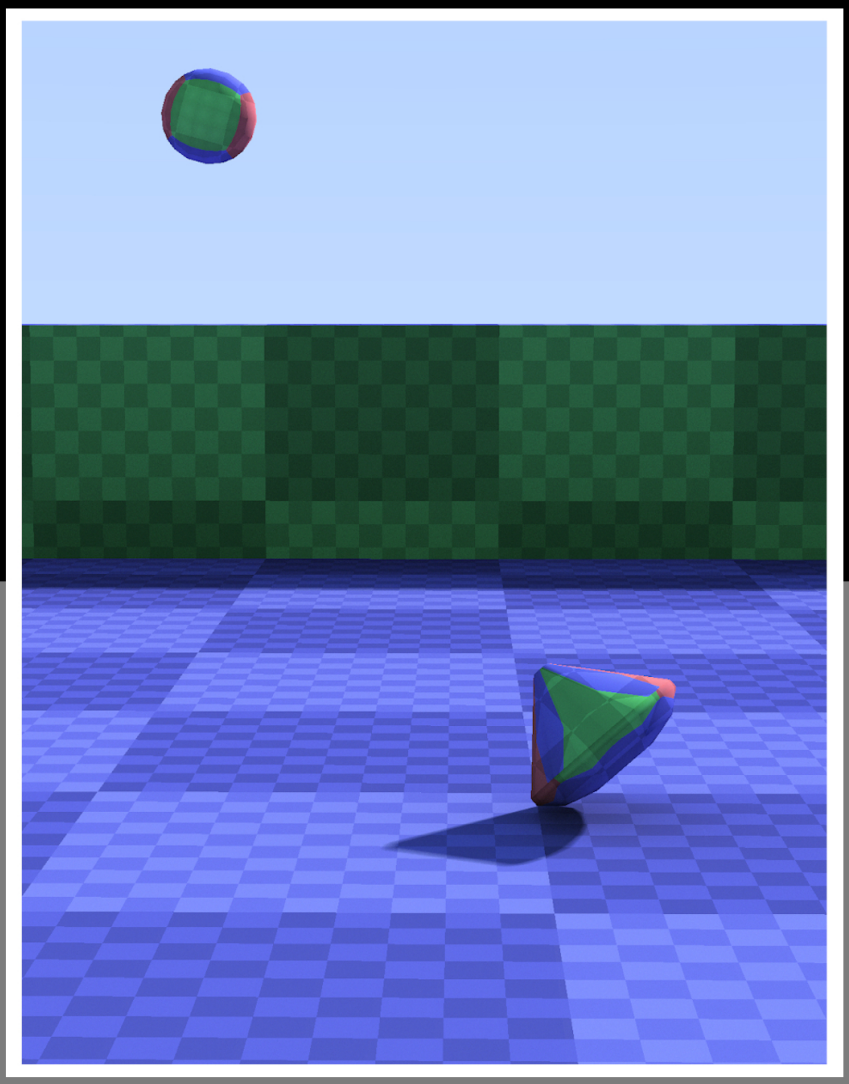
Game Physics

The Next Week



Gregory Hodges

Game Physics: The Next Week

Gregory Hodges
Version 1.07

Contents

1 Overview

In the last book we wrote a very basic impulse based rigid body simulation. However, you probably noticed that it's lacking a lot of features. The most noticeable is that it only checked collisions with spherical bodies.

In this book, we're going to address this issue. The first few chapters will be about developing the Gilbert-Johnson-Keerthi (GJK) algorithm. Which is an elegant and effective algorithm for detecting collisions between convex shapes. Then we'll discuss how the algorithm can be extended for continuous collision detection with conservative advance.

Also, I want to go ahead and define the term simplex here. A simplex is the simplest geometric shape that can be formed from n+1 points, where n is the number of dimensions. So, in zero dimensions, the simplex is a point. In one dimension, the simplex is a line segment. In two dimensions, it's a triangle. And in three dimensions it's a tetrahedron.

A quick reference for simplex:
0-simplex = point
1-simplex = line segment
2-simplex = triangle
3-simplex = tetrahedron

2 Shape Class Revisited

There's a handful of functions that we'll need to add to the base shape class in order to extend our collision detection to general convex shapes.

```
class Shape {
public:
  virtual void Build( const Vec3 * pts, const int num ) {}

  virtual Vec3 Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float bias
    ) const = 0;

  virtual Mat3 InertiaTensor() const = 0;

  virtual Bounds GetBounds( const Vec3 & pos, const Quat & orient ) const = 0;
  virtual Bounds GetBounds() const = 0;

  virtual Vec3 GetCenterOfMass() const { return m_centerOfMass; }

  virtual float FastestLinearSpeed( const Vec3 & angularVelocity, const Vec3 & dir ) const {
    return 0.0f; }

  enum shapeType_t {
    SHAPE_SPHERE,
    SHAPE_BOX,
    SHAPE_CONVEX,
  };
  virtual shapeType_t GetType() const = 0;

protected:
  Vec3 m_centerOfMass;
};
```

Notice the new Support function and the new FastestLinearSpeed functions. We'll need these for collision detection between general convex shapes and for continuous collision detection between convex shapes.

Now, that we have these, let's go ahead and see what they look like for the Sphere shape:

```
class ShapeSphere : public Shape {
public:
  explicit ShapeSphere( const float radius ) : m_radius( radius ) {
    m_centerOfMass.Zero();
  }

  Vec3 Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float bias ) const
    override;

  Mat3 InertiaTensor() const override;

  Bounds GetBounds( const Vec3 & pos, const Quat & orient ) const override;
  Bounds GetBounds() const override;

  shapeType_t GetType() const override { return SHAPE_SPHERE; }

public:
  float m_radius;
};
```

Notice that we're only overriding the Support function. That's because the support function gives us a point on the convex shape that is furthest in a particular direction. While the FastestLinearSpeed is a function that is only useful for non-spherical shapes.

The FastestLinearSpeed function is used for continuous collision detection. It's necessary for objects that are "long". Since a "long" object that's rotating might hit other objects, even though its linear velocity is zero. And for spheres, we don't care how quickly they rotate; we only care about their linear velocity. But we'll get more into this when we cover conservative advance later.

So, let's have a look at the Support function for spheres:

```
Vec3 ShapeSphere::Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float
    bias ) const {
```

```
    return ( pos + dir * ( m_radius + bias ) );
}
```

The dir vector is assumed to be normalized. Also, take note of the bias term. This is used to expand or shrink the apparent size of the shape. Which is necessary for accurately calculating the contact normal. But, we'll discuss it more in a few chapters from now.

3 Box Shape

The first new shape we're going to implement is the box shape. Boxes are so common in games, there's a joke of "speed to crate" in a game. Most games will have some sort of obvious box/crate within the first minutes of the game, and then there'll be boxes littered throughout the rest. Therefore we really need to know how to simulate them. It'll also help ease us into general convex hulls.

We can think of boxes as oriented bounds, or we can think of boxes as convex hulls that have eight points defining the corners. We are going to think of them as the latter, since that'll be a natural way of moving into general convex hulls.

And without further ado, the class declaration is:

```
class ShapeBox : public Shape {
public:
  explicit ShapeBox( const Vec3 * pts, const int num ) {
    Build( pts, num );
  }
  void Build( const Vec3 * pts, const int num ) override;

  Vec3 Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float bias ) const
    override;

  Mat3 InertiaTensor() const override;

  Bounds GetBounds( const Vec3 & pos, const Quat & orient ) const override;
  Bounds GetBounds() const override;

  float FastestLinearSpeed( const Vec3 & angularVelocity, const Vec3 & dir ) const override;

  shapeType_t GetType() const override { return SHAPE_BOX; }

public:
  std::vector< Vec3 > m_points;
  Bounds m_bounds;
};
```

Let's first have a look at the Build function. This will be where we build the bounds and store the points of the box:

```
void ShapeBox::Build( const Vec3 * pts, const int num ) {
  for ( int i = 0; i < num; i++ ) {
    m_bounds.Expand( pts[ i ] );
  }

  m_points.clear();
  m_points.push_back( Vec3( m_bounds.mins.x, m_bounds.mins.y, m_bounds.mins.z ) );
  m_points.push_back( Vec3( m_bounds.maxs.x, m_bounds.mins.y, m_bounds.mins.z ) );
  m_points.push_back( Vec3( m_bounds.mins.x, m_bounds.maxs.y, m_bounds.mins.z ) );
  m_points.push_back( Vec3( m_bounds.mins.x, m_bounds.mins.y, m_bounds.maxs.z ) );

  m_points.push_back( Vec3( m_bounds.maxs.x, m_bounds.maxs.y, m_bounds.maxs.z ) );
  m_points.push_back( Vec3( m_bounds.mins.x, m_bounds.maxs.y, m_bounds.maxs.z ) );
  m_points.push_back( Vec3( m_bounds.maxs.x, m_bounds.mins.y, m_bounds.maxs.z ) );
  m_points.push_back( Vec3( m_bounds.maxs.x, m_bounds.maxs.y, m_bounds.mins.z ) );

  m_centerOfMass = ( m_bounds.maxs + m_bounds.mins ) * 0.5f;
}
```

The inertia tensor for a box is defined as:

$$\begin{pmatrix} \frac{1}{12}M(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}M(w^2+d^2) & 0 \\ 0 & 0 & \frac{1}{12}M(w^2+h^2) \end{pmatrix} \tag{1}$$

However, we can't guarantee that the box will be centered about the origin. Well, we can, this is a tutorial book. However, we should go ahead and address situations where this won't happen. And in order to handle them we need to take advantage of the parallel axis theorem. Recall the definition of the inertia tensor is:

$$\mathbf{I_{ij}} \equiv \sum_k m_k \cdot (\vec{r_k} \cdot \vec{r_k} \cdot \delta_{ij} - x_i \cdot x_j) \tag{2}$$

And the tensor form of the parallel axis theorem is:

$$\mathbf{J_{ij}} \equiv \mathbf{I_{ij}} + M * (|R|^2 \delta_{ij} - R_i R_j) \tag{3}$$

Which gives us the code for the InertiaTensor function:

```
Mat3 ShapeBox::InertiaTensor() const {
  // Inertia tensor for box centered around zero
  const float dx = m_bounds.maxs.x - m_bounds.mins.x;
  const float dy = m_bounds.maxs.y - m_bounds.mins.y;
  const float dz = m_bounds.maxs.z - m_bounds.mins.z;

  Mat3 tensor;
  tensor.Zero();
  tensor.rows[ 0 ][ 0 ] = ( dy * dy + dz * dz ) / 12.0f;
  tensor.rows[ 1 ][ 1 ] = ( dx * dx + dz * dz ) / 12.0f;
  tensor.rows[ 2 ][ 2 ] = ( dx * dx + dy * dy ) / 12.0f;

  // Now we need to use the parallel axis theorem to get the inertia tensor for a box
  // that is not centered around the origin

  Vec3 cm;
  cm.x = ( m_bounds.maxs.x + m_bounds.mins.x ) * 0.5f;
  cm.y = ( m_bounds.maxs.y + m_bounds.mins.y ) * 0.5f;
  cm.z = ( m_bounds.maxs.z + m_bounds.mins.z ) * 0.5f;

  const Vec3 R = Vec3( 0, 0, 0 ) - cm;  // the displacement from center of mass to the origin
  const float R2 = R.GetLengthSqr();
  Mat3 patTensor;
  patTensor.rows[ 0 ] = Vec3( R2 - R.x * R.x,   R.x * R.y,    R.x * R.z );
  patTensor.rows[ 1 ] = Vec3(   R.y * R.x,  R2 - R.y * R.y,   R.y * R.z );
  patTensor.rows[ 2 ] = Vec3(   R.z * R.x,    R.z * R.y,  R2 - R.z * R.z );

  // Now we need to add the center of mass tensor and the parallel axis theorem tensor together;
  tensor += patTensor;
  return tensor;
}
```

Next let's have a look at the support function. Remember, the support function only takes in a direction, and returns the vertex on the shape that is furthest in that direction. We can make a pretty simple brute force version of that for our boxes like so:

```
Vec3 ShapeBox::Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float bias
    ) const {
  // Find the point in furthest in direction
  Vec3 maxPt = orient.RotatePoint( m_points[ 0 ] ) + pos;
  float maxDist = dir.Dot( maxPt );
  for ( int i = 1; i < m_points.size(); i++ ) {
    const Vec3 pt = orient.RotatePoint( m_points[ i ] ) + pos;
    const float dist = dir.Dot( pt );

    if ( dist > maxDist ) {
      maxDist = dist;
      maxPt = pt;
    }
  }

  Vec3 norm = dir;
  norm.Normalize();
  norm *= bias;

  return maxPt + norm;
}
```

Of course we're going to need the bounds functions for use in the broadphase. This is also a very straight forward method:

```
Bounds ShapeBox::GetBounds( const Vec3 & pos, const Quat & orient ) const {
  Vec3 corners[ 8 ];
  corners[ 0 ] = Vec3( m_bounds.mins.x, m_bounds.mins.y, m_bounds.mins.z );
  corners[ 1 ] = Vec3( m_bounds.mins.x, m_bounds.mins.y, m_bounds.maxs.z );
  corners[ 2 ] = Vec3( m_bounds.mins.x, m_bounds.maxs.y, m_bounds.mins.z );
  corners[ 3 ] = Vec3( m_bounds.maxs.x, m_bounds.mins.y, m_bounds.mins.z );

  corners[ 4 ] = Vec3( m_bounds.maxs.x, m_bounds.maxs.y, m_bounds.maxs.z );
  corners[ 5 ] = Vec3( m_bounds.maxs.x, m_bounds.maxs.y, m_bounds.mins.z );
  corners[ 6 ] = Vec3( m_bounds.maxs.x, m_bounds.mins.y, m_bounds.maxs.z );
  corners[ 7 ] = Vec3( m_bounds.mins.x, m_bounds.maxs.y, m_bounds.maxs.z );

  Bounds bounds;
  for ( int i = 0; i < 8; i++ ) {
    corners[ i ] = orient.RotatePoint( corners[ i ] ) + pos;
    bounds.Expand( corners[ i ] );
  }

  return bounds;
}

Bounds ShapeBox::GetBounds() const {
  return m_bounds;
}
```

And now the FastestLinearSpeed function. As we mentioned in the previous chapter, this is going to be used for continuous collision detection. And it takes in a direction and the angular velocity of the object and returns to us the velocity of the vertex traveling the fastest in that direction.

```
float ShapeBox::FastestLinearSpeed( const Vec3 & angularVelocity, const Vec3 & dir ) const {
  float maxSpeed = 0.0f;
  for ( int i = 0; i < m_points.size(); i++ ) {
    Vec3 r = m_points[ i ] - m_centerOfMass;
    Vec3 linearVelocity = angularVelocity.Cross( r );
    float speed = dir.Dot( linearVelocity );
    if ( speed > maxSpeed ) {
      maxSpeed = speed;
    }
  }
  return maxSpeed;
}
```

4 Convex Hulls

If we want to start modeling bodies that are more complicated than spheres and boxes then we need to figure out how to collide general convex shapes.

Technically a convex hull is the intersection of a set of infinite half spaces. But for our needs it'll just be an array of points. And it's going to be pretty easy to add this shape to the code, since we already have a base shape class:

```
class ShapeConvex : public Shape {
public:
  explicit ShapeConvex( const Vec3 * pts, const int num ) {
    Build( pts, num );
  }
  void Build( const Vec3 * pts, const int num ) override;

  Vec3 Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float bias ) const
    override;

  Mat3 InertiaTensor() const override;

  Bounds GetBounds( const Vec3 & pos, const Quat & orient ) const override;
  Bounds GetBounds() const override { return m_bounds; }

  float FastestLinearSpeed( const Vec3 & angularVelocity, const Vec3 & dir ) const override;

  shapeType_t GetType() const override { return SHAPE_CONVEX; }

public:
  std::vector< Vec3 > m_points;
  Bounds m_bounds;
  Mat3 m_inertiaTensor;
};

Bounds ShapeConvex::GetBounds( const Vec3 & pos, const Quat & orient ) const {
  Vec3 corners[ 8 ];
  corners[ 0 ] = Vec3( m_bounds.mins.x, m_bounds.mins.y, m_bounds.mins.z );
  corners[ 1 ] = Vec3( m_bounds.mins.x, m_bounds.mins.y, m_bounds.maxs.z );
  corners[ 2 ] = Vec3( m_bounds.mins.x, m_bounds.maxs.y, m_bounds.mins.z );
  corners[ 3 ] = Vec3( m_bounds.maxs.x, m_bounds.mins.y, m_bounds.mins.z );

  corners[ 4 ] = Vec3( m_bounds.maxs.x, m_bounds.maxs.y, m_bounds.maxs.z );
  corners[ 5 ] = Vec3( m_bounds.maxs.x, m_bounds.maxs.y, m_bounds.mins.z );
  corners[ 6 ] = Vec3( m_bounds.maxs.x, m_bounds.mins.y, m_bounds.maxs.z );
  corners[ 7 ] = Vec3( m_bounds.mins.x, m_bounds.maxs.y, m_bounds.maxs.z );

  Bounds bounds;
  for ( int i = 0; i < 8; i++ ) {
    corners[ i ] = orient.RotatePoint( corners[ i ] ) + pos;
    bounds.Expand( corners[ i ] );
  }

  return bounds;
}

float ShapeConvex::FastestLinearSpeed( const Vec3 & angularVelocity, const Vec3 & dir ) const {
  float maxSpeed = 0.0f;
  for ( int i = 0; i < m_points.size(); i++ ) {
    Vec3 r = m_points[ i ] - m_centerOfMass;
    Vec3 linearVelocity = angularVelocity.Cross( r );
    float speed = dir.Dot( linearVelocity );
    if ( speed > maxSpeed ) {
      maxSpeed = speed;
    }
  }
  return maxSpeed;
}
```

Now, you might think that the Build function for this class should be pretty simple. After all, we're just going to pass in an array of points and store an array of points, so can't we just copy it and that's it?

Unfortunately, it's not that simple. Because we also need to construct the inertia tensor. And, we need to be certain that the only points we store, are points that are on the surface of the convex hull.

So how do we do all this? Well, we basically have to slowly build up connections of the points. If any points are found to be inside the convex hull, then they're discarded. We can do this by building a simplex (tetrahedron) and then expanding it out to include all the points that are outside of it.

To build the tetrahedron we take four basic steps:

1. Find a point that is the furthest in a particular direction (later we will call this a support point)
2. Find another point that is furthest in the opposite direction of the point from step 1
3. Find a third point that is furthest from the axis of the points formed from 1 & 2
4. Then we find the point that is furthest from the plane formed from the previous points
5. Finally, we build the "connections" of those points. But really, we're just recording triangle indices.

And here's all the code for that:

```
/*
================================
FindPointFurthestInDir
================================
*/
int FindPointFurthestInDir( const Vec3 * pts, const int num, const Vec3 & dir ) {
  int maxIdx = 0;
  float maxDist = dir.Dot( pts[ 0 ] );
  for ( int i = 1; i < num; i++ ) {
    float dist = dir.Dot( pts[ i ] );
    if ( dist > maxDist ) {
      maxDist = dist;
      maxIdx = i;
    }
  }
  return maxIdx;
}

/*
================================
DistanceFromLine
================================
*/
float DistanceFromLine( const Vec3 & a, const Vec3 & b, const Vec3 & pt ) {
  Vec3 ab = b - a;
  ab.Normalize();

  Vec3 ray = pt - a;
  Vec3 projection = ab * ray.Dot( ab ); // project the ray onto ab
  Vec3 perpindicular = ray - projection;
  return perpindicular.GetMagnitude();
}

/*
================================
FindPointFurthestFromLine
================================
*/
Vec3 FindPointFurthestFromLine( const Vec3 * pts, const int num, const Vec3 & ptA, const Vec3 &
    ptB ) {
  int maxIdx = 0;
  float maxDist = DistanceFromLine( ptA, ptB, pts[ 0 ] );
  for ( int i = 1; i < num; i++ ) {
    float dist = DistanceFromLine( ptA, ptB, pts[ i ] );
    if ( dist > maxDist ) {
      maxDist = dist;
      maxIdx = i;
    }
  }
  return pts[ maxIdx ];
```

```
}

/*
================================
DistanceFromTriangle
================================
*/
float DistanceFromTriangle( const Vec3 & a, const Vec3 & b, const Vec3 & c, const Vec3 & pt ) {
  Vec3 ab = b - a;
  Vec3 ac = c - a;
  Vec3 normal = ab.Cross( ac );
  normal.Normalize();

  Vec3 ray = pt - a;
  float dist = ray.Dot( normal );
  return dist;
}

/*
================================
FindPointFurthestFromTriangle
================================
*/
Vec3 FindPointFurthestFromTriangle( const Vec3 * pts, const int num, const Vec3 & ptA, const Vec3
    & ptB, const Vec3 & ptC ) {
  int maxIdx = 0;
  float maxDist = DistanceFromTriangle( ptA, ptB, ptC, pts[ 0 ] );
  for ( int i = 1; i < num; i++ ) {
    float dist = DistanceFromTriangle( ptA, ptB, ptC, pts[ i ] );
    if ( dist * dist > maxDist * maxDist ) {
      maxDist = dist;
      maxIdx = i;
    }
  }
  return pts[ maxIdx ];
}

struct tri_t {
  int a;
  int b;
  int c;
};

/*
================================
BuildTetrahedron
================================
*/
void BuildTetrahedron( const Vec3 * verts, const int num, std::vector< Vec3 > & hullPts, std::
    vector< tri_t > & hullTris ) {
  hullPts.clear();
  hullTris.clear();

  Vec3 points[ 4 ];

  int idx = FindPointFurthestInDir( verts, num, Vec3( 1, 0, 0 ) );
  points[ 0 ] = verts[ idx ];
  idx = FindPointFurthestInDir( verts, num, points[ 0 ] * -1.0f );
  points[ 1 ] = verts[ idx ];
  points[ 2 ] = FindPointFurthestFromLine( verts, num, points[ 0 ], points[ 1 ] );
  points[ 3 ] = FindPointFurthestFromTriangle( verts, num, points[ 0 ], points[ 1 ], points[ 2 ] )
    ;

  // This is important for making sure the ordering is CCW for all faces.
  float dist = DistanceFromTriangle( points[ 0 ], points[ 1 ], points[ 2 ], points[ 3 ] );
  if ( dist > 0.0f ) {
    std::swap( points[ 0 ], points[ 1 ] );
  }

  // Build the tetrahedron
  hullPts.push_back( points[ 0 ] );
```

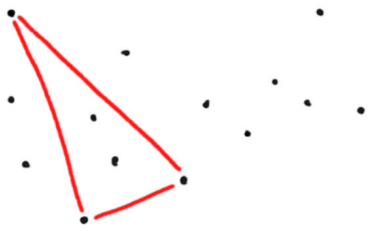
```
  hullPts.push_back( points[ 1 ] );
  hullPts.push_back( points[ 2 ] );
  hullPts.push_back( points[ 3 ] );

  tri_t tri;
  tri.a = 0;
  tri.b = 1;
  tri.c = 2;
  hullTris.push_back( tri );

  tri.a = 0;
  tri.b = 2;
  tri.c = 3;
  hullTris.push_back( tri );

  tri.a = 2;
  tri.b = 1;
  tri.c = 3;
  hullTris.push_back( tri );

  tri.a = 1;
  tri.b = 0;
  tri.c = 3;
  hullTris.push_back( tri );
}
```

Sweet, we have the function that can build the tetrahedron. An important note is the check at the end of the BuildTetrahedron function. We need to make sure that every face of the tetrahedron is in counter-clockwise order (CCW), since we'll use that to calculate the normals of each face, and our convex shapes have outward facing normals.

So, now that we have the most basic convex hull possible (a tetrahedron). We need to expand it out to include all the other points that define the surface of the shape.

Here's an outline of the steps that we need to perform to expand the tetrahedron to the convex hull:

1. Remove any internal points
   (a) There may be some points that are already inside the tetrahedron, so we need to discard them, since we only want the points that are on the surface of the convex hull.
2. Then we choose any point that is still external to the hull, and find the point furthest in that direction.
3. Then we find all the triangles that face this point and discard them.
4. Find the dangling edges, and use them to create new triangles.
5. Go back to step 1 and repeat until there are no external points.
6. Remove any points that are not referenced by any triangles.



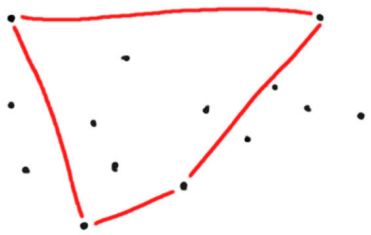


Figure 1: EPA

And here's the code for that function:

```
void ExpandConvexHull( std::vector< Vec3 > & hullPoints, std::vector< tri_t > & hullTris, const
    std::vector< Vec3 > & verts ) {
  std::vector< Vec3 > externalVerts = verts;
  RemoveInternalPoints( hullPoints, hullTris, externalVerts );

  while ( externalVerts.size() > 0 ) {
    int ptIdx = FindPointFurthestInDir( externalVerts.data(), (int)externalVerts.size(),
    externalVerts[ 0 ] );

    Vec3 pt = externalVerts[ ptIdx ];

    // remove this element
    externalVerts.erase( externalVerts.begin() + ptIdx );

    AddPoint( hullPoints, hullTris, pt );

    RemoveInternalPoints( hullPoints, hullTris, externalVerts );
  }

  RemoveUnreferencedVerts( hullPoints, hullTris );
}
```

Okay, there's a few functions in here that we'll need to add. Let's start with the RemoveInternalPoints. This is pretty brute force, we will just iterate through the "externalVerts" list and check if any of them are inside the current convex hull. If a point is inside the hull, then we will remove it from the list:

```
void RemoveInternalPoints( const std::vector< Vec3 > & hullPoints, const std::vector< tri_t > &
    hullTris, std::vector< Vec3 > & checkPts ) {
  for ( int i = 0; i < checkPts.size(); i++ ) {
    const Vec3 & pt = checkPts[ i ];

    bool isExternal = false;
    for ( int t = 0; t < hullTris.size(); t++ ) {
      const tri_t & tri = hullTris[ t ];
      const Vec3 & a = hullPoints[ tri.a ];
      const Vec3 & b = hullPoints[ tri.b ];
      const Vec3 & c = hullPoints[ tri.c ];

      // If the point is in front of any triangle then it's external
      float dist = DistanceFromTriangle( a, b, c, pt );
      if ( dist > 0.0f ) {
        isExternal = true;
        break;
      }
    }

    // if it's not external, then it's inside the polyhedron and should be removed
    if ( !isExternal ) {
      checkPts.erase( checkPts.begin() + i );
      i--;
    }
  }
```

```
  // Also remove any points that are just a little too close to the hull points
  for ( int i = 0; i < checkPts.size(); i++ ) {
    const Vec3 & pt = checkPts[ i ];

    bool isTooClose = false;
    for ( int j = 0; j < hullPoints.size(); j++ ) {
      Vec3 hullPt = hullPoints[ j ];
      Vec3 ray = hullPt - pt;
      if ( ray.GetLengthSqr() < 0.01f * 0.01f ) { // 1cm is too close
        isTooClose = true;
        break;
      }
    }

    if ( isTooClose ) {
      checkPts.erase( checkPts.begin() + i );
      i--;
    }
  }
}
```

Now, let's talk about the add point function. By all means this will be the most complicated part of this section. This is where we will have to loop over all triangles and determine if the point is on the positive side of the triangle, and if it is, delete the triangle. Then once all triangles are deleted, find the dangling edges and build new triangles:

```
struct edge_t {
  int a;
  int b;

  bool operator == ( const edge_t & rhs ) const {
    return ( ( a == rhs.a && b == rhs.b ) || ( a == rhs.b && b == rhs.a ) );
  }
};

/*
================================
IsEdgeUnique
This will compare the incoming edge with all the edges in the facing tris and then return true if
    it's unique
================================
*/
bool IsEdgeUnique( const std::vector< tri_t > & tris, const std::vector< int > & facingTris, const
     int ignoreTri, const edge_t & edge ) {
  for ( int i = 0; i < facingTris.size(); i++ ) {
    const int triIdx = facingTris[ i ];
    if ( ignoreTri == triIdx ) {
      continue;
    }

    const tri_t & tri = tris[ triIdx ];

    edge_t edges[ 3 ];
    edges[ 0 ].a = tri.a;
    edges[ 0 ].b = tri.b;

    edges[ 1 ].a = tri.b;
    edges[ 1 ].b = tri.c;

    edges[ 2 ].a = tri.c;
    edges[ 2 ].b = tri.a;

    for ( int e = 0; e < 3; e++ ) {
      if ( edge == edges[ e ] ) {
        return false;
      }
    }
  }
  return true;
```

```
}

/*
================================
AddPoint
================================
*/
void AddPoint( std::vector< Vec3 > & hullPoints, std::vector< tri_t > & hullTris, const Vec3 & pt
    ) {
  // This point is outside
  // Now we need to remove old triangles and build new ones

  // Find all the triangles that face this point
  std::vector< int > facingTris;
  for ( int i = (int)hullTris.size() - 1; i >= 0; i-- ) {
    const tri_t & tri = hullTris[ i ];

    const Vec3 & a = hullPoints[ tri.a ];
    const Vec3 & b = hullPoints[ tri.b ];
    const Vec3 & c = hullPoints[ tri.c ];

    const float dist = DistanceFromTriangle( a, b, c, pt );
    if ( dist > 0.0f ) {
      facingTris.push_back( i );
    }
  }

  // Now find all edges that are unique to the tris, these will be the edges that form the new
    triangles
  std::vector< edge_t > uniqueEdges;
  for ( int i = 0; i < facingTris.size(); i++ ) {
    const int triIdx = facingTris[ i ];
    const tri_t & tri = hullTris[ triIdx ];

    edge_t edges[ 3 ];
    edges[ 0 ].a = tri.a;
    edges[ 0 ].b = tri.b;

    edges[ 1 ].a = tri.b;
    edges[ 1 ].b = tri.c;

    edges[ 2 ].a = tri.c;
    edges[ 2 ].b = tri.a;

    for ( int e = 0; e < 3; e++ ) {
      if ( IsEdgeUnique( hullTris, facingTris, triIdx, edges[ e ] ) ) {
        uniqueEdges.push_back( edges[ e ] );
      }
    }
  }

  // now remove the old facing tris
  for ( int i = 0; i < facingTris.size(); i++ ) {
    hullTris.erase( hullTris.begin() + facingTris[ i ] );
  }

  // Now add the new point
  hullPoints.push_back( pt );
  const int newPtIdx = (int)hullPoints.size() - 1;

  // Now add triangles for each unique edge
  for ( int i = 0; i < uniqueEdges.size(); i++ ) {
    const edge_t & edge = uniqueEdges[ i ];

    tri_t tri;
    tri.a = edge.a;
    tri.b = edge.b;
    tri.c = newPtIdx;
    hullTris.push_back( tri );
  }
}
```

Finally, let's talk about the RemoveUnreferencedVerts function. You may have noticed that when we removed triangles from the convex hull, that we didn't remove the points that were referenced by the triangle. This is a really simple function, we only need to loop over each point and then check if it's referenced by any triangles, if it isn't, then we remove it.

```
void RemoveUnreferencedVerts( std::vector< Vec3 > & hullPoints, std::vector< tri_t > & hullTris )
    {
  for ( int i = 0; i < hullPoints.size(); i++ ) {

    bool isUsed = false;
    for ( int j = 0; j < hullTris.size(); j++ ) {
      const tri_t & tri = hullTris[ j ];

      if ( tri.a == i || tri.b == i || tri.c == i ) {
        isUsed = true;
        break;
      }
    }

    if ( isUsed ) {
      continue;
    }

    for ( int j = 0; j < hullTris.size(); j++ ) {
      tri_t & tri = hullTris[ j ];
      if ( tri.a > i ) {
        tri.a--;
      }
      if ( tri.b > i ) {
        tri.b--;
      }
      if ( tri.c > i ) {
        tri.c--;
      }
    }

    hullPoints.erase( hullPoints.begin() + i );
    i--;
  }
}
```

Okay, so that was a whole lot of code, so take your time to properly parse it. But with all these useful functions, we can now add the BuildConvexHull function.

```
void BuildConvexHull( const std::vector< Vec3 > & verts, std::vector< Vec3 > & hullPts, std::
    vector< tri_t > & hullTris ) {
  if ( verts.size() < 4 ) {
    return;
  }

  // Build a tetrahedron
  BuildTetrahedron( verts.data(), (int)verts.size(), hullPts, hullTris );

  ExpandConvexHull( hullPts, hullTris, verts );
}
```

Finally, we've managed to take in a set of vertices, removed any vertices that are not on the surface of the convex hull, and built the list of triangles that composes the surface of the convex hull. The only thing we need to do now is build the inertia tensor for the shape. However, this is large enough of a topic, that we'll do it in the next chapter.

5 Inertia Tensor Revisited

There's a handful of ways we could go about calculating the inertia tensor for a convex hull. We could decompose the entire convex hull into a set of tetrahedrons, calculate the inertia tensor for each simplex, and then add those together. But instead what we will do is perform a 3D integration over the bounds of the convex hull and if the sample point is outside the hull, we ignore it, but if it's inside the hull then we treat it as a point mass and accumulate its moment of inertia. Recall that the definition of moment of inertia is:

$$\mathbf{I_{ij}} \equiv \sum_k m_k \cdot (\vec{r_k} \cdot \vec{r_k} \cdot \delta_{ij} - x_i \cdot x_j) \tag{4}$$

Where $\vec{r_i}$ is the distance from the center of mass for the sample point mass, m_i. So, the first thing we need to do is calculate the center of mass. This will actually be a very similar algorithm to the inertia tensor, we'll integrate over the bounds and only accumulate the sample points that are inside the convex hull. Recall that the center of mass is:

$$x_{cm} = \frac{\sum_i m_i * x_i}{\sum_i m_i} \tag{5}$$

Okay, now let's calculate the center of mass of the convex hull:

```
/*
================================
IsExternal
================================
*/
bool IsExternal( const std::vector< Vec3 > & pts, const std::vector< tri_t > & tris, const Vec3 &
    pt ) {
  bool isExternal = false;
  for ( int t = 0; t < tris.size(); t++ ) {
    const tri_t & tri = tris[ t ];
    const Vec3 & a = pts[ tri.a ];
    const Vec3 & b = pts[ tri.b ];
    const Vec3 & c = pts[ tri.c ];

    // If the point is in front of any triangle then it's external
    float dist = DistanceFromTriangle( a, b, c, pt );
    if ( dist > 0.0f ) {
      isExternal = true;
      break;
    }
  }

  return isExternal;
}

/*
================================
CalculateCenterOfMass
================================
*/
Vec3 CalculateCenterOfMass( const std::vector< Vec3 > & pts, const std::vector< tri_t > & tris ) {
  const int numSamples = 100;

  Bounds bounds;
  bounds.Expand( pts.data(), pts.size() );

  Vec3 cm( 0.0f );
  const float dx = bounds.WidthX() / (float)numSamples;
  const float dy = bounds.WidthY() / (float)numSamples;
  const float dz = bounds.WidthZ() / (float)numSamples;

  int sampleCount = 0;
  for ( float x = bounds.mins.x; x < bounds.maxs.x; x += dx ) {
    for ( float y = bounds.mins.y; y < bounds.maxs.y; y += dy ) {
      for ( float z = bounds.mins.z; z < bounds.maxs.z; z += dz ) {
        Vec3 pt( x, y, z );
```

```
        if ( IsExternal( pts, tris, pt ) ) {
          continue;
        }

        cm += pt;
        sampleCount++;
      }
    }
  }

  cm /= (float)sampleCount;
  return cm;
}
```

And finally we can calculate the inertia tensor:

```
Mat3 CalculateInertiaTensor( const std::vector< Vec3 > & pts, const std::vector< tri_t > & tris,
    const Vec3 & cm ) {
  const int numSamples = 100;

  Bounds bounds;
  bounds.Expand( pts.data(), (int)pts.size() );

  Mat3 tensor;
  tensor.Zero();

  const float dx = bounds.WidthX() / (float)numSamples;
  const float dy = bounds.WidthY() / (float)numSamples;
  const float dz = bounds.WidthZ() / (float)numSamples;

  int sampleCount = 0;
  for ( float x = bounds.mins.x; x < bounds.maxs.x; x += dx ) {
    for ( float y = bounds.mins.y; y < bounds.maxs.y; y += dy ) {
      for ( float z = bounds.mins.z; z < bounds.maxs.z; z += dz ) {
        Vec3 pt( x, y, z );

        if ( IsExternal( pts, tris, pt ) ) {
          continue;
        }

        // Get the point relative to the center of mass
        pt -= cm;

        tensor.rows[ 0 ][ 0 ] += pt.y * pt.y + pt.z * pt.z;
        tensor.rows[ 1 ][ 1 ] += pt.z * pt.z + pt.x * pt.x;
        tensor.rows[ 2 ][ 2 ] += pt.x * pt.x + pt.y * pt.y;

        tensor.rows[ 0 ][ 1 ] += -1.0f * pt.x * pt.y;
        tensor.rows[ 0 ][ 2 ] += -1.0f * pt.x * pt.z;
        tensor.rows[ 1 ][ 2 ] += -1.0f * pt.y * pt.z;

        tensor.rows[ 1 ][ 0 ] += -1.0f * pt.x * pt.y;
        tensor.rows[ 2 ][ 0 ] += -1.0f * pt.x * pt.z;
        tensor.rows[ 2 ][ 1 ] += -1.0f * pt.y * pt.z;

        sampleCount++;
      }
    }
  }

  tensor *= 1.0f / (float)sampleCount;
  return tensor;
}
```

6 Finalizing the convex hull

Now that we have a means of constructing a convex hull from a set of points and then calculating the center of mass and inertia tensor, we can finally put it all together. And the best part is that we've already done all the hard work. So the Build function for the convex shape turns into:

```
void ShapeConvex::Build( const Vec3 * pts, const int num ) {
  m_points.clear();
  m_points.reserve( num );
  for ( int i = 0; i < num; i++ ) {
    m_points.push_back( pts[ i ] );
  }

  // Expand into a convex hull
  std::vector< Vec3 > hullPoints;
  std::vector< tri_t > hullTriangles;
  BuildConvexHull( m_points, hullPoints, hullTriangles );
  m_points = hullPoints;

  // Expand the bounds
  m_bounds.Clear();
  m_bounds.Expand( m_points.data(), m_points.size() );

  m_centerOfMass = CalculateCenterOfMass( hullPoints, hullTriangles );

  m_inertiaTensor = CalculateInertiaTensor( hullPoints, hullTriangles, m_centerOfMass );
}
```

Yeah, that's mostly it. This chapter gets to be super short. Now, we just have to test it. Let's replace the ground from the last book with a flat box shape, and then add some dynamic bodies. But first let's go ahead and define the geometry for some new shapes we'll be using.

```
static const float w = 50;
static const float h = 25;

Vec3 g_boxGround[] = {
  Vec3(-w,-h, 0 ),
  Vec3( w,-h, 0 ),
  Vec3(-w, h, 0 ),
  Vec3( w, h, 0 ),

  Vec3(-w,-h,-1 ),
  Vec3( w,-h,-1 ),
  Vec3(-w, h,-1 ),
  Vec3( w, h,-1 ),
};

Vec3 g_boxWall0[] = {
  Vec3(-1,-h, 0 ),
  Vec3( 1,-h, 0 ),
  Vec3(-1, h, 0 ),
  Vec3( 1, h, 0 ),

  Vec3(-1,-h, 5 ),
  Vec3( 1,-h, 5 ),
  Vec3(-1, h, 5 ),
  Vec3( 1, h, 5 ),
};

Vec3 g_boxWall1[] = {
  Vec3(-w,-1, 0 ),
  Vec3( w,-1, 0 ),
  Vec3(-w, 1, 0 ),
  Vec3( w, 1, 0 ),

  Vec3(-w,-1, 5 ),
  Vec3( w,-1, 5 ),
  Vec3(-w, 1, 5 ),
  Vec3( w, 1, 5 ),
};
```

```
Vec3 g_boxUnit[] = {
  Vec3(-1,-1,-1 ),
  Vec3( 1,-1,-1 ),
  Vec3(-1, 1,-1 ),
  Vec3( 1, 1,-1 ),

  Vec3(-1,-1, 1 ),
  Vec3( 1,-1, 1 ),
  Vec3(-1, 1, 1 ),
  Vec3( 1, 1, 1 ),
};

static const float t = 0.25f;
Vec3 g_boxSmall[] = {
  Vec3(-t,-t,-t ),
  Vec3( t,-t,-t ),
  Vec3(-t, t,-t ),
  Vec3( t, t,-t ),

  Vec3(-t,-t, t ),
  Vec3( t,-t, t ),
  Vec3(-t, t, t ),
  Vec3( t, t, t ),
};

static const float l = 3.0f;
Vec3 g_boxBeam[] = {
  Vec3(-l,-t,-t ),
  Vec3( l,-t,-t ),
  Vec3(-l, t,-t ),
  Vec3( l, t,-t ),

  Vec3(-l,-t, t ),
  Vec3( l,-t, t ),
  Vec3(-l, t, t ),
  Vec3( l, t, t ),
};

Vec3 g_diamond[ 7 * 8 ];
void FillDiamond() {
  Vec3 pts[ 4 + 4 ];
  pts[ 0 ] = Vec3( 0.1f, 0, -1 );
  pts[ 1 ] = Vec3( 1, 0, 0 );
  pts[ 2 ] = Vec3( 1, 0, 0.1f );
  //pts[ 3 ] = Vec3( 1, 0, 0.2f );
  pts[ 3 ] = Vec3( 0.4f, 0, 0.4f );

  const float pi = acosf( -1.0f );
  const Quat quatHalf( Vec3( 0, 0, 1 ), 2.0f * pi * 0.125f * 0.5f );
  pts[ 4 ] = Vec3( 0.8f, 0, 0.3f );
  pts[ 4 ] = quatHalf.RotatePoint( pts[ 4 ] );
  pts[ 5 ] = quatHalf.RotatePoint( pts[ 1 ] );
  pts[ 6 ] = quatHalf.RotatePoint( pts[ 2 ] );

  const Quat quat( Vec3( 0, 0, 1 ), 2.0f * pi * 0.125f );
  int idx = 0;
  for ( int i = 0; i < 7; i++ ) {
    g_diamond[ idx ] = pts[ i ];
    idx++;
  }

  Quat quatAccumulator;
  for ( int i = 1; i < 8; i++ ) {
    quatAccumulator = quatAccumulator * quat;
    for ( int pt = 0; pt < 7; pt++ ) {
      g_diamond[ idx ] = quatAccumulator.RotatePoint( pts[ pt ] );
      idx++;
    }
  }
}
```

Now, let's go ahead and use our box class and new geometry to define what will become the bodies defining the boundaries of our standard sandbox.

```
void AddStandardSandBox( std::vector< Body > & bodies ) {
  Body body;

  body.m_position = Vec3( 0, 0, 0 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity.Zero();
  body.m_angularVelocity.Zero();
  body.m_invMass = 0.0f;
  body.m_elasticity = 0.5f;
  body.m_friction = 0.5f;
  body.m_shape = new ShapeBox( g_boxGround, sizeof( g_boxGround ) / sizeof( Vec3 ) );
  bodies.push_back( body );

  body.m_position = Vec3( 50, 0, 0 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity.Zero();
  body.m_angularVelocity.Zero();
  body.m_invMass = 0.0f;
  body.m_elasticity = 0.5f;
  body.m_friction = 0.0f;
  body.m_shape = new ShapeBox( g_boxWall0, sizeof( g_boxWall0 ) / sizeof( Vec3 ) );
  bodies.push_back( body );

  body.m_position = Vec3(-50, 0, 0 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity.Zero();
  body.m_angularVelocity.Zero();
  body.m_invMass = 0.0f;
  body.m_elasticity = 0.5f;
  body.m_friction = 0.0f;
  body.m_shape = new ShapeBox( g_boxWall0, sizeof( g_boxWall0 ) / sizeof( Vec3 ) );
  bodies.push_back( body );

  body.m_position = Vec3( 0, 25, 0 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity.Zero();
  body.m_angularVelocity.Zero();
  body.m_invMass = 0.0f;
  body.m_elasticity = 0.5f;
  body.m_friction = 0.0f;
  body.m_shape = new ShapeBox( g_boxWall1, sizeof( g_boxWall1 ) / sizeof( Vec3 ) );
  bodies.push_back( body );

  body.m_position = Vec3( 0,-25, 0 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity.Zero();
  body.m_angularVelocity.Zero();
  body.m_invMass = 0.0f;
  body.m_elasticity = 0.5f;
  body.m_friction = 0.0f;
  body.m_shape = new ShapeBox( g_boxWall1, sizeof( g_boxWall1 ) / sizeof( Vec3 ) );
  bodies.push_back( body );
}
```

Finally, let's go ahead and initialize the scene with a convex hull and our new sandbox shapes.

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

  body.m_position = Vec3( 0, 0, 10 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity = Vec3( 0, 0, 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 );

    AddStandardSandBox( m_bodies );
}
```

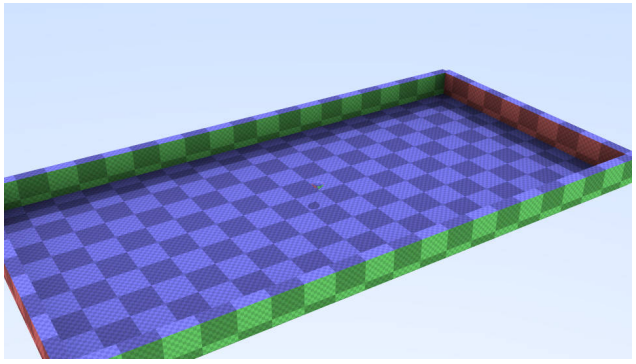


Figure 2: Standard Sandbox

Now, this is great. We have a method for creating general convex shapes. Unfortunately, we have yet to add any code that can check for collisions between convex shapes. That is what the next several chapters will be all about.

7 Minkowski Sums

What in the world is a Minkowski sum and what the heck does it have to do with collision detection? Well, let's answer the first question, and then we'll get to the second question.

The Minkwoski sum is what happens when you have two sets of points A and B. And you add every point in A with every point in B. And in the fancy pants mathy math lingo:

$$A + B = \{a + b | a \in A, b \in B\} \tag{6}$$

Definitions do one thing. But, personally, I think that it's a lot easier to understand it when it's visualized such as in figure 3

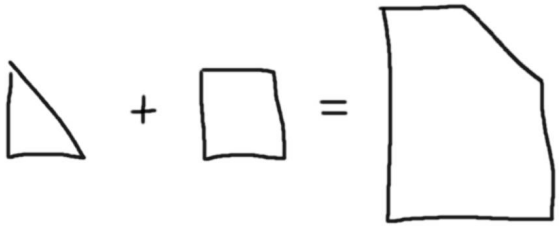


Figure 3: Minkowski Sum

It's important to note that the Minkowski sum of two convex sets, is itself a convex set. Why is that so important? Well, we already know it's very easy to determine if a point is inside a convex hull. All that needs to be done is to check if the point is on the positive side of any face, if it is then it's outside, if it isn't then it's inside. This is going to be useful for determining if two convex hulls are overlapping.

So how do we use this property to determine if two hulls are intersecting? Well, if we subtract every point in set B from every point in set A, then if they overlap, some of those points will map to the origin.

*A side note, some people call this the Minkowski "difference". But there's technically no such thing. What we're technically doing is negating set B and then performing the Minkowski sum between A and -B.

This makes for a pretty trivial algorithm. Let's review how it works:

1. Take the Minkowski "difference" between A and B
2. Check if the origin is contained by the new convex set

Now, you might be thinking that's insane. Sure, the algorithm is simple. But building a convex hull, as we already know, is not exactly fast. And since we want to write a simulation that needs to run in real time, we need something slightly more efficient. This is where GJK enters, because they figured out the clever bits.

8 Signed Volumes

Before we actually get into the meat of the GJK algorithm, I'd like to go over an algorithm known as Signed Volumes.

A key part of the GJK algorithm is the projection of the origin onto the 1, 2, or 3-simplex. The reason we need this, is because GJK needs to know the shortest distance/direction to the origin for any given simplex. And so, when we work on the GJK algorithm, it'd be nice if we already had this utility available.

Recall that the 1-simplex is a line segment, the 2-simplex is a triangle, and the 3-simplex is a tetrahedron.

So, how do we go about projecting the origin onto those shapes? Or more generally, how do we project any point onto a simplex?

The signed volumes approach is to find which axis-aligned plane maximizes the projected area or length of the simplex. So what the heck does that mean? Well, let's use a specific example.

Suppose we have a 1-simplex or line segment. We project it onto the x-axis, the y-axis, and the z-axis. Then, we determine which projection has the greatest length. We then choose that axis and project our point onto it as well. In this space, we can then easily calculate the barycentric coordinates of the point onto the 1-simplex. Which then gives us the projection of the point onto the 1-simplex.

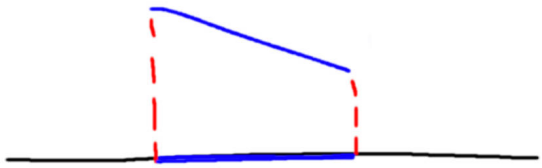


Figure 4: 1-simplex projection

And here's the associated code:

```
Vec2 SignedVolume1D( const Vec3 & s1, const Vec3 & s2 ) {
  Vec3 ab = s2 - s1;  // Ray from a to b
  Vec3 ap = Vec3( 0.0f ) - s1;  // Ray from a to origin
  Vec3 p0 = s1 + ab * ab.Dot( ap ) / ab.GetLengthSqr(); // projection of the origin onto the line

  // Choose the axis with the greatest difference/length
  int idx = 0;
  float mu_max = 0;
  for ( int i = 0; i < 3; i++ ) {
    float mu = s2[ i ] - s1[ i ];
    if ( mu * mu > mu_max * mu_max ) {
      mu_max = mu;
      idx = i;
    }
  }

  // Project the simplex points and projected origin onto the axis with greatest length
  const float a = s1[ idx ];
  const float b = s2[ idx ];
  const float p = p0[ idx ];

  // Get the signed distance from a to p and from p to b
  const float C1 = p - a;
  const float C2 = b - p;

  // if p is between [a,b]
  if ( ( p > a && p < b ) || ( p > b && p < a ) ) {
    Vec2 lambdas;
    lambdas[ 0 ] = C2 / mu_max;
    lambdas[ 1 ] = C1 / mu_max;
    return lambdas;
```

```
  }

  // if p is on the far side of a
  if ( ( a <= b && p <= a ) || ( a >= b && p >= a ) ) {
    return Vec2( 1.0f, 0.0f );
  }

  // p must be on the far side of b
  return Vec2( 0.0f, 1.0f );
}
```

What about the 2-simplex or triangle? We do the same thing. Only we project the triangle onto the xy-plane, the yz-plane, and the zx-plane. Then we determine which projection has the greatest area. Then project the point onto that plane. If the point is inside the triangle, then calculate the barycentric coordinates and be done. If it's outside the triangle, then project it onto the edges of the triangle using the 1-simplex projection method, and use the closest projection.

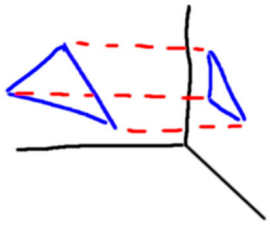


Figure 5: 2-simplex projection

And the associated code:

```
int CompareSigns( float a, float b ) {
  if ( a > 0.0f && b > 0.0f ) {
    return 1;
  }
  if ( a < 0.0f && b < 0.0f ) {
    return 1;
  }
  return 0;
}

Vec3 SignedVolume2D( const Vec3 & s1, const Vec3 & s2, const Vec3 & s3 ) {
  Vec3 normal = ( s2 - s1 ).Cross( s3 - s1 );
  Vec3 p0 = normal * s1.Dot( normal ) / normal.GetLengthSqr();

  // Find the axis with the greatest projected area
  int idx = 0;
  float area_max = 0;
  for ( int i = 0; i < 3; i++ ) {
    int j = ( i + 1 ) % 3;
    int k = ( i + 2 ) % 3;

    Vec2 a = Vec2( s1[ j ], s1[ k ] );
    Vec2 b = Vec2( s2[ j ], s2[ k ] );
    Vec2 c = Vec2( s3[ j ], s3[ k ] );
    Vec2 ab = b - a;
    Vec2 ac = c - a;

    float area = ab.x * ac.y - ab.y * ac.x;
    if ( area * area > area_max * area_max ) {
      idx = i;
      area_max = area;
    }
  }
```

```
  // Project onto the appropriate axis
  int x = ( idx + 1 ) % 3;
  int y = ( idx + 2 ) % 3;
  Vec2 s[ 3 ];
  s[ 0 ] = Vec2( s1[ x ], s1[ y ] );
  s[ 1 ] = Vec2( s2[ x ], s2[ y ] );
  s[ 2 ] = Vec2( s3[ x ], s3[ y ] );
  Vec2 p = Vec2( p0[ x ], p0[ y ] );

  // Get the sub-areas of the triangles formed from the projected origin and the edges
  Vec3 areas;
  for ( int i = 0; i < 3; i++ ) {
    int j = ( i + 1 ) % 3;
    int k = ( i + 2 ) % 3;

    Vec2 a = p;
    Vec2 b = s[ j ];
    Vec2 c = s[ k ];
    Vec2 ab = b - a;
    Vec2 ac = c - a;

    areas[ i ] = ab.x * ac.y - ab.y * ac.x;
  }

  // If the projected origin is inside the triangle, then return the barycentric points
  if ( CompareSigns( area_max, areas[ 0 ] ) > 0 && CompareSigns( area_max, areas[ 1 ] ) > 0 &&
    CompareSigns( area_max, areas[ 2 ] ) > 0 ) {
    Vec3 lambdas = areas / area_max;
    return lambdas;
  }

  // If we make it here, then we need to project onto the edges and determine the closest point
  float dist = 1e10;
  Vec3 lambdas = Vec3( 1, 0, 0 );
  for ( int i = 0; i < 3; i++ ) {
    int k = ( i + 1 ) % 3;
    int l = ( i + 2 ) % 3;

    Vec3 edgesPts[ 3 ];
    edgesPts[ 0 ] = s1;
    edgesPts[ 1 ] = s2;
    edgesPts[ 2 ] = s3;

    Vec2 lambdaEdge = SignedVolume1D( edgesPts[ k ], edgesPts[ l ] );
    Vec3 pt = edgesPts[ k ] * lambdaEdge[ 0 ] + edgesPts[ l ] * lambdaEdge[ 1 ];
    if ( pt.GetLengthSqr() < dist ) {
      dist = pt.GetLengthSqr();
      lambdas[ i ] = 0;
      lambdas[ k ] = lambdaEdge[ 0 ];
      lambdas[ l ] = lambdaEdge[ 1 ];
    }
  }

  return lambdas;
}
```

And finally we need to handle the 3-simplex case. Simply determine if the point is inside the tetrahedron, if it is then calculate its barycentric coordinates. If it isn't, then check each face for which projection is closest.

```
Vec4 SignedVolume3D( const Vec3 & s1, const Vec3 & s2, const Vec3 & s3, const Vec3 & s4 ) {
  Mat4 M;
  M.rows[ 0 ] = Vec4( s1.x, s2.x, s3.x, s4.x );
  M.rows[ 1 ] = Vec4( s1.y, s2.y, s3.y, s4.y );
  M.rows[ 2 ] = Vec4( s1.z, s2.z, s3.z, s4.z );
  M.rows[ 3 ] = Vec4( 1.0f, 1.0f, 1.0f, 1.0f );

  Vec4 C4;
  C4[ 0 ] = M.Cofactor( 3, 0 );
  C4[ 1 ] = M.Cofactor( 3, 1 );
  C4[ 2 ] = M.Cofactor( 3, 2 );
```

```
  C4[ 3 ] = M.Cofactor( 3, 3 );

  const float detM = C4[ 0 ] + C4[ 1 ] + C4[ 2 ] + C4[ 3 ];

  // If the barycentric coordinates put the origin inside the simplex, then return them
  if ( CompareSigns( detM, C4[ 0 ] ) > 0 && CompareSigns( detM, C4[ 1 ] ) > 0 && CompareSigns(
    detM, C4[ 2 ] ) > 0 && CompareSigns( detM, C4[ 3 ] ) > 0 ) {
    Vec4 lambdas = C4 * ( 1.0f / detM );
    return lambdas;
  }

  // If we get here, then we need to project the origin onto the faces and determine the closest
    one
  Vec4 lambdas;
  float dist = 1e10;
  for ( int i = 0; i < 4; i++ ) {
    int j = ( i + 1 ) % 4;
    int k = ( i + 2 ) % 4;

    Vec3 facePts[ 4 ];
    facePts[ 0 ] = s1;
    facePts[ 1 ] = s2;
    facePts[ 2 ] = s3;
    facePts[ 3 ] = s4;

    Vec3 lambdasFace = SignedVolume2D( facePts[ i ], facePts[ j ], facePts[ k ] );
    Vec3 pt = facePts[ i ] * lambdasFace[ 0 ] + facePts[ j ] * lambdasFace[ 1 ] + facePts[ k ] *
    lambdasFace[ 2 ];
    if ( pt.GetLengthSqr() < dist ) {
      dist = pt.GetLengthSqr();
      lambdas.Zero();
      lambdas[ i ] = lambdasFace[ 0 ];
      lambdas[ j ] = lambdasFace[ 1 ];
      lambdas[ k ] = lambdasFace[ 2 ];
    }
  }

  return lambdas;
}
```

Before we move on to the GJK algorithm. We should test these utiltiy functions to make sure they work as expected. Remember, they project a point onto the simplex and return the barycentric coordinates of the projection.

```
void TestSignedVolumeProjection() {
  const Vec3 orgPts[ 4 ] = {
    Vec3( 0, 0, 0 ),
    Vec3( 1, 0, 0 ),
    Vec3( 0, 1, 0 ),
    Vec3( 0, 0, 1 ),
  };
  Vec3 pts[ 4 ];
  Vec4 lambdas;
  Vec3 v;

  for ( int i = 0; i < 4; i++ ) {
    pts[ i ] = orgPts[ i ] + Vec3( 1, 1, 1 );
  }
  lambdas = SignedVolume3D( pts[ 0 ], pts[ 1 ], pts[ 2 ], pts[ 3 ] );
  v.Zero();
  for ( int i = 0; i < 4; i++ ) {
    v += pts[ i ] * lambdas[ i ];
  }
  printf( "lambdas: %.3f %.3f %.3f %.3f        v: %.3f %.3f %.3f\n",
    lambdas.x, lambdas.y, lambdas.z, lambdas.w,
    v.x, v.y, v.z
  );

  for ( int i = 0; i < 4; i++ ) {
    pts[ i ] = orgPts[ i ] + Vec3( -1, -1, -1 ) * 0.25f;
  }
```

```
  lambdas = SignedVolume3D( pts[ 0 ], pts[ 1 ], pts[ 2 ], pts[ 3 ] );
  v.Zero();
  for ( int i = 0; i < 4; i++ ) {
    v += pts[ i ] * lambdas[ i ];
  }
  printf( "lambdas: %.3f %.3f %.3f %.3f        v: %.3f %.3f %.3f\n",
    lambdas.x, lambdas.y, lambdas.z, lambdas.w,
    v.x, v.y, v.z
  );

  for ( int i = 0; i < 4; i++ ) {
    pts[ i ] = orgPts[ i ] + Vec3( -1, -1, -1 );
  }
  lambdas = SignedVolume3D( pts[ 0 ], pts[ 1 ], pts[ 2 ], pts[ 3 ] );
  v.Zero();
  for ( int i = 0; i < 4; i++ ) {
    v += pts[ i ] * lambdas[ i ];
  }
  printf( "lambdas: %.3f %.3f %.3f %.3f        v: %.3f %.3f %.3f\n",
    lambdas.x, lambdas.y, lambdas.z, lambdas.w,
    v.x, v.y, v.z
  );

  for ( int i = 0; i < 4; i++ ) {
    pts[ i ] = orgPts[ i ] + Vec3( 1, 1, -0.5f );
  }
  lambdas = SignedVolume3D( pts[ 0 ], pts[ 1 ], pts[ 2 ], pts[ 3 ] );
  v.Zero();
  for ( int i = 0; i < 4; i++ ) {
    v += pts[ i ] * lambdas[ i ];
  }
  printf( "lambdas: %.3f %.3f %.3f %.3f        v: %.3f %.3f %.3f\n",
    lambdas.x, lambdas.y, lambdas.z, lambdas.w,
    v.x, v.y, v.z
  );

  pts[ 0 ] = Vec3( 51.1996613f, 26.1989613f, 1.91339576f );
  pts[ 1 ] = Vec3( -51.0567360f, -26.0565681f, -0.436143428f );
  pts[ 2 ] = Vec3( 50.8978920f, -24.1035538f, -1.04042661f );
  pts[ 3 ] = Vec3( -49.1021080f, 25.8964462f, -1.04042661f );
  lambdas = SignedVolume3D( pts[ 0 ], pts[ 1 ], pts[ 2 ], pts[ 3 ] );
  v.Zero();
  for ( int i = 0; i < 4; i++ ) {
    v += pts[ i ] * lambdas[ i ];
  }
  printf( "lambdas: %.3f %.3f %.3f %.3f        v: %.3f %.3f %.3f\n",
    lambdas.x, lambdas.y, lambdas.z, lambdas.w,
    v.x, v.y, v.z
  );
}
```

If everything is working properly, the above test code will give this output:

```
lambdas: 1.000 0.000 0.000 0.000        v: 1.000 1.000 1.000
lambdas: 0.250 0.250 0.250 0.250        v: 0.000 0.000 0.000
lambdas: 0.000 0.333 0.333 0.333        v: -0.667 -0.667 -0.667
lambdas: 0.500 0.000 0.000 0.500        v: 1.000 1.000 0.000
lambdas: 0.290 0.302 0.206 0.202        v: 0.000 -0.000 -0.000
```

A final note, that I think is worth mentioning, is that we are projecting the origin onto all sides of each simplex, every time. And for people who have implemented a GJK algorithm before, may know that this isn't necessary. That's because of how we build up the simplex over time.

GJK is an iterative algorithm, that builds a simplex up over time. And we don't need to check older parts of the simplex, because they were already checked in a previous iteration.

Why are we doing it this way? Why do a brute force "check all sides every time" instead of the "only check the new parts of the simplex"?

It would in fact be more optimal to only check the new features. However, I think that optimization would obfuscate the code. And since this is meant to be an informal introduction to the subject, I'd rather keep the code as clean as possible.

9 Gilbert-Johnson-Keerthi (GJK)

So what is this GJK algorithm all about and why is it so special? Recall that if we take the Minkowski "difference" of two convex shapes then we can easily determine if they intersect.

Well, as it turns out, we don't actually need to build the entire convex hull of the Minkowski "difference". We only need to build the simplex that contains the origin. That significantly increases the performance when the objects intersect. But, when the objects don't intersect, the algorithm will early out, so it's even faster when there's no intersection.

You may recall that when we built the convex hull from a set of points, we would first find a point that was furthest in a particular direction. It was mentioned that was called a support point. These support points become pretty vital to the GJK algorithm. So let's go ahead and review the support functions in the shape classes:

```
class Shape {
public:
  virtual Mat3 InertiaTensor() const = 0;

  virtual Bounds GetBounds( const Vec3 & pos, const Quat & orient ) const = 0;
  virtual Bounds GetBounds() const = 0;

  virtual Vec3 GetCenterOfMass() const { return m_centerOfMass; }

  enum shapeType_t {
    SHAPE_SPHERE,
    SHAPE_BOX,
    SHAPE_CONVEX,
  };
  virtual shapeType_t GetType() const = 0;

  virtual Vec3 Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float bias
    ) const = 0;

  virtual float FastestLinearSpeed( const Vec3 & angularVelocity, const Vec3 & dir ) const {
    return 0.0f; }

protected:
  Vec3 m_centerOfMass;
};

/*
================================
ShapeSphere::Support
================================
*/
Vec3 ShapeSphere::Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float
    bias ) const {
  return ( pos + dir * ( m_radius + bias ) );
}

/*
================================
ShapeBox::Support
================================
*/
Vec3 ShapeBox::Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float bias
    ) const {
  // Find the point in furthest in direction
  Vec3 maxPt = orient.RotatePoint( m_points[ 0 ] ) + pos;
  float maxDist = dir.Dot( maxPt );
  for ( int i = 1; i < m_points.size(); i++ ) {
    const Vec3 pt = orient.RotatePoint( m_points[ i ] ) + pos;
    const float dist = dir.Dot( pt );

    if ( dist > maxDist ) {
      maxDist = dist;
      maxPt = pt;
    }
  }
```

```
  Vec3 norm = dir;
  norm.Normalize();
  norm *= bias;

  return maxPt + norm;
}

/*
================================
ShapeConvex::Support
================================
*/
Vec3 ShapeConvex::Support( const Vec3 & dir, const Vec3 & pos, const Quat & orient, const float
    bias ) const {
  // Find the point in furthest in direction
  Vec3 maxPt = orient.RotatePoint( m_points[ 0 ] ) + pos;
  float maxDist = dir.Dot( maxPt );
  for ( int i = 1; i < m_points.size(); i++ ) {
    const Vec3 pt = orient.RotatePoint( m_points[ i ] ) + pos;
    const float dist = dir.Dot( pt );

    if ( dist > maxDist ) {
      maxDist = dist;
      maxPt = pt;
    }
  }

  Vec3 norm = dir;
  norm.Normalize();
  norm *= bias;

  return maxPt + norm;
}
```

These functions take in a position and orientation, because the shape is defined in model/local space, but we'll want the support point to be in world space. This function is very similar to the FindPointFurthestInDir function. But there's a curious difference, the bias parameter. For right now, it's not very important, as we're just going to pass in zero. However, later it will become important and we'll properly address it then.

You also may be curious why we added a support function for the sphere shape. And that's because this algorithm works for all convex shapes. Not just convex hulls, but any and all convex shapes, you only have to define a support function to get it to work.

Okay, let's really get into the meat of the algorithm. Let's have a look at a 2D illustration of the algorithm. The Minkowski "difference" between two intersecting convex shapes looks something like figure 6.

Figure 6: Minkowski Sum

As we can see, these shapes clearly intersect, and the origin is clearly inside the new convex set. Again, we don't actually want to calculate this Minkowski sum. So we start by getting a random support point on C (the minkowski difference). And we can do that by getting the support point on A and B then taking the difference, because that will be the support point on C. And the code for this is:

```
struct point_t {
  Vec3 xyz; // The point on the minkowski sum
  Vec3 ptA; // The point on bodyA
  Vec3 ptB; // The point on bodyB

  point_t() : xyz( 0.0f ), ptA( 0.0f ), ptB( 0.0f ) {}

  const point_t & operator = ( const point_t & rhs ) {
    xyz = rhs.xyz;
    ptA = rhs.ptA;
    ptB = rhs.ptB;
    return *this;
  }

  bool operator == ( const point_t & rhs ) const {
    return ( ( ptA == rhs.ptA ) && ( ptB == rhs.ptB ) && ( xyz == rhs.xyz ) );
  }
};

point_t Support( const Body * bodyA, const Body * bodyB, Vec3 dir, const float bias ) {
  dir.Normalize();

  point_t point;

  // Find the point in A furthest in direction
  point.ptA = bodyA->m_shape->Support( dir, bodyA->m_position, bodyA->m_orientation, bias );

  dir *= -1.0f;

  // Find the point in B furthest in the opposite direction
  point.ptB = bodyB->m_shape->Support( dir, bodyB->m_position, bodyB->m_orientation, bias );

  // Return the point, in the minkowski sum, furthest in the direction
  point.xyz = point.ptA - point.ptB;
  return point;
}
```

Quickly let's talk about the point_t struct. The reason it exists is because we want more than just the point on C, we also want the points on A and B that created it. This will be essential when we want to construct the contact between A and B. But we'll get to that a little later.

So, now that we have a support point on C, we simply need another support point. But what direction do we choose? We choose the direction that points towards the origin with shortest distance. This will give us our next support point, as illustrated in figure 7.

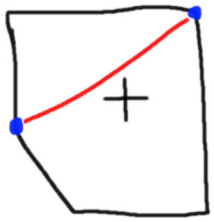


Figure 7: Minkowski Support point 2

Now, we need another support point, and again we choose the search direction by using the direction that points towards the origin with shortest distance. This gives the next support point.

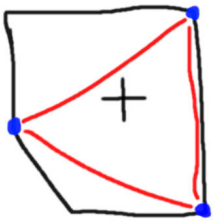


Figure 8: Minkowski Support point 3

Now, if the origin was inside the triangle (or 3-simplex), we'd be done. We'd know that the point was inside and therefore there's an intersection. If it isn't inside the triangle, then we need to throw away a support point and keep searching. We'd simply find which edge of the triangle has a normal that faces the origin, and throw away the other edges, then look for a support point in the normal direction of the remaining edge. Then we'd loop until either the 3-simplex contains the origin, or until we run out of new support points. If we never find a 3-simplex containing the origin, then there's no intersection.

Extending this to full 3D isn't too difficult, it just requires a few extra checks. We do the exact same thing, where we get the 0-simplex, then the 1-simplex, then the 2-simplex. Once we have the triangle (or 2-simplex), that's where the algorithm diverges from the 2D example.

We simply use the normal of the triangle to determine which side of the triangle the origin is on. Then we use that normal (positive if the origin is on that side, negative if the origin is on the other side) to find a new support point and build the 3-simplex (tetrahedron).

If the origin is inside the 3-simplex, then we're done, the intersection occurs. If it isn't, then just like before we have to search for a new support point. And then we continue looping until either the origin is inside the tetrahedron or we run out of new support points.

Okay that was a lot of text trying to describe what's happening. So let's review the algorithm:

1. Find any random support point.
2. Choose a new search direction (in this case, it's the negative of the point's position).
3. Find a new support point with the new search direction.
4. Check if we already have this support point. If we do, then we can't expand any further. Return false.
5. Check if this new point is on the "far" side of the origin from the simplex. If it's on the "same" side, then there's no intersection. Return false.
6. Check if the origin is inside the simplex. If it is, return true. If it isn't, then find a new search direction.
7. Check that the projection of the origin on the simplex is closer than the previous projection. If we're not getting closer to the origin, there's no intersection. Return false.
8. Remove any points from the simplex that do not support the projection of the origin onto the simplex.
9. If there's four simplex points, then there's an intersection. Return true.
10. Go to step 3.

Finally, putting this all in code form:

```
/*
================================
SimplexSignedVolumes

Projects the origin onto the simplex to acquire the new search direction,
also checks if the origin is "inside" the simplex.
================================
```

```
*/
bool SimplexSignedVolumes( point_t * pts, const int num, Vec3 & newDir, Vec4 & lambdasOut ) {
  const float epsilonf = 0.0001f * 0.0001f;
  lambdasOut.Zero();

  bool doesIntersect = false;
  switch ( num ) {
    default:
    case 2: {
      Vec2 lambdas = SignedVolume1D( pts[ 0 ].xyz, pts[ 1 ].xyz );
      Vec3 v( 0.0f );
      for ( int i = 0; i < 2; i++ ) {
        v += pts[ i ].xyz * lambdas[ i ];
      }
      newDir = v * -1.0f;
      doesIntersect = ( v.GetLengthSqr() < epsilonf );
      lambdasOut[ 0 ] = lambdas[ 0 ];
      lambdasOut[ 1 ] = lambdas[ 1 ];
    } break;
    case 3: {
      Vec3 lambdas = SignedVolume2D( pts[ 0 ].xyz, pts[ 1 ].xyz, pts[ 2 ].xyz );
      Vec3 v( 0.0f );
      for ( int i = 0; i < 3; i++ ) {
        v += pts[ i ].xyz * lambdas[ i ];
      }
      newDir = v * -1.0f;
      doesIntersect = ( v.GetLengthSqr() < epsilonf );
      lambdasOut[ 0 ] = lambdas[ 0 ];
      lambdasOut[ 1 ] = lambdas[ 1 ];
      lambdasOut[ 2 ] = lambdas[ 2 ];
    } break;
    case 4: {
      Vec4 lambdas = SignedVolume3D( pts[ 0 ].xyz, pts[ 1 ].xyz, pts[ 2 ].xyz, pts[ 3 ].xyz );
      Vec3 v( 0.0f );
      for ( int i = 0; i < 4; i++ ) {
        v += pts[ i ].xyz * lambdas[ i ];
      }
      newDir = v * -1.0f;
      doesIntersect = ( v.GetLengthSqr() < epsilonf );
      lambdasOut[ 0 ] = lambdas[ 0 ];
      lambdasOut[ 1 ] = lambdas[ 1 ];
      lambdasOut[ 2 ] = lambdas[ 2 ];
      lambdasOut[ 3 ] = lambdas[ 3 ];
    } break;
  };

  return doesIntersect;
}

/*
================================
HasPoint

Checks whether the new point already exists in the simplex
================================
*/
bool HasPoint( const point_t simplexPoints[ 4 ], const point_t & newPt ) {
  const float precision = 1e-6f;

  for ( int i = 0; i < 4; i++ ) {
    Vec3 delta = simplexPoints[ i ].xyz - newPt.xyz;
    if ( delta.GetLengthSqr() < precision * precision ) {
      return true;
    }
  }
  return false;
}

/*
================================
SortValids
```

```
Sorts the valid support points to the beginning of the array
================================
*/
void SortValids( point_t simplexPoints[ 4 ], Vec4 & lambdas ) {
  bool valids[ 4 ];
  for ( int i = 0; i < 4; i++ ) {
    valids[ i ] = true;
    if ( lambdas[ i ] == 0.0f ) {
      valids[ i ] = false;
    }
  }

  Vec4 validLambdas( 0.0f );
  int validCount = 0;
  point_t validPts[ 4 ];
  memset( validPts, 0, sizeof( point_t ) * 4 );
  for ( int i = 0; i < 4; i++ ) {
    if ( valids[ i ] ) {
      validPts[ validCount ] = simplexPoints[ i ];
      validLambdas[ validCount ] = lambdas[ i ];
      validCount++;
    }
  }

  // Copy the valids back into simplexPoints
  for ( int i = 0; i < 4; i++ ) {
    simplexPoints[ i ] = validPts[ i ];
    lambdas[ i ] = validLambdas[ i ];
  }
}

/*
================================
NumValids
================================
*/
static int NumValids( const Vec4 & lambdas ) {
  int num = 0;
  for ( int i = 0; i < 4; i++ ) {
    if ( 0.0f != lambdas[ i ] ) {
      num++;
    }
  }
  return num;
}

/*
================================
GJK_DoesIntersect
================================
*/
bool GJK_DoesIntersect( const Body * bodyA, const Body * bodyB ) {
  const Vec3 origin( 0.0f );

  int numPts = 1;
  point_t simplexPoints[ 4 ];
  simplexPoints[ 0 ] = Support( bodyA, bodyB, Vec3( 1, 1, 1 ), 0.0f );

  float closestDist = 1e10f;
  bool doesContainOrigin = false;
  Vec3 newDir = simplexPoints[ 0 ].xyz * -1.0f;
  do {
    // Get the new point to check on
    point_t newPt = Support( bodyA, bodyB, newDir, 0.0f );

    // If the new point is the same as a previous point, then we can't expand any further
    if ( HasPoint( simplexPoints, newPt ) ) {
      break;
    }
```

```
    simplexPoints[ numPts ] = newPt;
    numPts++;

    // If this new point hasn't moved passed the origin, then the
    // origin cannot be in the set. And therefore there is no collision.
    float dotdot = newDir.Dot( newPt.xyz - origin );
    if ( dotdot < 0.0f ) {
      break;
    }

    Vec4 lambdas;
    doesContainOrigin = SimplexSignedVolumes( simplexPoints, numPts, newDir, lambdas );
    if ( doesContainOrigin ) {
      break;
    }

    // Check that the new projection of the origin onto the simplex is closer than the previous
    float dist = newDir.GetLengthSqr();
    if ( dist >= closestDist ) {
      break;
    }
    closestDist = dist;

    // Use the lambdas that support the new search direction, and invalidate any points that don't
     support it
    SortValids( simplexPoints, lambdas );
    numPts = NumValids( lambdas );
    doesContainOrigin = ( 4 == numPts );
  } while ( !doesContainOrigin );

  return doesContainOrigin;
}
```

Okay, this is an important algorithm. So, really take your time to examine it.

Something I neglected to mention before is why we need to check that the new projected point is closer to the origin than the previous projection. This is because of floating point errors.

Imagine the convex hull of the minkowski difference (also known as configuration space object or CSO). If the CSO has a face that's a quad, then if the origin is close to the diagonal of the quad, then we can have a situation where we constantly flip between two different support points. Then we'll end up looping forever. So, an easy check to prevent that is to make sure we're always moving the simplex closer to the origin.

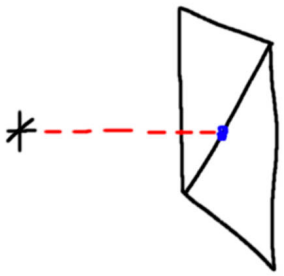


Figure 9: simplex infinite toggle bug

10 Expanding Polytope Algorithm (EPA)

So this is great. We have an algorithm that tells if two convex shapes intersect. But we need more than that. We need the contact point on A and the point on B and the contact normal between them. Otherwise we won't be able to properly resolve the collision between the two bodies.

How do we get that information?

Well, it turns out that the point on the surface of the convex set C, that is closest to the origin, will give us the necessary information. That point, maps to the point on A and B of greatest penetration. And the direction from the origin to that point on C, gives the contact normal.

Figure 10: Minkowski Collision Normal

Now, is there an algorithm that we can exploit to find the surface point on C that is closest to the origin?

Yes, and that is called the expanding polytope algorithm. Imagine we have the situation where we found a simplex that contains the origin, therefore we know there's an intersection, but we don't yet have any of the surfaces on C, such as in figure 8.

Well, we just need to find the surface on C that is closest to the origin. To do this, we measure the distance from the origin to all sides of the simplex. Whichever surface of the simplex is closest, we throw that away, then we expand in the direction of the normal of that surface. What we're doing is building the convex hull of C, but we're only interested in building the part that is closest to the origin.

Once the closest surface can no longer be expanded further away from the origin, then we have the surface that is closest to the origin. And that surface will contain our contact point. All we have to do now is cast a ray from the origin to the surface. Whatever point that is, is our contact point.

We just need to calculate the barycentric points of the intersection with that face. Then use those to reconstruct the points on A and B.

So, writing out the code for this, we get:

```
/*
================================
BarycentricCoordinates

This borrows our signed volume code to perform the barycentric coordinates.
================================
*/
Vec3 BarycentricCoordinates( Vec3 s1, Vec3 s2, Vec3 s3, const Vec3 & pt ) {
  s1 = s1 - pt;
  s2 = s2 - pt;
  s3 = s3 - pt;

  Vec3 normal = ( s2 - s1 ).Cross( s3 - s1 );
  Vec3 p0 = normal * s1.Dot( normal ) / normal.GetLengthSqr();

  // Find the axis with the greatest projected area
  int idx = 0;
  float area_max = 0;
  for ( int i = 0; i < 3; i++ ) {
    int j = ( i + 1 ) % 3;
    int k = ( i + 2 ) % 3;

    Vec2 a = Vec2( s1[ j ], s1[ k ] );
```

```
    Vec2 b = Vec2( s2[ j ], s2[ k ] );
    Vec2 c = Vec2( s3[ j ], s3[ k ] );
    Vec2 ab = b - a;
    Vec2 ac = c - a;

    float area = ab.x * ac.y - ab.y * ac.x;
    if ( area * area > area_max * area_max ) {
      idx = i;
      area_max = area;
    }
  }

  // Project onto the appropriate axis
  int x = ( idx + 1 ) % 3;
  int y = ( idx + 2 ) % 3;
  Vec2 s[ 3 ];
  s[ 0 ] = Vec2( s1[ x ], s1[ y ] );
  s[ 1 ] = Vec2( s2[ x ], s2[ y ] );
  s[ 2 ] = Vec2( s3[ x ], s3[ y ] );
  Vec2 p = Vec2( p0[ x ], p0[ y ] );

  // Get the sub-areas of the triangles formed from the projected origin and the edges
  Vec3 areas;
  for ( int i = 0; i < 3; i++ ) {
    int j = ( i + 1 ) % 3;
    int k = ( i + 2 ) % 3;

    Vec2 a = p;
    Vec2 b = s[ j ];
    Vec2 c = s[ k ];
    Vec2 ab = b - a;
    Vec2 ac = c - a;

    areas[ i ] = ab.x * ac.y - ab.y * ac.x;
  }

  Vec3 lambdas = areas / area_max;
  if ( !lambdas.IsValid() ) {
    lambdas = Vec3( 1, 0, 0 );
  }
  return lambdas;
}

/*
================================
NormalDirection
================================
*/
Vec3 NormalDirection( const tri_t & tri, const std::vector< point_t > & points ) {
  const Vec3 & a = points[ tri.a ].xyz;
  const Vec3 & b = points[ tri.b ].xyz;
  const Vec3 & c = points[ tri.c ].xyz;

  Vec3 ab = b - a;
  Vec3 ac = c - a;
  Vec3 normal = ab.Cross( ac );
  normal.Normalize();
  return normal;
}

/*
================================
SignedDistanceToTriangle
================================
*/
float SignedDistanceToTriangle( const tri_t & tri, const Vec3 & pt, const std::vector< point_t > &
    points ) {
  const Vec3 normal = NormalDirection( tri, points );
  const Vec3 & a = points[ tri.a ].xyz;
  const Vec3 a2pt = pt - a;
  const float dist = normal.Dot( a2pt );
```

```
  return dist;
}

/*
================================
ClosestTriangle
================================
*/
int ClosestTriangle( const std::vector< tri_t > & triangles, const std::vector< point_t > & points
    ) {
  float minDistSqr = 1e10;

  int idx = -1;
  for ( int i = 0; i < triangles.size(); i++ ) {
    const tri_t & tri = triangles[ i ];

    float dist = SignedDistanceToTriangle( tri, Vec3( 0.0f ), points );
    float distSqr = dist * dist;
    if ( distSqr < minDistSqr ) {
      idx = i;
      minDistSqr = distSqr;
    }
  }

  return idx;
}

/*
================================
HasPoint
================================
*/
bool HasPoint( const Vec3 & w, const std::vector< tri_t > triangles, const std::vector< point_t >
    & points ) {
  const float epsilons = 0.001f * 0.001f;
  Vec3 delta;

  for ( int i = 0; i < triangles.size(); i++ ) {
    const tri_t & tri = triangles[ i ];

    delta = w - points[ tri.a ].xyz;
    if ( delta.GetLengthSqr() < epsilons ) {
      return true;
    }
    delta = w - points[ tri.b ].xyz;
    if ( delta.GetLengthSqr() < epsilons ) {
      return true;
    }
    delta = w - points[ tri.c ].xyz;
    if ( delta.GetLengthSqr() < epsilons ) {
      return true;
    }
  }
  return false;
}

/*
================================
RemoveTrianglesFacingPoint
================================
*/
int RemoveTrianglesFacingPoint( const Vec3 & pt, std::vector< tri_t > & triangles, const std::
    vector< point_t > & points ) {
  int numRemoved = 0;
  for ( int i = 0; i < triangles.size(); i++ ) {
    const tri_t & tri = triangles[ i ];

    float dist = SignedDistanceToTriangle( tri, pt, points );
    if ( dist > 0.0f ) {
      // This triangle faces the point.  Remove it.
      triangles.erase( triangles.begin() + i );
```

```
      i--;
      numRemoved++;
    }
  }
  return numRemoved;
}

/*
================================
FindDanglingEdges
================================
*/
void FindDanglingEdges( std::vector< edge_t > & danglingEdges, const std::vector< tri_t > &
    triangles ) {
  danglingEdges.clear();

  for ( int i = 0; i < triangles.size(); i++ ) {
    const tri_t & tri = triangles[ i ];

    edge_t edges[ 3 ];
    edges[ 0 ].a = tri.a;
    edges[ 0 ].b = tri.b;

    edges[ 1 ].a = tri.b;
    edges[ 1 ].b = tri.c;

    edges[ 2 ].a = tri.c;
    edges[ 2 ].b = tri.a;

    int counts[ 3 ];
    counts[ 0 ] = 0;
    counts[ 1 ] = 0;
    counts[ 2 ] = 0;

    for ( int j = 0; j < triangles.size(); j++ ) {
      if ( j == i ) {
        continue;
      }

      const tri_t & tri2 = triangles[ j ];

      edge_t edges2[ 3 ];
      edges2[ 0 ].a = tri2.a;
      edges2[ 0 ].b = tri2.b;

      edges2[ 1 ].a = tri2.b;
      edges2[ 1 ].b = tri2.c;

      edges2[ 2 ].a = tri2.c;
      edges2[ 2 ].b = tri2.a;

      for ( int k = 0; k < 3; k++ ) {
        if ( edges[ k ] == edges2[ 0 ] ) {
          counts[ k ]++;
        }
        if ( edges[ k ] == edges2[ 1 ] ) {
          counts[ k ]++;
        }
        if ( edges[ k ] == edges2[ 2 ] ) {
          counts[ k ]++;
        }
      }
    }

    // An edge that isn't shared, is dangling
    for ( int k = 0; k < 3; k++ ) {
      if ( 0 == counts[ k ] ) {
        danglingEdges.push_back( edges[ k ] );
      }
    }
  }
```

```
}

/*
================================
EPA_Expand
================================
*/
float EPA_Expand( const Body * bodyA, const Body * bodyB, const float bias, const point_t
    simplexPoints[ 4 ], Vec3 & ptOnA, Vec3 & ptOnB ) {
  std::vector< point_t > points;
  std::vector< tri_t > triangles;
  std::vector< edge_t > danglingEdges;

  Vec3 center( 0.0f );
  for ( int i = 0; i < 4; i++ ) {
    points.push_back( simplexPoints[ i ] );
    center += simplexPoints[ i ].xyz;
  }
  center *= 0.25f;

  // Build the triangles
  for ( int i = 0; i < 4; i++ ) {
    int j = ( i + 1 ) % 4;
    int k = ( i + 2 ) % 4;
    tri_t tri;
    tri.a = i;
    tri.b = j;
    tri.c = k;

    int unusedPt = ( i + 3 ) % 4;
    float dist = SignedDistanceToTriangle( tri, points[ unusedPt ].xyz, points );

    // The unused point is always on the negative/inside of the triangle.. make sure the normal
    points away
    if ( dist > 0.0f ) {
      std::swap( tri.a, tri.b );
    }

    triangles.push_back( tri );
  }

  //
  //  Expand the simplex to find the closest face of the CSO to the origin
  //
  while ( 1 ) {
    const int idx = ClosestTriangle( triangles, points );
    Vec3 normal = NormalDirection( triangles[ idx ], points );

    const point_t newPt = Support( bodyA, bodyB, normal, bias );

    // if w already exists, then just stop
    // because it means we can't expand any further
    if ( HasPoint( newPt.xyz, triangles, points ) ) {
      break;
    }

    float dist = SignedDistanceToTriangle( triangles[ idx ], newPt.xyz, points );
    if ( dist <= 0.0f ) {
      break;  // can't expand
    }

    const int newIdx = (int)points.size();
    points.push_back( newPt );

    // Remove Triangles that face this point
    int numRemoved = RemoveTrianglesFacingPoint( newPt.xyz, triangles, points );
    if ( 0 == numRemoved ) {
      break;
    }

    // Find Dangling Edges
```

```
    danglingEdges.clear();
    FindDanglingEdges( danglingEdges, triangles );
    if ( 0 == danglingEdges.size() ) {
      break;
    }

    // In theory the edges should be a proper CCW order
    // So we only need to add the new point as 'a' in order
    // to create new triangles that face away from origin
    for ( int i = 0; i < danglingEdges.size(); i++ ) {
      const edge_t & edge = danglingEdges[ i ];

      tri_t triangle;
      triangle.a = newIdx;
      triangle.b = edge.b;
      triangle.c = edge.a;

      // Make sure it's oriented properly
      float dist = SignedDistanceToTriangle( triangle, center, points );
      if ( dist > 0.0f ) {
        std::swap( triangle.b, triangle.c );
      }

      triangles.push_back( triangle );
    }
  }

  // Get the projection of the origin on the closest triangle
  const int idx = ClosestTriangle( triangles, points );
  const tri_t & tri = triangles[ idx ];
  Vec3 ptA_w = points[ tri.a ].xyz;
  Vec3 ptB_w = points[ tri.b ].xyz;
  Vec3 ptC_w = points[ tri.c ].xyz;
  Vec3 lambdas = BarycentricCoordinates( ptA_w, ptB_w, ptC_w, Vec3( 0.0f ) );

  // Get the point on shape A
  Vec3 ptA_a = points[ tri.a ].ptA;
  Vec3 ptB_a = points[ tri.b ].ptA;
  Vec3 ptC_a = points[ tri.c ].ptA;
  ptOnA = ptA_a * lambdas[ 0 ] + ptB_a * lambdas[ 1 ] + ptC_a * lambdas[ 2 ];

  // Get the point on shape B
  Vec3 ptA_b = points[ tri.a ].ptB;
  Vec3 ptB_b = points[ tri.b ].ptB;
  Vec3 ptC_b = points[ tri.c ].ptB;
  ptOnB = ptA_b * lambdas[ 0 ] + ptB_b * lambdas[ 1 ] + ptC_b * lambdas[ 2 ];

  // Return the penetration distance
  Vec3 delta = ptOnB - ptOnA;
  return delta.GetMagnitude();
}
```

As you can see, we use the results from GJK to build the tetrahedron containing the origin. Then we just loop over all the surfaces, to expand in the direction of the closest surface. Once we can no longer expand, we have the closest surface.

You may recall that earlier we defined the struct point_t to contain the associated support points in A and B as well as the result in C. You can probably see why we did this now. Once we have the surface on C that's closest to the origin, we use the barycentric coordinates to calculate the points on A and B.

And finally, we use the bias term. Why in the world do we use a bias here? The reason, is that if the two sets A and B are just barely touching, then the origin will be on the surface of C. This causes a problem for us. Since this will map the point on A and the point on B to be nearly the same point. And if those two points are too close to each other, then we can't accurately calculate the contact normal.

To curtail this problem, we simply use the bias to mildly expand the sets A and B. So when we first check if the two are overlapping, we expand them, then build the contact information. And this solves the problem of calculating an accurate contact normal.

Okay, since this EPA function does need a full tetrahedron. Let's go ahead and modify our GJK function to fully form the 3-simplex and then directly pass it into the EPA algorithm. This way we have a one stop

function for both determining if an intersection happens and the calculating the contact points.

```
bool GJK_DoesIntersect( const Body * bodyA, const Body * bodyB, const float bias, Vec3 & ptOnA,
    Vec3 & ptOnB ) {
  const Vec3 origin( 0.0f );

  int numPts = 1;
  point_t simplexPoints[ 4 ];
  simplexPoints[ 0 ] = Support( bodyA, bodyB, Vec3( 1, 1, 1 ), 0.0f );

  float closestDist = 1e10f;
  bool doesContainOrigin = false;
  Vec3 newDir = simplexPoints[ 0 ].xyz * -1.0f;
  do {
    // Get the new point to check on
    point_t newPt = Support( bodyA, bodyB, newDir, 0.0f );

    // If the new point is the same as a previous point, then we can't expand any further
    if ( HasPoint( simplexPoints, newPt ) ) {
      break;
    }

    simplexPoints[ numPts ] = newPt;
    numPts++;

    // If this new point hasn't moved passed the origin, then the
    // origin cannot be in the set. And therefore there is no collision.
    float dotdot = newDir.Dot( newPt.xyz - origin );
    if ( dotdot < 0.0f ) {
      break;
    }

    Vec4 lambdas;
    doesContainOrigin = SimplexSignedVolumes( simplexPoints, numPts, newDir, lambdas );
    if ( doesContainOrigin ) {
      break;
    }

    // Check that the new projection of the origin onto the simplex is closer than the previous
    float dist = newDir.GetLengthSqr();
    if ( dist >= closestDist ) {
      break;
    }
    closestDist = dist;

    // Use the lambdas that support the new search direction, and invalidate any points that don't
     support it
    SortValids( simplexPoints, lambdas );
    numPts = NumValids( lambdas );
    doesContainOrigin = ( 4 == numPts );
  } while ( !doesContainOrigin );

  if ( !doesContainOrigin ) {
    return false;
  }

  //
  //  Check that we have a 3-simplex (EPA expects a tetrahedron)
  //
  if ( 1 == numPts ) {
    Vec3 searchDir = simplexPoints[ 0 ].xyz * -1.0f;
    point_t newPt = Support( bodyA, bodyB, searchDir, 0.0f );
    simplexPoints[ numPts ] = newPt;
    numPts++;
  }
  if ( 2 == numPts ) {
    Vec3 ab = simplexPoints[ 1 ].xyz - simplexPoints[ 0 ].xyz;
    Vec3 u, v;
    ab.GetOrtho( u, v );

    Vec3 newDir = u;
```

```
    point_t newPt = Support( bodyA, bodyB, newDir, 0.0f );
    simplexPoints[ numPts ] = newPt;
    numPts++;
  }
  if ( 3 == numPts ) {
    Vec3 ab = simplexPoints[ 1 ].xyz - simplexPoints[ 0 ].xyz;
    Vec3 ac = simplexPoints[ 2 ].xyz - simplexPoints[ 0 ].xyz;
    Vec3 norm = ab.Cross( ac );

    Vec3 newDir = norm;
    point_t newPt = Support( bodyA, bodyB, newDir, 0.0f );
    simplexPoints[ numPts ] = newPt;
    numPts++;
  }

  //
  // Expand the simplex by the bias amount
  //

  // Get the center point of the simplex
  Vec3 avg = Vec3( 0, 0, 0 );
  for ( int i = 0; i < 4; i++ ) {
    avg += simplexPoints[ i ].xyz;
  }
  avg *= 0.25f;

  // Now expand the simplex by the bias amount
  for ( int i = 0; i < numPts; i++ ) {
    point_t & pt = simplexPoints[ i ];

    Vec3 dir = pt.xyz - avg;  // ray from "center" to witness point
    dir.Normalize();
    pt.ptA += dir * bias;
    pt.ptB -= dir * bias;
    pt.xyz = pt.ptA - pt.ptB;
  }

  //
  // Perform EPA expansion of the simplex to find the closest face on the CSO
  //
  EPA_Expand( bodyA, bodyB, bias, simplexPoints, ptOnA, ptOnB );
  return true;
}
```

11 Closest Points

Before we put all this together into our intersection tests. I'd like to discuss the closest points gjk algorithm.

It's going to turn out, that even when objects aren't intersecting, it's very useful to know the closest points on the two.

As you can probably guess, if we have the face on the surface of the CSO that's closest to the origin, then even if there's no intersection; we can use it to get the closest points.

This modified GJK function will assume that there's no intersection. That means we can remove a whole lot of early out checks. Because we actually just want to build and crawl the simplex until it can no longer get closer to the origin. Which makes its code simpler than the full GJK:

```
void GJK_ClosestPoints( const Body * bodyA, const Body * bodyB, Vec3 & ptOnA, Vec3 & ptOnB ) {
  const Vec3 origin( 0.0f );

  float closestDist = 1e10f;
  const float bias = 0.0f;

  int numPts = 1;
  point_t simplexPoints[ 4 ];
  simplexPoints[ 0 ] = Support( bodyA, bodyB, Vec3( 1, 1, 1 ), bias );

  Vec4 lambdas = Vec4( 1, 0, 0, 0 );
  Vec3 newDir = simplexPoints[ 0 ].xyz * -1.0f;
  do {
    // Get the new point to check on
    point_t newPt = Support( bodyA, bodyB, newDir, bias );

    // If the new point is the same as a previous point, then we can't expand any further
    if ( HasPoint( simplexPoints, newPt ) ) {
      break;
    }

    // Add point and get new search direction
    simplexPoints[ numPts ] = newPt;
    numPts++;

    SimplexSignedVolumes( simplexPoints, numPts, newDir, lambdas );
    SortValids( simplexPoints, lambdas );
    numPts = NumValids( lambdas );

    // Check that the new projection of the origin onto the simplex is closer than the previous
    float dist = newDir.GetLengthSqr();
    if ( dist >= closestDist ) {
      break;
    }
    closestDist = dist;
  } while ( numPts < 4 );

  ptOnA.Zero();
  ptOnB.Zero();
  for ( int i = 0; i < 4; i++ ) {
    ptOnA += simplexPoints[ i ].ptA * lambdas[ i ];
    ptOnB += simplexPoints[ i ].ptB * lambdas[ i ];
  }
}
```

12 Putting it all together

Now, we just need to update our intersection function, and we should be good to go. All we need is to check if the GJK returns true, and if it does, run the EPA:

```
bool SphereSphereStatic( const ShapeSphere * sphereA, const ShapeSphere * sphereB, const Vec3 &
    posA, const Vec3 & posB, Vec3 & ptOnA, Vec3 & ptOnB ) {
  const Vec3 ab = posB - posA;
  Vec3 norm = ab;
  norm.Normalize();

  ptOnA = posA + norm * sphereA->m_radius;
  ptOnB = posB - norm * sphereB->m_radius;

  const float radiusAB = sphereA->m_radius + sphereB->m_radius;
  const float lengthSquare = ab.GetLengthSqr();
  if ( lengthSquare <= ( radiusAB * radiusAB ) ) {
    return true;
  }

  return false;
}

bool Intersect( Body * bodyA, Body * bodyB, contact_t & contact ) {
  contact.bodyA = bodyA;
  contact.bodyB = bodyB;
  contact.timeOfImpact = 0.0f;

  if ( bodyA->m_shape->GetType() == Shape::SHAPE_SPHERE && bodyB->m_shape->GetType() == Shape::
    SHAPE_SPHERE ) {
    const ShapeSphere * sphereA = (const ShapeSphere *)bodyA->m_shape;
    const ShapeSphere * sphereB = (const ShapeSphere *)bodyB->m_shape;

    Vec3 posA = bodyA->m_position;
    Vec3 posB = bodyB->m_position;

    if ( SphereSphereStatic( sphereA, sphereB, posA, posB, contact.ptOnA_WorldSpace, contact.
    ptOnB_WorldSpace ) ) {
      contact.normal = posA - posB;
      contact.normal.Normalize();

      contact.ptOnA_LocalSpace = bodyA->WorldSpaceToBodySpace( contact.ptOnA_WorldSpace );
      contact.ptOnB_LocalSpace = bodyB->WorldSpaceToBodySpace( contact.ptOnB_WorldSpace );

      Vec3 ab = bodyB->m_position - bodyA->m_position;
      float r = ab.GetMagnitude() - ( sphereA->m_radius + sphereB->m_radius );
      contact.separationDistance = r;
      return true;
    }
  } else {
    Vec3 ptOnA;
    Vec3 ptOnB;
    const float bias = 0.001f;
    if ( GJK_DoesIntersect( bodyA, bodyB, bias, ptOnA, ptOnB ) ) {
      // There was an intersection, so get the contact data
      Vec3 normal = ptOnB - ptOnA;
      normal.Normalize();

      ptOnA -= normal * bias;
      ptOnB += normal * bias;

      contact.normal = normal;

      contact.ptOnA_WorldSpace = ptOnA;
      contact.ptOnB_WorldSpace = ptOnB;

      contact.ptOnA_LocalSpace = bodyA->WorldSpaceToBodySpace( contact.ptOnA_WorldSpace );
      contact.ptOnB_LocalSpace = bodyB->WorldSpaceToBodySpace( contact.ptOnB_WorldSpace );

      Vec3 ab = bodyB->m_position - bodyA->m_position;
```

```
      float r = ( ptOnA - ptOnB ).GetMagnitude();
      contact.separationDistance = -r;
      return true;
    }

    // There was no collision, but we still want the contact data, so get it
    GJK_ClosestPoints( bodyA, bodyB, ptOnA, ptOnB );
    contact.ptOnA_WorldSpace = ptOnA;
    contact.ptOnB_WorldSpace = ptOnB;

    contact.ptOnA_LocalSpace = bodyA->WorldSpaceToBodySpace( contact.ptOnA_WorldSpace );
    contact.ptOnB_LocalSpace = bodyB->WorldSpaceToBodySpace( contact.ptOnB_WorldSpace );

    Vec3 ab = bodyB->m_position - bodyA->m_position;
    float r = ( ptOnA - ptOnB ).GetMagnitude();
    contact.separationDistance = r;
  }
  return false;
}
```

As you can see, we don't throw away our sphere-sphere intersection test. Although GJK would work for sphere-sphere tests, the custom check is faster than GJK. In a production quality physics engine we would write a custom shape per shape collision check to get the fastest possible intersection tests (ie capsules, cylinders, etc would all get their own custom collision code). And we'd only use GJK when we don't have a faster intersection test available. However, since this is only an informal introduction to physics simulation, I will defer those discussions to the references in the further reading section.

Now, we only need to update our continuous collision detection algorithms. Then we can finally check if these objects will intersect.

13 CCD Revisited - Conservative Advance

It's great that we finally have static collision between all of our shapes. However, we only have continuous collision for our spheres. We need to develop an algorithm for general convex shapes. Fortunately, there's already an algorithm for this out there. It's called conservative advance.

The algorithm is actually pretty simple. We simply find the closest distance between two objects, and then we calculate the fastest velocity on A and the fastest velocity on B, both projected onto the ray separating the two bodies. Using these velocities we can determine the minimum time required for the two shapes to come into contact.

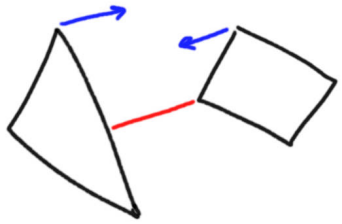


Figure 11: conservative advance

Then we just step the two bodies forward in time, and check for collision. If they collide, we're done. If they don't collide then we repeat this process until we've run out of time for the frame:

```
bool ConservativeAdvance( Body * bodyA, Body * bodyB, float dt, contact_t & contact ) {
  contact.bodyA = bodyA;
  contact.bodyB = bodyB;

  float toi = 0.0f;

  int numIters = 0;

  // Advance the positions of the bodies until they touch or there's not time left
  while ( dt > 0.0f ) {
    // Check for intersection
    bool didIntersect = Intersect( bodyA, bodyB, contact );
    if ( didIntersect ) {
      contact.timeOfImpact = toi;
      bodyA->Update( -toi );
      bodyB->Update( -toi );
      return true;
    }

    ++numIters;
    if ( numIters > 10 ) {
      break;
    }

    // Get the vector from the closest point on A to the closest point on B
    Vec3 ab = contact.ptOnB_WorldSpace - contact.ptOnA_WorldSpace;
    ab.Normalize();

    // project the relative velocity onto the ray of shortest distance
    Vec3 relativeVelocity = bodyA->m_linearVelocity - bodyB->m_linearVelocity;
    float orthoSpeed = relativeVelocity.Dot( ab );

    // Add to the orthoSpeed the maximum angular speeds of the relative shapes
    float angularSpeedA = bodyA->m_shape->FastestLinearSpeed( bodyA->m_angularVelocity, ab );
    float angularSpeedB = bodyB->m_shape->FastestLinearSpeed( bodyB->m_angularVelocity, ab * -1.0f
     );
    orthoSpeed += angularSpeedA + angularSpeedB;
    if ( orthoSpeed <= 0.0f ) {
```

```
      break;
    }

    float timeToGo = contact.separationDistance / orthoSpeed;
    if ( timeToGo > dt ) {
      break;
    }

    dt -= timeToGo;
    toi += timeToGo;
    bodyA->Update( timeToGo );
    bodyB->Update( timeToGo );
  }

  // unwind the clock
  bodyA->Update( -toi );
  bodyB->Update( -toi );
  return false;
}
```

A quick note here is that we do have a max number of iterations. This is to avoid situations where the bodies rotate very quickly, but don't translate. Because with high rates of rotation, we'll sit here calculating tiny time steps, but then the bodies don't actually move towards one another. So, to avoid sitting there looping forever, we put a limit on the number of iterations.

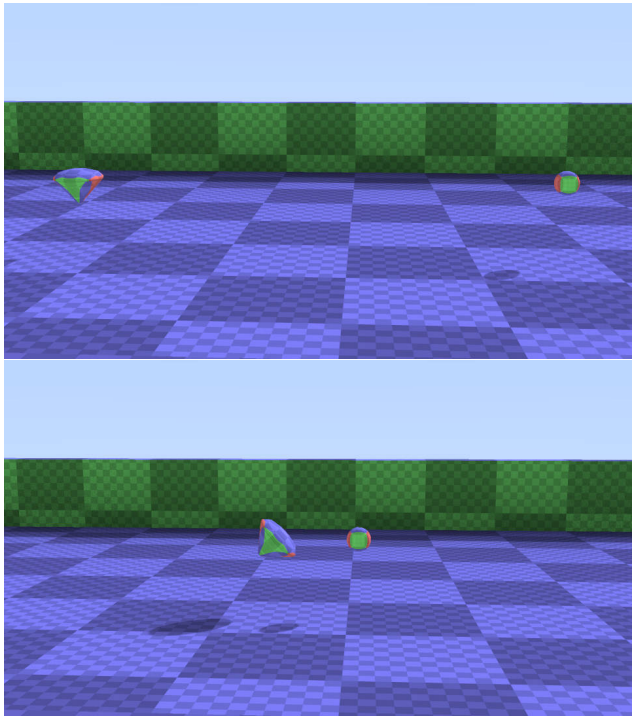
Now, let's put it all together to finalize continuous collision detection for all shapes:

```
bool Intersect( Body * bodyA, Body * bodyB, const float dt, contact_t & contact ) {
  contact.bodyA = bodyA;
  contact.bodyB = bodyB;

  if ( bodyA->m_shape->GetType() == Shape::SHAPE_SPHERE && bodyB->m_shape->GetType() == Shape::
    SHAPE_SPHERE ) {
    const ShapeSphere * sphereA = (const ShapeSphere *)bodyA->m_shape;
    const ShapeSphere * sphereB = (const ShapeSphere *)bodyB->m_shape;

    Vec3 posA = bodyA->m_position;
    Vec3 posB = bodyB->m_position;

    Vec3 velA = bodyA->m_linearVelocity;
    Vec3 velB = bodyB->m_linearVelocity;

    if ( SphereSphereDynamic( sphereA, sphereB, posA, posB, velA, velB, dt, contact.
    ptOnA_WorldSpace, contact.ptOnB_WorldSpace, contact.timeOfImpact ) ) {
      // Step bodies forward to get local space collision points
      bodyA->Update( contact.timeOfImpact );
      bodyB->Update( contact.timeOfImpact );

      // Convert world space contacts to local space
      contact.ptOnA_LocalSpace = bodyA->WorldSpaceToBodySpace( contact.ptOnA_WorldSpace );
      contact.ptOnB_LocalSpace = bodyB->WorldSpaceToBodySpace( contact.ptOnB_WorldSpace );

      contact.normal = bodyA->m_position - bodyB->m_position;
      contact.normal.Normalize();

      // Unwind time step
      bodyA->Update( -contact.timeOfImpact );
      bodyB->Update( -contact.timeOfImpact );

      // Calculate the separation distance
      Vec3 ab = bodyB->m_position - bodyA->m_position;
      float r = ab.GetMagnitude() - ( sphereA->m_radius + sphereB->m_radius );
      contact.separationDistance = r;
      return true;
    }
  } else {
    // Use GJK to perform conservative advancement
    bool result = ConservativeAdvance( bodyA, bodyB, dt, contact );
    return result;
  }
  return false;
```

```
}
```

Now we can setup a situation where we have two fast moving bodies and compare the teleportation bug and how conservative advance fixes it:

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

  body.m_position = Vec3( 10, 0, 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, 0, 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 );

  AddStandardSandBox( m_bodies );
}
```



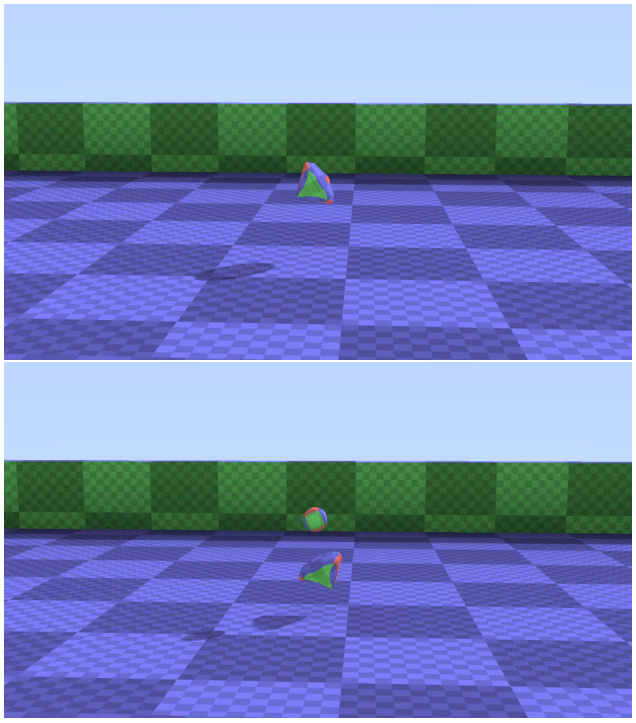
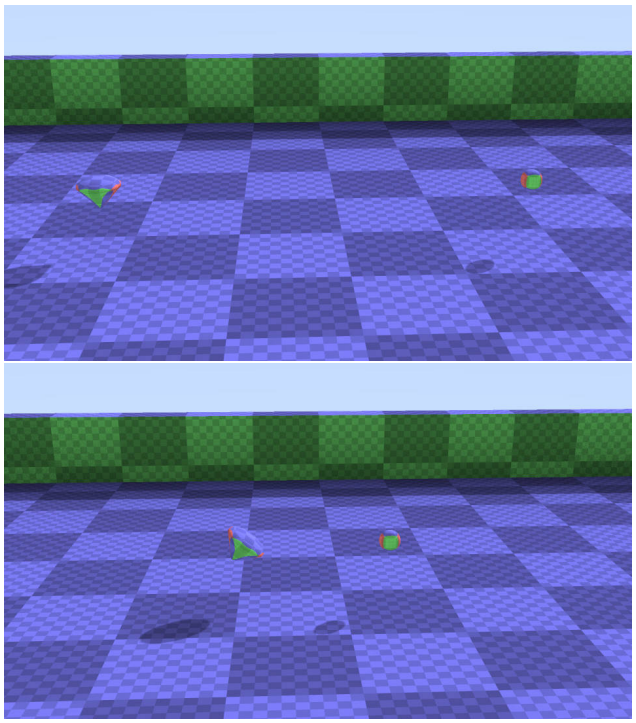


Figure 12: CCD Off

As you can see, when there's no continuous collision detection, the two bodies completely interpenetrate between the frames. And the collision isn't detected until the start of the next frame, when the two bodies are inside each other.



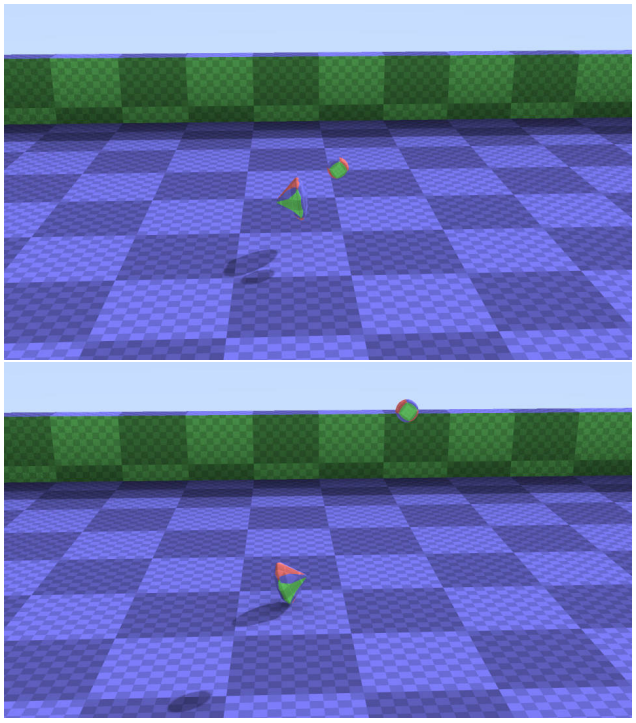


Figure 13: CCD On

And now with proper continuous collision detection, the collision that occurs between the time steps (or between the frames), is properly detected and handled.

14 Bonus: Monte Carlo Calculation of the Inertia Tensor

I'm considering this chapter as a bonus chapter, since you can choose to ignore it and still complete the other books in the series. However, even if you don't implement this chapter, it is likely still worth reading through it.

The purpose of this chapter is to discuss an alternative method for calculating the inertia tensor for generic shapes. The numerical integrator we previously used, for calculating the inertia tensor, samples the bounding box of the shape in a linear fashion. While this integrator works for a large number of samples, it can be slow to converge for some shapes, since it might over/under sample the shape.

We can illustrate this issue with an example. Suppose we want to approximate the value for π. We know that the area of a circle is $\pi * r^2$, where r is the radius of the circle. And the area of a square is s^2, where s is the length of the side.

Then if we inscribe a quarter of a circle of radius 1 inside a square of side 1, like in figure 14

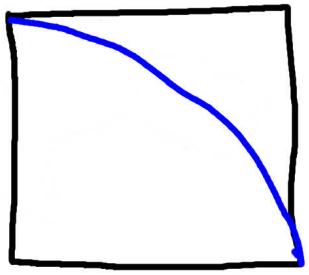


Figure 14: Inscribed Circle

Then we know that area of the quarter circle is $\frac{pi}{4}$ and the area of the square is 1. And if we sample across the square in a uniform fashion, then the ratio of the between the areas should equal the ratio of the samples that fall inside circle and the total number of samples taken.

$$\frac{A_{circle}}{A_{square}} = \frac{N_{circle}}{N_{total}} \tag{7}$$

Using that relationship, and the known areas of the circle and square, we can easily calculate pi:

$$\pi = 4 * \frac{N_{circle}}{N_{total}} \tag{8}$$

Now, suppose we choose to integrate this in a linear fashion. We might choose to take a total of 100 samples, 10 evenly spaced columns of 10 evenly spaced rows. From a visual inspection, we can see that the first row hardly samples the circle.

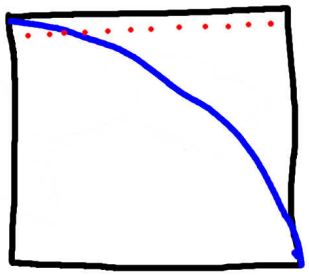


Figure 15: Sampled Row

As we can see, the first row under samples the circle. This means it'll take many samples before we can get a good approximation to π. Is there a better way?

A method that tends to converge quicker is known as Monte Carlo integration. Instead of sampling the space in a uniform grid, we will use a random number generator to sample the space.

So, how do we generate random numbers? We could use the rand functions provided in the C library, or we could use some higher quality generators provided in the more modern C++ libraries. In this case, we'll use the mersenne twister and a uniform distribution.

```
class Random {
public:
  static float Get() { return m_distribution( m_generator ); }

private:
  static std::mt19937 m_generator;
  static std::uniform_real_distribution< float > m_distribution;
};

std::mt19937 Random::m_generator( 0 );
std::uniform_real_distribution< float > Random::m_distribution( 0.0f, 1.0f );
```

And now let's go ahead and test this on our pi example with the following code:

```
float Pi_Uniform( int samplesX, int samplesY ) {
  int counter = 0;
  for ( int y = 0; y < samplesY; y++ ) {
    for ( int x = 0; x < samplesX; x++ ) {
      float xx = float( x ) / float( samplesX );
      float yy = float( y ) / float( samplesY );
      float x2 = xx * xx;
      float y2 = yy * yy;
      if ( x2 + y2 < 1.0f ) {
        counter++;
      }
    }
  }
  float pi = 4.0f * float( counter ) / float( samplesX * samplesY );
  return pi;
}

float Pi_MonteCarlo( int numSamples ) {
  int counter = 0;
  for ( int i = 0; i < numSamples; i++ ) {
    float xx = Random::Get();
    float yy = Random::Get();
    float x2 = xx * xx;
    float y2 = yy * yy;
    if ( x2 + y2 < 1.0f ) {
      counter++;
    }
  }
  float pi = 4.0f * float( counter ) / float( numSamples );
  return pi;
}

void TestPI() {
  int samples[ 4 ] = { 10, 100, 1000, 10000 };
  for ( int i = 0; i < 4; i++ ) {
    float pi_uniform = Pi_Uniform( samples[ i ], samples[ i ] );
    float pi_monte = Pi_MonteCarlo( samples[ i ] * samples[ i ] );
    int totalSamples = samples[ i ] * samples[ i ];
    printf( "uniform: \%f     monte carlo: \%f     Total Samples: \%i\n", pi_uniform, pi_monte,
    totalSamples );
  }
}
```

Which produces the following output:

```
uniform: 3.440000     monte carlo: 3.000000     Total Samples: 100
uniform: 3.179600     monte carlo: 3.155200     Total Samples: 10000
```

```
uniform: 3.145528      monte carlo: 3.140840      Total Samples: 1000000
uniform: 3.141990      monte carlo: 3.141635      Total Samples: 100000000
```

And as you can see, the monte carlo method approaches the true value of π "quicker" than the uniform sampling.

If you replace the old uniform sampling for the inertia tensor with this monte carlo method, then you don't need as many samples to get an accurate value. This can help reduce the amount of time spent calculating the inertia tensor.

To see an example of how to apply this technique, here's the code for calculating the inertia tensor of a convex hull using monte carlo integration:

```
static Vec3 CalculateCenterOfMassMonteCarlo( const std::vector< Vec3 > & pts, const std::vector<
    tri_t > & tris ) {
  const int numSamples = 10000;

  Bounds bounds;
  bounds.Expand( pts.data(), (int)pts.size() );

  Vec3 cm( 0.0f );
  int sampleCount = 0;
  for ( int i = 0; i < numSamples; i++ ) {
    Vec3 pt;
    pt.x = bounds.mins.x + Random::Get() * bounds.WidthX();
    pt.y = bounds.mins.y + Random::Get() * bounds.WidthY();
    pt.z = bounds.mins.z + Random::Get() * bounds.WidthZ();

    if ( IsExternal( pts, tris, pt ) ) {
      continue;
    }

    cm += pt;
    sampleCount++;
  }

  cm /= (float)sampleCount;
  return cm;
}

Mat3 CalculateInertiaTensorMonteCarlo( const std::vector< Vec3 > & pts, const std::vector< tri_t >
    & tris, const Vec3 & cm ) {
  const int numSamples = 10000;

  Bounds bounds;
  bounds.Expand( pts.data(), (int)pts.size() );

  Mat3 tensor;
  tensor.Zero();

  int sampleCount = 0;
  for ( int i = 0; i < numSamples; i++ ) {
    Vec3 pt;
    pt.x = bounds.mins.x + Random::Get() * bounds.WidthX();
    pt.y = bounds.mins.y + Random::Get() * bounds.WidthY();
    pt.z = bounds.mins.z + Random::Get() * bounds.WidthZ();

    if ( IsExternal( pts, tris, pt ) ) {
      continue;
    }

    // Get the point relative to the center of mass
    pt -= cm;

    tensor.rows[ 0 ][ 0 ] += pt.y * pt.y + pt.z * pt.z;
    tensor.rows[ 1 ][ 1 ] += pt.z * pt.z + pt.x * pt.x;
    tensor.rows[ 2 ][ 2 ] += pt.x * pt.x + pt.y * pt.y;

    tensor.rows[ 0 ][ 1 ] += -1.0f * pt.x * pt.y;
    tensor.rows[ 0 ][ 2 ] += -1.0f * pt.x * pt.z;
    tensor.rows[ 1 ][ 2 ] += -1.0f * pt.y * pt.z;
```

```
    tensor.rows[ 1 ][ 0 ] += -1.0f * pt.x * pt.y;
    tensor.rows[ 2 ][ 0 ] += -1.0f * pt.x * pt.z;
    tensor.rows[ 2 ][ 1 ] += -1.0f * pt.y * pt.z;

    sampleCount++;
  }

  tensor *= 1.0f / (float)sampleCount;
  return tensor;
}
```

15 Bonus: Exact Inertia Tensors for Convex Hulls

Okay, so we have two integration methods now for approximating the inertia tensors of shapes. However, it might be nice to not need such brute force integration methods.

Well, as it turns out, a convex hull can be pretty easily decomposed into a set of tetrahedrons. Then if we calculate the inertia tensor for the tetrahedrons, we should be able to add them all up. This should not only be more accurate than our integration methods, but faster as well.

To decompose the convex hull into a set of tetrahedrons, we simply average all the vertices of the convex hull together. This is guaranteed to give us a point inside the hull. Then we can use our set of surface triangles, in conjunction with this new point, to create a set of tetrahedrons. And all those tetrahedrons, together, will be the fully decomposed convex hull.

The next step, after decomposition, is to calculate the center of mass. This is pretty easy. The center of mass of a tetrahedron, with uniform density, is the average of its four corner points. Then we can use a weighted sum of the center of masses, multiplied by the volumes of each tetrahedron, to compute the center of mass of the convex hull.

So what is the volume of a tetrahedron? It's described by a rather simple equation, $\frac{1}{3}Ah$, where A is the area of one of its sides, and h is the projected height of the fourth point from that triangle. This produces a very simple function for calcuating the volume:

```
float TetrahedronVolume( const Vec3 & a, const Vec3 & b, const Vec3 & c, const Vec3 & d ) {
  const Vec3 ad = d - a;
  const Vec3 bd = d - b;
  const Vec3 cd = d - c;
  float numerator = ad.Dot( bd.Cross( cd ) );
  float volume = numerator / 6.0f;

  return fabsf( volume );
}
```

Now, we can calculate the center of mass of a convex hull with this code:

```
Vec3 CalculateCenterOfMassTetrahedron( const std::vector< Vec3 > & pts, const std::vector< tri_t >
    & tris ) {
  std::vector< Vec3 > cms;
  std::vector< float > volumes;
  float totalVolume = 0.0f;
  cms.reserve( tris.size() );
  volumes.reserve( tris.size() );

  Vec3 centerish( 0.0f );
  for ( int i = 0; i < pts.size(); i++ ) {
    centerish += pts[ i ];
  }
  centerish *= 1.0f / float( pts.size() );

  for ( int i = 0; i < tris.size(); i++ ) {
    const tri_t & tri = tris[ i ];

    const Vec3 & ptA = centerish;
    const Vec3 & ptB = pts[ tri.a ];
    const Vec3 & ptC = pts[ tri.b ];
    const Vec3 & ptD = pts[ tri.c ];

    const Vec3 centerOfMassOfThisSimplex = ( ptA + ptB + ptC + ptD ) * 0.25f;
    const float volume = TetrahedronVolume( ptA, ptB, ptC, ptD );
    cms.push_back( centerOfMassOfThisSimplex );
    volumes.push_back( volume );
    totalVolume += volume;
  }

  Vec3 cm( 0.0f );
  for ( int i = 0; i < cms.size(); i++ ) {
    cm += cms[ i ] * volumes[ i ];
  }
  cm *= 1.0f / totalVolume;

  return cm;
```

```
}
```

And now, how do we calculate the inertia tensor? Well, as it turns out the inertia tensor of the regular tetrahedron is well known. And it can be transformed using a matrix that is defined by the four vertices of the tetrahedron. And this produces a maybe not so simple function for calculating the inertia tensor of any tetrahedron (btw, I'm hand waving my explanation here, and I'll defer to the paper in the further reading section for the full description):

```
Mat3 InertiaTensorTetrahedron( const Vec3 & ptA, const Vec3 & ptB, const Vec3 & ptC, const Vec3 &
    ptD ) {
  const Vec3 pts[ 4 ] = { ptA, ptB, ptC, ptD };

  Mat3 mat;
  mat.rows[0] = Vec3( pts[ 1 ].x - pts[ 0 ].x, pts[ 2 ].x - pts[ 0 ].x, pts[ 3 ].x - pts[ 0 ].x );
  mat.rows[1] = Vec3( pts[ 1 ].y - pts[ 0 ].y, pts[ 2 ].y - pts[ 0 ].y, pts[ 3 ].y - pts[ 0 ].y );
  mat.rows[2] = Vec3( pts[ 1 ].z - pts[ 0 ].z, pts[ 2 ].z - pts[ 0 ].z, pts[ 3 ].z - pts[ 0 ].z );
  const float DetJ = fabsf( mat.Determinant() );

  const float density = 1.0f;
  const float mu = density;

  float xx = 0;
  float yy = 0;
  float zz = 0;

  float xy = 0;
  float xz = 0;
  float yz = 0;

  for ( int i = 0; i < 4; i++ ) {
    for ( int j = i; j < 4; j++ ) {
      // diagonals
      xx += pts[ i ].x * pts[ j ].x;
      yy += pts[ i ].y * pts[ j ].y;
      zz += pts[ i ].z * pts[ j ].z;

      // off-diagonals
      xy += pts[ i ].x * pts[ j ].y + pts[ j ].x * pts[ i ].y;
      xz += pts[ i ].x * pts[ j ].z + pts[ j ].x * pts[ i ].z;
      yz += pts[ i ].y * pts[ j ].z + pts[ j ].y * pts[ i ].z;
    }
  }

  const float a = mu * DetJ * ( yy + zz ) / 60.0f;
  const float b = mu * DetJ * ( xx + zz ) / 60.0f;
  const float c = mu * DetJ * ( xx + yy ) / 60.0f;

  const float aprime = mu * DetJ * yz / 120.0f;
  const float bprime = mu * DetJ * xz / 120.0f;
  const float cprime = mu * DetJ * xy / 120.0f;

  Mat3 InteriaTensor;
  InteriaTensor.rows[ 0 ] = Vec3(   a,    -cprime,  -bprime );
  InteriaTensor.rows[ 1 ] = Vec3( -cprime,    b,    -aprime );
  InteriaTensor.rows[ 2 ] = Vec3( -bprime,  -aprime,    c );
  return InteriaTensor;
}
```

Now we have everything we need to calculate the exact inertia tensor for the convex hull. We simply get the tensor for each tetrahedron, and then add them up!

```
Mat3 CalculateInertiaTensorTetrahedron( const std::vector< Vec3 > & pts, const std::vector< tri_t
    > & tris, const Vec3 & cm ) {
  Mat3 inertiaTensor;
  inertiaTensor.Zero();
  float totalVolume = 0.0f;
  for ( int i = 0; i < tris.size(); i++ ) {
    const tri_t & tri = tris[ i ];

    const Vec3 ptA = cm - cm;
```

```
    const Vec3 ptB = pts[ tri.a ] - cm;
    const Vec3 ptC = pts[ tri.b ] - cm;
    const Vec3 ptD = pts[ tri.c ] - cm;

    Mat3 tensor = InertiaTensorTetrahedron( ptA, ptB, ptC, ptD );
    inertiaTensor += tensor;

    const float volume = TetrahedronVolume( ptA, ptB, ptC, ptD );
    totalVolume += volume;
  }

  inertiaTensor *= 1.0f / totalVolume;
  return inertiaTensor;
}
```

It's worth noting, that we've changed our definitions of the tetrahedrons. We're no longer using the original decomposition, but instead we've changed it to where each tetrahedron uses the center of mass as a defining point. We also shift each tetrahedron by the center of mass, so that we're calculating the inertia tensor in the center of mass frame of the shape.

16 Bonus: Analytical Methods for finding Inertia Tensors

It's great that we have these computational methods for finding inertia tensors. But, it's also important to discuss analytical methods for solving the inertia tensor. Earlier in this series, we just pulled the exact inertia tensor for the sphere and box shapes out of a hat.

Well, as it turns out we can use the definition of the inertia tensor and the mathematical tools from calculus to pre-determine the inertia tensor for certain shapes. Recall that the definition of the inertia tensor for a collection of particles is:

$$\mathbf{I_{ij}} \equiv \sum_k m_k \cdot (\vec{r_k} \cdot \vec{r_k} \cdot \delta_{ij} - x_i \cdot x_j) \tag{9}$$

And to extend this definition to a continuous distribution of mass:

$$\mathbf{I_{ij}} \equiv \int \rho(\vec{r}) \cdot (\vec{r} \cdot \vec{r} \cdot \delta_{ij} - x_i \cdot x_j) dV \tag{10}$$

Where $\rho(\vec{r})$ is a function that describes the density of the object.

To demonstrate how to use this equation, let's look at the derivation of some familiar inertia tensors.

16.1 Derivation of the Inertia Tensor for a Box

Given a box centered around the origin with dimensions (w, d, h), total mass M, and uniform density. Then we will have the following set of equations:

$$\mathbf{I_{00}} = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} x^2+y^2+z^2-x^2 dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} y^2+z^2 dxdydz$$
$$\mathbf{I_{11}} = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} x^2+y^2+z^2-y^2 dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} x^2+z^2 dxdydz$$
$$\mathbf{I_{22}} = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} x^2+y^2+z^2-z^2 dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} x^2+y^2 dxdydz$$

$$\mathbf{I_{01}} = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} -x*y dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} -y*x dxdydz = \mathbf{I_{10}}$$
$$\mathbf{I_{02}} = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} -x*z dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} -z*x dxdydz = \mathbf{I_{20}}$$
$$\mathbf{I_{12}} = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} -y*z dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} -z*y dxdydz = \mathbf{I_{21}}$$

Let's go ahead and solve $\mathbf{I_{00}}$:

$$
\begin{aligned}
\mathbf{I_{00}} &= \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} y^2 + z^2 dxdydz \\
&= \frac{M}{V} * w \int_{-h/2}^{h/2}\int_{-d/2}^{d/2} y^2 + z^2 dydz \\
&= \frac{M}{V} * w \int_{-h/2}^{h/2} [\frac{1}{3}y^3 + yz^2]_{-d/2}^{d/2} dz \\
&= \frac{M}{V} * w \int_{-h/2}^{h/2} \frac{d^3}{12} + d * z^2 dz \\
&= \frac{M}{V} * w (\frac{d^3}{12} * h + d * \frac{h^3}{12}) \\
&= \frac{M}{w * h * d} * w (\frac{d^3}{12} * h + d * \frac{h^3}{12}) \\
&= \frac{M}{12}(d^2 + h^2)
\end{aligned}
$$

Similarly, we can solve for $\mathbf{I_{11}}$ and $\mathbf{I_{22}}$ to get:

$$
\begin{aligned}
\mathbf{I_{11}} &= \frac{M}{12}(w^2 + h^2) \\
\mathbf{I_{22}} &= \frac{M}{12}(w^2 + d^2)
\end{aligned}
$$

Now, to solve for the diagonal terms:

$$
\begin{aligned}
\mathbf{I_{01}} &= \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2}\int_{-w/2}^{w/2} -x * y dxdydz \\
&= \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2} [-\frac{x^2}{2} * y]_{-w/2}^{w/2} dydz \\
&= \frac{M}{V}\int_{-h/2}^{h/2}\int_{-d/2}^{d/2} (-\frac{w^2}{8} + \frac{w^2}{8}) * y dydz \\
&= 0
\end{aligned}
$$

This can be shown to be true for all the non-diagonal terms. And so the total inertia tensor is:

$$
\begin{pmatrix}
\frac{1}{12}M(h^2 + d^2) & 0 & 0 \\
0 & \frac{1}{12}M(w^2 + d^2) & 0 \\
0 & 0 & \frac{1}{12}M(w^2 + h^2)
\end{pmatrix} \tag{11}
$$

16.2 Derivation of the Inertia Tensor for a Cylinder

Although we haven't implemented a cylinder shape in this book, it is a natural example to examine the use of cylindrical coordinates for calculating an inertia tensor.

The equations relating cylindrical coordinates to cartesian coordinates are:

$$
\begin{aligned}
x &= r\cos(\theta) \\
y &= r\sin(\theta) \\
z &= z
\end{aligned}
$$

So, given a cylinder of radius R and height h, then the equations for the elements of the inertia tensor are:

$$\mathbf{I_{00}} = \frac{M}{V}\int\int\int y^2 + z^2 dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-\pi}^{\pi}\int_0^R (r^2\sin^2(\theta) + z^2) r dr d\theta dz$$

$$\mathbf{I_{11}} = \frac{M}{V}\int\int\int x^2 + z^2 dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-\pi}^{\pi}\int_0^R (r^2\cos^2(\theta) + z^2) r dr d\theta dz$$

$$\mathbf{I_{22}} = \frac{M}{V}\int\int\int x^2 + y^2 dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-\pi}^{\pi}\int_0^R (r^2\cos^2(\theta) + r^2\sin^2(\theta)) r dr d\theta dz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-\pi}^{\pi}\int_0^R r^3 dr d\theta dz$$

$$\mathbf{I_{01}} = \frac{M}{V}\int\int\int -x * y dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-\pi}^{\pi}\int_0^R -r^3\cos(\theta)\sin(\theta) dr d\theta dz = \mathbf{I_{10}}$$

$$\mathbf{I_{02}} = \frac{M}{V}\int\int\int -x * z dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-\pi}^{\pi}\int_0^R -z * r^2\cos(\theta) dr d\theta dz = \mathbf{I_{20}}$$

$$\mathbf{I_{12}} = \frac{M}{V}\int\int\int -y * z dxdydz = \frac{M}{V}\int_{-h/2}^{h/2}\int_{-\pi}^{\pi}\int_0^R -z * r^2\sin(\theta) dr d\theta dz = \mathbf{I_{21}}$$

Evaluating these integrals, and knowing that the volume of a cylinder is given by $h\pi R^2$. Then the inertia tensor is:

$$\begin{pmatrix} \frac{1}{12}M(3R^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12}M(3R^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{2}MR^2 \end{pmatrix} \tag{12}$$

16.3 Derivation of the Inertia Tensor for a Sphere

Now that we've seen an example in cylindrical coordinates, let's have a look at an example in spherical coordinates.

The equations relating spherical coordinates to cartesian coordinates are:

$$x = r\cos(\theta)\sin(\phi)$$
$$y = r\sin(\theta)\sin(\phi)$$
$$z = r\cos(\phi)$$

This gives the following integrals for the elements of the inertia tensor:

$$\mathbf{I_{00}} = \frac{M}{V}\int\int\int y^2 + z^2 dxdydz = \frac{M}{V}\int_0^{\pi}\int_{-\pi}^{\pi}\int_0^R (r^2\sin^2(\theta)\sin^2(\phi) + r^2\cos^2(\phi)) r^2\sin(\phi) dr d\theta d\phi$$

$$\mathbf{I_{11}} = \frac{M}{V}\int\int\int x^2 + z^2 dxdydz = \frac{M}{V}\int_0^{\pi}\int_{-\pi}^{\pi}\int_0^R (r^2\cos^2(\theta)\sin^2(\phi) + r^2\cos^2(\phi)) r^2\sin(\phi) dr d\theta d\phi$$

$$\begin{aligned}\mathbf{I_{22}} = \frac{M}{V}\int\int\int x^2 + y^2 dxdydz &= \frac{M}{V}\int_0^{\pi}\int_{-\pi}^{\pi}\int_0^R (r^2\cos^2(\theta)\sin^2(\phi) + r^2\sin^2(\theta)\sin^2(\phi)) r^2\sin(\phi) dr d\theta d\phi \\ &= \frac{M}{V}\int_0^{\pi}\int_{-\pi}^{\pi}\int_0^R r^4\sin^3(\phi) dr d\theta d\phi\end{aligned}$$

$$\mathbf{I_{01}} = \frac{M}{V}\int\int\int -x * y dxdydz = \frac{M}{V}\int_0^{\pi}\int_{-\pi}^{\pi}\int_0^R -r^4\cos(\theta)\sin(\theta)\sin^3(\phi) dr d\theta d\phi = \mathbf{I_{10}}$$

$$\mathbf{I_{02}} = \frac{M}{V}\int\int\int -x * z dxdydz = \frac{M}{V}\int_0^{\pi}\int_{-\pi}^{\pi}\int_0^R -r^4\cos(\theta)\cos(\phi)\sin^2(\phi) dr d\theta d\phi = \mathbf{I_{20}}$$

$$\mathbf{I_{12}} = \frac{M}{V}\int\int\int -y * z dxdydz = \frac{M}{V}\int_0^{\pi}\int_{-\pi}^{\pi}\int_0^R -r^4\sin(\theta)\cos(\phi)\sin^2(\phi) dr d\theta d\phi = \mathbf{I_{21}}$$

Again, knowing that the volume of a sphere is $\frac{4}{3}\pi R^3$, then the inertia tensor evaluates to:

$$\begin{pmatrix} \frac{2}{5}MR^2 & 0 & 0 \\ 0 & \frac{2}{5}MR^2 & 0 \\ 0 & 0 & \frac{2}{5}MR^2 \end{pmatrix} \tag{13}$$

16.4 Derivation of the Inertia Tensor for a Half Sphere

While we could stop our discussion there, it's worth another example. So how about the half sphere?

If the bottom of the half sphere is centered on the origin then the integrals become:

$$\mathbf{I_{00}} = \frac{M}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R (r^2\sin^2(\theta)\sin^2(\phi) + r^2\cos^2(\phi))r^2\sin(\phi)drd\theta d\phi$$
$$\mathbf{I_{11}} = \frac{M}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R (r^2\cos^2(\theta)\sin^2(\phi) + r^2\cos^2(\phi))r^2\sin(\phi)drd\theta d\phi$$
$$\mathbf{I_{22}} = \frac{M}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R r^4\sin^3(\phi)drd\theta d\phi$$

$$\mathbf{I_{01}} = \frac{M}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R -r^4\cos(\theta)\sin(\theta)\sin^3(\phi)drd\theta d\phi = \mathbf{I_{10}}$$
$$\mathbf{I_{02}} = \frac{M}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R -r^4\cos(\theta)\cos(\phi)\sin^2(\phi)drd\theta d\phi = \mathbf{I_{20}}$$
$$\mathbf{I_{12}} = \frac{M}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R -r^4\sin(\theta)\cos(\phi)\sin^2(\phi)drd\theta d\phi = \mathbf{I_{21}}$$

As you can see, they've hardly changed. The only difference is the limits of integration for the variable ϕ. We only need to integrate about the the top half of the sphere instead of the whole sphere.

It's also important to note that the volume is also halved, $\frac{2}{3}\pi R^3$. And this gives us the following inertia tensor:

$$\begin{pmatrix} \frac{2}{5}MR^2 & 0 & 0 \\ 0 & \frac{2}{5}MR^2 & 0 \\ 0 & 0 & \frac{2}{5}MR^2 \end{pmatrix} \tag{14}$$

Now, this inertia tensor is not centered about the center of mass of the half sphere. But we would like to know what the inertia tensor about the center of mass would be. To calculate this, we can use the parallel axis theorem, but before we can apply it, we need to find the center of mass. Recall that the center of mass is defined as:

$$x_{cm} = \frac{\sum_i m_i * x_i}{\sum_i m_i} \tag{15}$$

or for a continuous distribution of mass:

$$\vec{x}_{cm} = \frac{\int \rho(\vec{r}) \cdot \vec{r} dV}{\int \rho(\vec{r}) dV} \tag{16}$$

And since our mass density is uniform then we get:

$$\vec{x}_{cm} = \frac{\int \frac{M}{V} \cdot \vec{r} dV}{\int \frac{M}{V} dV} \tag{17}$$

The densities then cancel and the integral in the denominator evaluates to the total volume of the half sphere:

$$\vec{x}_{cm} = \frac{1}{V}\int \vec{r} dV \tag{18}$$

This means that we'll get the following equations:

$$\begin{aligned}
x_{cm} &= \frac{1}{V}\int\int\int x dx dy dz = \frac{1}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R r^3 \cos(\theta)\sin^2(\phi) dr d\theta d\phi \\
y_{cm} &= \frac{1}{V}\int\int\int y dx dy dz = \frac{1}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R r^3 \sin(\theta)\sin^2(\phi) dr d\theta d\phi \\
z_{cm} &= \frac{1}{V}\int\int\int z dx dy dz = \frac{1}{V}\int_0^{\pi/2}\int_{-\pi}^{\pi}\int_0^R r^3 \cos(\phi)\sin(\phi) dr d\theta d\phi
\end{aligned}$$

It's straightforward to see that x_{cm} and y_{cm} are both zero. But for z_{cm} we get:

$$\begin{aligned}
z_{cm} &= \frac{1}{V}\frac{1}{4}R^4 * 2 * \pi \int_0^{\pi/2} \cos(\phi)\sin(\phi) dr d\theta d\phi \\
&= \frac{1}{V}\frac{1}{4}R^4 * 2 * \pi * \frac{1}{2} \\
&= \frac{1}{V}\frac{1}{4}R^4\pi
\end{aligned}$$

And recall that the volume of the half sphere is just $\frac{2}{3}\pi R^3$ which gives us the following for the center of mass:

$$\begin{aligned}
x_{cm} &= 0 \\
y_{cm} &= 0 \\
z_{cm} &= \frac{3}{8}R
\end{aligned}$$

Now we can take advantage of the parallel axis theorem to recenter the inertia tensor about the center of mass. The parallel axis theorem, in tensor form, can be written as:

$$\mathbf{J_{ij}} \equiv \mathbf{I_{ij}} + M(\vec{r}\cdot\vec{r}\cdot\delta_{ij} - x_i \cdot x_j) \tag{19}$$

where $\mathbf{I_{ij}}$ is the inertia tensor in the center of mass frame, $\mathbf{J_{ij}}$ is the inertia tensor relative to the non-center of mass point, $\vec{r}$ is the vector from the center of mass to the new point, and x_i are the elements of $\vec{r}$. Solving for the center of mass tensor yields:

$$\mathbf{I_{ij}} = \mathbf{J_{ij}} - M(\vec{r}\cdot\vec{r}\cdot\delta_{ij} - x_i \cdot x_j) \tag{20}$$

Since $\vec{r} = (0, 0, -\frac{3}{8}R)$ then, all the non-diagonal terms do not change, and the diagonal terms become:

$$\begin{aligned}
\mathbf{I_{00}} &= \mathbf{J_{00}} - M(\frac{3}{8}R)^2 \\
\mathbf{I_{11}} &= \mathbf{J_{11}} - M(\frac{3}{8}R)^2 \\
\mathbf{I_{22}} &= \mathbf{J_{22}}
\end{aligned}$$

Which means the center of mass inertia tensor is:

$$MR^2 \begin{pmatrix} \frac{2}{5} - (\frac{3}{8})^2 & 0 & 0 \\ 0 & \frac{2}{5} - (\frac{3}{8})^2 & 0 \\ 0 & 0 & \frac{2}{5} \end{pmatrix} \tag{21}$$

16.5 Derivation of the Inertia Tensor for a Capsule

So you may have wondered why we bothered with the calculation of the inertia tensor for the cylinder and half sphere. While it could be argued that it was a good exercise, the reality is that we needed it for calculating the inertia tensor of the capsule. Capsules tend to be a commonly used primitive in physics simulations, and so it's very much worth covering how to derive their inertia.

A capsule can be subdivided into a cylinder and two half sphere end caps. Therefore the inertia tensor can be described as:

$$\mathbf{I}_{capsule} = \mathbf{I}_{cylinder} + \mathbf{I}_{halfSphereTop} + \mathbf{I}_{halfSphereBottom} \tag{22}$$

Now, $\mathbf{I}_{cylinder}$, is just the familiar one that we've already derived in a previous section. However, we need to be a little careful with $\mathbf{I}_{halfSphereTop}$ and $\mathbf{I}_{halfSphereBottom}$, since they're not the exact same as the center of mass inertia tensor for the half sphere. Both $\mathbf{I}_{halfSphereTop}$ and $\mathbf{I}_{halfSphereBottom}$ are translated away from the origin, so we'll need to apply the parallel axis theorem, and $\mathbf{I}_{halfSphereBottom}$ is a mirrored version of $\mathbf{I}_{halfSphereTop}$.

Let's first examine $\mathbf{I}_{halfSphereTop}$. This is simply the same as the center of mass half sphere inertia tensor, with the parallel axis theorem applied. The vector that we need to apply will require a distance of half the cylinder height and the distance of the center of mass of the half sphere. This makes the displacement vector:

$$\vec{r} = \begin{pmatrix} 0 \\ 0 \\ \frac{h}{2} + \frac{3}{8}R \end{pmatrix} \tag{23}$$

Applying the parallel axis theorem with this vector gives:

$$\mathbf{J_{00}} = \mathbf{I_{00}} + M(\frac{h}{2} + \frac{3}{8}R)^2 = \mathbf{I_{00}} + M((\frac{h}{2})^2 + (\frac{3}{8}R)^2 + \frac{3}{8}hR)$$
$$\mathbf{J_{11}} = \mathbf{I_{11}} + M(\frac{h}{2} + \frac{3}{8}R)^2 = \mathbf{I_{11}} + M((\frac{h}{2})^2 + (\frac{3}{8}R)^2 + \frac{3}{8}hR)$$
$$\mathbf{J_{22}} = \mathbf{I_{22}}$$

Which makes the top half sphere's inertia tensor:

$$\mathbf{I}_{halfSphereTop} = M_{halfSphereTop} \begin{pmatrix} \frac{2}{5}R^2 + \frac{h^2}{4} + \frac{3}{8}hR & 0 & 0 \\ 0 & \frac{2}{5}R^2 + \frac{h^2}{4} + \frac{3}{8}hR & 0 \\ 0 & 0 & \frac{2}{5}R^2 \end{pmatrix} \tag{24}$$

Now, we could go through this same procedure for calculating the bottom half sphere's inertia tensor. Or we could use the argument of symmetry to state that it's equal to the top half sphere's inertia tensor. And since it requires less typing on my part to use the argument of symmetry, it is what we'll use here. This gives the final inertia tensor as:

$$\mathbf{I}_{capsule} = \mathbf{I}_{cylinder} + 2 * \mathbf{I}_{halfSphere} \tag{25}$$

Or fully written out as:

$$\begin{pmatrix} M_{cy}(\frac{R^2}{4} + \frac{h^2}{12}) + M_{hs}(\frac{4}{5}R^2 + \frac{h^2}{2} + \frac{3}{4}hR) & 0 & 0 \\ 0 & M_{cy}(\frac{R^2}{4} + \frac{h^2}{12}) + M_{hs}(\frac{4}{5}R^2 + \frac{h^2}{2} + \frac{3}{4}hR) & 0 \\ 0 & 0 & (\frac{1}{2}M_{cy} + \frac{4}{5}M_{hs})R^2 \end{pmatrix}$$

And it's important to remember that M_{cy} is just the mass of the cylinder portion, and M_{hs} is just the mass of one of the half spheres. So the total mass of the capsule is:

$$M_{capsule} = M_{cy} + 2M_{hs} \tag{26}$$

17 Bonus: Derivation of the definition of the Inertia Tensor

In this chapter we're going to derive the defintion of the inertia tensor from first principles. Recall that the inertia tensor is:

$$\mathbf{I_{ij}} \equiv \sum_k m_k \cdot (\vec{r_k} \cdot \vec{r_k} \cdot \delta_{ij} - x_i \cdot x_j) \tag{27}$$

Now, it's great that we've learned how to use this equation. But sadly we just pulled it out of thin air. Which means we don't really know if it's true or not.

So, let's assume we have a single particle that is moving. The angular momentum for that particle is defined as:

$$\begin{aligned}\vec{L} &= \vec{r} \times \vec{p} \\ &= \vec{r} \times m\frac{d\vec{r}}{dt} \\ &= m(\vec{r} \times \frac{d\vec{r}}{dt})\end{aligned}$$

Now let's assume we have a collection of particles that are rigidly connected to each other. And for our convenience, their center of mass coincides with the origin and they're rotating about it with angular velocity $\vec{\omega}$.

$$\begin{aligned}\vec{L} &= \sum_k m_k \vec{r}_k \times \frac{d\vec{r}_k}{dt} \\ &= \sum_k m_k \vec{r}_k \times (\vec{\omega} \times \vec{r}_k)\end{aligned}$$

And if we use the following identity for the vector triple product:

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

Then this gives us:

$$\vec{L} = \sum_k m_k[(\vec{r}_k \cdot \vec{r}_k)\vec{\omega} - (\vec{r}_k \cdot \vec{\omega})\vec{r}_k]$$

Now, let's ignore the sum for a moment and only focus on the angular momentum of a specific particle

$$\vec{L}_k = m_k[(\vec{r}_k \cdot \vec{r}_k)\vec{\omega} - (\vec{r}_k \cdot \vec{\omega})\vec{r}_k]$$

With the malice of foresight, we know that we want to obtain a matrix such that when we multiply it by $\vec{\omega}$ we get the above vector result. We can also see that the above equation has two distinct parts $(\vec{r}_k \cdot \vec{r}_k)\vec{\omega}$ and $(\vec{r}_k \cdot \vec{\omega})\vec{r}_k$. So, to make our lives easier, let's tackle each part by itself. Let's begin with the first one. And since we're working on a single particle at a time, I'm going to drop the k subscript for now:

$$\begin{aligned}(\vec{r} \cdot \vec{r})\vec{\omega} &= \begin{pmatrix}(r_x r_x + r_y r_y + r_z r_z)\omega_x \\ (r_x r_x + r_y r_y + r_z r_z)\omega_y \\ (r_x r_x + r_y r_y + r_z r_z)\omega_z\end{pmatrix} \\ &= \begin{pmatrix}r_x r_x + r_y r_y + r_z r_z & 0 & 0 \\ 0 & r_x r_x + r_y r_y + r_z r_z & 0 \\ 0 & 0 & r_x r_x + r_y r_y + r_z r_z\end{pmatrix}\begin{pmatrix}\omega_x \\ \omega_y \\ \omega_z\end{pmatrix} \\ &= \begin{pmatrix}r_x r_x + r_y r_y + r_z r_z & 0 & 0 \\ 0 & r_x r_x + r_y r_y + r_z r_z & 0 \\ 0 & 0 & r_x r_x + r_y r_y + r_z r_z\end{pmatrix}\vec{\omega}\end{aligned}$$

Now, let's have a look at the second part $(\vec{r} \cdot \vec{\omega})\vec{r}$:

$$
\begin{aligned}
(\vec{r} \cdot \vec{\omega})\vec{r} &= \begin{pmatrix} (r_x\omega_x + r_y\omega_y + r_z\omega_z)r_x \\ (r_x\omega_x + r_y\omega_y + r_z\omega_z)r_y \\ (r_x\omega_x + r_y\omega_y + r_z\omega_z)r_z \end{pmatrix} \\
&= \begin{pmatrix} r_xr_x\omega_x + r_yr_x\omega_y + r_zr_x\omega_z \\ r_xr_y\omega_x + r_yr_y\omega_y + r_zr_y\omega_z \\ r_xr_z\omega_x + r_yr_z\omega_y + r_zr_z\omega_z \end{pmatrix} \\
&= \begin{pmatrix} r_xr_x & r_yr_x & r_zr_x \\ r_xr_y & r_yr_y & r_zr_y \\ r_xr_z & r_yr_z & r_zr_z \end{pmatrix} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \\
&= \begin{pmatrix} r_xr_x & r_yr_x & r_zr_x \\ r_xr_y & r_yr_y & r_zr_y \\ r_xr_z & r_yr_z & r_zr_z \end{pmatrix} \vec{\omega}
\end{aligned}
$$

And it should be pretty easy to see what's going to happen when we subtract these two:

$$
(\vec{r} \cdot \vec{r})\vec{\omega} - (\vec{r} \cdot \vec{\omega})\vec{r} = \begin{pmatrix} r_yr_y + r_zr_z & -r_yr_x & -r_zr_x \\ -r_xr_y & r_xr_x + r_zr_z & -r_zr_y \\ -r_xr_z & -r_yr_z & r_xr_x + r_yr_y \end{pmatrix} \vec{\omega} = (\vec{r} \cdot \vec{r} \cdot \delta_{ij} - x_i \cdot x_j)\vec{\omega}
$$

And simply by re-introducing the k subscript and summing over all particles gives us the inertia tensor:

$$
\mathbf{I_{ij}} \equiv \sum_k m_k \cdot (\vec{r_k} \cdot \vec{r_k} \cdot \delta_{ij} - x_i \cdot x_j)
$$

Finally, you know that we didn't just make this stuff up. And it's plane to see that:

$$
\vec{L} = \vec{r} \times \vec{p} = \mathbf{I} \cdot \vec{\omega}
$$

18 Conclusion

Alright! That's it for now. If you've made it this far, then congratulations! Hopefully you've learned a lot. And maybe you're willing to keep going and learn even more. The next book is going to be all about constraints. Constraints are finally going to give us ragdolls, motors, and stable box stacking.

19 Further Reading & References

"Classical Mechanics" - Goldstein, Poole, Safko

"Explicit Exact Formulas for the 3-D Tetrahedron Inertia Tensor in Terms of its Vertex Coordinates" - Tonon 2004

"A fast procedure for computing the distance between complex objects in three-dimensional space" - Gilbert, Johnson, Keerthi 1988

"Implementing GJK" - Casey Muratori 2006

"Collision Detection in Interactive 3D Environments" - Gino Van Den Bergen

"Improving the GJK algorithm for faster and more reliable distance queries between convex objects" - Montanari, Petrinic, Barbieri 2017

"Real-time Collision Detection" - Christer Ericson

"Continuous Collision Detection and Physics" - Erwin Coumans 2005