ ) {
    m_data[ i ] = 0.0f;
  }
}
```

## 3 MatN class

The MatN class is very similar to the Mat3 class. Only it's an NxN matrix instead of a 3x3 matrix:

```
class MatN {
public:
  MatN() : numDimensions( 0 ) {}
  MatN( int N );
  MatN( const MatN & rhs ) {
    *this = rhs;
  }
  MatN( const MatMN & rhs ) {
    *this = rhs;
  }
  ~MatN() { delete[] rows; }

  const MatN & operator = ( const MatN & rhs );
  const MatN & operator = ( const MatMN & rhs );

  void Identity();
  void Zero();
  void Transpose();

  void operator *= ( float rhs );
  VecN operator * ( const VecN & rhs );
  MatN operator * ( const MatN & rhs );

public:
  int    numDimensions;
  VecN *   rows;
};

inline MatN::MatN( int N ) {
  numDimensions = N;
  rows = new VecN[ N ];
  for ( int i = 0; i < N; i++ ) {
    rows[ i ] = VecN( N );
  }
}

inline const MatN & MatN::operator = ( const MatN & rhs ) {
  numDimensions = rhs.numDimensions;
  rows = new VecN[ numDimensions ];
  for ( int i = 0; i < numDimensions; i++ ) {
    rows[ i ] = rhs.rows[ i ];
  }
  return *this;
}

inline const MatN & MatN::operator = ( const MatMN & rhs ) {
  if ( rhs.M != rhs.N ) {
    return *this;
  }

  numDimensions = rhs.N;
  rows = new VecN[ numDimensions ];
  for ( int i = 0; i < numDimensions; i++ ) {
    rows[ i ] = rhs.rows[ i ];
  }
  return *this;
}

inline void MatN::Zero() {
  for ( int i = 0; i < numDimensions; i++ ) {
    rows[ i ].Zero();
  }
}

inline void MatN::Identity() {
  for ( int i = 0; i < numDimensions; i++ ) {
    rows[ i ].Zero();
    rows[ i ][ i ] = 1.0f;
  }
}

inline void MatN::Transpose() {
  MatN tmp( numDimensions );

  for ( int i = 0; i < numDimensions; i++ ) {
    for ( int j = 0; j < numDimensions; j++ ) {
      tmp.rows[ i ][ j ] = rows[ j ][ i ];
    }
  }

  *this = tmp;
}

inline void MatN::operator *= ( float rhs ) {
  for ( int i = 0; i < numDimensions; i++ ) {
    rows[ i ] *= rhs;
  }
}

inline VecN MatN::operator * ( const VecN & rhs ) {
  VecN tmp( numDimensions );

  for ( int i = 0; i < numDimensions; i++ ) {
    tmp[ i ] = rows[ i ].Dot( rhs );
  }

  return tmp;
}

inline MatN MatN::operator * ( const MatN & rhs ) {
  MatN tmp( numDimensions );
  tmp.Zero();

  for ( int i = 0; i < numDimensions; i++ ) {
    for ( int j = 0; j < numDimensions; j++ ) {
      tmp.rows[ i ][ j ] += rows[ i ][ j ] * rhs.rows[ j ][ i ];
    }
  }

  return tmp;
}
```

## 4 MatMN class

The MatMN class is a generalization of the MatN class. The MatN class is a square matrix. The MatrixMN class is not necessarily square:

```
class MatMN {
public:
  MatMN() : M( 0 ), N( 0 ) {}
  MatMN( int M, int N );
  MatMN( const MatMN & rhs ) {
    *this = rhs;
  }
  ~MatMN() { delete[] rows; }

  const MatMN & operator = ( const MatMN & rhs );
  const MatMN & operator *= ( float rhs );
  VecN operator * ( const VecN & rhs ) const;
  MatMN operator * ( const MatMN & rhs ) const;
  MatMN operator * ( const float rhs ) const;

  void Zero();
  MatMN Transpose() const;

public:
  int     M;   // M rows
  int     N;   // N columns
  VecN *  rows;
};

inline MatMN::MatMN( int _M, int _N ) {
  M = _M;
  N = _N;
  rows = new VecN[ M ];
  for ( int m = 0; m < M; m++ ) {
    rows[ m ] = VecN( N );
  }
}

inline const MatMN & MatMN::operator = ( const MatMN & rhs ) {
  M = rhs.M;
  N = rhs.N;
  rows = new VecN[ M ];
  for ( int m = 0; m < M; m++ ) {
    rows[ m ] = rhs.rows[ m ];
  }
  return *this;
}

inline const MatMN & MatMN::operator *= ( float rhs ) {
  for ( int m = 0; m < M; m++ ) {
    rows[ m ] *= rhs;
  }
  return *this;
}

inline VecN MatMN::operator * ( const VecN & rhs ) const {
  // Check that the incoming vector is of the correct dimension
  if ( rhs.N != N ) {
    return rhs;
  }

  VecN tmp( M );
  for ( int m = 0; m < M; m++ ) {
    tmp[ m ] = rhs.Dot( rows[ m ] );
  }
  return tmp;
}

inline MatMN MatMN::operator * ( const MatMN & rhs ) const {
  // Check that the incoming matrix of the correct dimension
  if ( rhs.M != N && rhs.N != M ) {
    return rhs;
  }

  MatMN tranposedRHS = rhs.Transpose();

  MatMN tmp( M, rhs.N );
  for ( int m = 0; m < M; m++ ) {
    for ( int n = 0; n < rhs.N; n++ ) {
      tmp.rows[ m ][ n ] = rows[ m ].Dot( tranposedRHS.rows[ n ] );
    }
  }
  return tmp;
}

inline MatMN MatMN::operator * ( const float rhs ) const {
  MatMN tmp = *this;
  for ( int m = 0; m < M; m++ ) {
    for ( int n = 0; n < N; n++ ) {
      tmp.rows[ m ][ n ] *= rhs;
    }
  }
  return tmp;
}

inline void MatMN::Zero() {
  for ( int m = 0; m < M; m++ ) {
    rows[ m ].Zero();
  }
}

inline MatMN MatMN::Transpose() const {
  MatMN tmp( N, M );
  for ( int m = 0; m < M; m++ ) {
    for ( int n = 0; n < N; n++ ) {
      tmp.rows[ n ][ m ] = rows[ m ][ n ];
    }
  }
  return tmp;
}
```

## 5 Linear Complimentary Problem

The linear complimentary problem (LCP) is a method to solve:

$$\mathbf{A} \cdot \vec{x} = \vec{b} \tag{1}$$

where **A** is a matrix and $\vec{x}$ and $\vec{b}$ are vectors, and $\vec{x}$ is our unknown.
And the function to solve this problem, is pretty simple:

```
VecN LCP_GaussSeidel( const MatN & A, const VecN & b ) {
  const int N = b.N;
  VecN x( N );
  x.Zero();

  for ( int iter = 0; iter < N; iter++ ) {
    for ( int i = 0; i < N; i++ ) {
      float dx = ( b[ i ] - A.rows[ i ].Dot( x ) ) / A.rows[ i ][ i ];
      if ( dx * 0.0f == dx * 0.0f ) {
        x[ i ] = x[ i ] + dx;
      }
    }
  }
  return x;
}
```

This particular implementation is known as the Gauss-Seidel method. Now, the only important thing that you need to know is that this is a special utiltiy function that we will use for solving constraints. However, if you'd like to know a little bit more, you can read the following bonus subsection.

### 5.1 Bonus: Comments on Gauss-Seidel

Something worth mentioning is that Gauss-Seidel does not solve all linear complimentary problems. It can only solve problems where the matrix is either symmetric positive definite or diagonally dominant. Now, what does that mean?

Well, symmetric positive definite just means that when you multiply the transpose of a vector by the matrix and the vector, the result is always greater than zero.

$$\vec{x}^T \mathbf{M} \vec{x} > 0 \tag{2}$$

And, a diagonally dominant matrix is one where the absolute value of the diagonal element is greater than the sum of the absolute values of the elements in the row.

$$|x_{ii}| \geq \sum_{j, j \neq i} |x_{ij}| \tag{3}$$

Let's look at a concrete example:

$$\mathbf{M} = \begin{pmatrix} 1 & 3 \\ 4 & 2 \end{pmatrix}$$

This matrix is neither definite positive, nor diagonally dominant. We can easily prove it too. Given a vector $\vec{v}$ such that

$$\vec{v} = \begin{pmatrix} -2 \\ 1 \end{pmatrix}$$

Then checking the requirement for positive definite we get

$$\vec{v}^T \mathbf{M} \vec{v} = -8$$

Which violates our requirement for positive definite. And therefore it is not a positive definite matrix.

What about diagonally dominant? Well, by inspection, it is easy to see that the diagonal elements are not greater, in absolute value, than the non-diagonal elements, and therefore it is not diagonally domninant.

So what happens when we try to use Gauss-Seidel to solve a problem involving such a matrix? Let's find out with some basic test code:

```
VecN LCP_GaussSeidelVerbose( const MatN & A, const VecN & b ) {
  const int N = b.N;
  VecN x( N );
  x.Zero();

  for ( int iter = 0; iter < 5; iter++ ) {
    printf( "iter%i: ", iter );

    for ( int i = 0; i < N; i++ ) {
      float dx = ( b[ i ] - A.rows[ i ].Dot( x ) ) / A.rows[ i ][ i ];
      if ( dx * 0.0f == dx * 0.0f ) {
        x[ i ] = x[ i ] + dx;
      }

      printf( "%.2f ", x[ i ] );
    }

    printf( "\n" );
  }
  return x;
}

void TestGaussSeidel() {
  MatN A( 2 );
  A.rows[ 0 ][ 0 ] = 1.0f;
  A.rows[ 0 ][ 1 ] = 3.0f;
  A.rows[ 1 ][ 0 ] = 4.0f;
  A.rows[ 1 ][ 1 ] = 2.0f;

  VecN b( 2 );
  b[ 0 ] = 6.0f;
  b[ 1 ] = 7.0f;

  LCP_GaussSeidelVerbose( A, b );
}
```

This produces the following output:

```
iter0: 6.00 -8.50
iter1: 31.50 -59.50
iter2: 184.50 -365.50
iter3: 1102.50 -2201.50
iter4: 6610.50 -13217.50
```

As you can see, the solver diverges and it cannot find the correct solution. Now, let's test it on a matrix that is diagonally dominant:

$$\mathbf{M} = \begin{pmatrix} 3 & 1 \\ 2 & 4 \end{pmatrix}$$

```
void TestGaussSeidel2() {
  MatN A( 2 );
  A.rows[ 0 ][ 0 ] = 3.0f;
  A.rows[ 0 ][ 1 ] = 1.0f;
  A.rows[ 1 ][ 0 ] = 2.0f;
  A.rows[ 1 ][ 1 ] = 4.0f;

  VecN b( 2 );
  b[ 0 ] = 6.0f;
  b[ 1 ] = 7.0f;

  LCP_GaussSeidelVerbose( A, b );
}
```

And this gives us the following output:

```
iter0: 2.00 0.75
iter1: 1.75 0.88
```

```
iter2: 1.71 0.90
iter3: 1.70 0.90
iter4: 1.70 0.90
```

Which, as we can see, coverges to the correct solution of ( 1.7, 0.9 ).

So, why are we using this? Because all the constraints that we need to solve will involve diagonally dominant matrices.

I'd also like to make a note that you can further optimize this method. For instance, if you happen to know what the correct solution should approximately be, then you could pass in a "guess" solution and iterate fewer times. This would be an optimization that is similar to what we will discuss in the chapter on warm starting.

# 6 Theory of Constraints

The next few subsections are only a discussion of the theory of constraints. We won't actually write any code until we've fully developed the math that we will need.

## 6.1 Bead on a wire

The traditional problem to introduce constraints is to consider a bead constrained to move along a wire. To make sure there's no confusion though, we do not care about the orientation of the bead on the wire. So, we are only concerned with the position of the bead.

To make this example even more concrete, let's imagine a wire that is a ring.

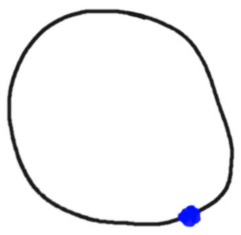


Figure 1: bead on wire

The question that we pose is how to calculate the "internal" forces that the wire applies to the bead, no matter the "external" forces applied.

Let's start getting more specific with how we define the constraint. The bead only has a position, $\vec{r}$, and is constrained to be a distance, $l$, from the origin:

$$x * x + y * y - l * l = 0 \tag{4}$$

This is what's called a holonomic constraint. Holonomic is a fancy pants name for a constraint that is dependent on position and time only, and not dependent on anything else such as velocity. A textbook might define holonomic as, any constraint that can be written in the form:

$$f(q_1, q_2, ..., q_n, t) = 0 \tag{5}$$

where the $q_n$'s are generalized coordinates.

So, re-writing our constraint function in vector form, we get:

$$C(r) = \vec{r} \cdot \vec{r} - l * l = 0 \tag{6}$$

This particular constraint is not dependent on time, therefore it's also known as scleronomous. If it were specifically dependent on time then it would be called rheonomorous. Alright, that's enough of a distraction with fancy pants names.

Let's see how this constraint changes over time. We can do this by taking the time derivative of the function:

$$\begin{aligned}\frac{dC}{dt} = \frac{d(\vec{r} \cdot \vec{r})}{dt} - \frac{d(l * l)}{dt} = 0 \\ = \frac{d\vec{r}}{dt} \cdot \vec{r} + \vec{r} \cdot \frac{d\vec{r}}{dt} \\ = 2 * \vec{r} \cdot \vec{v}\end{aligned}$$

$$\begin{aligned}\frac{d^2C}{dt^2} &= \frac{d(2*\vec{r}\cdot\vec{v})}{dt} = 0\\ &= 2*\frac{d\vec{r}}{dt}\cdot v + 2*\vec{r}\cdot\frac{d\vec{v}}{dt}\\ &= 2*\vec{v}\cdot\vec{v} + 2*\vec{r}\cdot\vec{a}\end{aligned}$$

Now, if the initial conditions of our bead (position and velocity) already satisfy the constraint conditions, then we can solve for the allowed acceleration of the bead. And also recall that:

$$\sum_i \vec{F_i} = m_i * \vec{a_i} \tag{7}$$

The sum of all forces is the total force. In this case, we have the internal forces of the constraint and external forces:

$$\begin{aligned}\vec{F}_{int} + \vec{F}_{ext} &= m*\vec{a}\\ \implies \vec{a} &= \frac{\vec{F}_{int} + \vec{F}_{ext}}{m}\end{aligned}$$

Then inserting this into the acceleration constraint equation we get:

$$\begin{aligned}0 &= \vec{v}\cdot\vec{v} + \vec{r}\cdot\frac{\vec{F}_{int} + \vec{F}_{ext}}{m}\\ \implies \vec{r}\cdot(\vec{F}_{int} + \vec{F}_{ext}) &= -m*\vec{v}\cdot\vec{v}\\ \implies \vec{F}_{int}\cdot\vec{r} + \vec{F}_{ext}\cdot\vec{r} &= -m*\vec{v}\cdot\vec{v}\\ \implies \vec{F}_{int}\cdot\vec{r} &= -m*\vec{v}\cdot\vec{v} - \vec{F}_{ext}\cdot\vec{r}\end{aligned}$$

Now, the next step is to invoke D'Alembert's principle. This principle states that forces of constraint do no work (they're conservative; the constraint forces do not add energy to the system). Another way of describing this is that the virtual work (work performed by the constraint) is zero. Note that work is defined as:

$$W = F*d$$

Where $d$ is the distance traveled and $F$ is the force applied in the direction of travel. Since the virtual work is zero, then at any given instant:

$$\vec{F}_{int}\cdot\vec{v} = 0$$

which means that the constraint force is always perpendicular to the allowed velocity.

Since we already have, from the velocity constraint, the condition that:

$$\vec{r}\cdot\vec{v} = 0$$

then this means:

$$\vec{F}_{int} = \lambda * \vec{r}$$

where lambda is a scalar. Plugging this back into the acceleration constraint equation we get:

$$\implies \lambda = -\frac{m*\vec{v}\cdot\vec{v} + \vec{F}_{ext}\cdot\vec{r}}{\vec{r}\cdot\vec{r}}$$

This is great. Now, we have a simple equation that let's us easily solve for the constraint force. And also since we derived this in vector notation, it naturally extends to 3D. But, in three dimensions, it's a distance constraint, not just a bead on a wire. So you can think of it as a particle trapped to the surface of a sphere.

## 6.2 Generalized Constraint Forces

We could go through this process for every single constraint we ever have to deal with in the future. But it would be nice if we could figure out some more abstract relationships that will allow us to solve constraints in some generic framework.

Let's start with the generic holonomic constraint equation:

$$C(q_1, q_2, ..., q_N) = 0 \tag{8}$$

Taking the derivative with respect to time gives:

$$\begin{aligned} \dot{C} &\equiv \frac{\partial C}{\partial q_i}\frac{\partial q_i}{\partial t} = 0 \\ \implies & \frac{dC}{dq} * v = 0 \end{aligned}$$

The first term in that equation is also known as the Jacobian matrix, therefore we define:

$$\begin{aligned} J &\equiv \frac{\partial C}{\partial q} \\ \implies & \dot{C} = J * v = 0 \end{aligned}$$

Taking the second time derivative yields:

$$\ddot{C} = \dot{J} * v + J * a = 0$$

Just as the $q_i$'s are introduced as the general coordinates, we will use $Q$ as the general forces. And since we're dealing with n-bodies, we will need more than just a single mass, and so we wish to introduce a mass matrix, $M$. This gives:

$$Q = M * a$$

and splitting the forces into the external and interal gives:

$$\begin{aligned} Q_{int} + Q_{ext} &= M * a \\ \implies & a = M^{-1} * (Q_{int} + Q_{ext}) \end{aligned}$$

And plugging this into the acceleration constraints:

$$\begin{aligned} \ddot{C} &= \dot{J} * v + J * M^{-1} * (Q_{int} + Q_{ext}) = 0 \\ \implies & J * M^{-1} * Q_{int} = -\dot{J} * v - J * M^{-1} * Q_{ext} \end{aligned}$$

Now, once again, invoking D'Alembert's principle of the virtual work being zero gives the condition:

$$Q_{int} * v = 0$$

and the condition from the velocity constraint implies:

$$Q_{int} = J^T * \lambda$$

It's important to note that lambda is now a vector quantity. And the reason we use lambda is because it's called a Lagrange multiplier, and lambda is just the traditional symbol used to represent it.

Now we can go ahead and solve for the lambda's:

$$J * M^{-1} * J^T * \lambda = -\dot{J} * v - J * M^{-1} * Q_{ext}$$
$$\implies \lambda = \frac{-\dot{J} * v - J * M^{-1} * Q_{ext}}{J * M^{-1} * J^T}$$

Wow, that wasn't so bad. In theory we now have all we need to make generalized constraints. Okay, good job everyone! We're done, right? Book over?

I mean, if a constraint can be described with a simple holonomic function, then we can use it to calculate the Jacobian. And once we have that, then it's seemingly straightforward to solve for the Lagrange multipliers. Right? So simple!

Well, as some say, the devil is in the details. So, we should probably start looking into how to turn our theory into a simulated reality.

## 6.3 Generalized Constraint Impulses

An important note we need to make here is that of impulses. The physics engine we've been developing thus far has been impulse based. So, we're not going to solve for constraint forces. Instead we'll be solving for constraint impulses.

Also, we're already going to be applying the external forces before we even bother to solve the constraint. Now, this is fine, because the information of the external force ends up in the velocity of the body. But this does simplify the equation of the lagrange multiplier to:

$$\lambda_{force} = \frac{-\dot{J} * v}{J * M^{-1} * J^T}$$

But this is still the lagrange multiplier for forces, not impulses. What are we to do about that? Well, if you recall, impulses and forces are related to each other by the equation:

$$J_{impulse} = \int F dt$$

Which means that we can solve for the impulse based lagrange multipliers with the following equation:

$$\lambda_{impulse} = \frac{-J * v}{J * M^{-1} * J^T}$$

## 6.4 Distance Constraint

Okay, so now we have a very simple equation that'll give us the impulse that a constraint would need to apply to a body, to make sure the constraint is not violated. All we need now is the Jacobian and the mass matrix. How do we go about finding those? Let's use the previously defined distance constraint as a concrete example.

Fortunately the mass matrix for a single particle is pretty simple and is defined as:

$$M = \begin{pmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix} \tag{9}$$

Now, we just need to figure out how to get the Jacobian. There's actually two ways of doing this. We could use the definition of the Jacobian:

$$J = \frac{\partial C}{\partial q} \tag{10}$$

Or, we could use the velocity constraint:

$$\dot{C} = J * v \tag{11}$$

The second method may not seem like it's easier. But it's deceptively effective. All we need to do is take the time derivative of the constraint equation and then factor out the velocities from all the other terms. What's left over is the Jacobian.

Let's examine the equation for the distance constraint:

$$C = \vec{r} \cdot \vec{r} - l * l = 0 \tag{12}$$

Taking the time derivative:

$$\begin{gathered} \implies r * \frac{dr}{dt} + \frac{dr}{dt} * r = 0 \\ \implies r * v + v * r = 0 \\ \implies 2 * r * v = 0 \end{gathered}$$

This implies that the Jacobian is:

$$J = 2 * r$$

Note: We could factor out the $2$, but that's up to you. I haven't bothered to do that. In the long run it doesn't matter, as long as we're consistent.

Believe it or not, that's really about all we need to discuss. Now we just have to go through the work of actually writing the code for the constraints that we'll need.

## 7 Base Constraint Class

For the rest of this book, all the constraints that we will be dealing with will contain exactly two bodies. And most of these constraints will have more in common than not. So, we will want a base constraint class.

```
class Constraint {
public:
  virtual void PreSolve( const float dt_sec ) {}
  virtual void Solve() {}
  virtual void PostSolve() {}

protected:
  MatMN GetInverseMassMatrix() const;
  VecN GetVelocities() const;
  void ApplyImpulses( const VecN & impulses );

public:
  Body * m_bodyA;
  Body * m_bodyB;

  Vec3 m_anchorA;   // The anchor location in bodyA's space
  Vec3 m_axisA;   // The axis direction in bodyA's space

  Vec3 m_anchorB;   // The anchor location in bodyB's space
  Vec3 m_axisB;   // The axis direction in bodyB's space
};
```

The first thing I want to discuss here are the common utility functions. Since our constraints will always be done between two bodies, it's convenient for us to have these common functions for building the mass matrix, acquiring the velocities, and applying the impulses we solve.

Also take note of the PreSolve, Solve, and PostSolve functions. We won't always use the PostSolve function, which is why it has a default definition that does nothing. The PreSolve function will be where we setup our jacobians. The Solve function is where we actually calculate and then apply the impulses that satisfy the constraint.

You might be asking why break the solver functions into phases like that? Well, it's going to turn out to be necessary for efficiently converging to the correct solution when multiple constraints are chaining many bodies together.

Now, let's have a closer look at what goes into the GetInverseMassMatrix function. We've already seen the mass matrix for a single point particle. But what would the mass matrix be for two point particles?

Well, as it turns out, we only need to add more rows and columns to the matrix. Like so:

$$\mathbf{M} = \begin{pmatrix} m_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & m_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & m_2 \end{pmatrix} \tag{13}$$

You might notice that there's a lot of zeros in this matrix now. This is commonly referred to as a sparse matrix. And since most of the arithmetic performed on the matrix won't actually do anything (since adding a 0 does nothing and multiplying by zero is always zero), then there's a lot of room for optimization by writing a specific sparse matrix class. However, we're not going to bother with that here. And as is commonly stated in other texts; I leave that as an exercise to the reader.

For a short hand way of writing these sparse diagonal matrices, we can also write it as:

$$\mathbf{M} = \begin{pmatrix} m_1\mathbf{I} \\ m_2\mathbf{I} \end{pmatrix} \tag{14}$$

where $\mathbf{I}$ is the identity matrix.

Now, it would be pretty boring if we were to only simulate point masses. What about extending this to include rigid bodies? Also, since we're now going to care about orientations, we'll need to add the inertia tensors to the mass matrix. Which we'll write in the sparse matrix notation as:

$$\mathbf{M} = \begin{pmatrix} m_1\mathbf{I} \\ \mathbf{I_1} \\ m_2\mathbf{I} \\ \mathbf{I_2} \end{pmatrix} \tag{15}$$

And finally, we can fill out that GetInverseMassMatrix function:

```
MatMN Constraint::GetInverseMassMatrix() const {
  MatMN invMassMatrix( 12, 12 );
  invMassMatrix.Zero();

  invMassMatrix.rows[ 0 ][ 0 ] = m_bodyA->m_invMass;
  invMassMatrix.rows[ 1 ][ 1 ] = m_bodyA->m_invMass;
  invMassMatrix.rows[ 2 ][ 2 ] = m_bodyA->m_invMass;

  Mat3 invInertiaA = m_bodyA->GetInverseInertiaTensorWorldSpace();
  for ( int i = 0; i < 3; i++ ) {
    invMassMatrix.rows[ 3 + i ][ 3 + 0 ] = invInertiaA.rows[ i ][ 0 ];
    invMassMatrix.rows[ 3 + i ][ 3 + 1 ] = invInertiaA.rows[ i ][ 1 ];
    invMassMatrix.rows[ 3 + i ][ 3 + 2 ] = invInertiaA.rows[ i ][ 2 ];
  }

  invMassMatrix.rows[ 6 ][ 6 ] = m_bodyB->m_invMass;
  invMassMatrix.rows[ 7 ][ 7 ] = m_bodyB->m_invMass;
  invMassMatrix.rows[ 8 ][ 8 ] = m_bodyB->m_invMass;

  Mat3 invInertiaB = m_bodyB->GetInverseInertiaTensorWorldSpace();
  for ( int i = 0; i < 3; i++ ) {
    invMassMatrix.rows[ 9 + i ][ 9 + 0 ] = invInertiaB.rows[ i ][ 0 ];
    invMassMatrix.rows[ 9 + i ][ 9 + 1 ] = invInertiaB.rows[ i ][ 1 ];
    invMassMatrix.rows[ 9 + i ][ 9 + 2 ] = invInertiaB.rows[ i ][ 2 ];
  }

  return invMassMatrix;
}
```

Now how about that GetVelocities function? We're now dealing with two bodies that have both linear and angular velocity:

$$\vec{v} = \begin{pmatrix} \vec{v}_1 \\ \vec{\omega}_1 \\ \vec{v}_2 \\ \vec{\omega}_2 \end{pmatrix} \tag{16}$$

And as you might expect, the code for that function will be:

```
VecN Constraint::GetVelocities() const {
  VecN q_dt( 12 );

  q_dt[ 0 ] = m_bodyA->m_linearVelocity.x;
  q_dt[ 1 ] = m_bodyA->m_linearVelocity.y;
  q_dt[ 2 ] = m_bodyA->m_linearVelocity.z;

  q_dt[ 3 ] = m_bodyA->m_angularVelocity.x;
  q_dt[ 4 ] = m_bodyA->m_angularVelocity.y;
  q_dt[ 5 ] = m_bodyA->m_angularVelocity.z;

  q_dt[ 6 ] = m_bodyB->m_linearVelocity.x;
  q_dt[ 7 ] = m_bodyB->m_linearVelocity.y;
  q_dt[ 8 ] = m_bodyB->m_linearVelocity.z;

  q_dt[ 9 ] = m_bodyB->m_angularVelocity.x;
  q_dt[ 10] = m_bodyB->m_angularVelocity.y;
  q_dt[ 11] = m_bodyB->m_angularVelocity.z;

  return q_dt;
}
```

And finally the ApplyImpulses is going to be a convenience function for applying the VecN impulses that we will find in the Solve function:

```
void Constraint::ApplyImpulses( const VecN & impulses ) {
  Vec3 forceInternalA( 0.0f );
  Vec3 torqueInternalA( 0.0f );
  Vec3 forceInternalB( 0.0f );
  Vec3 torqueInternalB( 0.0f );

  forceInternalA[ 0 ] = impulses[ 0 ];
  forceInternalA[ 1 ] = impulses[ 1 ];
  forceInternalA[ 2 ] = impulses[ 2 ];

  torqueInternalA[ 0 ] = impulses[ 3 ];
  torqueInternalA[ 1 ] = impulses[ 4 ];
  torqueInternalA[ 2 ] = impulses[ 5 ];

  forceInternalB[ 0 ] = impulses[ 6 ];
  forceInternalB[ 1 ] = impulses[ 7 ];
  forceInternalB[ 2 ] = impulses[ 8 ];

  torqueInternalB[ 0 ] = impulses[ 9 ];
  torqueInternalB[ 1 ] = impulses[ 10];
  torqueInternalB[ 2 ] = impulses[ 11];

  m_bodyA->ApplyImpulseLinear( forceInternalA );
  m_bodyA->ApplyImpulseAngular( torqueInternalA );

  m_bodyB->ApplyImpulseLinear( forceInternalB );
  m_bodyB->ApplyImpulseAngular( torqueInternalB );
}
```

That pretty much wraps up the base Constraint class. I hope its utility functions are clear to you. We're going to be using them for the rest of the book.

To clear any confusion that might be going on with this class, let's go ahead and immediately use it in a simple example.

## 8 Rigid Body Distance Constraint

Okay, so far the distance constraint we've been building has assumed the bodies are just point particles. Obviously, we'll need to extend it to handle rigid bodies and not just point particles, otherwise we won't be able to do anything interesting with it.

Also, since we're now going to care about orientations, we'll need to add the inertia tensors to the mass matrix. Which we'll write in the sparse matrix notation as:

$$\mathbf{M} = \begin{pmatrix} m_1\mathbf{I} \\ \mathbf{I_1} \\ m_2\mathbf{I} \\ \mathbf{I_2} \end{pmatrix} \tag{17}$$

Likewise, we'll now need to add the angular velocities to the velocity vector too:

$$\vec{v} = \begin{pmatrix} \vec{v}_1 \\ \vec{\omega}_1 \\ \vec{v}_2 \\ \vec{\omega}_2 \end{pmatrix} \tag{18}$$

And now the forces that we solve for will also contain torques as well as linear forces.

One last thing before actually writing out the code for this constraint. We should add anchor points. After all, if we want to do something such as a ragdoll later, then in order to simulate a joint properly, we'll need to know where the anchor point is relative to each body.

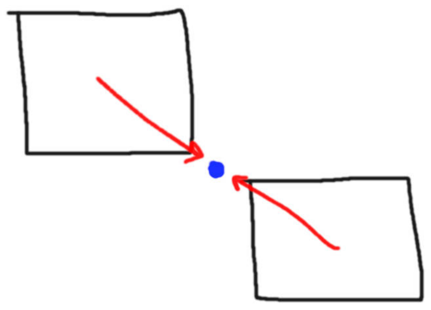


Figure 2: anchored bodies

The above image is what two anchored bodies look like when the constraint is satisfied. However, this is unlikely to be the case in our physics simulation, after all that's why we need to know how to solve for the corrective impulses. So, let's have a look at this situation when the two bodies are violating the constraint:

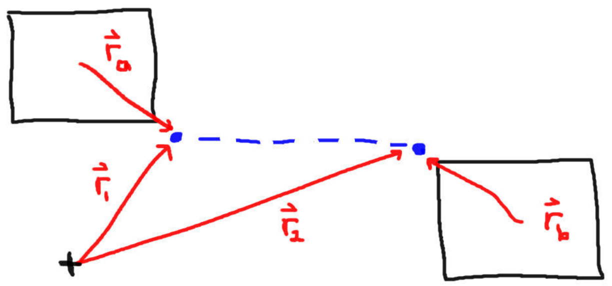


Figure 3: anchored bodies violated

Looking at this new figure, we can see that each body defines the anchor point relative to itself. This is because we don't really care where in the world the anchor is located, just that the two bodies are anchored together. So we have $\vec{r}_1$ which is the anchor point in world space, relative to bodyA, and we have $\vec{r}_2$, the anchor point in world space, relative to bodyB. And then $\vec{r}_a$ is the vector pointing from bodyA's center of mass to the anchor point, and $\vec{r}_b$ is the vector pointing from bodyB's center of mass to the anchor point.

Now, that we have a concept of anchor points. The Jacobian is going to look a little different. We'll just go ahead and re-derive it here:

$$\begin{aligned} C &= (\vec{r}_2 - \vec{r}_1)\cdot(\vec{r}_2 - \vec{r}_1) = 0 \\ &= \vec{r}_2\cdot\vec{r}_2 - \vec{r}_2\cdot\vec{r}_1 - \vec{r}_1\cdot\vec{r}_2 + \vec{r}_1\cdot\vec{r}_1 \\ \implies \dot{C} &= \frac{d\vec{r}_2}{dt}\cdot\vec{r}_2 + \vec{r}_2\cdot\frac{d\vec{r}_2}{dt} - \frac{d\vec{r}_2}{dt}\cdot\vec{r}_1 - \vec{r}_1\cdot\frac{d\vec{r}_2}{dt} \\ &+ \frac{d\vec{r}_1}{dt}\cdot\vec{r}_1 + \vec{r}_1\cdot\frac{d\vec{r}_1}{dt} - \frac{d\vec{r}_1}{dt}\cdot\vec{r}_2 - \vec{r}_2\cdot\frac{d\vec{r}_1}{dt} \\ &= 2*(\vec{r}_2 - \vec{r}_1)\cdot\frac{d\vec{r}_2}{dt} + 2*(\vec{r}_1 - \vec{r}_2)\cdot\frac{d\vec{r}_1}{dt} \end{aligned}$$

Now it's important to note that for an anchor point attached to a body, the time derivative of the point isn't just the linear velocity of the body, but also the rotation of the body:

$$\frac{d\vec{r}_1}{dt} = \vec{v}_1 + \vec{\omega}_1 \times \vec{r}_a \tag{19}$$

$$\frac{d\vec{r}_2}{dt} = \vec{v}_2 + \vec{\omega}_2 \times \vec{r}_b \tag{20}$$

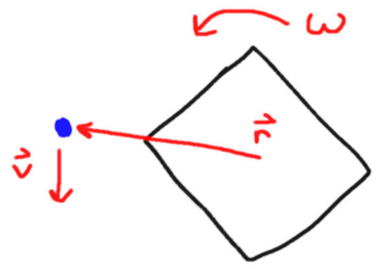


Figure 4: rotating anchor point

plugging this into the above velocity constraints gives:

$$\begin{aligned} \implies \dot{C} &= 2(\vec{r}_1 - \vec{r}_2)\cdot(\vec{v}_1 + \vec{\omega}_1 \times \vec{r}_a) + 2(\vec{r}_2 - \vec{r}_1)\cdot(\vec{v}_2 + \vec{\omega}_2 \times \vec{r}_b) \\ &= 2(\vec{r}_1 - \vec{r}_2)\cdot\vec{v}_1 + 2(\vec{r}_1 - \vec{r}_2)\cdot\vec{\omega}_1 \times \vec{r}_a + 2(\vec{r}_2 - \vec{r}_1)\cdot\vec{v}_2 + 2(\vec{r}_2 - \vec{r}_1)\cdot\vec{\omega}_2 \times \vec{r}_b \end{aligned}$$

Now using some vector identities we can reorder those cross products:

$$\vec{a}\cdot(\vec{b}\times\vec{c}) = \vec{b}\cdot(\vec{c}\times\vec{a}) = \vec{c}\cdot(\vec{a}\times\vec{b}) \tag{21}$$

$$\implies \dot{C} = 2(\vec{r}_1 - \vec{r}_2)\cdot\vec{v}_1 + \vec{\omega}_1\cdot\vec{r}_a \times 2(\vec{r}_1 - \vec{r}_2) + 2(\vec{r}_2 - \vec{r}_1)\cdot\vec{v}_2 + \vec{\omega}_2\cdot\vec{r}_b \times 2(\vec{r}_2 - \vec{r}_1)$$

And re-writing this in matrix form gives:

$$\implies \dot{C} = \begin{pmatrix} 2(\vec{r}_1 - \vec{r}_2), & 2\vec{r}_a \times (\vec{r}_1 - \vec{r}_2), & 2(\vec{r}_2 - \vec{r}_1), & 2\vec{r}_b \times (\vec{r}_2 - \vec{r}_1) \end{pmatrix} \begin{pmatrix} \vec{v}_1 \\ \vec{\omega}_1 \\ \vec{v}_2 \\ \vec{\omega}_2 \end{pmatrix}$$

So, by inspection, we can see that this gives us the following jacobian:

$$\implies \mathbf{J} = \big(2(\vec{r}_1 - \vec{r}_2), \quad 2\vec{r}_a \times (\vec{r}_1 - \vec{r}_2), \quad 2(\vec{r}_2 - \vec{r}_1), \quad 2\vec{r}_b \times (\vec{r}_2 - \vec{r}_1)\big) \tag{22}$$

Alright, with that, let's have a look at the code:

```
class ConstraintDistance : public Constraint {
public:
  ConstraintDistance() : Constraint(), m_Jacobian( 1, 12 ) {}

  void PreSolve( const float dt_sec ) override;
  void Solve() override;

private:
  MatMN m_Jacobian;
};

void ConstraintDistance::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 r = worldAnchorB - worldAnchorA;
  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;

  m_Jacobian.Zero();

  Vec3 J1 = ( a - b ) * 2.0f;
  m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

  Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
  m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

  Vec3 J3 = ( b - a ) * 2.0f;
  m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

  Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
  m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
  m_Jacobian.rows[ 0 ][ 10] = J4.y;
  m_Jacobian.rows[ 0 ][ 11] = J4.z;
}

void ConstraintDistance::Solve() {
  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities();
  const MatMN invMassMatrix = GetInverseMassMatrix();
  const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
  VecN rhs = m_Jacobian * q_dt * -1.0f;

  // Solve for the Lagrange multipliers
  const VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

  // Apply the impulses
  const VecN impulses = JacobianTranspose * lambdaN;
  ApplyImpulses( impulses );
}
```

Finally, all that just to make a simple constraint example. As can be seen, those helper functions from the base constraint class are not only useful, but they also make the code very clean.

You might be wondering what happened to Q_ext? Well, as it turns out, we're not going to change the update loop of the scene very much. So the bodies will continue to receive impulses from external sources (such as gravity). Those external impulses will translate to velocities on the bodies. And when we run our constraint solver, it'll calculate the exact impulses necessary to remove velocities that are violating the constraint. So, it turns out we don't need Q_ext.

And since we're talking about updates to the Scene class. Let's go ahead and get into those details.

The first thing we need to do is add a container to hold our constraints as a member of the Scene class:

```
std::vector< Constraint * > m_constraints;
```

Next we need to make sure the Scene::Update function is calling the Solve member functions of all constraints in the scene:

```
void Scene::Update( const float dt_sec ) {
  // Gravity impulse
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];
    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }

  //
  // Broadphase (build potential collision pairs)
  //
  std::vector< collisionPair_t > collisionPairs;
  BroadPhase( m_bodies.data(), (int)m_bodies.size(), collisionPairs, dt_sec );

  //
  //  NarrowPhase (perform actual collision detection)
  //
  int numContacts = 0;
  contact_t * contacts = (contact_t *)alloca( sizeof( contact_t ) * collisionPairs.size() );
  for ( int i = 0; i < collisionPairs.size(); i++ ) {
    const collisionPair_t & pair = collisionPairs[ i ];
    Body * bodyA = &m_bodies[ pair.a ];
    Body * bodyB = &m_bodies[ pair.b ];

    // Skip body pairs with infinite mass
    if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
      continue;
    }

    // Check for intersection
    contact_t contact;
    if ( Intersect( bodyA, bodyB, dt_sec, contact ) ) {
      contacts[ numContacts ] = contact;
      numContacts++;
    }
  }

  // Sort the times of impact from first to last
  if ( numContacts > 1 ) {
    qsort( contacts, numContacts, sizeof( contact_t ), CompareContacts );
  }

  //
  //  Solve Constraints
  //
  for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->PreSolve( dt_sec );
  }

  for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->Solve();
  }
  for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->PostSolve();
  }

  //
  // Apply ballistic impulses
  //
  float accumulatedTime = 0.0f;
  for ( int i = 0; i < numContacts; i++ ) {
    contact_t & contact = contacts[ i ];
    const float dt = contact.timeOfImpact - accumulatedTime;

    // Position update
    for ( int j = 0; j < m_bodies.size(); j++ ) {
      m_bodies[ j ].Update( dt );
    }

    ResolveContact( contact );
    accumulatedTime += dt;
  }

  // Update the positions for the rest of this frame's time
  const float timeRemaining = dt_sec - accumulatedTime;
  if ( timeRemaining > 0.0f ) {
    for ( int i = 0; i < m_bodies.size(); i++ ) {
      m_bodies[ i ].Update( timeRemaining );
    }
  }
}
```

That should look familiar, with the exception of the part where we loop over the constraints and solve them. You might be asking why do we have three separate loops? Wouldn't it be more efficient to just have a single loop? Well, we'll get to that.

Alright! Now, let's add some bodies and a constraint to see this thing in action!

```
void Scene::Initialize() {
  Body body;

  body.m_position = Vec3( 0.0f, 0.0f, 5.0f );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
  body.m_invMass = 0.0f;
  body.m_elasticity = 1.0f;
  m_bodies.push_back( body );
  Body * bodyA = &m_bodies[ m_bodies.size() - 1 ];

  body.m_position = Vec3( 1.0f, 0.0f, 5.0f );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
  body.m_invMass = 1.0f;
  body.m_elasticity = 1.0f;
  m_bodies.push_back( body );
  Body * bodyB = &m_bodies[ m_bodies.size() - 1 ];

  const Vec3 jointWorldSpaceAnchor = bodyA->m_position;


  ConstraintDistance * joint = new ConstraintDistance();

  joint->m_bodyA   = bodyA;
  joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

  joint->m_bodyB   = bodyB;
  joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
  m_constraints.push_back( joint );

  //
  //  Standard floor and walls
  //
  AddStandardSandBox( m_bodies );
}
```

Now, running this we should see one body dangling below another like a physical pendulum.

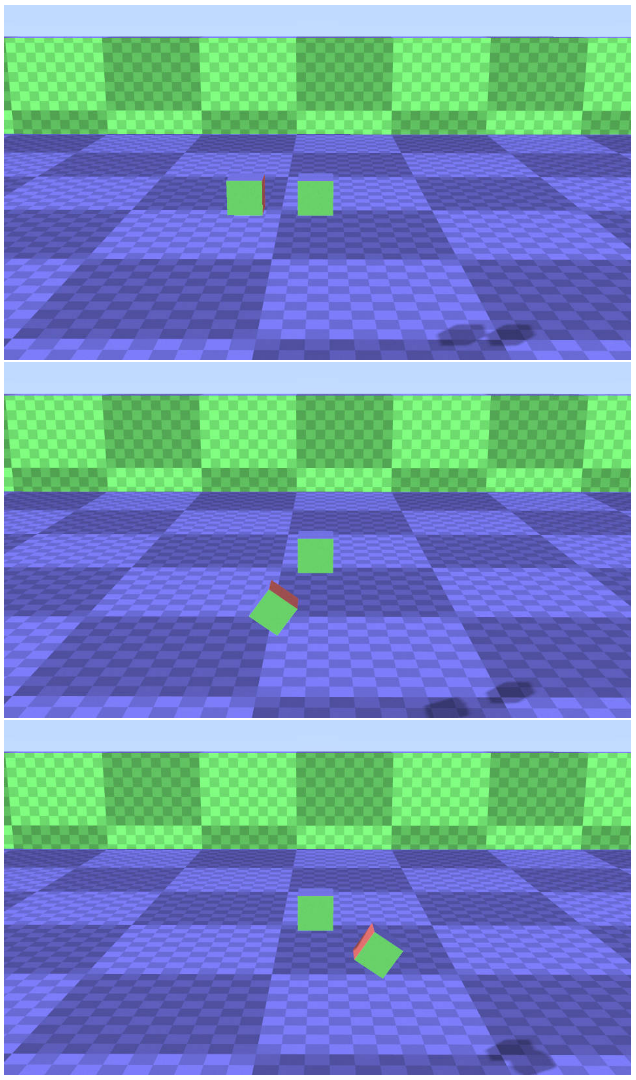


Figure 5: Physical Pendulum

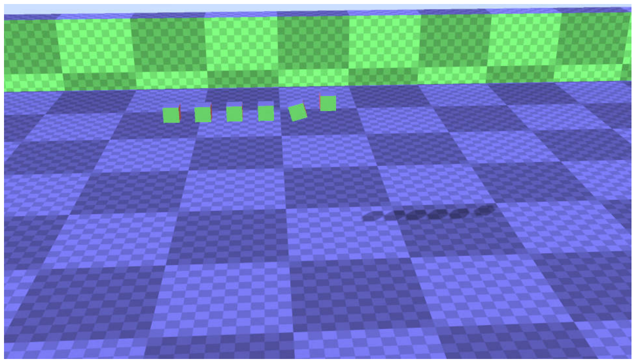
# 9 Chaining Constraints Together

Well, now that we have a basic distance constraint, we need to chain some together. Because why wouldn't you?

So, let's go ahead and modify that initialize function:

```
void Scene::Initialize() {
  Body body;

  //
  // Build a chain for funsies
  //
  const int numJoints = 5;
  for ( int i = 0; i < numJoints; i++ ) {
    if ( i == 0 ) {
      body.m_position = Vec3( 0.0f, 5.0f, (float)numJoints + 3.0f );
      body.m_orientation = Quat( 0, 0, 0, 1 );
      body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
      body.m_invMass = 0.0f;
      body.m_elasticity = 1.0f;
      m_bodies.push_back( body );
    } else {
      body.m_invMass = 1.0f;
    }

    body.m_linearVelocity = Vec3( 0, 0, 0 );

    Body * bodyA = &m_bodies[ m_bodies.size() - 1 ];
    const Vec3 jointWorldSpaceAnchor = bodyA->m_position;

    ConstraintDistance * joint = new ConstraintDistance();

    joint->m_bodyA  = &m_bodies[ m_bodies.size() - 1 ];
    joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

    body.m_position = joint->m_bodyA->m_position + Vec3( 1, 0, 0 );
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
    body.m_invMass = 1.0f;
    body.m_elasticity = 1.0f;
    m_bodies.push_back( body );

    joint->m_bodyB  = &m_bodies[ m_bodies.size() - 1 ];
    joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

    m_constraints.push_back( joint );
  }

  //
  //  Standard floor and walls
  //
  AddStandardSandBox( m_bodies );
}
```

Now let's run this thing. Hopefully it just works.



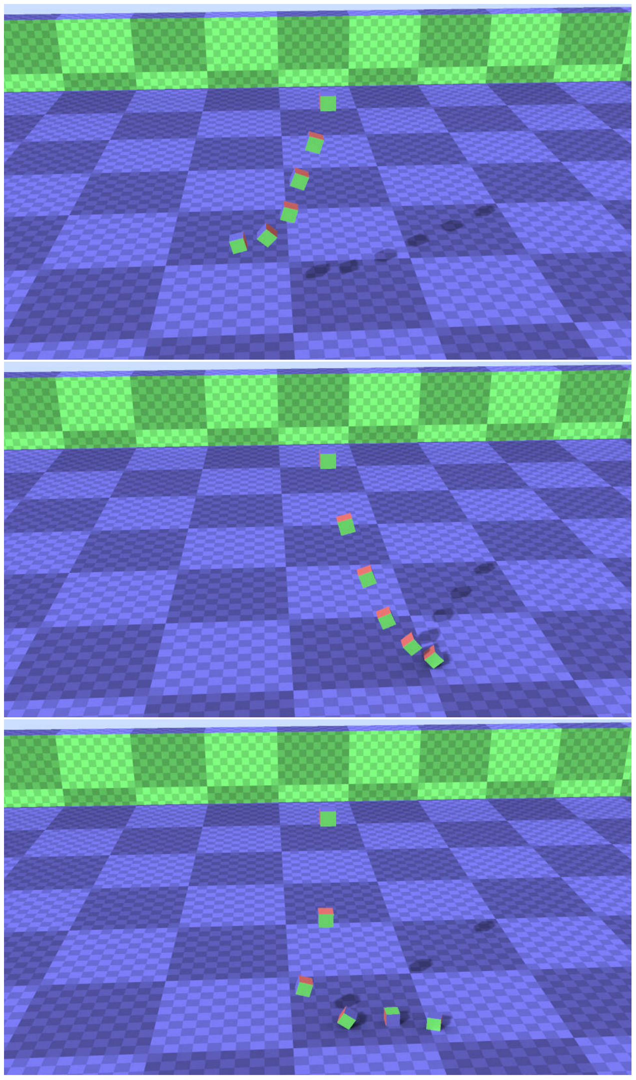


Figure 6: Physical Pendulum Chain

Okay, so it doesn't exactly work. It kinda works. What's going on here?

What's basically happening here is we have five bodies that are pulled to the ground by gravity. But, we're only solving constraints between two bodies at a time.

So, for the top constraint, we solve it and the second to top body is stopped. But then, the next constraint solves the velocities between the second and third body, so this pulls the body down.

What are we to do about this?

Well, we could take all our constraints and build one giant constraint equation and then solve all the constraints at once. This is known as a global solver. And it's not what we will be doing here.

Don't get me wrong. Using global solvers can do wonders. However, they require that we maintain a graph of constraint dependencies. Now, for a few bodies chained together, it's not a problem to maintain a graph. But it's something I'd rather avoid for this book.

Instead we're going to take an iterative approach. The basic idea behind this is to simply solve our constraints multiple times in a single frame.

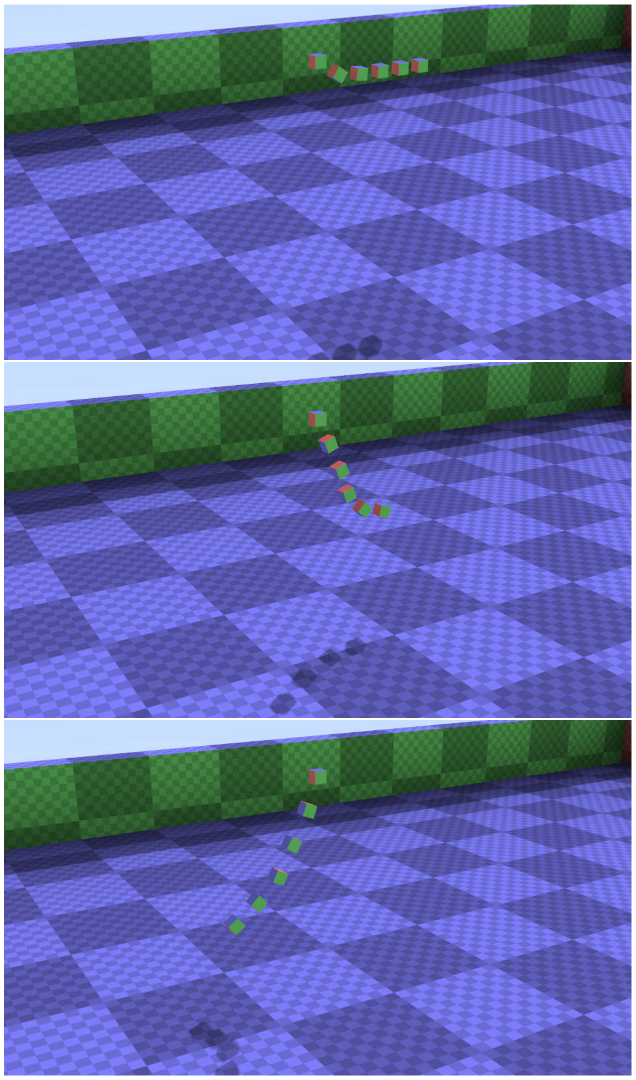
Remember how I mentioned that solving the second constraint invalidates the solution of the first constraint? Well, if we went back and re-solved the first constraint, it would be fixed. But then the second constraint is invalid. So you solve that again. And as you continue to do this, the constraints should slowly converge to the correct solution.

So let's go ahead and try this iterative approach:

```
void Scene::Update( const float dt_sec ) {
  // Gravity impulse
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];
    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }

  //
  // Broadphase (build potential collision pairs)
  //
  std::vector< collisionPair_t > collisionPairs;
  BroadPhase( m_bodies.data(), (int)m_bodies.size(), collisionPairs, dt_sec );

  //
  //  NarrowPhase (perform actual collision detection)
  //
  int numContacts = 0;
  contact_t * contacts = (contact_t *)alloca( sizeof( contact_t ) * collisionPairs.size() );
  for ( int i = 0; i < collisionPairs.size(); i++ ) {
    const collisionPair_t & pair = collisionPairs[ i ];
    Body * bodyA = &m_bodies[ pair.a ];
    Body * bodyB = &m_bodies[ pair.b ];

    // Skip body pairs with infinite mass
    if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
      continue;
    }

    // Check for intersection
    contact_t contact;
    if ( Intersect( bodyA, bodyB, dt_sec, contact ) ) {
      contacts[ numContacts ] = contact;
      numContacts++;
    }
  }

  // Sort the times of impact from first to last
  if ( numContacts > 1 ) {
    qsort( contacts, numContacts, sizeof( contact_t ), CompareContacts );
  }

  //
  //  Solve Constraints
  //
  for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->PreSolve( dt_sec );
  }

  const int maxIters = 50;
  for ( int iters = 0; iters < maxIters; iters++ ) {
    for ( int i = 0; i < m_constraints.size(); i++ ) {
      m_constraints[ i ]->Solve();
    }
  }

  for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->PostSolve();
  }

  //
  // Apply ballistic impulses
  //
  float accumulatedTime = 0.0f;
  for ( int i = 0; i < numContacts; i++ ) {
    contact_t & contact = contacts[ i ];
    const float dt = contact.timeOfImpact - accumulatedTime;

    // Position update
    for ( int j = 0; j < m_bodies.size(); j++ ) {
      m_bodies[ j ].Update( dt );
    }
    ResolveContact( contact );
    accumulatedTime += dt;
  }

  // Update the positions for the rest of this frame's time
  const float timeRemaining = dt_sec - accumulatedTime;
  if ( timeRemaining > 0.0f ) {
    for ( int i = 0; i < m_bodies.size(); i++ ) {
      m_bodies[ i ].Update( timeRemaining );
    }
  }
}
```

Now you can see why we keep the loops for the PreSolve, Solve, and PostSolve functions separate. There's no need to run the PreSolve and PostSolve functions multiple times. But this iterative approach requires that we call Solve multiple times.

So, in the above Update function, I added fifty iterations to the solvers. And that gets us pretty close to the correct solution. So let's have a look at the results:



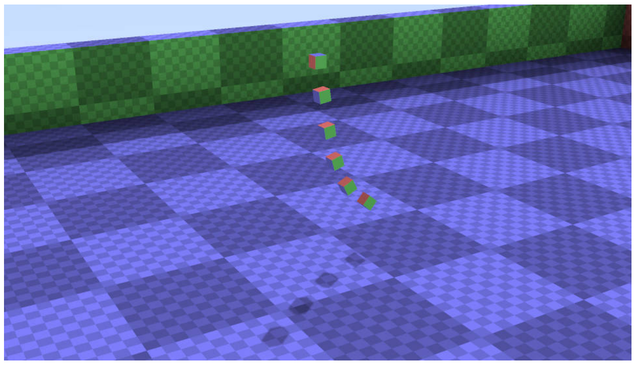


Figure 7: Physical Pendulum Chain Improved

As you can see, it is improved. The chain doesn't just immediately hit the bottom. But it's not perfect.
So what do we do? Give up and use global solvers? I mean, fifty iterations is rather absurd and won't be performant enough for a real time solution.
There's gotta be something we can do that's more efficient. And that's the topic of the next chapter.

## 10 Warm Starting

Warm starting is a method for stabilizing constraints that takes advantage of the high frame rate of real-time simulations. Basically, if we cache off the previous frame's constraint forces we can re-apply them this frame before calculating the new solutions for the constraints.

After all, from frame to frame, bodies don't typically move very much. So, using the previous frame's solutions to "warm start" this frame's solutions, should get us to the correct stable solution in just a few frames.

So, let's go ahead and add this feature to our constraints and then go back to looping over our constraints only once per frame:

```
class ConstraintDistance : public Constraint {
public:
  ConstraintDistance() : Constraint(),  m_cachedLambda( 1 ), m_Jacobian( 1, 12 ) {
    m_cachedLambda.Zero();
  }

  void PreSolve( const float dt_sec ) override;
  void Solve() override;

private:
  MatMN m_Jacobian;

  VecN m_cachedLambda;
};

void ConstraintDistance::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 r = worldAnchorB - worldAnchorA;
  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;

  m_Jacobian.Zero();

  Vec3 J1 = ( a - b ) * 2.0f;
  m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

  Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
  m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

  Vec3 J3 = ( b - a ) * 2.0f;
  m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

  Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
  m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
  m_Jacobian.rows[ 0 ][ 10] = J4.y;
  m_Jacobian.rows[ 0 ][ 11] = J4.z;

  //
  // Apply warm starting from last frame
  //
  const VecN impulses = m_Jacobian.Transpose() * m_cachedLambda;
  ApplyImpulses( impulses );
}

void ConstraintDistance::Solve() {
	const MatMN JacobianTranspose = m_Jacobian.Transpose();

	// Build the system of equations
	const VecN q_dt = GetVelocities();
	const MatMN invMassMatrix = GetInverseMassMatrix();
	const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
	VecN rhs = m_Jacobian * q_dt * -1.0f;

	// Solve for the Lagrange multipliers
	const VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

	// Apply the impulses
	const VecN impulses = JacobianTranspose * lambdaN;
	ApplyImpulses( impulses );

	// Accumulate the impulses for warm starting
	m_cachedLambda += lambdaN;
}
```

Now, let's test this code with a reduced number of iterations. Try reducing the number from fifty to five and see what this does.

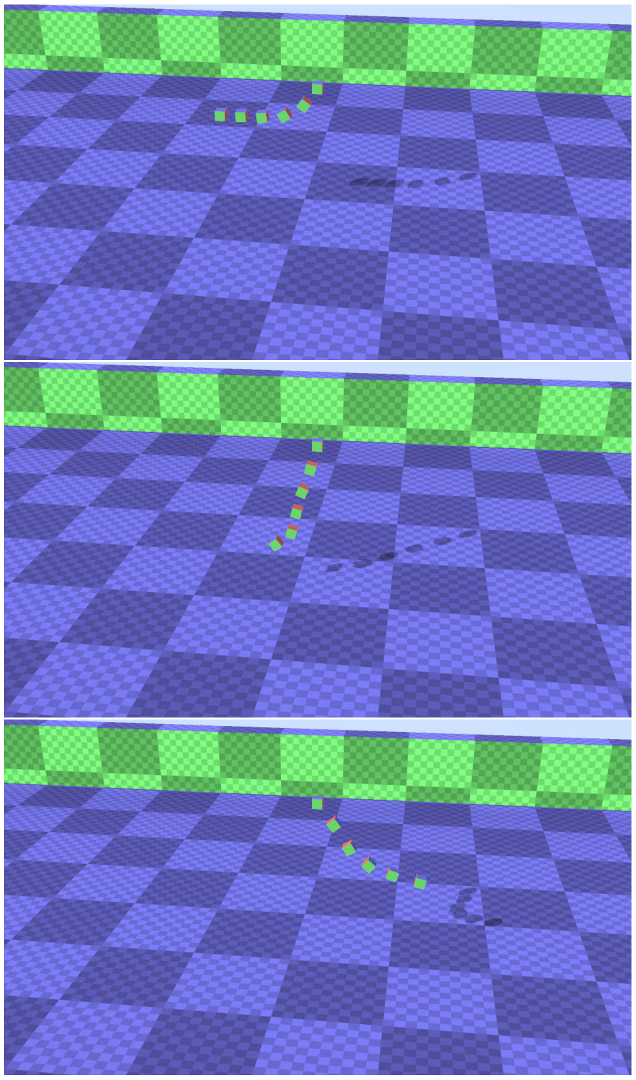


Figure 8: Warm Starting

Look at that! It's just as stable as the crazy high number of iterations. This iterative approach is starting to look promising.

## 10.1 Ragdolls + Warm Starting

At this moment you might be very tempted to try your hand at your very first "ragdoll". You could set the first body's invMass to one, and then it'll fall with the other bodies.

In the current state of the physics engine, you may see the bodies hit the ground and then immediately freak out and fly all over.

So now, let's try to make our very first "ragdoll" and let the bodies hit the ground. Go ahead and set the first body's invMass to one.

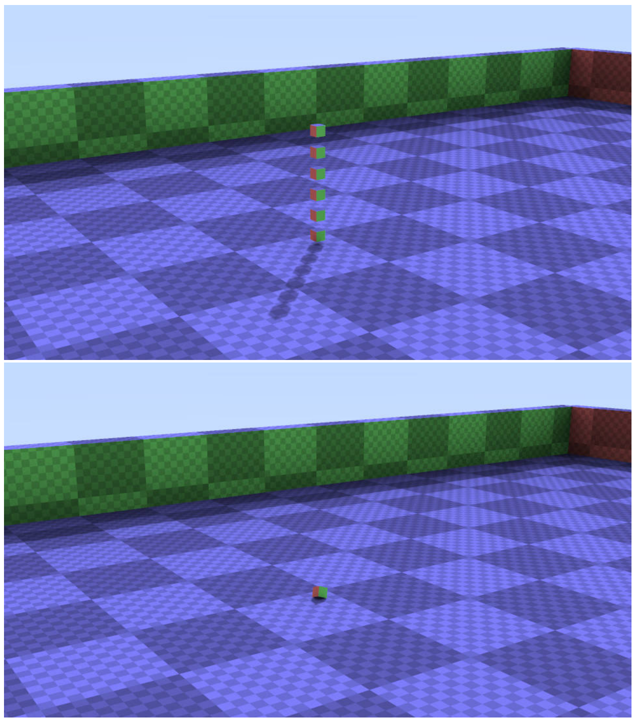


Figure 9: Bad Ragdoll

Whoah, that's not good. What's happening?

Well, when the chain hits the ground, it hits with enough force that the constraints will have rather large corrective impulses. And then the warm starting ends up freaking out the following frame. Fortunately, we can pretty easily check for this problem and prevent it. So, let's just override the PostSolve function and add a check:

```
void ConstraintDistance::PostSolve() {
  // Limit the warm starting to reasonable limits
  if ( m_cachedLambda[ 0 ] * 0.0f != m_cachedLambda[ 0 ] * 0.0f ) {
    m_cachedLambda[ 0 ] = 0.0f;
  }
  const float limit = 1e5f;
  if ( m_cachedLambda[ 0 ] > limit ) {
    m_cachedLambda[ 0 ] = limit;
  }
  if ( m_cachedLambda[ 0 ] < -limit ) {
    m_cachedLambda[ 0 ] = -limit;
  }
}
```

So now if we re-run our simulation. We get some nice warm starting and it won't freak out if there's a very large impulse for one frame.

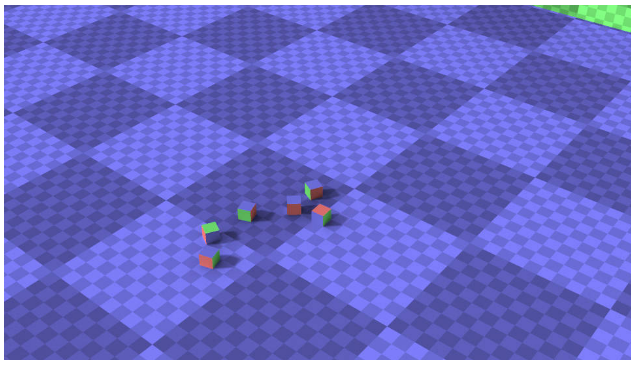


Figure 10: Stable Ragdoll

It's probably also worth noting that in your simulation you may wish to have "breakable" constraints. You've probably seen these in action, especially if you've played a game with a dismemberment feature.

To make a breakable constraint, you could expose the max impulse to designers, and then if that impulse is reached, permanently disable this constraint. Or even better, just remove the constraint from the simulation all together.

# 11 Baumgarte Stabilization

I'm not sure if you have noticed, but the constraint may have been a little unstable. Or maybe you really noticed and you've been going insane thinking that you can't get this to work and that you're a total failure and you'll never be able to accomplish the task of understanding the contents of this book.

Okay, that was dark for a moment. But, in case you have had some stability troubles, that is completely normal and expected. It happens due to the linearization of the simulation.

Basically, we calculate a constraint force that gets the body's velocities to move tangent to the constraint. Then when we update the position, the dynamic body will have moved slightly away from the static body, as in the figure below:

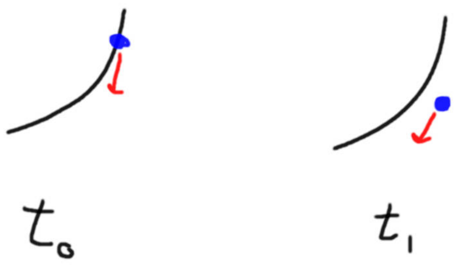


Figure 11: constraint drift

In order to fix this situation, we need some sort of stabilization method. A fairly common one is known as Baumgarte stabilization. And it's pretty easy to implement too.

We want to introduce a restorative force and we can do that by introducing an extra term to the velocity constraint:

$$\dot{C} = -\frac{\beta}{\Delta t} \cdot C \tag{23}$$

where $\beta \in [0, 1)$.

You can think of this in a pretty simple way. If you consider that $C$ is just how much distance the constraint is being violated, and $\Delta t$ as the amount of time in a given step of the simulation, then $-\frac{C}{\Delta t}$ is the exact velocity required to close the gap in a single time step.

Now, the reason for the $\beta$-factor is because a full correction actually adds a significant amount of energy into the system, which leads to instability. So you need a $\beta$ that corrects for the constraint drift but doesn't add too much energy.

There's a handful of ways to approach choosing the $\beta$-factor. You can hand tune them, in which case you may wish to open it up as a parameter to your designers (or not, I've heard of some people having bad experiences with this). Or you can try an automated approach. But for us, it'll be good enough to just hand tune them.

So let's go ahead and add that code to our constraints. However, for this book, we're only going to add it to the distance component of our constraints.

```
class ConstraintDistance : public Constraint {
public:
  ConstraintDistance() : Constraint(),
    m_cachedLambda( 1 ),
    m_Jacobian( 1, 12 ) {
    m_cachedLambda.Zero();
    m_baumgarte = 0.0f;
  }

  void PreSolve( const float dt_sec ) override;
  void Solve() override;
  void PostSolve() override;

private:
  MatMN m_Jacobian;

  VecN m_cachedLambda;
  float m_baumgarte;
};

void ConstraintDistance::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 r = worldAnchorB - worldAnchorA;
  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;

  m_Jacobian.Zero();

  Vec3 J1 = ( a - b ) * 2.0f;
  m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

  Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
  m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

  Vec3 J3 = ( b - a ) * 2.0f;
  m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

  Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
  m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
  m_Jacobian.rows[ 0 ][ 10] = J4.y;
  m_Jacobian.rows[ 0 ][ 11] = J4.z;

  //
  // Apply warm starting from last frame
  //
  const VecN impulses = m_Jacobian.Transpose() * m_cachedLambda;
  ApplyImpulses( impulses );

  //
  //  Calculate the baumgarte stabilization
  //
  float C = r.Dot( r );
  C = std::max( 0.0f, C - 0.01f );
  const float Beta = 0.05f;
  m_baumgarte = ( Beta / dt_sec ) * C;
}

void ConstraintDistance::Solve() {
  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities();
  const MatMN invMassMatrix = GetInverseMassMatrix();
  const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
  VecN rhs = m_Jacobian * q_dt * -1.0f;
  rhs[ 0 ] -= m_baumgarte;

  // Solve for the Lagrange multipliers
  const VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

  // Apply the impulses
  const VecN impulses = JacobianTranspose * lambdaN;
  ApplyImpulses( impulses );

  // Accumulate the impulses for warm starting
  m_cachedLambda += lambdaN;
}

void ConstraintDistance::PostSolve() {
  // Limit the warm starting to reasonable limits
  if ( m_cachedLambda[ 0 ] * 0.0f != m_cachedLambda[ 0 ] * 0.0f ) {
    m_cachedLambda[ 0 ] = 0.0f;
  }
  const float limit = 1e5f;
  if ( m_cachedLambda[ 0 ] > limit ) {
    m_cachedLambda[ 0 ] = limit;
  }
  if ( m_cachedLambda[ 0 ] < -limit ) {
    m_cachedLambda[ 0 ] = -limit;
  }
}
```

And now letting this thing run, it's solid. We finally did it. We have a completely stable simulation.

## 11.1 Slop

Before we move on to the next constraint, I want to talk about this curious line of code:

```
  C = std::max( 0.0f, C - 0.01f );
```

What in the world is that about?

Well, as previously mentioned, this stabilization method adds a little bit of energy to the system. And if we use it to enforce the constraint completely, then our constraint will never completely stabilize, and it'll always jitter a little. The simplest way to avoid that jitter, is to add a little slop; where we don't bother with the baumgarte stabilization until the constraint is violated beyond this tiny tolerance.

## 12 Penetration Constraint

The next constraint we will cover is the penetration constraint. And ironically, it's probably the easiest constraint to implement too (next to the distance constraint).

You might be curious why we're even bothering. After all we have a solution for penetration already. When we resolve our contacts we project them outside of each other.

In my view, that solution is a hack. It's great for getting started. Especially, when we didn't have the framework yet to figure out another solution. But it has a lot of problems.

In case you hadn't noticed yet, we can't stack boxes. If you attempt to do so, the stack will very quickly fall apart. And no shapes will come to rest on a slope. The only way to solve these problems is with constraints and manifolds.

So, let's go ahead and define the penetration constraint:

$$C = \vec{n} \cdot (\vec{r}_2 - \vec{r}_1) \geq 0 \tag{24}$$

Notice this is not holonomic since it uses a greater or equal to zero, and not just equal to zero. Handling this is not very different from an equality constraint. We simply check to see if the constraint is violated. If it is, then we enforce the constraint, and if it isn't, then we don't.

So let's go ahead and derive the Jacobian of the constraint:

$$\begin{aligned}\dot{C} &= \frac{d}{dt}(\vec{n} \cdot (\vec{r}_2 - \vec{r}_1)) \\ &= \vec{n} \cdot \frac{d\vec{r}_2}{dt} - \vec{n} \cdot \frac{d\vec{r}_1}{dt}\end{aligned}$$

recall:

$$\frac{d\vec{r}}{dt} = \vec{v} + \vec{\omega} \times \vec{r}$$

giving us:

$$\begin{aligned}\implies \dot{C} &= \vec{n} \cdot \vec{v}_2 + \vec{n} \cdot (\vec{\omega}_2 \times \vec{r}_2) - \vec{n} \cdot \vec{v}_1 - \vec{n} \cdot (\vec{\omega}_1 \times \vec{r}_1) \\ &= \vec{n} \cdot \vec{v}_2 + \vec{\omega}_2 \cdot (\vec{r}_2 \times \vec{n}) - \vec{n} \cdot \vec{v}_1 - \vec{\omega}_1 \cdot (\vec{r}_1 \times \vec{n}) \\ &= \begin{pmatrix}-\vec{n}, & -\vec{r}_1 \times \vec{n}, & \vec{n}, & \vec{r}_2 \times \vec{n}\end{pmatrix} \begin{pmatrix}\vec{v}_1 \\ \vec{\omega}_1 \\ \vec{v}_2 \\ \vec{\omega}_2\end{pmatrix}\end{aligned}$$

And, we now have our jacobian:

$$J = \begin{pmatrix}-\vec{n}, & -\vec{r}_1 \times \vec{n}, & \vec{n}, & \vec{r}_2 \times \vec{n}\end{pmatrix} \tag{25}$$

We should probably also add friction to this constraint as well. Since it'll be very common for there to be friction between penetrating objects. As it turns out, we can model friction in much the same we do the penetration, but with the tangential vectors $\vec{u}$ and $\vec{v}$ instead of the normal. And we will clamp the output forces calculated to be within the limits of $\mu m g$.

And the code will look something like:

```
class ConstraintPenetration : public Constraint {
public:
  ConstraintPenetration() : Constraint(), m_cachedLambda( 3 ), m_Jacobian( 3, 12 ) {
    m_cachedLambda.Zero();
    m_baumgarte = 0.0f;
    m_friction = 0.0f;
  }

  void PreSolve( const float dt_sec ) override;
  void Solve() override;

  VecN m_cachedLambda;
  Vec3 m_normal;    // in Body A's local space

  MatMN m_Jacobian;

  float m_baumgarte;
  float m_friction;
};

void ConstraintPenetration::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;

  const float frictionA = m_bodyA->m_friction;
  const float frictionB = m_bodyB->m_friction;
  m_friction = frictionA * frictionB;

  Vec3 u;
  Vec3 v;
  m_normal.GetOrtho( u, v );

  // Convert tangent space from model space to world space
  Vec3 normal = m_bodyA->m_orientation.RotatePoint( m_normal );
  u = m_bodyA->m_orientation.RotatePoint( u );
  v = m_bodyA->m_orientation.RotatePoint( v );

  //
  //  Penetration Constraint
  //
  m_Jacobian.Zero();

  // First row is the primary distance constraint that holds the anchor points together
  Vec3 J1 = normal * -1.0f;
  m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

  Vec3 J2 = ra.Cross( normal * -1.0f );
  m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

  Vec3 J3 = normal * 1.0f;
  m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

  Vec3 J4 = rb.Cross( normal * 1.0f );
  m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
  m_Jacobian.rows[ 0 ][ 10] = J4.y;
  m_Jacobian.rows[ 0 ][ 11] = J4.z;

  //
  //  Friction Jacobians
  //
  if ( m_friction > 0.0f ) {
    Vec3 J1 = u * -1.0f;
    m_Jacobian.rows[ 1 ][ 0 ] = J1.x;
    m_Jacobian.rows[ 1 ][ 1 ] = J1.y;
    m_Jacobian.rows[ 1 ][ 2 ] = J1.z;
    Vec3 J2 = ra.Cross( u * -1.0f );
    m_Jacobian.rows[ 1 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 1 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 1 ][ 5 ] = J2.z;

    Vec3 J3 = u * 1.0f;
    m_Jacobian.rows[ 1 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 1 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 1 ][ 8 ] = J3.z;

    Vec3 J4 = rb.Cross( u * 1.0f );
    m_Jacobian.rows[ 1 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 1 ][ 10] = J4.y;
    m_Jacobian.rows[ 1 ][ 11] = J4.z;
  }
  if ( m_friction > 0.0f ) {
    Vec3 J1 = v * -1.0f;
    m_Jacobian.rows[ 2 ][ 0 ] = J1.x;
    m_Jacobian.rows[ 2 ][ 1 ] = J1.y;
    m_Jacobian.rows[ 2 ][ 2 ] = J1.z;

    Vec3 J2 = ra.Cross( v * -1.0f );
    m_Jacobian.rows[ 2 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 2 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 2 ][ 5 ] = J2.z;

    Vec3 J3 = v * 1.0f;
    m_Jacobian.rows[ 2 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 2 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 2 ][ 8 ] = J3.z;

    Vec3 J4 = rb.Cross( v * 1.0f );
    m_Jacobian.rows[ 2 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 2 ][ 10] = J4.y;
    m_Jacobian.rows[ 2 ][ 11] = J4.z;
  }

  //
  // Apply warm starting from last frame
  //
  const VecN impulses = m_Jacobian.Transpose() * m_cachedLambda;
  ApplyImpulses( impulses );

  //
  //  Calculate the baumgarte stabilization
  //
  float C = ( b - a ).Dot( normal );
  C = std::min( 0.0f, C + 0.02f );  // Add slop
  float Beta = 0.25f;
  m_baumgarte = Beta * C / dt_sec;
}

void ConstraintPenetration::Solve() {
  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities();
  const MatMN invMassMatrix = GetInverseMassMatrix();
  const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
  VecN rhs = m_Jacobian * q_dt * -1.0f;
  rhs[ 0 ] -= m_baumgarte;

  // Solve for the Lagrange multipliers
  VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

  // Accumulate the impulses and clamp to within the constraint limits
  VecN oldLambda = m_cachedLambda;
  m_cachedLambda += lambdaN;
  const float lambdaLimit = 0.0f;
  if ( m_cachedLambda[ 0 ] < lambdaLimit ) {
    m_cachedLambda[ 0 ] = lambdaLimit;
  }
  if ( m_friction > 0.0f ) {
    const float umg = m_friction * 10.0f * 1.0f / ( m_bodyA->m_invMass + m_bodyB->m_invMass );
    const float normalForce = fabsf( lambdaN[ 0 ] * m_friction );
    const float maxForce = ( umg > normalForce ) ? umg : normalForce;

    if ( m_cachedLambda[ 1 ] > maxForce ) {
      m_cachedLambda[ 1 ] = maxForce;
    }
    if ( m_cachedLambda[ 1 ] < -maxForce ) {
      m_cachedLambda[ 1 ] = -maxForce;
    }

    if ( m_cachedLambda[ 2 ] > maxForce ) {
      m_cachedLambda[ 2 ] = maxForce;
    }
    if ( m_cachedLambda[ 2 ] < -maxForce ) {
      m_cachedLambda[ 2 ] = -maxForce;
    }
  }
  lambdaN = m_cachedLambda - oldLambda;

  // Apply the impulses
  const VecN impulses = JacobianTranspose * lambdaN;
  ApplyImpulses( impulses );
}
```

Now, let's go ahead and test our approach for penetration constraints. This is going to be a little different from our other constraints. Since, these won't be permanent. After all, if two objects penetrate one frame, it doesn't mean they'll touch for the rest of the simulation. So, for now these will be one offs that exist only for a single frame.

And here's what adding that to the Scene::Update function looks like:

```
void Scene::Update( const float dt_sec ) {
  std::vector< ConstraintPenetration > penetrationConstraints;

  // Gravity impulse
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];
    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }

  //
  // Broadphase (build potential collision pairs)
  //
  std::vector< collisionPair_t > collisionPairs;
  BroadPhase( m_bodies.data(), (int)m_bodies.size(), collisionPairs, dt_sec );

  //
  //  NarrowPhase (perform actual collision detection)
  //
  int numContacts = 0;
  contact_t * contacts = (contact_t *)alloca( sizeof( contact_t ) * collisionPairs.size() );
  for ( int i = 0; i < collisionPairs.size(); i++ ) {
    const collisionPair_t & pair = collisionPairs[ i ];
    Body * bodyA = &m_bodies[ pair.a ];
    Body * bodyB = &m_bodies[ pair.b ];

    // Skip body pairs with infinite mass
    if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
      continue;
    }

    // Check for intersection
    contact_t contact;
    if ( Intersect( bodyA, bodyB, dt_sec, contact ) ) {
      if ( 0.0f == contact.timeOfImpact ) {
        // Static contact
```

```
            ConstraintPenetration constraint;
            constraint.m_bodyA = contact.bodyA;
            constraint.m_bodyB = contact.bodyB;

            constraint.m_anchorA = contact.ptOnA_LocalSpace;
            constraint.m_anchorB = contact.ptOnB_LocalSpace;

            // Get the normal in BodyA's space
            Vec3 normal = constraint.m_bodyA->m_orientation.Inverse().RotatePoint( contact.normal *
    -1.0f );

            constraint.m_normal = normal;
            constraint.m_normal.Normalize();

            penetrationConstraints.push_back( constraint );
        } else {
            // Ballistic contact
            contacts[ numContacts ] = contact;
            numContacts++;
        }
    }
}

// Sort the times of impact from first to last
if ( numContacts > 1 ) {
    qsort( contacts, numContacts, sizeof( contact_t ), CompareContacts );
}

//
//  Solve Constraints
//
for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->PreSolve( dt_sec );
}
for ( int i = 0; i < penetrationConstraints.size(); i++ ) {
    penetrationConstraints[ i ].PreSolve( dt_sec );
}

const int maxIters = 5;
for ( int iters = 0; iters < maxIters; iters++ ) {
    for ( int i = 0; i < m_constraints.size(); i++ ) {
        m_constraints[ i ]->Solve();
    }
    for ( int i = 0; i < penetrationConstraints.size(); i++ ) {
        penetrationConstraints[ i ].Solve();
    }
}

for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->PostSolve();
}
for ( int i = 0; i < penetrationConstraints.size(); i++ ) {
    penetrationConstraints[ i ].PostSolve();
}

//
// Apply ballistic impulses
//
float accumulatedTime = 0.0f;
for ( int i = 0; i < numContacts; i++ ) {
    contact_t & contact = contacts[ i ];
    const float dt = contact.timeOfImpact - accumulatedTime;

    // Position update
    for ( int j = 0; j < m_bodies.size(); j++ ) {
        m_bodies[ j ].Update( dt );
    }

    ResolveContact( contact );
    accumulatedTime += dt;
}
```

```
  // Update the positions for the rest of this frame's time
  const float timeRemaining = dt_sec - accumulatedTime;
  if ( timeRemaining > 0.0f ) {
    for ( int i = 0; i < m_bodies.size(); i++ ) {
      m_bodies[ i ].Update( timeRemaining );
    }
  }
}
```

Notice that all of our t = 0 contacts now become penetration constraints. And all of our t > 0 contacts remain in the ballistic contact resolution section. Our update function is starting to look a little cleaner now.

Let's go ahead and test this with what is, in some circles, considered to be the ultimate test in a physics simulation... a stack of boxes:

```
void Scene::Initialize() {
  Body body;

  //
  //  Stack of Boxes
  //
  int x = 0;
  int y = 0;
  const int stackHeight = 5;
  for ( int z = 0; z < stackHeight; z++ ) {
    float offset = ( ( z & 1 ) == 0 ) ? 0.0f : 0.15f;
    float xx = (float)x + offset;
    float yy = (float)y + offset;
    float delta = 0.04f;
    float scaleHeight = 2.0f + delta;
    float deltaHeight = 1.0f + delta;
    body.m_position = Vec3( (float)xx * scaleHeight, (float)yy * scaleHeight, deltaHeight + (float
    )z * scaleHeight );
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_shape = new ShapeBox( g_boxUnit, sizeof( g_boxUnit ) / sizeof( Vec3 ) );
    body.m_invMass = 1.0f;
    body.m_elasticity = 0.5f;
    body.m_friction = 0.5f;
    m_bodies.push_back( body );
  }

  //
  //  Standard floor and walls
  //
  AddStandardSandBox( m_bodies );
}
```

Now, remember, our old code, using impulses and projection method, a stable stack of boxes was impossible. It would jitter and almost immediately fall over.

So, how do penetration constraints stack up?

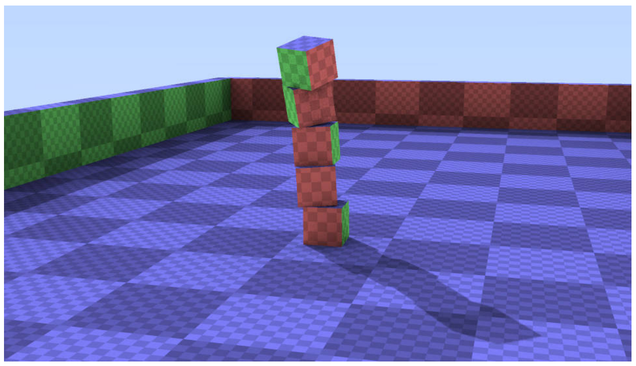


Figure 12: Unstable Stack

As you can see. It still jitters. If you had tested this situation before, then you may notice this is a little better than what we had before. However, it still isn't exactly stable.

What's going on here? I thought the whole reason we were doing this was to get a stable stack?

Well, a part of the problem is that we can't take advantage of warm starting. These constraints only exist for a single frame. The other problem is that GJK only gives us a single contact point between two bodies, but bodies could have infinitely more. And using more contacts should help make it more stable.

So, how do we get it to where we really want it?

## 13 Manifolds/Contact Caching

In order to get this stack stable, we're going to add contact caching. And we'll be doing that with the inclusion of a new data structure called a manifold.

For us, a collision manifold will be a collection of contacts between two bodies. Even though our bodies are composed of convex shapes, they may have multiple contacts. Let's look at the example of two boxes in figure 13.

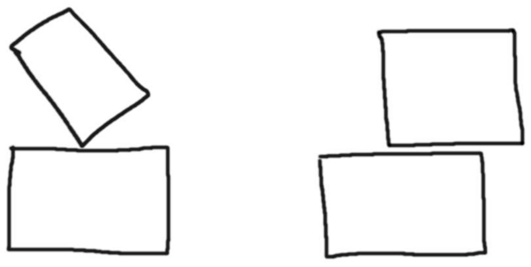


Figure 13: box contacts

One can see that for two colliding boxes, they may have as few as one contact point, or infinitely many. However, we will restrict our manifolds to having a maximum of four contacts.

Now, we only calculate a single contact every frame. So, to build the manifold, we will have to cache off the contacts from previous frames. Of course, this also means that contacts may become invalid as the simulation steps forward. Therefore, every frame, we will need to check for and remove outdated contacts while adding the new ones to the manifold.

Let's have a look at what this class might look like:

```
class Manifold {
public:
  Manifold() : m_bodyA( NULL ), m_bodyB( NULL ), m_numContacts( 0 ) {}

  void AddContact( const contact_t & contact );
  void RemoveExpiredContacts();

  void PreSolve( const float dt_sec );
  void Solve();
  void PostSolve();

  contact_t GetContact( const int idx ) const { return m_contacts[ idx ]; }
  int GetNumContacts() const { return m_numContacts; }

private:
  static const int MAX_CONTACTS = 4;
  contact_t m_contacts[ MAX_CONTACTS ];

  int m_numContacts;

  Body * m_bodyA;
  Body * m_bodyB;

  ConstraintPenetration m_constraints[ MAX_CONTACTS ];

  friend class ManifoldCollector;
};
```

Also, we're going to need a class that makes it easy for us to manage the manifolds. After all, when we add a contact it'd be nice to have a helper class that checks if a previously existing manifold should accept the new contact or if we need to create a new manifold. And, if all contacts in a manifold have expired, then we should remove the manifold from the collection.

```
class ManifoldCollector {
public:
  ManifoldCollector() {}

  void AddContact( const contact_t & contact );

  void PreSolve( const float dt_sec );
  void Solve();
  void PostSolve();

  void RemoveExpired();
  void Clear() { m_manifolds.clear(); } // For resetting the demo

public:
  std::vector< Manifold > m_manifolds;
};

/*
================================
ManifoldCollector::AddContact
================================
*/
void ManifoldCollector::AddContact( const contact_t & contact ) {
  // Try to find the previously existing manifold for contacts between these two bodies
  int foundIdx = -1;
  for ( int i = 0; i < m_manifolds.size(); i++ ) {
    const Manifold & manifold = m_manifolds[ i ];
    bool hasA = ( manifold.m_bodyA == contact.bodyA || manifold.m_bodyB == contact.bodyA );
    bool hasB = ( manifold.m_bodyA == contact.bodyB || manifold.m_bodyB == contact.bodyB );
    if ( hasA && hasB ) {
      foundIdx = i;
      break;
    }
  }

  // Add contact to manifolds
  if ( foundIdx >= 0 ) {
    m_manifolds[ foundIdx ].AddContact( contact );
  } else {
    Manifold manifold;
    manifold.m_bodyA = contact.bodyA;
    manifold.m_bodyB = contact.bodyB;

    manifold.AddContact( contact );
    m_manifolds.push_back( manifold );
  }
}

/*
================================
ManifoldCollector::RemoveExpired
================================
*/
void ManifoldCollector::RemoveExpired() {
  // Remove expired manifolds
  for ( int i = (int)m_manifolds.size() - 1; i >= 0; i-- ) {
    Manifold & manifold = m_manifolds[ i ];
    manifold.RemoveExpiredContacts();

    if ( 0 == manifold.m_numContacts ) {
      m_manifolds.erase( m_manifolds.begin() + i );
    }
  }
}

/*
================================
ManifoldCollector::PreSolve
================================
*/
```

```
void ManifoldCollector::PreSolve( const float dt_sec ) {
  for ( int i = 0; i < m_manifolds.size(); i++ ) {
    m_manifolds[ i ].PreSolve( dt_sec );
  }
}

/*
================================
ManifoldCollector::Solve
================================
*/
void ManifoldCollector::Solve() {
  for ( int i = 0; i < m_manifolds.size(); i++ ) {
    m_manifolds[ i ].Solve();
  }
}

/*
================================
Manifold::PostSolve
================================
*/
void ManifoldCollector::PostSolve() {
  for ( int i = 0; i < m_manifolds.size(); i++ ) {
    m_manifolds[ i ].PostSolve();
  }
}
```

Hopefully the ManifoldCollector class is straightforward. It just acts as a convenient little interface for our scene to manage manifolds between bodies.

And while we're on the subject, let's go ahead and add a ManifoldCollector to the scene:

```
ManifoldCollector m_manifolds;
```

```
void Scene::Update( const float dt_sec ) {
  m_manifolds.RemoveExpired();

  // Gravity impulse
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];
    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }

  //
  // Broadphase (build potential collision pairs)
  //
  std::vector< collisionPair_t > collisionPairs;
  BroadPhase( m_bodies.data(), (int)m_bodies.size(), collisionPairs, dt_sec );

  //
  //  NarrowPhase (perform actual collision detection)
  //
  int numContacts = 0;
  contact_t * contacts = (contact_t *)alloca( sizeof( contact_t ) * collisionPairs.size() );
  for ( int i = 0; i < collisionPairs.size(); i++ ) {
    const collisionPair_t & pair = collisionPairs[ i ];
    Body * bodyA = &m_bodies[ pair.a ];
    Body * bodyB = &m_bodies[ pair.b ];

    // Skip body pairs with infinite mass
    if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
      continue;
    }

    // Check for intersection
    contact_t contact;
    if ( Intersect( bodyA, bodyB, dt_sec, contact ) ) {
      if ( 0.0f == contact.timeOfImpact ) {
        // Static contact
        m_manifolds.AddContact( contact );
      } else {
        // Ballistic contact
        contacts[ numContacts ] = contact;
        numContacts++;
      }
    }
  }

  // Sort the times of impact from first to last
  if ( numContacts > 1 ) {
    qsort( contacts, numContacts, sizeof( contact_t ), CompareContacts );
  }

  //
  //  Solve Constraints
  //
  for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->PreSolve( dt_sec );
  }
  m_manifolds.PreSolve( dt_sec );

  const int maxIters = 5;
  for ( int iters = 0; iters < maxIters; iters++ ) {
    for ( int i = 0; i < m_constraints.size(); i++ ) {
      m_constraints[ i ]->Solve();
    }
    m_manifolds.Solve();
  }

  for ( int i = 0; i < m_constraints.size(); i++ ) {
    m_constraints[ i ]->PostSolve();
  }
  m_manifolds.PostSolve();


  //
  // Apply ballistic impulses
  //
  float accumulatedTime = 0.0f;
  for ( int i = 0; i < numContacts; i++ ) {
    contact_t & contact = contacts[ i ];
    const float dt = contact.timeOfImpact - accumulatedTime;

    // Position update
    for ( int j = 0; j < m_bodies.size(); j++ ) {
      m_bodies[ j ].Update( dt );
    }

    ResolveContact( contact );
    accumulatedTime += dt;
  }

  // Update the positions for the rest of this frame's time
  const float timeRemaining = dt_sec - accumulatedTime;
  if ( timeRemaining > 0.0f ) {
    for ( int i = 0; i < m_bodies.size(); i++ ) {
      m_bodies[ i ].Update( timeRemaining );
    }
  }
}
```

Alright, adding a ManifoldCollector was pretty easy. And say congratulations to yourself, because you won't ever have to visit the Scene::Update function again for the rest of the book. It's been finalized!

And with that out of the way, let's go ahead and examine the meaty details of the Manifold class.

Let's examine the RemoveExpiredContacts function first. This is where we'll determine if there's any contacts that are invalid and then remove them. There's a handful of strategies one can take here. Although the strategy we will have, will simply be based upon distance and direction. If the original contact normal no longer points in the same direction as the current normal, and if the penetration distance has become positive or if the parallel distance is too great, then we remove the contact:

```
void Manifold::RemoveExpiredContacts() {
  // remove any contacts that have drifted too far
  for ( int i = 0; i < m_numContacts; i++ ) {
    contact_t & contact = m_contacts[ i ];

    Body * bodyA = contact.bodyA;
    Body * bodyB = contact.bodyB;

    // Get the tangential distance of the point on A and the point on B
    const Vec3 a = bodyA->BodySpaceToWorldSpace( contact.ptOnA_LocalSpace );
    const Vec3 b = bodyB->BodySpaceToWorldSpace( contact.ptOnB_LocalSpace );

    Vec3 normal = m_constraints[ i ].m_normal;
    normal = bodyA->m_orientation.RotatePoint( normal );

    // Calculate the tangential separation and penetration depth
    const Vec3 ab = b - a;
    float penetrationDepth = normal.Dot( ab );
    Vec3 abNormal = normal * penetrationDepth;
    Vec3 abTangent = ab - abNormal;

    // If the tangential displacement is less than a specific threshold, it's okay to keep it
    const float distanceThreshold = 0.02f;
    if ( abTangent.GetLengthSqr() < distanceThreshold * distanceThreshold && penetrationDepth <=
    0.0f ) {
      continue;
    }

    // This contact has moved beyond its threshold and should be removed
    for ( int j = i; j < MAX_CONTACTS - 1; j++ ) {
      m_constraints[ j ] = m_constraints[ j + 1 ];
      m_contacts[ j ] = m_contacts[ j + 1 ];
      if ( j >= m_numContacts ) {
        m_constraints[ j ].m_cachedLambda.Zero();
      }
    }
    m_numContacts--;
    i--;
  }
}
```

And now let's look at the AddContact function. This one is a little more interesting. In fact there's a handful of strategies that may cause troubles.

1. What to do when a new constraint is physically very close to an old constraint?
2. What to do when we're already storing the maximum allowed constraints?

For item 1, we keep the old constraint and discard the new one. The reason is primarily for warm starting. The old constraint is likely to have converged on the correct solution.

For item 2, there's a couple of strategies that could be taken. A common strategy is to keep the contact of greatest penetration, then keep the contacts that are furthest from it. However, we'll do a slightly simpler version of this and just keep the contacts that are furthest from each other.

```
void Manifold::AddContact( const contact_t & contact_old ) {
  // Make sure the contact's BodyA and BodyB are of the correct order
  contact_t contact = contact_old;
  if ( contact_old.bodyA != m_bodyA || contact_old.bodyB != m_bodyB ) {
    contact.ptOnA_LocalSpace = contact_old.ptOnB_LocalSpace;
    contact.ptOnB_LocalSpace = contact_old.ptOnA_LocalSpace;
    contact.ptOnA_WorldSpace = contact_old.ptOnB_WorldSpace;
    contact.ptOnB_WorldSpace = contact_old.ptOnA_WorldSpace;

    contact.bodyA = m_bodyA;
    contact.bodyB = m_bodyB;
  }
  // If this contact is close to another contact, then keep the old contact
  for ( int i = 0; i < m_numContacts; i++ ) {
    const Body * bodyA = m_contacts[ i ].bodyA;
    const Body * bodyB = m_contacts[ i ].bodyB;

    const Vec3 oldA = bodyA->BodySpaceToWorldSpace( m_contacts[ i ].ptOnA_LocalSpace );
    const Vec3 oldB = bodyB->BodySpaceToWorldSpace( m_contacts[ i ].ptOnB_LocalSpace );

    const Vec3 newA = contact.bodyA->BodySpaceToWorldSpace( contact.ptOnA_LocalSpace );
    const Vec3 newB = contact.bodyB->BodySpaceToWorldSpace( contact.ptOnB_LocalSpace );

    const Vec3 aa = newA - oldA;
    const Vec3 bb = newB - oldB;

    const float distanceThreshold = 0.02f;
    if ( aa.GetLengthSqr() < distanceThreshold * distanceThreshold ) {
      return;
    }
    if ( bb.GetLengthSqr() < distanceThreshold * distanceThreshold ) {
      return;
    }
  }

  // If we're all full on contacts, then keep the contacts that are furthest away from each other
  int newSlot = m_numContacts;
  if ( newSlot >= MAX_CONTACTS ) {
    Vec3 avg = Vec3( 0, 0, 0 );
    avg += m_contacts[ 0 ].ptOnA_LocalSpace;
    avg += m_contacts[ 1 ].ptOnA_LocalSpace;
    avg += m_contacts[ 2 ].ptOnA_LocalSpace;
    avg += m_contacts[ 3 ].ptOnA_LocalSpace;
    avg += contact.ptOnA_LocalSpace;
    avg *= 0.2f;

    float minDist = ( avg - contact.ptOnA_LocalSpace ).GetLengthSqr();
    int newIdx = -1;
    for ( int i = 0; i < MAX_CONTACTS; i++ ) {
      float dist2 = ( avg - m_contacts[ i ].ptOnA_LocalSpace ).GetLengthSqr();

      if ( dist2 < minDist ) {
        minDist = dist2;
        newIdx = i;
      }
    }

    if ( -1 != newIdx ) {
      newSlot = newIdx;
    } else {
      return;
    }
  }

  m_contacts[ newSlot ] = contact;

  m_constraints[ newSlot ].m_bodyA = contact.bodyA;
  m_constraints[ newSlot ].m_bodyB = contact.bodyB;
  m_constraints[ newSlot ].m_anchorA = contact.ptOnA_LocalSpace;
  m_constraints[ newSlot ].m_anchorB = contact.ptOnB_LocalSpace;

  // Get the normal in BodyA's space
  Vec3 normal = m_bodyA->m_orientation.Inverse().RotatePoint( contact.normal * -1.0f );
  m_constraints[ newSlot ].m_normal = normal;
  m_constraints[ newSlot ].m_normal.Normalize();

  m_constraints[ newSlot ].m_cachedLambda.Zero();

  if ( newSlot == m_numContacts ) {
    m_numContacts++;
  }
}
```

And of course we need to add our Pre, Post, and Solve function calls:

```
/*
================================
Manifold::PreSolve
================================
*/
void Manifold::PreSolve( const float dt_sec ) {
  for ( int i = 0; i < m_numContacts; i++ ) {
    m_constraints[ i ].PreSolve( dt_sec );
  }
}

/*
================================
Manifold::Solve
================================
*/
void Manifold::Solve() {
  for ( int i = 0; i < m_numContacts; i++ ) {
    m_constraints[ i ].Solve();
  }
}

/*
================================
Manifold::PostSolve
================================
*/
void Manifold::PostSolve() {
  for ( int i = 0; i < m_numContacts; i++ ) {
    m_constraints[ i ].PostSolve();
  }
}
```

The combination of contact caching and warm starting should, at least in theory, stabilize our stacks. So, let's test it:

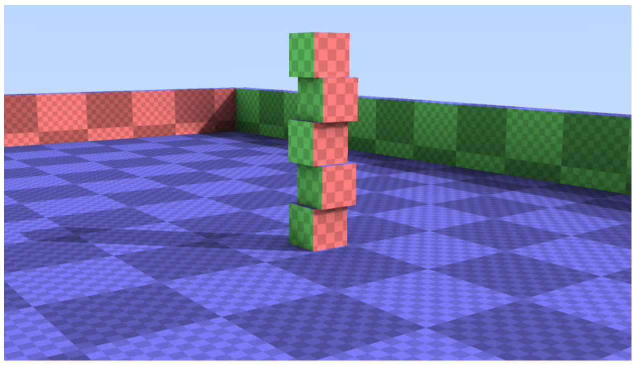


Figure 14: Stable Stack

Alright! Look at that! We've got a stable stack!

## 14 Hinge Constraint

Now that we have a proper distance constraint between two bodies with an anchor point. We can start making more interesting constraints that we'll eventually be able to combine together to make a ragdoll.

The next most simple constraint is the hinge constraint. There's at least three ways, that I can think of, to implement this constraint.

The first is the basic vector constraint:

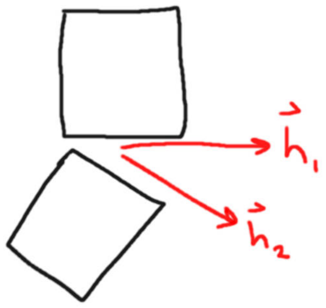


Figure 15: hinge vector constraint

This is probably the most intuitive implementation. We simply define the hinge axis in each body's local space, and then incorporate the angle between them in world space into the constraint equation. This will create a restorative torque to keep the two axis aligned. Let's go ahead and derive the Jacobian.

The vector hinge constraint can be described by a dot product between the hinge vectors of the two bodies:

$$C = \vec{h}_1 \cdot \vec{h}_2 = 0 \tag{26}$$

Taking the time derivative gives:

$$\dot{C} = \frac{d\vec{h}_1}{dt} \cdot \vec{h}_2 + \vec{h}_1 \cdot \frac{d\vec{h}_2}{dt}$$

And since these hinge vectors are directions, not positions. Then their time derivative is only dependent on their body's angular velocity:

$$\frac{d\vec{h}_1}{dt} = \vec{\omega}_1 \times \vec{h}_1 \tag{27}$$

$$\frac{d\vec{h}_2}{dt} = \vec{\omega}_2 \times \vec{h}_2 \tag{28}$$

$$\implies \dot{C} = \begin{pmatrix} 0, & \vec{h}_1 \times \vec{h}_2, & 0, & \vec{h}_2 \times \vec{h}_1 \end{pmatrix} \begin{pmatrix} \vec{v}_1 \\ \vec{\omega}_1 \\ \vec{v}_2 \\ \vec{\omega}_2 \end{pmatrix}$$

And now we have the jacobian:

$$\mathbf{J} = \begin{pmatrix} 0, & \vec{h}_1 \times \vec{h}_2, & 0, & \vec{h}_2 \times \vec{h}_1 \end{pmatrix} \tag{29}$$

It shouldn't be too difficult to go ahead and code this up. However, it has a downside that's probably unexpected. It's severely wobbly. This happens because for small angles, the sine is approximately equal to the angle. This linearization of the sine function prevents the restorative torque from accurately correcting the angle.

We can increase the accuracy by instead finding two vectors that are orthogonal to the axis and each other. Then we use those two vectors of one body and make sure they're co-planar to the two vectors in the other body's frame.

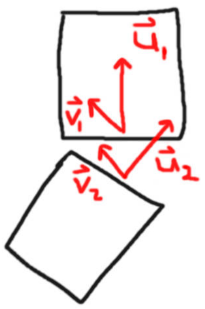


Figure 16: hinge axis constraint

Then we can go ahead and derive its Jacobians. Let's start with the two constraint equations that describe this:

$$C_1 = \vec{h}_1 \cdot \vec{u}_2 = 0 \tag{30}$$
$$C_2 = \vec{h}_1 \cdot \vec{v}_2 = 0 \tag{31}$$

Taking the time derivative to get the velocity constraint for the first constraint equation:

$$\begin{aligned}\dot{C}_1 &= \frac{d\vec{h}_1}{dt} \cdot \vec{u}_2 + \vec{h}_1 \cdot \frac{d\vec{u}_2}{dt} = 0 \\ &= (\vec{\omega}_1 \times \vec{h}_1) \cdot \vec{u}_2 + \vec{h}_1 \cdot (\vec{\omega}_2 \times \vec{u}_2) \\ &= \vec{\omega}_1 \cdot (\vec{h}_1 \times \vec{u}_2) + \vec{\omega}_2 \cdot (\vec{u}_2 \times \vec{h}_1) \\ => \dot{C}_1 &= \begin{pmatrix} 0, & \vec{h}_1 \times \vec{u}_2, & 0, & \vec{u}_2 \times \vec{h}_1 \end{pmatrix} \begin{pmatrix} \vec{v}_1 \\ \vec{\omega}_1 \\ \vec{v}_2 \\ \vec{\omega}_2 \end{pmatrix}\end{aligned}$$

It's a very similar derivation for the other constraint $C_2$. And this implies the two Jacobians are:

$$\mathbf{J}_1 = \begin{pmatrix} 0, & \vec{h}_1 \times \vec{u}_2, & 0, & \vec{u}_2 \times \vec{h}_1 \end{pmatrix} \tag{32}$$
$$\mathbf{J}_2 = \begin{pmatrix} 0, & \vec{h}_1 \times \vec{v}_2, & 0, & \vec{v}_2 \times \vec{h}_1 \end{pmatrix} \tag{33}$$

Once again, there's a problem with this formulation of the constraint also. While this constraint is no longer wobbly, it suffers from chirality. Basically, its mirror image also satisfies the constraint:

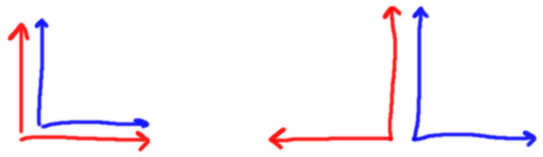


Figure 17: hinge axis chirality

This means that if one body is hit hard enough, then it could flip its orientation and then remain stable in this bad orientation. You may have even seen this in some games, where some ragdoll ends up in a bad state and a limb is twisted in an ugly way. While such a bug isn't the end of the world it is something we'd like to avoid.

So how are we supposed to eliminate these issues? This is where quaternions come to the rescue. Quaternions will allow us to restrict the relative orientation of the two bodies in a way that is uniquely stable.

## 15 Quaternions as Matrices

Before we actually dive into the quaternion constraints. We should mention a method of interpreting quaternions as vectors and matrices. If instead of thinking about quaternions as extensions to imaginary numbers:

$$q = q_w + i \cdot q_x + j \cdot q_y + k \cdot q_z \tag{34}$$

But instead intepreted them as a four dimensional vector:

$$q = \begin{pmatrix} q_w \\ q_x \\ q_y \\ q_z \end{pmatrix} \tag{35}$$

Then the quaternion product between two quaterions $a$ and $b$ would produce the vector:

$$a \cdot b = \begin{pmatrix} a_w \cdot b_w - a_x \cdot b_x - a_y \cdot b_y - a_z \cdot b_z \\ a_w \cdot b_x + a_x \cdot b_x + a_y \cdot b_y - a_z \cdot b_z \\ a_w \cdot b_y - a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z \\ a_w \cdot b_z + a_x \cdot b_x - a_y \cdot b_y + a_z \cdot b_z \end{pmatrix} \tag{36}$$

Then we could introduce the function $L(q)$ and $R(q)$ that takes in a 4d vector and outputs a 4x4 matrix:

$$L(q) = \begin{pmatrix} q_w & -q_x & -q_y & -q_z \\ q_x & q_w & -q_z & q_y \\ q_y & q_z & q_w & -q_x \\ q_z & -q_y & q_x & q_w \end{pmatrix} \tag{37}$$

$$R(q) = \begin{pmatrix} q_w & -q_x & -q_y & -q_z \\ q_x & q_w & q_z & -q_y \\ q_y & -q_z & q_w & q_x \\ q_z & q_y & -q_x & q_w \end{pmatrix} \tag{38}$$

And these have the curious property:

$$a \cdot b = L(a)b = R(b)a \tag{39}$$

This may not seem important right now. But we are very much about to take advantage of these properties in the next several chapters.

If you're curious why I'm calling these $L$ and $R$. It's because the $L$ matrix is a quaternion multiply from the left hand side, and the $R$ matrix is a quaternion multiply from the right hand side.

And we can add these functions to the base constraint class like so:

```
Mat4 Constraint::Left( const Quat & q ) {
  Mat4 L;
  L.rows[ 0 ] = Vec4( q.w, -q.x, -q.y, -q.z );
  L.rows[ 1 ] = Vec4( q.x,  q.w, -q.z,  q.y );
  L.rows[ 2 ] = Vec4( q.y,  q.z,  q.w, -q.x );
  L.rows[ 3 ] = Vec4( q.z, -q.y,  q.x,  q.w );
  return L.Transpose();
}

Mat4 Constraint::Right( const Quat & q ) {
  Mat4 R;
  R.rows[ 0 ] = Vec4( q.w, -q.x, -q.y, -q.z );
  R.rows[ 1 ] = Vec4( q.x,  q.w,  q.z, -q.y );
  R.rows[ 2 ] = Vec4( q.y, -q.z,  q.w,  q.x );
  R.rows[ 3 ] = Vec4( q.z,  q.y, -q.x,  q.w );
  return R.Transpose();
}
```

## 16 Quaternion Constraint

The math for quaternions isn't something that's typically taught, and that makes it a little more obscure. But I assure you, it's really not bad. So let's go ahead and figure out constraints in terms of quaternions. Let us start with the definition of the relative orientation between two bodies. If we use $q_1$ to define the orientation of body1 and $q_2$ to define the orientation of body2, then the transformation quaternion $q_r$ that takes us from body1's orientation to body2's orientation is:

$$q_r = q_1^* \cdot q_2 \tag{40}$$

where $q_1^*$ is the complex conjugate of $q_1$

Then the initial relative orientation between the two bodies is from the intial setup:

$$q_0 = q_{1_0}^* \cdot q_{2_0} \tag{41}$$

Then we can define a locking constraint (resticting relative orientation) to be:

$$C = q_r \cdot q_0^* = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{42}$$

And to convert this constraint into vector form, we can define a "projection" matrix:

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{43}$$

Then this modifies our constraint equation:

$$C = P \cdot (q_r \cdot q_0^*) = 0 \tag{44}$$

A quick note before pulling the mathematical crank to find the jacobian. The time derivative of a quaternion is defined to be:

$$\frac{dq}{dt} = \frac{1}{2}\omega \cdot q \tag{45}$$

Where $\omega$ is treated as a quaternion, with only imaginary components:

$$\omega = \begin{pmatrix} 0 \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{46}$$

Now taking the time derivative of the constraint equation yields:

$$\begin{aligned}
\dot{C} &= \frac{d}{dt}(P \cdot (q_r \cdot q_0^*)) \\
&= P \cdot (\frac{dq_r}{dt} \cdot q_0^* + q_r \cdot \frac{dq_0}{dt}) \\
&= P \cdot (\frac{dq_r}{dt} \cdot q_0^*) \\
&= P \cdot (\frac{d(q_1^* \cdot q_2)}{dt} \cdot q_0^*) \\
&= P \cdot (\frac{dq_1^*}{dt} \cdot q_2 \cdot q_0^* + q_1^* \cdot \frac{dq_2}{dt} \cdot q_0^*) \\
&= P \cdot (\frac{1}{2}(\omega_1 \cdot q_1)^* \cdot q_2 \cdot q_0^* + q_1^* \cdot \frac{1}{2}\omega_2 \cdot q_2 \cdot q_0^*) \\
&= \frac{1}{2}P \cdot (q_1^* \cdot \omega_1^* \cdot q_2 \cdot q_0^* + q_1^* \cdot \omega_2 \cdot q_2 \cdot q_0^*)
\end{aligned}$$

And noting that since $\omega$ is only composed of imaginary components then $\omega^* = -\omega$:

$$\implies \dot{C} = \frac{1}{2} P \cdot (-q_1^* \cdot \omega_1 \cdot q_2 \cdot q_0^* + q_1^* \cdot \omega_2 \cdot q_2 \cdot q_0^*)$$

And now we can use the mathemagic of the left hand matrix operator $L$ and the right hand matrix operator $R$ from the previous chapter. And also the fact that we can use the transpose of matrix $P$ to convert the vector form of $\omega$ into the quaternion form

$$\omega = P^T \cdot \vec{\omega} \tag{47}$$

then we can write the velocity constraint in matrix form:

$$\implies \dot{C} = \frac{1}{2} P \cdot (L(q_1^*) R(q_2 \cdot q_0^*) \cdot P^T \cdot \omega_2 - L(q_1^*) \cdot R(q_2 \cdot q_0^*) \cdot P^T \cdot \omega_1)$$

$$= \begin{pmatrix} 0, & -\frac{1}{2} P \cdot L(q_1^*) \cdot R(q_2 \cdot q_0^*) \cdot P^T, & 0, & \frac{1}{2} P \cdot L(q_1^*) \cdot R(q_2 \cdot q_0^*) \cdot P^T \end{pmatrix} \begin{pmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{pmatrix}$$

And as always, from inspection we get the jacobian:

$$J = \begin{pmatrix} 0, & -\frac{1}{2} P \cdot L(q_1^*) \cdot R(q_2 \cdot q_0^*) \cdot P^T, & 0, & \frac{1}{2} P \cdot L(q_1^*) \cdot R(q_2 \cdot q_0^*) \cdot P^T \end{pmatrix}$$

While this constraint is very useful for attaching bodies together, we're not actually going to code this one up. Since for this set of tutorials we don't actually need this. But it was important to go through the exercise of inspecting the math for it. Since we'll build off this for the quaternion hinge constraint.

Actually, I've changed my mind. You might as well see the code for this constraint:

```
class ConstraintOrientation : public Constraint {
public:
  ConstraintOrientation() : Constraint(), m_Jacobian( 4, 12 ) {
    m_baumgarte = 0.0f;
  }

  void PreSolve( const float dt_sec ) override;
  void Solve() override;

  Quat m_q0;      // The initial relative quaternion q1^-1 * q2

  MatMN m_Jacobian;

  float m_baumgarte;
};

/*
================================
ConstraintOrientation::PreSolve
================================
*/
void ConstraintOrientation::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 r = worldAnchorB - worldAnchorA;
  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;
```

```
// Get the orientation information of the bodies
const Quat q1 = m_bodyA->m_orientation;
const Quat q2 = m_bodyB->m_orientation;
const Quat q0_inv = m_q0.Inverse();
const Quat q1_inv = q1.Inverse();

const Vec3 u = Vec3( 1, 0, 0 );
const Vec3 v = Vec3( 0, 1, 0 );
const Vec3 w = Vec3( 0, 0, 1 );

Mat4 P;
P.rows[ 0 ] = Vec4( 0, 0, 0, 0 );
P.rows[ 1 ] = Vec4( 0, 1, 0, 0 );
P.rows[ 2 ] = Vec4( 0, 0, 1, 0 );
P.rows[ 3 ] = Vec4( 0, 0, 0, 1 );
Mat4 P_T = P.Transpose(); // I know it's pointless to do this with our particular matrix
  implementations.  But I like its self commenting.

const Mat4 MatA = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * -0.5f;
const Mat4 MatB = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * 0.5f;

//
//  The distance constraint
//

m_Jacobian.Zero();

// First row is the primary distance constraint that holds the anchor points together
Vec3 J1 = ( a - b ) * 2.0f;
m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

Vec3 J3 = ( b - a ) * 2.0f;
m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
m_Jacobian.rows[ 0 ][ 10] = J4.y;
m_Jacobian.rows[ 0 ][ 11] = J4.z;

//
// The quaternion jacobians
//
const int idx = 1;

Vec4 tmp;
{
  J1.Zero();
  m_Jacobian.rows[ 1 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 1 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 1 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, u.x, u.y, u.z );
  J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 1 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 1 ][ 5 ] = J2.z;

  J3.Zero();
  m_Jacobian.rows[ 1 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 1 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 1 ][ 8 ] = J3.z;
```

```
    tmp = MatB * Vec4( 0, u.x, u.y, u.z );
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 1 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 1 ][ 10] = J4.y;
    m_Jacobian.rows[ 1 ][ 11] = J4.z;
  }
  {
    J1.Zero();
    m_Jacobian.rows[ 2 ][ 0 ] = J1.x;
    m_Jacobian.rows[ 2 ][ 1 ] = J1.y;
    m_Jacobian.rows[ 2 ][ 2 ] = J1.z;

    tmp = MatA * Vec4( 0, v.x, v.y, v.z );
    J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 2 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 2 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 2 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 2 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 2 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 2 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, v.x, v.y, v.z );
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 2 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 2 ][ 10] = J4.y;
    m_Jacobian.rows[ 2 ][ 11] = J4.z;
  }
  {
    J1.Zero();
    m_Jacobian.rows[ 3 ][ 0 ] = J1.x;
    m_Jacobian.rows[ 3 ][ 1 ] = J1.y;
    m_Jacobian.rows[ 3 ][ 2 ] = J1.z;

    tmp = MatA * Vec4( 0, w.x, w.y, w.z );
    J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 3 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 3 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 3 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 3 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 3 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 3 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, w.x, w.y, w.z );
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 3 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 3 ][ 10] = J4.y;
    m_Jacobian.rows[ 3 ][ 11] = J4.z;
  }

  //
  //  Calculate the baumgarte stabilization
  //
  float C = r.Dot( r );
  const float Beta = 0.5f;
  m_baumgarte = ( Beta / dt_sec ) * C;
}

/*
================================
ConstraintOrientation::Solve
================================
*/
void ConstraintOrientation::Solve() {
  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities();
```

```
    const MatMN invMassMatrix = GetInverseMassMatrix();
    const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
    VecN rhs = m_Jacobian * q_dt * -1.0f;
    rhs[ 0 ] -= m_baumgarte;

    // Solve for the Lagrange multipliers
    VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

    // Apply the impulses
    const VecN impulses = JacobianTranspose * lambdaN;
    ApplyImpulses( impulses );
}
```

## 17 Quaternion Hinge Constraint

Now that we've managed our way through the basics of quaternion constraints. Let's see if we can modify it to generate a hinge constraint.

The hinge will have a particular axis, $\vec{h}$, that the relative quaternion is free to rotate about. And let the vectors $\vec{u}$ and $\vec{v}$ be orthonormal to $\vec{h}$, so that three vectors form an orthonormal basis. Then in body1's space the vectors are defined as $\vec{h}_1, \vec{u}_1, \vec{v}_1$ and in body2's space as $\vec{h}_2, \vec{u}_2, \vec{v}_2$

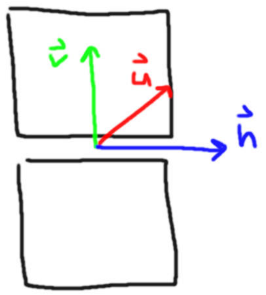


Figure 18: hinge setup

Then this gives us two constraints:

$$C_1 = P \cdot (q_r \cdot q_0^*) \cdot \vec{u} = 0 \tag{48}$$
$$C_2 = P \cdot (q_r \cdot q_0^*) \cdot \vec{v} = 0 \tag{49}$$

Let's go ahead and pull the mathematical crank for the first equation, to find its jacobian, and then we'll just be mindful that it's a similar derivation for the second.

$$\dot{C}_1 = \frac{d}{dt}(P \cdot (q_r \cdot q_0^*) \cdot \vec{u}) = 0 \tag{50}$$
$$= P \cdot \frac{d(q_1^* \cdot q_2)}{dt} \cdot q_0 \cdot \vec{u} + P \cdot (q_r \cdot q_0^*) \cdot \frac{d\vec{u}}{dt} \tag{51}$$
$$= P \cdot \frac{dq_1^*}{dt} \cdot q_2 \cdot q_0^* \cdot \vec{u} + P \cdot q_1^* \cdot \frac{dq_2}{dt} \cdot q_0^* \cdot \vec{u} + P \cdot (q_r \cdot q_0^*) \cdot \frac{d\vec{u}}{dt} \tag{52}$$

recall that:

$$\frac{d\vec{u}}{dt} = \vec{\omega} \times \vec{r},$$
$$\frac{dq}{dt} = \frac{1}{2}\omega \cdot q$$

but since $\vec{r} = 0$ here, then so is the time derivative of $\vec{u}$. So continuing

$$\frac{d\vec{u}}{dt} = \vec{\omega} \times \vec{r} = 0$$

$$\implies \dot{C}_1 = \frac{1}{2}P \cdot ((\omega_1 \cdot q_1)^* \cdot q_2 \cdot q_0^* \cdot \vec{u} + q_1^* \cdot \omega_2 \cdot q_2 \cdot q_0^* \cdot \vec{u})$$
$$= \frac{1}{2}P \cdot (q_1^* \cdot \omega_1^* \cdot q_2 \cdot q_0^* \cdot \vec{u} + q_1^* \cdot \omega_2 \cdot q_2 \cdot q_0^* \cdot \vec{u})$$
$$= \frac{1}{2}P \cdot (-q_1^* \cdot \omega_1 \cdot q_2 \cdot q_0^* \cdot \vec{u} + q_1^* \cdot \omega_2 \cdot q_2 \cdot q_0^* \cdot \vec{u})$$

And using our magical matrix operators

$$\implies \dot{C}_1 = \frac{1}{2}P \cdot (-L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{u} \cdot \vec{\omega_1} + L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{u} \cdot \omega_2)$$

$$= \begin{pmatrix} 0, & -\frac{1}{2}P \cdot L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{u}, & 0, & \frac{1}{2}P \cdot L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{u} \end{pmatrix} \begin{pmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{pmatrix}$$

And this implies that our jacobians for this constraint are simply:

$$J_1 = \begin{pmatrix} 0, & -\frac{1}{2}P \cdot L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{u}, & 0, & \frac{1}{2}P \cdot L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{u} \end{pmatrix}$$
$$J_2 = \begin{pmatrix} 0, & -\frac{1}{2}P \cdot L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{v}, & 0, & \frac{1}{2}P \cdot L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{v} \end{pmatrix}$$

Alright, let's translate this into code:

```
class ConstraintHingeQuat : public Constraint {
public:
  ConstraintHingeQuat() : Constraint(), m_cachedLambda( 3 ), m_Jacobian( 3, 12 ) {
    m_cachedLambda.Zero();
    m_baumgarte = 0.0f;
  }
  void PreSolve( const float dt_sec ) override;
  void Solve() override;
  void PostSolve() override;

  Quat q0;  // The initial relative quaternion q1^-1 * q2

  VecN m_cachedLambda;
  MatMN m_Jacobian;

  float m_baumgarte;
};

/*
================================
ConstraintHingeQuat::PreSolve
================================
*/
void ConstraintHingeQuat::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 r = worldAnchorB - worldAnchorA;
  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;

  // Get the orientation information of the bodies
  const Quat q1 = m_bodyA->m_orientation;
  const Quat q2 = m_bodyB->m_orientation;
  const Quat q0_inv = q0.Inverse();
  const Quat q1_inv = q1.Inverse();

  // This axis is defined in the local space of bodyA
  Vec3 u;
  Vec3 v;
  Vec3 hingeAxis = m_axisA;
  hingeAxis.GetOrtho( u, v );

  Mat4 P;
  P.rows[ 0 ] = Vec4( 0, 0, 0, 0 );
```

```
P.rows[ 1 ] = Vec4( 0, 1, 0, 0 );
P.rows[ 2 ] = Vec4( 0, 0, 1, 0 );
P.rows[ 3 ] = Vec4( 0, 0, 0, 1 );
Mat4 P_T = P.Transpose(); // I know it's pointless to do this with our particular matrix
  implementations.  But I like its self commenting.

const Mat4 MatA = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * -0.5f;
const Mat4 MatB = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * 0.5f;

const MatMN invMassMatrix = GetInverseMassMatrix();

m_Jacobian.Zero();

//
//  The distance constraint
//
Vec3 J1 = ( a - b ) * 2.0f;
m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

Vec3 J3 = ( b - a ) * 2.0f;
m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
m_Jacobian.rows[ 0 ][ 10] = J4.y;
m_Jacobian.rows[ 0 ][ 11] = J4.z;

//
// The quaternion jacobians
//
const int idx = 1;

Vec4 tmp;
{
  J1.Zero();
  m_Jacobian.rows[ 1 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 1 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 1 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, u.x, u.y, u.z );
  J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 1 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 1 ][ 5 ] = J2.z;

  J3.Zero();
  m_Jacobian.rows[ 1 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 1 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 1 ][ 8 ] = J3.z;

  tmp = MatB * Vec4( 0, u.x, u.y, u.z );
  J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 9 ] = J4.x;
  m_Jacobian.rows[ 1 ][ 10] = J4.y;
  m_Jacobian.rows[ 1 ][ 11] = J4.z;
}
{
  J1.Zero();
  m_Jacobian.rows[ 2 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 2 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 2 ][ 2 ] = J1.z;
```

```
    tmp = MatA * Vec4( 0, v.x, v.y, v.z );
    J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 2 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 2 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 2 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 2 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 2 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 2 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, v.x, v.y, v.z );
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 2 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 2 ][ 10] = J4.y;
    m_Jacobian.rows[ 2 ][ 11] = J4.z;
  }

  //
  // Apply warm starting from last frame
  //
  const VecN impulses = m_Jacobian.Transpose() * m_cachedLambda;
  ApplyImpulses( impulses );

  //
  //  Calculate the baumgarte stabilization
  //
  float C = r.Dot( r );
  C = std::max( 0.0f, C - 0.01f );
  const float Beta = 0.05f;
  m_baumgarte = ( Beta / dt_sec ) * C;
}

/*
================================
ConstraintHingeQuat::Solve
================================
*/
void ConstraintHingeQuat::Solve() {
  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities();
  const MatMN invMassMatrix = GetInverseMassMatrix();
  const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
  VecN rhs = m_Jacobian * q_dt * -1.0f;
  rhs[ 0 ] -= m_baumgarte;

  // Solve for the Lagrange multipliers
  const VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

  // Apply the impulses
  const VecN impulses = JacobianTranspose * lambdaN;
  ApplyImpulses( impulses );

  // Accumulate the impulses for warm starting
  m_cachedLambda += lambdaN;
}

/*
================================
ConstraintHingeQuat::PostSolve
================================
*/
void ConstraintHingeQuat::PostSolve() {
  // Limit the warm starting to reasonable limits
  for ( int i = 0; i < m_cachedLambda.N; i++ ) {
    if ( m_cachedLambda[ i ] * 0.0f != m_cachedLambda[ i ] * 0.0f ) {
      m_cachedLambda[ i ] = 0.0f;
    }
    const float limit = 20.0f;
```

```
      if ( m_cachedLambda[ i ] > limit ) {
        m_cachedLambda[ i ] = limit;
      }
      if ( m_cachedLambda[ i ] < -limit ) {
        m_cachedLambda[ i ] = -limit;
      }
  }
}
```

## 18 Limited Hinge Constraint

The next step we should probably take is to add limits to the constraint. As wonderful as it is that we have the hinge, most hinges in the real world come with limits.

You can define the limits in a number of ways. You may wish to define an axis that represents the "zero" angle of rotation. However, we will presume that the initial orientation of our bodies is the "zero".

Something else that will be unique about this extra constraint is that it's no longer of the form:

$$C = 0 \tag{53}$$

but instead it's of the form:

$$C \leq 0 \tag{54}$$

Which is actually not really that different from our more familiar constraints. It just means that we first need to calculate the relative angle, and check if the constraint is violated. If it is violated, then we apply the inequality constraint as though it were an equality constraint. If the constraint is not violated, then we simply ignore it.

So how do we calculate the relative angle? First we begin with the relative quaterion. Recall that the relative quaternion of the initial setup is:

$$q_0 = q_{1_0}^* \cdot q_{2_0} \tag{55}$$

And that the relative quaternion for any time later is:

$$q_r = q_1^* \cdot q_2 \tag{56}$$

Now what we need is a quaternion that tells us how much of a "difference" there is between these two quaternions. Which gives us the new quaternion $q_{rr}$:

$$q_{rr} = q_r \cdot q_0^* \tag{57}$$

If we use the interpretation of quaternions as an axis of rotation and an angle of rotation, then we can find how much our bodies have rotated about each other with:

$$\theta = 2 * \text{arcsin}(\vec{q}_{rr} \cdot \vec{h}) \tag{58}$$

Where we are only interested in using the $ijk$ components of $q_{rr}$ as a vector in $\Re^3$, and $\vec{h}$ is the hinge axis in bodyA's local space.

Now, all we need to do is calculate the amount of rotation around the hinge axis with that equation. Then check if it violates the constraint's limits. And only if there's a violation, do we need to bother with the extra constraint.

I should probably also mention what the extra constraint is actually defined to be. Recall from the last chapter the constraints were:

$$C_1 = P \cdot (q_r \cdot q_0^*) \cdot \vec{u} = 0 \tag{59}$$
$$C_2 = P \cdot (q_r \cdot q_0^*) \cdot \vec{v} = 0 \tag{60}$$

And the extra constraint that we need to enforce, but only if there's too much rotation, is:

$$C_1 = P \cdot (q_r \cdot q_0^*) \cdot \vec{h} = 0 \tag{61}$$

The only other thing we need to be careful with, is that the constraint force that we calculate from this extra constraint, must be restorative. This way it only pushes the bodies back into the allowed region. And we can do that with a simple bounds check on the $\lambda$'s

And translating into code:

```
class ConstraintHingeQuatLimited : public Constraint {
public:
  ConstraintHingeQuatLimited() : Constraint(), m_cachedLambda( 4 ), m_Jacobian( 4, 12 ) {
    m_cachedLambda.Zero();
    m_baumgarte = 0.0f;
    m_isAngleViolated = false;
    m_relativeAngle = 0.0f;
  }
  void PreSolve( const float dt_sec ) override;
  void Solve() override;
  void PostSolve() override;

  Quat m_q0;  // The initial relative quaternion q1^-1 * q2

  VecN m_cachedLambda;
  MatMN m_Jacobian;

  float m_baumgarte;

  bool m_isAngleViolated;
  float m_relativeAngle;
};


/*
================================
ConstraintHingeQuatLimited::PreSolve
================================
*/
void ConstraintHingeQuatLimited::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 r = worldAnchorB - worldAnchorA;
  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;

  // Get the orientation information of the bodies
  const Quat q1 = m_bodyA->m_orientation;
  const Quat q2 = m_bodyB->m_orientation;
  const Quat q0_inv = m_q0.Inverse();
  const Quat q1_inv = q1.Inverse();

  // This axis is defined in the local space of bodyA
  Vec3 u;
  Vec3 v;
  Vec3 hingeAxis = m_axisA;
  hingeAxis.GetOrtho( u, v );

  Mat4 P;
  P.rows[ 0 ] = Vec4( 0, 0, 0, 0 );
  P.rows[ 1 ] = Vec4( 0, 1, 0, 0 );
  P.rows[ 2 ] = Vec4( 0, 0, 1, 0 );
  P.rows[ 3 ] = Vec4( 0, 0, 0, 1 );
  Mat4 P_T = P.Transpose(); // I know it's pointless to do this with our particular matrix
    implementations.  But I like its self commenting.

  const Mat4 MatA = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * -0.5f;
  const Mat4 MatB = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * 0.5f;

  const float pi = acosf( -1.0f );
  const Quat qr = q1_inv * q2;
  const Quat qrr = qr * q0_inv;
  const float relativeAngle = 2.0f * asinf( qrr.xyz().Dot( hingeAxis ) ) * 180.0f / pi;;
```

```
// Check if there's an angle violation
m_isAngleViolated = false;
if ( relativeAngle > 45.0f ) {
  m_isAngleViolated = true;
}
if ( relativeAngle < -45.0f ) {
  m_isAngleViolated = true;
}
m_relativeAngle = relativeAngle;

//
// First row is the primary distance constraint that holds the anchor points together
//
m_Jacobian.Zero();

Vec3 J1 = ( a - b ) * 2.0f;
m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

Vec3 J3 = ( b - a ) * 2.0f;
m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
m_Jacobian.rows[ 0 ][ 10] = J4.y;
m_Jacobian.rows[ 0 ][ 11] = J4.z;

// The quaternion jacobians
const int idx = 1;

Vec4 tmp;
{
  J1.Zero();
  m_Jacobian.rows[ 1 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 1 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 1 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, u.x, u.y, u.z );
  J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 1 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 1 ][ 5 ] = J2.z;

  J3.Zero();
  m_Jacobian.rows[ 1 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 1 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 1 ][ 8 ] = J3.z;

  tmp = MatB * Vec4( 0, u.x, u.y, u.z );
  J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 9 ] = J4.x;
  m_Jacobian.rows[ 1 ][ 10] = J4.y;
  m_Jacobian.rows[ 1 ][ 11] = J4.z;
}
{
  J1.Zero();
  m_Jacobian.rows[ 2 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 2 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 2 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, v.x, v.y, v.z );
  J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 2 ][ 3 ] = J2.x;
```

```
    m_Jacobian.rows[ 2 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 2 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 2 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 2 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 2 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, v.x, v.y, v.z );
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 2 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 2 ][ 10] = J4.y;
    m_Jacobian.rows[ 2 ][ 11] = J4.z;
  }
  if ( m_isAngleViolated ) {
    J1.Zero();
    m_Jacobian.rows[ 3 ][ 0 ] = J1.x;
    m_Jacobian.rows[ 3 ][ 1 ] = J1.y;
    m_Jacobian.rows[ 3 ][ 2 ] = J1.z;

    tmp = MatA * Vec4( 0, hingeAxis.x, hingeAxis.y, hingeAxis.z );
    J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 3 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 3 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 3 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 3 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 3 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 3 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, hingeAxis.x, hingeAxis.y, hingeAxis.z );
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 3 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 3 ][ 10] = J4.y;
    m_Jacobian.rows[ 3 ][ 11] = J4.z;
  }

  //
  // Apply warm starting from last frame
  //
  const VecN impulses = m_Jacobian.Transpose() * m_cachedLambda;
  ApplyImpulses( impulses );

  //
  //  Calculate the baumgarte stabilization
  //
  float C = r.Dot( r );
  C = std::max( 0.0f, C - 0.01f );
  const float Beta = 0.05f;
  m_baumgarte = ( Beta / dt_sec ) * C;
}

/*
================================
ConstraintHingeQuatLimited::Solve
================================
*/
void ConstraintHingeQuatLimited::Solve() {
  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities();
  const MatMN invMassMatrix = GetInverseMassMatrix();
  const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
  VecN rhs = m_Jacobian * q_dt * -1.0f;
  rhs[ 0 ] -= m_baumgarte;

  // Solve for the Lagrange multipliers
  VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );
  // Clamp the torque from the angle constraint.
  // We need to make sure it's a restorative torque.
  if ( m_isAngleViolated ) {
    if ( m_relativeAngle > 0.0f ) {
      lambdaN[ 3 ] = std::min( 0.0f, lambdaN[ 3 ] );
    }
    if ( m_relativeAngle < 0.0f ) {
      lambdaN[ 3 ] = std::max( 0.0f, lambdaN[ 3 ] );
    }
  }

  // Apply the impulses
  const VecN impulses = JacobianTranspose * lambdaN;
  ApplyImpulses( impulses );

  // Accumulate the impulses for warm starting
  m_cachedLambda += lambdaN;
}

/*
================================
ConstraintHingeQuatLimited::PostSolve
================================
*/
void ConstraintHingeQuatLimited::PostSolve() {
  // Limit the warm starting to reasonable limits
  for ( int i = 0; i < m_cachedLambda.N; i++ ) {
    if ( i > 0 ) {
      m_cachedLambda[ i ] = 0.0f;
    }

    if ( m_cachedLambda[ i ] * 0.0f != m_cachedLambda[ i ] * 0.0f ) {
      m_cachedLambda[ i ] = 0.0f;
    }
    const float limit = 20.0f;
    if ( m_cachedLambda[ i ] > limit ) {
      m_cachedLambda[ i ] = limit;
    }
    if ( m_cachedLambda[ i ] < -limit ) {
      m_cachedLambda[ i ] = -limit;
    }
  }
}
```

## 19 Constant Velocity Constraint

The next constraint that I want to examine is the constant velocity constraint. This is the typical joint that's used in the axle of front wheel drive vehicles. So, if you like working on cars, then you're already familiar with it. And for everyone else, we'll be using them to represent ball and socket joints.

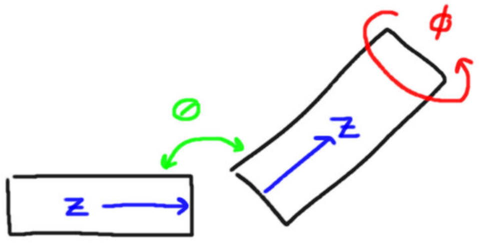


Figure 19: constant velocity constraint

As you can see in figure 19, we can define this constraint with the angles $\theta$ and $\phi$. The constant velocity constraint allows freedom of movement in the $\theta$ direction, but it restricts movement in the $\phi$ direction.

Since we already know that quaternions are a natural fit for orientation based constraints, we will formulate this constraint with quaternions too.

Given two bodies that are aligned along their $\vec{z}_1$ and $\vec{z}_2$ axes. Then we can define the constraint with quaternions as:

$$C = P \cdot (q_r \cdot q_0^*) \cdot \vec{z} = 0 \tag{62}$$

Which of course will result in a very familiar jacobian:

$$J = \left(0, \quad -\tfrac{1}{2}P \cdot L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{z}, \quad 0, \quad \tfrac{1}{2}P \cdot L(q_1^*)R(q_2 \cdot q_0^*) \cdot P^T \cdot \vec{z}\right)$$

And now the code:

```
class ConstraintConstantVelocity : public Constraint {
public:
  ConstraintConstantVelocity() : Constraint(), m_cachedLambda( 2 ), m_Jacobian( 2, 12 ) {
    m_cachedLambda.Zero();
    m_baumgarte = 0.0f;
  }
  void PreSolve( const float dt_sec ) override;
  void Solve() override;
  void PostSolve() override;

  Quat m_q0;  // The initial relative quaternion q1 * q2^-1

  VecN m_cachedLambda;
  MatMN m_Jacobian;

  float m_baumgarte;
};

/*
================================
ConstraintConstantVelocity::PreSolve
================================
*/
void ConstraintConstantVelocity::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );
```

```
// Get the world space position of the hinge from B's orientation
const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

const Vec3 r = worldAnchorB - worldAnchorA;
const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
const Vec3 a = worldAnchorA;
const Vec3 b = worldAnchorB;

// Get the orientation information of the bodies
const Quat q1 = m_bodyA->m_orientation;
const Quat q2 = m_bodyB->m_orientation;
const Quat q0_inv = m_q0.Inverse();
const Quat q1_inv = q1.Inverse();

// This axis is defined in the local space of bodyA
Vec3 u;
Vec3 v;
Vec3 cv = m_axisA;
cv.GetOrtho( u, v );

Mat4 P;
P.rows[ 0 ] = Vec4( 0, 0, 0, 0 );
P.rows[ 1 ] = Vec4( 0, 1, 0, 0 );
P.rows[ 2 ] = Vec4( 0, 0, 1, 0 );
P.rows[ 3 ] = Vec4( 0, 0, 0, 1 );
Mat4 P_T = P.Transpose(); // I know it's pointless to do this with our particular matrix
  implementations.  But I like its self commenting.

const Mat4 MatA = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * -0.5f;
const Mat4 MatB = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * 0.5f;

m_Jacobian.Zero();

//
// Distance constraint
//
Vec3 J1 = ( a - b ) * 2.0f;
m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

Vec3 J3 = ( b - a ) * 2.0f;
m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
m_Jacobian.rows[ 0 ][ 10] = J4.y;
m_Jacobian.rows[ 0 ][ 11] = J4.z;

//
// The quaternion jacobians
//
{
  const int idx = 1;

  Vec4 tmp;

  J1.Zero();
  m_Jacobian.rows[ 1 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 1 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 1 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, cv.x, cv.y, cv.z ) * -0.5f;
```

```
    J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 1 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 1 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 1 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 1 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 1 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 1 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, cv.x, cv.y, cv.z ) * 0.5f;
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 1 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 1 ][ 10] = J4.y;
    m_Jacobian.rows[ 1 ][ 11] = J4.z;
  }

  //
  // Apply warm starting from last frame
  //
  const VecN impulses = m_Jacobian.Transpose() * m_cachedLambda;
  ApplyImpulses( impulses );

  //
  //  Calculate the baumgarte stabilization
  //
  float C = r.Dot( r );
  C = std::max( 0.0f, C - 0.01f );
  const float Beta = 0.05f;
  m_baumgarte = ( Beta / dt_sec ) * C;
}

/*
================================
ConstraintConstantVelocity::Solve
================================
*/
void ConstraintConstantVelocity::Solve() {
  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities();
  const MatMN invMassMatrix = GetInverseMassMatrix();
  const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
  VecN rhs = m_Jacobian * q_dt * -1.0f;
  rhs[ 0 ] -= m_baumgarte;

  // Solve for the Lagrange multipliers
  const VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

  // Apply the impulses
  const VecN impulses = JacobianTranspose * lambdaN;
  ApplyImpulses( impulses );

  // Accumulate the impulses for warm starting
  m_cachedLambda += lambdaN;
}

/*
================================
ConstraintConstantVelocity::PostSolve
================================
*/
void ConstraintConstantVelocity::PostSolve() {
  // Limit the warm starting to reasonable limits
  for ( int i = 0; i < m_cachedLambda.N; i++ ) {
    if ( m_cachedLambda[ i ] * 0.0f != m_cachedLambda[ i ] * 0.0f ) {
      m_cachedLambda[ i ] = 0.0f;
    }
    const float limit = 20.0f;
    if ( m_cachedLambda[ i ] > limit ) {
        m_cachedLambda[ i ] = limit;
      }
      if ( m_cachedLambda[ i ] < -limit ) {
        m_cachedLambda[ i ] = -limit;
      }
    }
}
```

## 20 Limited Constant Velocity Constraint

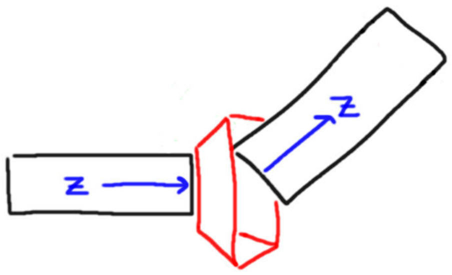


Figure 20: pyramid limit

Of course the next thing we need to do to this constraint is limit it. There's more than one way to limit this constraint, such as a conical limit or an elliptical limit. But the limit that we're going to implement is a pyramid limit (see figure 20).

A good way to think of this constraint is as a reverse limited hinge constraint. In that constraint we had the following three constraints:

$$C_1 = P \cdot (q_r \cdot q_0^*) \cdot \vec{u} = 0 \tag{63}$$
$$C_2 = P \cdot (q_r \cdot q_0^*) \cdot \vec{v} = 0 \tag{64}$$
$$C_3 = P \cdot (q_r \cdot q_0^*) \cdot \vec{h} = 0 \tag{65}$$

Where we always enforced the first two constraints, but only enforced the third when the limits on the angle were violated. And, for the limited cv joint, we're still going to have three constraints:

$$C_1 = P \cdot (q_r \cdot q_0^*) \cdot \vec{u} = 0 \tag{66}$$
$$C_2 = P \cdot (q_r \cdot q_0^*) \cdot \vec{v} = 0 \tag{67}$$
$$C_3 = P \cdot (q_r \cdot q_0^*) \cdot \vec{cv} = 0 \tag{68}$$

But we always enforce the third constraint, and only enforce the first two when they violate their limits.
So naturally, the code should look pretty familiar:

```
class ConstraintConstantVelocityLimited : public Constraint {
public:
  ConstraintConstantVelocityLimited() : Constraint(), m_cachedLambda( 4 ), m_Jacobian( 4, 12 ) {
    m_cachedLambda.Zero();
    m_baumgarte = 0.0f;
    m_isAngleViolatedU = false;
    m_isAngleViolatedV = false;
    m_angleU = 0.0f;
    m_angleV = 0.0f;
  }
  void PreSolve( const float dt_sec ) override;
  void Solve() override;
  void PostSolve() override;

  Quat m_q0; // The initial relative quaternion q1^-1 * q2

  VecN m_cachedLambda;
  MatMN m_Jacobian;

  float m_baumgarte;

  bool m_isAngleViolatedU;
  bool m_isAngleViolatedV;
  float m_angleU;
  float m_angleV;
};

/*
================================
ConstraintConstantVelocityLimited::PreSolve
================================
*/
void ConstraintConstantVelocityLimited::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 r = worldAnchorB - worldAnchorA;
  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;

  // Get the orientation information of the bodies
  const Quat q1 = m_bodyA->m_orientation;
  const Quat q2 = m_bodyB->m_orientation;
  const Quat q0_inv = m_q0.Inverse();
  const Quat q1_inv = q1.Inverse();

  // This axis is defined in the local space of bodyA
  Vec3 u;
  Vec3 v;
  Vec3 cv = m_axisA;
  cv.GetOrtho( u, v );

  Mat4 P;
  P.rows[ 0 ] = Vec4( 0, 0, 0, 0 );
  P.rows[ 1 ] = Vec4( 0, 1, 0, 0 );
  P.rows[ 2 ] = Vec4( 0, 0, 1, 0 );
  P.rows[ 3 ] = Vec4( 0, 0, 0, 1 );
  const Mat4 P_T = P.Transpose(); // I know it's pointless to do this with our particular matrix
    implementations.  But I like its self commenting.

  const Mat4 MatA = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * -0.5f;
  const Mat4 MatB = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * 0.5f;

  // Check the constraint's angular limits
  const float pi = acosf( -1.0f );
  const Quat qr = q1_inv * q2;
  const Quat qrr = qr * q0_inv;
  m_angleU = 2.0f * asinf( qrr.xyz().Dot( u ) ) * 180.0f / pi;
  m_angleV = 2.0f * asinf( qrr.xyz().Dot( v ) ) * 180.0f / pi;

  m_isAngleViolatedU = false;
  if ( m_angleU > 45.0f ) {
    m_isAngleViolatedU = true;
  }
  if ( m_angleU < -45.0f ) {
    m_isAngleViolatedU = true;
  }

  m_isAngleViolatedV = false;
  if ( m_angleV > 45.0f ) {
    m_isAngleViolatedV = true;
  }
  if ( m_angleV < -45.0f ) {
    m_isAngleViolatedV = true;
  }

  //
  // First row is the primary distance constraint that holds the anchor points together
  //
```

```
m_Jacobian.Zero();

Vec3 J1 = ( a - b ) * 2.0f;
m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

Vec3 J3 = ( b - a ) * 2.0f;
m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
m_Jacobian.rows[ 0 ][ 10] = J4.y;
m_Jacobian.rows[ 0 ][ 11] = J4.z;

//
// The quaternion jacobians
//
{
  const int idx = 1;

  Vec4 tmp;

  J1.Zero();
  m_Jacobian.rows[ 1 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 1 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 1 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, cv.x, cv.y, cv.z ) * -0.5f;
  J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 1 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 1 ][ 5 ] = J2.z;

  J3.Zero();
  m_Jacobian.rows[ 1 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 1 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 1 ][ 8 ] = J3.z;

  tmp = MatB * Vec4( 0, cv.x, cv.y, cv.z ) * 0.5f;
  J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 9 ] = J4.x;
  m_Jacobian.rows[ 1 ][ 10] = J4.y;
  m_Jacobian.rows[ 1 ][ 11] = J4.z;
}

if ( m_isAngleViolatedU ) {
  const int idx = 1;
  Vec4 tmp;

  J1.Zero();
  m_Jacobian.rows[ 2 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 2 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 2 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, u.x, u.y, u.z ) * -0.5f;
  J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 2 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 2 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 2 ][ 5 ] = J2.z;

  J3.Zero();
  m_Jacobian.rows[ 2 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 2 ][ 7 ] = J3.y;
```

```
    m_Jacobian.rows[ 2 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, u.x, u.y, u.z ) * 0.5f;
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 2 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 2 ][ 10] = J4.y;
    m_Jacobian.rows[ 2 ][ 11] = J4.z;
  }

  if ( m_isAngleViolatedV ) {
    const int idx = 1;
    Vec4 tmp;

    J1.Zero();
    m_Jacobian.rows[ 3 ][ 0 ] = J1.x;
    m_Jacobian.rows[ 3 ][ 1 ] = J1.y;
    m_Jacobian.rows[ 3 ][ 2 ] = J1.z;

    tmp = MatA * Vec4( 0, v.x, v.y, v.z ) * -0.5f;
    J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 3 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 3 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 3 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 3 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 3 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 3 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, v.x, v.y, v.z ) * 0.5f;
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 3 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 3 ][ 10] = J4.y;
    m_Jacobian.rows[ 3 ][ 11] = J4.z;
  }

  //
  // Apply warm starting from last frame
  //
  const VecN impulses = m_Jacobian.Transpose() * m_cachedLambda;
  ApplyImpulses( impulses );

  //
  //  Calculate the baumgarte stabilization
  //
  float C = r.Dot( r );
  C = std::max( 0.0f, C - 0.01f );
  const float Beta = 0.05f;
  m_baumgarte = ( Beta / dt_sec ) * C;
}

/*
================================
ConstraintConstantVelocityLimited::Solve
================================
*/
void ConstraintConstantVelocityLimited::Solve() {
  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities();
  const MatMN invMassMatrix = GetInverseMassMatrix();
  const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
  VecN rhs = m_Jacobian * q_dt * -1.0f;
  rhs[ 0 ] -= m_baumgarte;

  // Solve for the Lagrange multipliers
  VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

  // Clamp the torque from the angle constraint.
  // We need to make sure it's a restorative torque.
  if ( m_isAngleViolatedU ) {
    if ( m_angleU > 0.0f ) {
      lambdaN[ 2 ] = std::min( 0.0f, lambdaN[ 2 ] );
    }
    if ( m_angleU < 0.0f ) {
      lambdaN[ 2 ] = std::max( 0.0f, lambdaN[ 2 ] );
    }
  }
  if ( m_isAngleViolatedV ) {
    if ( m_angleV > 0.0f ) {
      lambdaN[ 3 ] = std::min( 0.0f, lambdaN[ 3 ] );
    }
    if ( m_angleV < 0.0f ) {
      lambdaN[ 3 ] = std::max( 0.0f, lambdaN[ 3 ] );
    }
  }

  // Apply the impulses
  const VecN impulses = JacobianTranspose * lambdaN;
  ApplyImpulses( impulses );

  // Accumulate the impulses for warm starting
  m_cachedLambda += lambdaN;
}

/*
================================
ConstraintConstantVelocityLimited::PostSolve
================================
*/
void ConstraintConstantVelocityLimited::PostSolve() {
  // Limit the warm starting to reasonable limits
  for ( int i = 0; i < m_cachedLambda.N; i++ ) {
    if ( i > 0 ) {
      m_cachedLambda[ i ] = 0.0f;
    }

    if ( m_cachedLambda[ i ] * 0.0f != m_cachedLambda[ i ] * 0.0f ) {
      m_cachedLambda[ i ] = 0.0f;
    }
    const float limit = 20.0f;
    if ( m_cachedLambda[ i ] > limit ) {
      m_cachedLambda[ i ] = limit;
    }
    if ( m_cachedLambda[ i ] < -limit ) {
      m_cachedLambda[ i ] = -limit;
    }
  }
}
```

Running this code, we can now properly emulate shoulders, hips, and necks in ragdolls.

# 21 Ragdoll

This chapter actually introduces nothing new. So feel free to skip it. The only thing we will be doing here is setting up some bodies and constraints to give us a very very basic ragdoll.

Here's the code:

```
static const float t2 = 0.25f;
static const float w2 = t2 * 2.0f;
static const float h3 = t2 * 4.0f;
Vec3 g_boxBody[] = {
  Vec3(-t2,-w2,-h3 ),
  Vec3( t2,-w2,-h3 ),
  Vec3(-t2, w2,-h3 ),
  Vec3( t2, w2,-h3 ),

  Vec3(-t2,-w2, h3 ),
  Vec3( t2,-w2, h3 ),
  Vec3(-t2, w2, h3 ),
  Vec3( t2, w2, h3 ),
};

static const float h2 = 0.25f;
Vec3 g_boxLimb[] = {
  Vec3(-h3,-h2,-h2 ),
  Vec3( h3,-h2,-h2 ),
  Vec3(-h3, h2,-h2 ),
  Vec3( h3, h2,-h2 ),

  Vec3(-h3,-h2, h2 ),
  Vec3( h3,-h2, h2 ),
  Vec3(-h3, h2, h2 ),
  Vec3( h3, h2, h2 ),
};


void Scene::Initialize() {
  Body body;

  //
  //  Build a ragdoll
  //

  // head
  body.m_position = Vec3( 0, 0, 5.5f );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
  body.m_invMass = 2.0f;
  body.m_elasticity = 1.0f;
  body.m_friction = 1.0f;
  m_bodies.push_back( body );

  // torso
  body.m_position = Vec3( 0, 0, 4 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeBox( g_boxBody, sizeof( g_boxBody ) / sizeof( Vec3 ) );
  body.m_invMass = 0.5f;
  body.m_elasticity = 1.0f;
  body.m_friction = 1.0f;
  m_bodies.push_back( body );

  // left arm
  body.m_position = Vec3( 0.0f, 2.0f, 4.75f );
  body.m_orientation = Quat( Vec3( 0, 0, 1 ), -3.1415f / 2.0f );
  body.m_shape = new ShapeBox( g_boxLimb, sizeof( g_boxLimb ) / sizeof( Vec3 ) );
  body.m_invMass = 1.0f;
  body.m_elasticity = 1.0f;
  body.m_friction = 1.0f;
  m_bodies.push_back( body );

  // right arm
```

```
body.m_position = Vec3( 0.0f, -2.0f, 4.75f );
body.m_orientation = Quat( Vec3( 0, 0, 1 ), 3.1415f / 2.0f );
body.m_shape = new ShapeBox( g_boxLimb, sizeof( g_boxLimb ) / sizeof( Vec3 ) );
body.m_invMass = 1.0f;
body.m_elasticity = 1.0f;
body.m_friction = 1.0f;
m_bodies.push_back( body );

// left leg
body.m_position = Vec3( 0.0f, 1.0f, 2.5f );
body.m_orientation = Quat( Vec3( 0, 1, 0 ), 3.1415f / 2.0f );
body.m_shape = new ShapeBox( g_boxLimb, sizeof( g_boxLimb ) / sizeof( Vec3 ) );
body.m_invMass = 1.0f;
body.m_elasticity = 1.0f;
body.m_friction = 1.0f;
m_bodies.push_back( body );

// right leg
body.m_position = Vec3( 0.0f, -1.0f, 2.5f );
body.m_orientation = Quat( Vec3( 0, 1, 0 ), 3.1415f / 2.0f );
body.m_shape = new ShapeBox( g_boxLimb, sizeof( g_boxLimb ) / sizeof( Vec3 ) );
body.m_invMass = 1.0f;
body.m_elasticity = 1.0f;
body.m_friction = 1.0f;
m_bodies.push_back( body );

const int idxHead = 0;
const int idxTorso = 1;
const int idxArmLeft = 2;
const int idxArmRight = 3;
const int idxLegLeft = 4;
const int idxLegRight = 5;

// Neck
{
  ConstraintHingeQuatLimited * joint = new ConstraintHingeQuatLimited();
  joint->m_bodyA = &m_bodies[ idxHead ];
  joint->m_bodyB = &m_bodies[ idxTorso ];

  const Vec3 jointWorldSpaceAnchor  = joint->m_bodyA->m_position + Vec3( 0, 0, -0.5f );
  joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
  joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

  joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 1, 0 ) );

  // Set the initial relative orientation
  joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

  m_constraints.push_back( joint );
}

// Shoulder Left
{
  ConstraintConstantVelocityLimited * joint = new ConstraintConstantVelocityLimited();
  joint->m_bodyB = &m_bodies[ idxArmLeft ];
  joint->m_bodyA = &m_bodies[ idxTorso ];

  const Vec3 jointWorldSpaceAnchor  = joint->m_bodyB->m_position + Vec3( 0, -1.0f, 0.0f );
  joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
  joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

  joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 1, 0 ) );

  // Set the initial relative orientation
  joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

  m_constraints.push_back( joint );
}

// Shoulder Right
{
    ConstraintConstantVelocityLimited * joint = new ConstraintConstantVelocityLimited();
    joint->m_bodyB = &m_bodies[ idxArmRight ];
    joint->m_bodyA = &m_bodies[ idxTorso ];

    const Vec3 jointWorldSpaceAnchor  = joint->m_bodyB->m_position + Vec3( 0, 1.0f, 0.0f );
    joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
    joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

    joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, -1, 0 ) );

    // Set the initial relative orientation
    joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

    m_constraints.push_back( joint );
  }

  // Hip Left
  {
    ConstraintHingeQuatLimited * joint = new ConstraintHingeQuatLimited();
    joint->m_bodyB = &m_bodies[ idxLegLeft ];
    joint->m_bodyA = &m_bodies[ idxTorso ];

    const Vec3 jointWorldSpaceAnchor  = joint->m_bodyB->m_position + Vec3( 0, 0, 0.5f );
    joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
    joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

    joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 1, 0 ) );

    // Set the initial relative orientation
    joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

    m_constraints.push_back( joint );
  }

  // Hip Right
  {
    ConstraintHingeQuatLimited * joint = new ConstraintHingeQuatLimited();
    joint->m_bodyB = &m_bodies[ idxLegRight ];
    joint->m_bodyA = &m_bodies[ idxTorso ];

    const Vec3 jointWorldSpaceAnchor  = joint->m_bodyB->m_position + Vec3( 0, 0, 0.5f );
    joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
    joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

    joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 1, 0 ) );

    // Set the initial relative orientation
    joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

    m_constraints.push_back( joint );
  }

  //
  //  Standard floor and walls
  //
  AddStandardSandBox( m_bodies );
}
```

And in action:

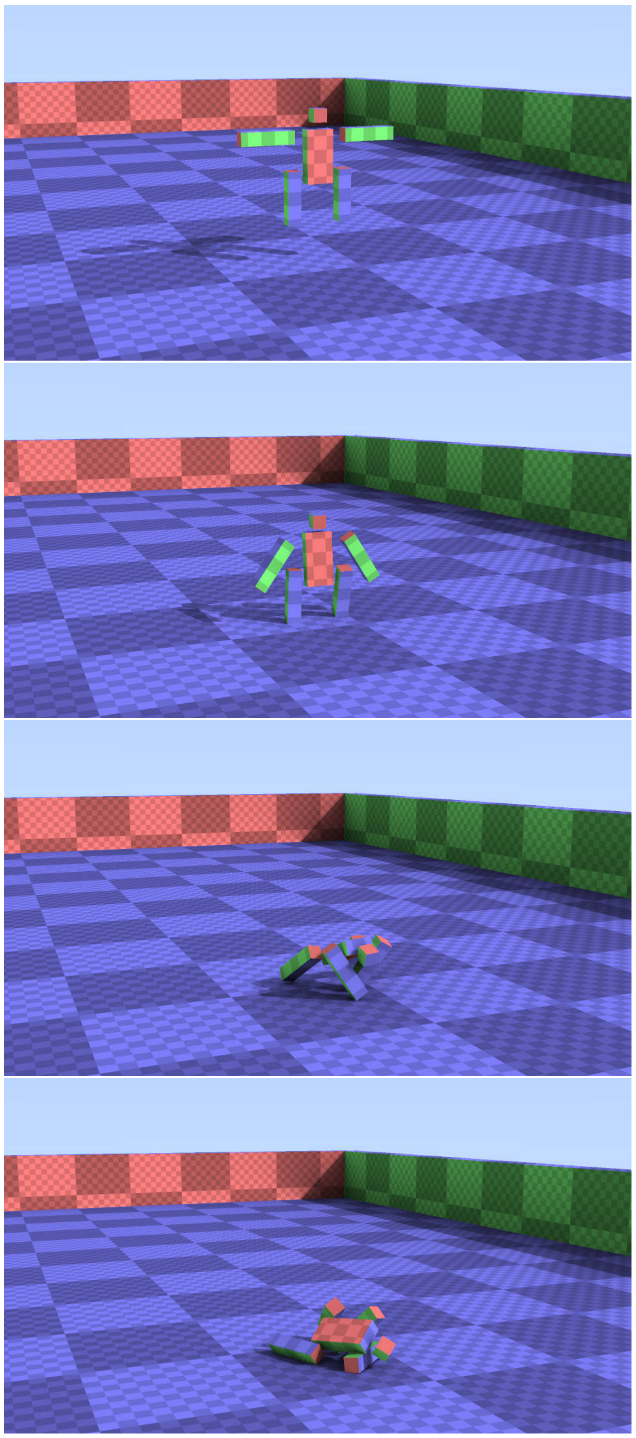


Figure 21: Ragdoll

## 22 Motors

Now that we've got ragdolls behind us; I'd like to discuss motors briefly. A motor can be implemented with the same framework we've built for constraints. The only difference between motors and constraints, is that instead of solving for a zero velocity, we solve for a constant velocity. Which only slightly modifies our equations:

$$\dot{C} = c_0 \tag{69}$$

And that's it. If we want the constraint to move/rotate at a constant rate, then we only need to set the velocity constraint to a non-zero constant. Let's modify the quaternion constraint to create a basic motor:

```
class ConstraintMotor : public Constraint {
public:
  ConstraintMotor() : Constraint(), m_Jacobian( 4, 12 ) {
    m_motorSpeed = 0.0f;
    m_motorAxis = Vec3( 0, 0, 1 );
    m_baumgarte = 0.0f;
  }

  void PreSolve( const float dt_sec ) override;
  void Solve() override;

  float m_motorSpeed;
  Vec3 m_motorAxis; // Motor Axis in BodyA's local space
  Quat m_q0;     // The initial relative quaternion q1^-1 * q2

  MatMN m_Jacobian;

  Vec3 m_baumgarte;
};

/*
================================
ConstraintMotor::PreSolve
================================
*/
void ConstraintMotor::PreSolve( const float dt_sec ) {
  // Get the world space position of the hinge from A's orientation
  const Vec3 worldAnchorA = m_bodyA->BodySpaceToWorldSpace( m_anchorA );

  // Get the world space position of the hinge from B's orientation
  const Vec3 worldAnchorB = m_bodyB->BodySpaceToWorldSpace( m_anchorB );

  const Vec3 r = worldAnchorB - worldAnchorA;
  const Vec3 ra = worldAnchorA - m_bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = worldAnchorB - m_bodyB->GetCenterOfMassWorldSpace();
  const Vec3 a = worldAnchorA;
  const Vec3 b = worldAnchorB;

  // Get the orientation information of the bodies
  const Quat q1 = m_bodyA->m_orientation;
  const Quat q2 = m_bodyB->m_orientation;
  const Quat q0_inv = m_q0.Inverse();
  const Quat q1_inv = q1.Inverse();

  const Vec3 motorAxis = m_bodyA->m_orientation.RotatePoint( m_motorAxis );
  Vec3 motorU;
  Vec3 motorV;
  motorAxis.GetOrtho( motorU, motorV );

  const Vec3 u = motorU;
  const Vec3 v = motorV;
  const Vec3 w = motorAxis;

  Mat4 P;
  P.rows[ 0 ] = Vec4( 0, 0, 0, 0 );
  P.rows[ 1 ] = Vec4( 0, 1, 0, 0 );
```

```
P.rows[ 2 ] = Vec4( 0, 0, 1, 0 );
P.rows[ 3 ] = Vec4( 0, 0, 0, 1 );
Mat4 P_T = P.Transpose(); // I know it's pointless to do this with our particular matrix
  implementations.  But I like its self commenting.

const Mat4 MatA = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * -0.5f;
const Mat4 MatB = P * Left( q1_inv ) * Right( q2 * q0_inv ) * P_T * 0.5f;

//
//  The distance constraint
//

m_Jacobian.Zero();

// First row is the primary distance constraint that holds the anchor points together
Vec3 J1 = ( a - b ) * 2.0f;
m_Jacobian.rows[ 0 ][ 0 ] = J1.x;
m_Jacobian.rows[ 0 ][ 1 ] = J1.y;
m_Jacobian.rows[ 0 ][ 2 ] = J1.z;

Vec3 J2 = ra.Cross( ( a - b ) * 2.0f );
m_Jacobian.rows[ 0 ][ 3 ] = J2.x;
m_Jacobian.rows[ 0 ][ 4 ] = J2.y;
m_Jacobian.rows[ 0 ][ 5 ] = J2.z;

Vec3 J3 = ( b - a ) * 2.0f;
m_Jacobian.rows[ 0 ][ 6 ] = J3.x;
m_Jacobian.rows[ 0 ][ 7 ] = J3.y;
m_Jacobian.rows[ 0 ][ 8 ] = J3.z;

Vec3 J4 = rb.Cross( ( b - a ) * 2.0f );
m_Jacobian.rows[ 0 ][ 9 ] = J4.x;
m_Jacobian.rows[ 0 ][ 10] = J4.y;
m_Jacobian.rows[ 0 ][ 11] = J4.z;

//
// The quaternion jacobians
//
const int idx = 1;

Vec4 tmp;
{
  J1.Zero();
  m_Jacobian.rows[ 1 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 1 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 1 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, u.x, u.y, u.z );
  J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 3 ] = J2.x;
  m_Jacobian.rows[ 1 ][ 4 ] = J2.y;
  m_Jacobian.rows[ 1 ][ 5 ] = J2.z;

  J3.Zero();
  m_Jacobian.rows[ 1 ][ 6 ] = J3.x;
  m_Jacobian.rows[ 1 ][ 7 ] = J3.y;
  m_Jacobian.rows[ 1 ][ 8 ] = J3.z;

  tmp = MatB * Vec4( 0, u.x, u.y, u.z );
  J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
  m_Jacobian.rows[ 1 ][ 9 ] = J4.x;
  m_Jacobian.rows[ 1 ][ 10] = J4.y;
  m_Jacobian.rows[ 1 ][ 11] = J4.z;
}
{
  J1.Zero();
  m_Jacobian.rows[ 2 ][ 0 ] = J1.x;
  m_Jacobian.rows[ 2 ][ 1 ] = J1.y;
  m_Jacobian.rows[ 2 ][ 2 ] = J1.z;

  tmp = MatA * Vec4( 0, v.x, v.y, v.z );
```

```
    J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 2 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 2 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 2 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 2 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 2 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 2 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, v.x, v.y, v.z );
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 2 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 2 ][ 10] = J4.y;
    m_Jacobian.rows[ 2 ][ 11] = J4.z;
  }
  {
    J1.Zero();
    m_Jacobian.rows[ 3 ][ 0 ] = J1.x;
    m_Jacobian.rows[ 3 ][ 1 ] = J1.y;
    m_Jacobian.rows[ 3 ][ 2 ] = J1.z;

    tmp = MatA * Vec4( 0, w.x, w.y, w.z );
    J2 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 3 ][ 3 ] = J2.x;
    m_Jacobian.rows[ 3 ][ 4 ] = J2.y;
    m_Jacobian.rows[ 3 ][ 5 ] = J2.z;

    J3.Zero();
    m_Jacobian.rows[ 3 ][ 6 ] = J3.x;
    m_Jacobian.rows[ 3 ][ 7 ] = J3.y;
    m_Jacobian.rows[ 3 ][ 8 ] = J3.z;

    tmp = MatB * Vec4( 0, w.x, w.y, w.z );
    J4 = Vec3( tmp[ idx + 0 ], tmp[ idx + 1 ], tmp[ idx + 2 ] );
    m_Jacobian.rows[ 3 ][ 9 ] = J4.x;
    m_Jacobian.rows[ 3 ][ 10] = J4.y;
    m_Jacobian.rows[ 3 ][ 11] = J4.z;
  }

  //
  //  Calculate the baumgarte stabilization
  //
  const float Beta = 0.05f;
  const float C = r.Dot( r );

  const Quat qr = m_bodyA->m_orientation.Inverse() * m_bodyB->m_orientation;
  const Quat qrA = qr * q0_inv; // Relative orientation in BodyA's space

  // Get the world space axis for the relative rotation
  const Vec3 axisA = m_bodyA->m_orientation.RotatePoint( qrA.xyz() );

  m_baumgarte.Zero();
  m_baumgarte[ 0 ] = ( Beta / dt_sec ) * C;
  m_baumgarte[ 1 ] = motorU.Dot( axisA ) * ( Beta / dt_sec );
  m_baumgarte[ 2 ] = motorV.Dot( axisA ) * ( Beta / dt_sec );
}

/*
================================
ConstraintMotor::Solve
================================
*/
void ConstraintMotor::Solve() {
  const Vec3 motorAxis = m_bodyA->m_orientation.RotatePoint( m_motorAxis );

  VecN w_dt( 12 );
  w_dt.Zero();
  w_dt[ 3 ] = motorAxis[ 0 ] * -m_motorSpeed;
  w_dt[ 4 ] = motorAxis[ 1 ] * -m_motorSpeed;
  w_dt[ 5 ] = motorAxis[ 2 ] * -m_motorSpeed;
  w_dt[ 9 ] = motorAxis[ 0 ] * m_motorSpeed;
  w_dt[ 10 ] = motorAxis[ 1 ] * m_motorSpeed;
  w_dt[ 11 ] = motorAxis[ 2 ] * m_motorSpeed;

  const MatMN JacobianTranspose = m_Jacobian.Transpose();

  // Build the system of equations
  const VecN q_dt = GetVelocities() - w_dt; // By subtracting by the desired velocity, the solver
    is tricked into applying the impulse to give us that velocity
  const MatMN invMassMatrix = GetInverseMassMatrix();
  const MatMN J_W_Jt = m_Jacobian * invMassMatrix * JacobianTranspose;
  VecN rhs = m_Jacobian * q_dt * -1.0f;
  for ( int i = 0; i < 3; i++ ) {
    rhs[ i ] -= m_baumgarte[ i ];
  }

  // Solve for the Lagrange multipliers
  VecN lambdaN = LCP_GaussSeidel( J_W_Jt, rhs );

  // Apply the impulses
  const VecN impulses = JacobianTranspose * lambdaN;
  ApplyImpulses( impulses );
}
```

The above motor is just a simple rotary motor that applies whatever impulse is required to get the bodies to rotate at a constant velocity relative to each other. But all kinds of different motors can be developed with the same principles.

Now let's see some action:

```
void Scene::Initialize() {
  Body body;

  //
  //  Motor
  //
  Vec3 motorPos = Vec3( 5, 0, 2 );
  Vec3 motorAxis = Vec3( 0, 0, 1 ).Normalize();
  Quat motorOrient = Quat( 1, 0, 0, 0 );

  body.m_position = motorPos;
  body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
  body.m_invMass = 0.0f;
  body.m_elasticity = 0.9f;
  body.m_friction = 0.5f;
  m_bodies.push_back( body );

  body.m_position = motorPos - motorAxis;
  body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
  body.m_orientation = motorOrient;
  body.m_shape = new ShapeBox( g_boxBeam, sizeof( g_boxBeam ) / sizeof( Vec3 ) );
  body.m_invMass = 0.01f;
  body.m_elasticity = 1.0f;
  body.m_friction = 0.5f;
  m_bodies.push_back( body );
  {
    ConstraintMotor * joint = new ConstraintMotor();
    joint->m_bodyA = &m_bodies[ m_bodies.size() - 2 ];
    joint->m_bodyB = &m_bodies[ m_bodies.size() - 1 ];

    const Vec3 jointWorldSpaceAnchor  = joint->m_bodyA->m_position;
    joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
    joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

    joint->m_motorSpeed = 2.0f;
    joint->m_motorAxis  = joint->m_bodyA->m_orientation.Inverse().RotatePoint( motorAxis );

    // Set the initial relative orientation (in bodyA's space)
    joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;
```

```
        m_constraints.push_back( joint );
    }

    body.m_position = motorPos + Vec3( 2, 0, 2 );
    body.m_linearVelocity = Vec3( 0, 0, 0 );
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_shape = new ShapeSphere( 1.0f );
    body.m_invMass = 1.0f;
    body.m_elasticity = 0.1f;
    body.m_friction = 0.9f;
    m_bodies.push_back( body );

    //
    //  Standard floor and walls
    //
    AddStandardSandBox( m_bodies );
}
```

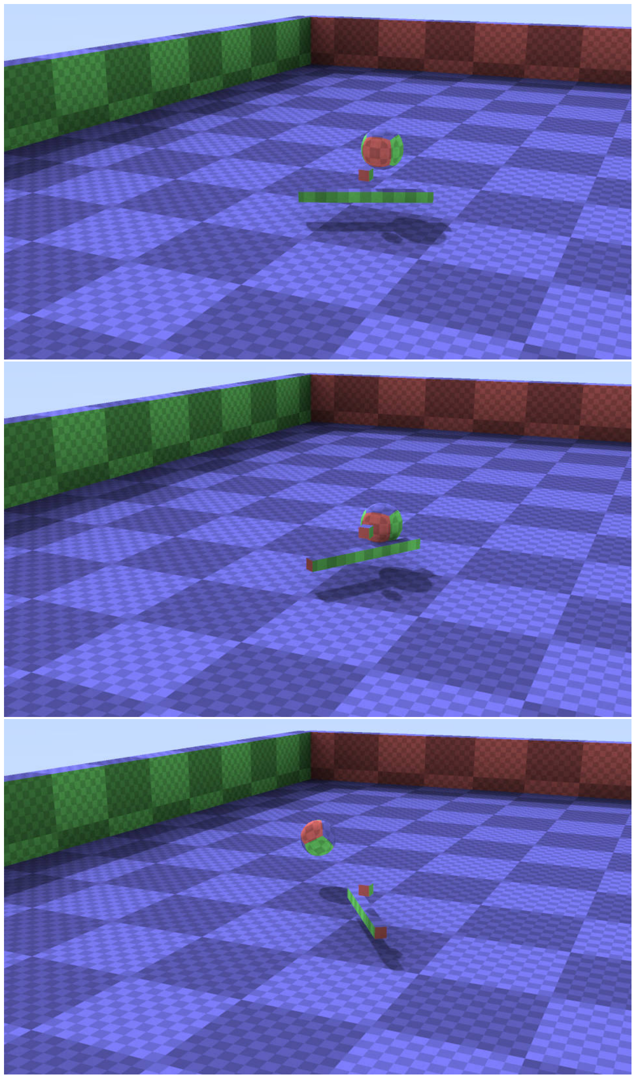


Figure 22: Motor

# 23 Movers

Now, motors can be great for a lot of different applications. But, they can suffer from the same stability issues all constraints can have. And sometimes all you want is a reliable moving platform. The simplest way to do that, is to directly set the velocity of an object with infinite mass.

A quick and dirty way for us to implement one of these is to just hack it into our constraint system. However, if you're making a game, your designers are likely to want a lot more control. In fact, they'll probably want to be able to lay down a spline and add custom commands.

But for our needs here, we just need to look at a simple example. So, let's make another custom constraint that controls a very simple mover.

```
class ConstraintMoverSimple : public Constraint {
public:
  ConstraintMoverSimple() : Constraint(), m_time( 0 ) {}

  void PreSolve( const float dt_sec ) override;

  float m_time;
};

void ConstraintMoverSimple::PreSolve( const float dt_sec ) {
  m_time += dt_sec;
  m_bodyA->m_linearVelocity.y = cosf( m_time * 0.25f ) * 4.0f;
}
```

And to test it, we'll need a new shape:

```
static const float t = 0.25f;
static const float l = 3.0f;

Vec3 g_boxPlatform[] = {
  Vec3(-l,-l,-t ),
  Vec3( l,-l,-t ),
  Vec3(-l, l,-t ),
  Vec3( l, l,-t ),

  Vec3(-l,-l, t ),
  Vec3( l,-l, t ),
  Vec3(-l, l, t ),
  Vec3( l, l, t ),
};
```

And finally adding some bodies:

```
void Scene::Initialize() {
  Body body;

  //
  //  Mover
  //
  body.m_position = Vec3( 10, 0, 5 );
  body.m_linearVelocity = Vec3( 0, 0, 0 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeBox( g_boxPlatform, sizeof( g_boxPlatform ) / sizeof( Vec3 ) );
  body.m_invMass = 0.0f;
  body.m_elasticity = 0.1f;
  body.m_friction = 0.9f;
  m_bodies.push_back( body );
  {
    ConstraintMoverSimple * mover = new ConstraintMoverSimple();
    mover->m_bodyA = &m_bodies[ m_bodies.size() - 1 ];

    m_constraints.push_back( mover );
  }

  body.m_position = Vec3( 10, 0, 6.3f );
  body.m_linearVelocity = Vec3( 0, 0, 0 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeBox( g_boxUnit, sizeof( g_boxUnit ) / sizeof( Vec3 ) );
  body.m_invMass = 1.0f;
  body.m_elasticity = 0.1f;
  body.m_friction = 0.9f;
  m_bodies.push_back( body );

  //
  //  Standard floor and walls
  //
  AddStandardSandBox( m_bodies );
}
```

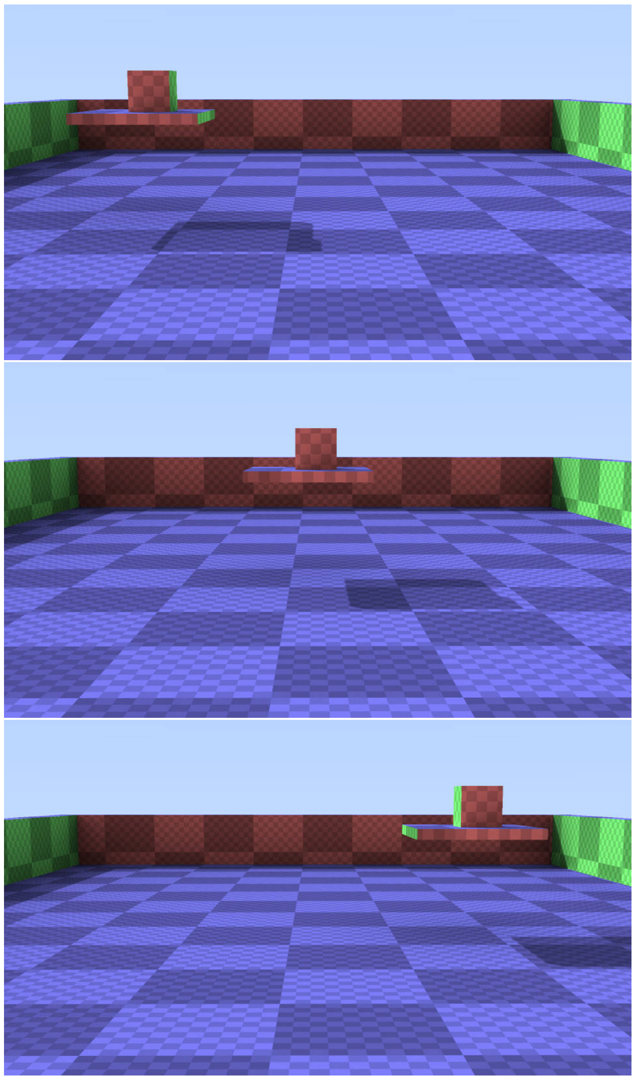


Figure 23: Mover

## 24 Conclusion

We now have all the features of a basic rigid body physics simulation. So, congratulations to yourself!

If all you wanted was to get a deeper understanding of how a physics engine might work under the hood, then hopefully you've gained that insight. And if you're interested in doing more physics programming, then now you have a nice little sandbox that you can extend for your own experimentation.

And just for funsies, let's setup a scene with everything in it!

```
void Scene::Initialize() {
  const float pi = acosf( -1.0f );
  Body body;

  //
  //  Build a ragdoll
  //
  {
    Vec3 offset = Vec3( -5, 0, 0 );

    // head
    body.m_position = Vec3( 0, 0, 5.5f ) + offset;
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
    body.m_invMass = 2.0f;
    body.m_elasticity = 1.0f;
    body.m_friction = 1.0f;
    m_bodies.push_back( body );

    // torso
    body.m_position = Vec3( 0, 0, 4 ) + offset;
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_shape = new ShapeBox( g_boxBody, sizeof( g_boxBody ) / sizeof( Vec3 ) );
    body.m_invMass = 0.5f;
    body.m_elasticity = 1.0f;
    body.m_friction = 1.0f;
    m_bodies.push_back( body );

    // left arm
    body.m_position = Vec3( 0.0f, 2.0f, 4.75f ) + offset;
    body.m_orientation = Quat( Vec3( 0, 0, 1 ), -3.1415f / 2.0f );
    body.m_shape = new ShapeBox( g_boxLimb, sizeof( g_boxLimb ) / sizeof( Vec3 ) );
    body.m_invMass = 1.0f;
    body.m_elasticity = 1.0f;
    body.m_friction = 1.0f;
    m_bodies.push_back( body );

    // right arm
    body.m_position = Vec3( 0.0f, -2.0f, 4.75f ) + offset;
    body.m_orientation = Quat( Vec3( 0, 0, 1 ), 3.1415f / 2.0f );
    body.m_shape = new ShapeBox( g_boxLimb, sizeof( g_boxLimb ) / sizeof( Vec3 ) );
    body.m_invMass = 1.0f;
    body.m_elasticity = 1.0f;
    body.m_friction = 1.0f;
    m_bodies.push_back( body );

    // left leg
    body.m_position = Vec3( 0.0f, 1.0f, 2.5f ) + offset;
    body.m_orientation = Quat( Vec3( 0, 1, 0 ), 3.1415f / 2.0f );
    body.m_shape = new ShapeBox( g_boxLimb, sizeof( g_boxLimb ) / sizeof( Vec3 ) );
    body.m_invMass = 1.0f;
    body.m_elasticity = 1.0f;
    body.m_friction = 1.0f;
    m_bodies.push_back( body );

    // right leg
    body.m_position = Vec3( 0.0f, -1.0f, 2.5f ) + offset;
    body.m_orientation = Quat( Vec3( 0, 1, 0 ), 3.1415f / 2.0f );
    body.m_shape = new ShapeBox( g_boxLimb, sizeof( g_boxLimb ) / sizeof( Vec3 ) );
    body.m_invMass = 1.0f;
    body.m_elasticity = 1.0f;
```

```
	body.m_friction = 1.0f;
	m_bodies.push_back( body );

	const int idxHead = 0;
	const int idxTorso = 1;
	const int idxArmLeft = 2;
	const int idxArmRight = 3;
	const int idxLegLeft = 4;
	const int idxLegRight = 5;

	// Neck
	{
		ConstraintHingeQuatLimited * joint = new ConstraintHingeQuatLimited();
		joint->m_bodyA = &m_bodies[ idxHead ];
		joint->m_bodyB = &m_bodies[ idxTorso ];

		const Vec3 jointWorldSpaceAnchor  = joint->m_bodyA->m_position + Vec3( 0, 0, -0.5f );
		joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
		joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

		joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 1, 0 ) );

		// Set the initial relative orientation
		joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

		m_constraints.push_back( joint );
	}

	// Shoulder Left
	{
		ConstraintConstantVelocityLimited * joint = new ConstraintConstantVelocityLimited();
		joint->m_bodyB = &m_bodies[ idxArmLeft ];
		joint->m_bodyA = &m_bodies[ idxTorso ];

		const Vec3 jointWorldSpaceAnchor  = joint->m_bodyB->m_position + Vec3( 0, -1.0f, 0.0f );
		joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
		joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

		joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 1, 0 ) );

		// Set the initial relative orientation
		joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

		m_constraints.push_back( joint );
	}

	// Shoulder Right
	{
		ConstraintConstantVelocityLimited * joint = new ConstraintConstantVelocityLimited();
		joint->m_bodyB = &m_bodies[ idxArmRight ];
		joint->m_bodyA = &m_bodies[ idxTorso ];

		const Vec3 jointWorldSpaceAnchor  = joint->m_bodyB->m_position + Vec3( 0, 1.0f, 0.0f );
		joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
		joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

		joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, -1, 0 ) );

		// Set the initial relative orientation
		joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

		m_constraints.push_back( joint );
	}

	// Hip Left
	{
		ConstraintHingeQuatLimited * joint = new ConstraintHingeQuatLimited();
		joint->m_bodyB = &m_bodies[ idxLegLeft ];
		joint->m_bodyA = &m_bodies[ idxTorso ];

		const Vec3 jointWorldSpaceAnchor  = joint->m_bodyB->m_position + Vec3( 0, 0, 0.5f );
		joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
		joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

		joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 1, 0 ) );

		// Set the initial relative orientation
		joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

		m_constraints.push_back( joint );
	}

	// Hip Right
	{
		ConstraintHingeQuatLimited * joint = new ConstraintHingeQuatLimited();
		joint->m_bodyB = &m_bodies[ idxLegRight ];
		joint->m_bodyA = &m_bodies[ idxTorso ];

		const Vec3 jointWorldSpaceAnchor  = joint->m_bodyB->m_position + Vec3( 0, 0, 0.5f );
		joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
		joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

		joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 1, 0 ) );

		// Set the initial relative orientation
		joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

		m_constraints.push_back( joint );
	}
}

//
// Build a chain for funsies
//
const int numJoints = 5;
for ( int i = 0; i < numJoints; i++ ) {
	if ( i == 0 ) {
		body.m_position = Vec3( 0.0f, 5.0f, (float)numJoints + 3.0f );
		body.m_orientation = Quat( 0, 0, 0, 1 );
		body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
		body.m_invMass = 0.0f;
		body.m_elasticity = 1.0f;
		m_bodies.push_back( body );
	} else {
		body.m_invMass = 1.0f;
	}

	body.m_linearVelocity = Vec3( 0, 0, 0 );

	Body * bodyA = &m_bodies[ m_bodies.size() - 1 ];

	const Vec3 jointWorldSpaceAnchor  = bodyA->m_position;
	const Vec3 jointWorldSpaceAxis    = Vec3( 0, 0, 1 ).Normalize();
	Mat3 jointWorldSpaceMatrix;
	jointWorldSpaceMatrix.rows[ 0 ] = jointWorldSpaceAxis;
	jointWorldSpaceMatrix.rows[ 0 ].GetOrtho( jointWorldSpaceMatrix.rows[ 1 ],
	 jointWorldSpaceMatrix.rows[ 2 ] );
	jointWorldSpaceMatrix.rows[ 2 ] = jointWorldSpaceAxis;
	jointWorldSpaceMatrix.rows[ 2 ].GetOrtho( jointWorldSpaceMatrix.rows[ 0 ],
	 jointWorldSpaceMatrix.rows[ 1 ] );
	Vec3 jointWorldSpaceAxisLimited = Vec3( 0, 1, -1 );
	jointWorldSpaceAxisLimited.Normalize();

	ConstraintDistance * joint = new ConstraintDistance();

	const float pi = acosf( -1.0f );

	joint->m_bodyA      = &m_bodies[ m_bodies.size() - 1 ];
	joint->m_anchorA    = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
	joint->m_axisA      = joint->m_bodyA->m_orientation.Inverse().RotatePoint( jointWorldSpaceAxis
	 );
```

```
    body.m_position = joint->m_bodyA->m_position - jointWorldSpaceAxis * 1.0f;
    body.m_position = joint->m_bodyA->m_position + Vec3( 1, 0, 0 );
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
    body.m_invMass = 1.0f;
    body.m_elasticity = 1.0f;
    m_bodies.push_back( body );

    joint->m_bodyB      = &m_bodies[ m_bodies.size() - 1 ];
    joint->m_anchorB    = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
    joint->m_axisB      = joint->m_bodyB->m_orientation.Inverse().RotatePoint( jointWorldSpaceAxis
     );

    m_constraints.push_back( joint );
  }

  //
  //  Stack of Boxes
  //
  const int stackHeight = 5;
  for ( int x = 0; x < 1; x++ ) {
    for ( int y = 0; y < 1; y++ ) {
      for ( int z = 0; z < stackHeight; z++ ) {
        float offset = ( ( z & 1 ) == 0 ) ? 0.0f : 0.15f;
        float xx = (float)x + offset;
        float yy = (float)y + offset;
        float delta = 0.04f;
        float scaleHeight = 2.0f + delta;
        float deltaHeight = 1.0f + delta;
        body.m_position = Vec3( (float)xx * scaleHeight, (float)yy * scaleHeight, deltaHeight + (
    float)z * scaleHeight );
        body.m_orientation = Quat( 0, 0, 0, 1 );
        body.m_shape = new ShapeBox( g_boxUnit, sizeof( g_boxUnit ) / sizeof( Vec3 ) );
        body.m_invMass = 1.0f;
        body.m_elasticity = 0.5f;
        body.m_friction = 0.5f;
        m_bodies.push_back( body );
      }
    }
  }

  //
  //  Sphere and Convex Hull
  //
  body.m_position = Vec3( -10.0f, 0.0f, 5.0f );
  body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeSphere( 1.0f );
  body.m_invMass = 1.0f;
  body.m_elasticity = 0.9f;
  body.m_friction = 0.5f;
  m_bodies.push_back( body );

  body.m_position = Vec3( -10.0f, 0.0f, 10.0f );
  body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeConvex( g_diamond, sizeof( g_diamond ) / sizeof( Vec3 ) );
  body.m_invMass = 1.0f;
  body.m_elasticity = 1.0f;
  body.m_friction = 0.5f;
  m_bodies.push_back( body );

  //
  //  Motor
  //
  Vec3 motorPos = Vec3( 5, 0, 2 );
  Vec3 motorAxis = Vec3( 0, 0, 1 ).Normalize();
  Quat motorOrient = Quat( 1, 0, 0, 0 );

  body.m_position = motorPos;
  body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
```

```
body.m_orientation = Quat( 0, 0, 0, 1 );
body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
body.m_invMass = 0.0f;
body.m_elasticity = 0.9f;
body.m_friction = 0.5f;
m_bodies.push_back( body );

body.m_position = motorPos - motorAxis;
body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
body.m_orientation = motorOrient;
body.m_shape = new ShapeBox( g_boxBeam, sizeof( g_boxBeam ) / sizeof( Vec3 ) );
body.m_invMass = 0.01f;
body.m_elasticity = 1.0f;
body.m_friction = 0.5f;
m_bodies.push_back( body );
{
  ConstraintMotor * joint = new ConstraintMotor();
  joint->m_bodyA = &m_bodies[ m_bodies.size() - 2 ];
  joint->m_bodyB = &m_bodies[ m_bodies.size() - 1 ];

  const Vec3 jointWorldSpaceAnchor  = joint->m_bodyA->m_position;
  joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
  joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

  joint->m_motorSpeed = 2.0f;
  joint->m_motorAxis  = joint->m_bodyA->m_orientation.Inverse().RotatePoint( motorAxis );

  // Set the initial relative orientation (in bodyA's space)
  joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

  m_constraints.push_back( joint );
}

//
//  Mover
//
body.m_position = Vec3( 10, 0, 5 );
body.m_linearVelocity = Vec3( 0, 0, 0 );
body.m_orientation = Quat( 0, 0, 0, 1 );
body.m_shape = new ShapeBox( g_boxPlatform, sizeof( g_boxPlatform ) / sizeof( Vec3 ) );
body.m_invMass = 0.0f;
body.m_elasticity = 0.1f;
body.m_friction = 0.9f;
m_bodies.push_back( body );
{
  ConstraintMoverSimple * mover = new ConstraintMoverSimple();
  mover->m_bodyA = &m_bodies[ m_bodies.size() - 1 ];

  m_constraints.push_back( mover );
}

body.m_position = Vec3( 10, 0, 6.3f );
body.m_linearVelocity = Vec3( 0, 0, 0 );
body.m_orientation = Quat( 0, 0, 0, 1 );
body.m_shape = new ShapeBox( g_boxUnit, sizeof( g_boxUnit ) / sizeof( Vec3 ) );
body.m_invMass = 1.0f;
body.m_elasticity = 0.1f;
body.m_friction = 0.9f;
m_bodies.push_back( body );

//
//  Hinge Constraint
//
body.m_position = Vec3( -2, -5, 6 );
body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
body.m_orientation = Quat( 0, 0, 0, 1 );
body.m_orientation = Quat( Vec3( 1, 1, 1 ), pi * 0.25f );
body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
body.m_invMass = 0.0f;
body.m_elasticity = 0.9f;
body.m_friction = 0.5f;
```

```
m_bodies.push_back( body );

body.m_position = Vec3( -2, -5, 5 );
body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
body.m_orientation = Quat( Vec3( 0, 1, 1 ), pi * 0.5f );
body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
body.m_invMass = 1.0f;
body.m_elasticity = 1.0f;
body.m_friction = 0.5f;
m_bodies.push_back( body );
{
  ConstraintHingeQuatLimited * joint = new ConstraintHingeQuatLimited();
  joint->m_bodyA = &m_bodies[ m_bodies.size() - 2 ];
  joint->m_bodyB = &m_bodies[ m_bodies.size() - 1 ];

  const Vec3 jointWorldSpaceAnchor  = joint->m_bodyA->m_position;
  joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
  joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

  joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 1, 0, 0 ) );

  // Set the initial relative orientation
  joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

  m_constraints.push_back( joint );
}

//
//  Constant Velocity Constraint
//
body.m_position = Vec3( 2, -5, 6 );
body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
body.m_orientation = Quat( 0, 0, 0, 1 );
body.m_orientation = Quat( Vec3( 1, 1, 1 ), pi * 0.5f );
body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
body.m_invMass = 0.0f;
body.m_elasticity = 0.9f;
body.m_friction = 0.5f;
m_bodies.push_back( body );

body.m_position = Vec3( 2, -5, 5 );
body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
body.m_orientation = Quat( Vec3( 0, 1, 1 ), pi * 0.5f );
body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
body.m_invMass = 1.0f;
body.m_elasticity = 1.0f;
body.m_friction = 0.5f;
m_bodies.push_back( body );
{
  ConstraintConstantVelocityLimited * joint = new ConstraintConstantVelocityLimited();
  joint->m_bodyA = &m_bodies[ m_bodies.size() - 2 ];
  joint->m_bodyB = &m_bodies[ m_bodies.size() - 1 ];

  const Vec3 jointWorldSpaceAnchor  = joint->m_bodyA->m_position;
  joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
  joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

  joint->m_axisA = joint->m_bodyA->m_orientation.Inverse().RotatePoint( Vec3( 0, 0, 1 ) );

  // Set the initial relative orientation
  joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

  m_constraints.push_back( joint );
}

//
//  Teleportation Bug Fix
//
body.m_position = Vec3( 10, -10, 3 );
body.m_orientation = Quat( 0, 0, 0, 1 );
body.m_linearVelocity = Vec3( -100, 0, 0 );
```

```
    body.m_angularVelocity = Vec3( 0.0f, 0.0f, 0.0f );
    body.m_invMass = 1.0f;
    body.m_elasticity = 0.5f;
    body.m_friction = 0.5f;
    body.m_shape = new ShapeSphere( 0.5f );
    m_bodies.push_back( body );

    body.m_position = Vec3( -10, -10, 3 );
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_linearVelocity = Vec3( 100, 0, 0 );
    body.m_angularVelocity = Vec3( 0, 10, 0 );
    body.m_invMass = 1.0f;
    body.m_elasticity = 0.5f;
    body.m_friction = 0.5f;
    body.m_shape = new ShapeConvex( g_diamond, sizeof( g_diamond ) / sizeof( Vec3 ) );
    m_bodies.push_back( body );

    //
    //  Orientation Constraint
    //
    body.m_position = Vec3( 5, 0, 5 );
    body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
    body.m_angularVelocity = Vec3( 0.0f, 0.0f, 0.0f );
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
    body.m_invMass = 0.0f;
    body.m_elasticity = 0.9f;
    body.m_friction = 0.5f;
    m_bodies.push_back( body );

    body.m_position = Vec3( 6, 0, 5 );
    body.m_linearVelocity = Vec3( 0.0f, 0.0f, 0.0f );
    body.m_angularVelocity = Vec3( 0.0f, 0.0f, 0.0f );
    body.m_orientation = Quat( 0, 0, 0, 1 );
    body.m_shape = new ShapeBox( g_boxSmall, sizeof( g_boxSmall ) / sizeof( Vec3 ) );
    body.m_invMass = 0.001f;
    body.m_elasticity = 1.0f;
    body.m_friction = 0.5f;
    m_bodies.push_back( body );
    {
        ConstraintOrientation * joint = new ConstraintOrientation();
        joint->m_bodyA = &m_bodies[ m_bodies.size() - 2 ];
        joint->m_bodyB = &m_bodies[ m_bodies.size() - 1 ];

        const Vec3 jointWorldSpaceAnchor  = joint->m_bodyA->m_position;
        joint->m_anchorA  = joint->m_bodyA->WorldSpaceToBodySpace( jointWorldSpaceAnchor );
        joint->m_anchorB  = joint->m_bodyB->WorldSpaceToBodySpace( jointWorldSpaceAnchor );

        // Set the initial relative orientation
        joint->m_q0 = joint->m_bodyA->m_orientation.Inverse() * joint->m_bodyB->m_orientation;

        m_constraints.push_back( joint );
    }

    //
    //  Standard floor and walls
    //
    AddStandardSandBox( m_bodies );
}
```

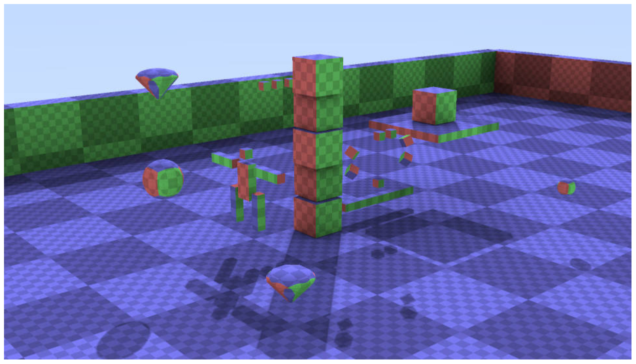


Figure 24: final

## 25 Further Reading & References

“Classical Mechanics” - Goldstein, Poole, Safko

“Physically Based Modeling: Principles and Practice, Constrained Dynamics” - David Baraff, Andrew Witkin

“Game Physics Pearls” - Gino van den Bergen, Dirk Gregorius

”Quaternion-Based Constraints” - Claude Lacoursiere

“Stabilization of Constraints and Integrals of motion in dynamical systems” - J. Baumgarte 1972

”3D Constraint Derivations for Impulse Solvers” -Marijin Tamis, 2015

”Iterative Dynamics with Temporal Coherence” -Erin Catto, 2005

”Baumgarte stabilisation over the SO(3) rotation group for control” - Sebastien Gros, Marion Zanon, Moritz Diehl