Contact Points Using Clipping

Posted on *November 17, 2011*

Many have asked “How do I get the contact points from GJK?” or similar on the SAT, GJK, and EPA posts. I’ve finally got around to creating a post on this topic. Contact point generation is a vital piece of many applications and is usually the next step after collision detection. Generating **good** contact points is crucial to predictable and life-like iteractions between bodies. In this post I plan to cover a clipping method that is used in Box2d and dyn4j. This is not the only method available and I plan to comment about the other methods near the end of the post.

Introduction

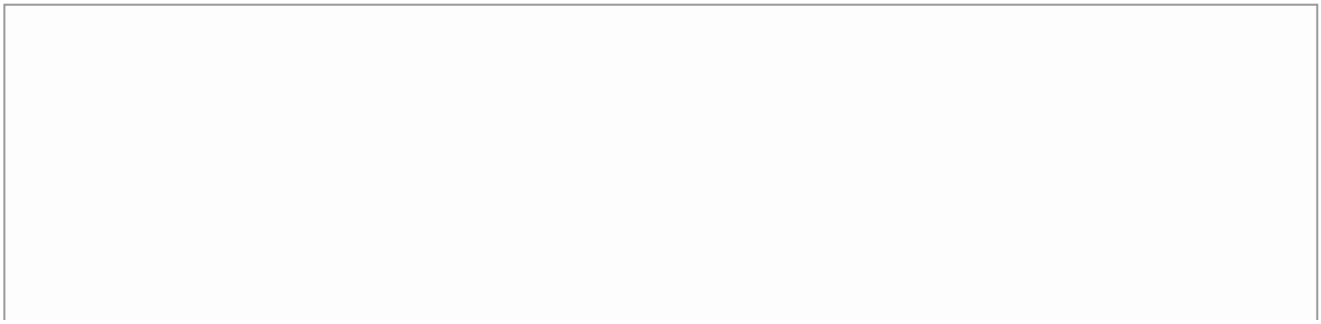
Most collision detection algorithms will return a separation normal and depth. Using this information we can translate the shapes directly to resolve the collision. Doing so does not exhibit real world physical behavior. As such, this isn’t sufficent for applications that want to model the physical world. To model real world iteractions effectively, we need to know **where** the collision occurred.

Contact points are usually world space points on the colliding shapes/bodies that represent where the collision is taking place. In the real world this would on the edge of two objects where they are touching. However, most simulations run collision detection routines on some interval allowing the objects overlap rather than touch. In this very common scenario, we must infer what the contact(s) should be.

> *More than one contact point is typically called a contact manifold or contact patch.*

Finding the Features

The first step is to identify the features of the shapes that are involved in the collision. We can find the collision feature of a shape by finding the farthest vertex in the shape. Then, we look at the adjacent two vertices to determine which edge is the “closest.” We determine the closest as the edge who is most perpendicular to the separation normal.



```
// step 1
// find the farthest vertex in
// the polygon along the separation normal
int c = vertices.length;
for (int i = 0; i < c; i++) {
  double projection = n.dot(v);
  if (projection > max) {
    max = projection;
    index = i;
  }
}

// step 2
// now we need to use the edge that
// is most perpendicular, either the
// right or the left
Vector2 v = vertices[index];
Vector2 v1 = v.next;
Vector2 v0 = v.prev;
// v1 to v
Vector2 l = v - v1;
// v0 to v
Vector2 r = v - v0;
// normalize
l.normalize();
r.normalize();
// the edge that is most perpendicular
// to n will have a dot product closer to zero
if (r.dot(n) <= l.dot(n)) {
  // the right edge is better
  // make sure to retain the winding direction
  return new Edge(v, v0, v);
} else {
  // the left edge is better
  // make sure to retain the winding direction
  return new Edge(v, v, v1);
}
// we return the maximum projection vertex (v)
// and the edge points making the best edge (v and either v0 or v1)
```

> *Be careful when computing the left and right (l and r in the code above) vectors as they both must point towards the maximum point. If one doesn't that edge may always be used since its pointing in the negative direction and the other is pointing in the positive direction.*

To obtain the correct feature we must know the direction of the separation normal ahead of time. Does it point from A to B or does it point from B to A? Its recommended that this is fixed, so for this post we will assume that the separation normal always points from A to B.

```
// find the "best" edge for shape A
Edge e1 = A.best(n);
// find the "best" edge for shape B
Edge e2 = B.best(-n);
```

Clipping

Now that we have the two edges involved in the collision, we can do a series of line/plane clips to get the contact manifold (all the contact points). To do so we need to identify the reference edge and incident edge. The reference edge is the edge most perpendicular to the separation normal. The reference edge will be used to clip the incident edge vertices to generate the contact manifold.

```
Edge ref, inc;
boolean flip = false;
if (abs(e1.dot(n)) <= abs(e2.dot(n))) {
  ref = e1;
  inc = e2;
} else {
  ref = e2;
  inc = e1;
  // we need to set a flag indicating that the reference
  // and incident edge were flipped so that when we do the final
  // clip operation, we use the right edge normal
  flip = true;
}
```

Now that we have identified the reference and incident edges we can begin clipping points. First we need to clip the incident edge's points by the first vertex in the reference edge. This is done by comparing the offset of the first vertex along the reference vector with the incident edge's offsets. Afterwards, the result of the previous clipping operation on the incident edge is clipped again using the second vertex of the reference edge. Finally, we check if the remaining points are past the reference edge along the reference edge's normal. In all, we perform three clipping operations.

```
// the edge vector
Vector2 refv = ref.edge;
refv.normalize();

double o1 = refv.dot(ref.v1);
// clip the incident edge by the first
// vertex of the reference edge
ClippedPoints cp = clip(inc.v1, inc.v2, refv, o1);
// if we dont have 2 points left then fail
if (cp.length < 2) return;

// clip whats left of the incident edge by the
// second vertex of the reference edge
// but we need to clip in the opposite direction
// so we flip the direction and offset
double o2 = refv.dot(ref.v2);
ClippedPoints cp = clip(cp[0], cp[1], -refv, -o2);
// if we dont have 2 points left then fail
if (cp.length < 2) return;

// get the reference edge normal
Vector2 refNorm = ref.cross(-1.0);
// if we had to flip the incident and reference edges
// then we need to flip the reference edge normal to
// clip properly
if (flip) refNorm.negate();
// get the largest depth
double max = refNorm.dot(ref.max);
// make sure the final points are not past this maximum
if (refNorm.dot(cp[0]) - max < 0.0) {
  cp.remove(cp[0]);
}
if (refNorm.dot(cp[1]) - max < 0.0) {
  cp.remove(cp[1]);
}
// return the valid points
return cp;
```

And the clip method:

```
// clips the line segment points v1, v2
// if they are past o along n
ClippedPoints clip(v1, v2, n, o) {
  ClippedPoints cp = new ClippedPoints();
  double d1 = n.dot(v1) - o;
  double d2 = n.dot(v2) - o;
  // if either point is past o along n
  // then we can keep the point
  if (d1 >= 0.0) cp.add(v1);
  if (d2 >= 0.0) cp.add(v2);
  // finally we need to check if they
  // are on opposing sides so that we can
  // compute the correct point
  if (d1 * d2 < 0.0) {
    // if they are on different sides of the
    // offset, d1 and d2 will be a (+) * (-)
    // and will yield a (-) and therefore be
    // less than zero
    // get the vector for the edge we are clipping
    Vector2 e = v2 - v1;
    // compute the location along e
    double u = d1 / (d1 - d2);
    e.multiply(u);
    e.add(v1);
    // add the point
    cp.add(e);
  }
}
```

> *Even though all the examples use box-box collisions, this method will work for any convex polytopes. See the end of the post for details on handling curved shapes.*

Example 1
Its best to start with a simple example explaining the process. Figure 1 shows a box vs. box collision with the collision information listed along with the winding direction of the vertices for both shapes. We have following data to begin with:

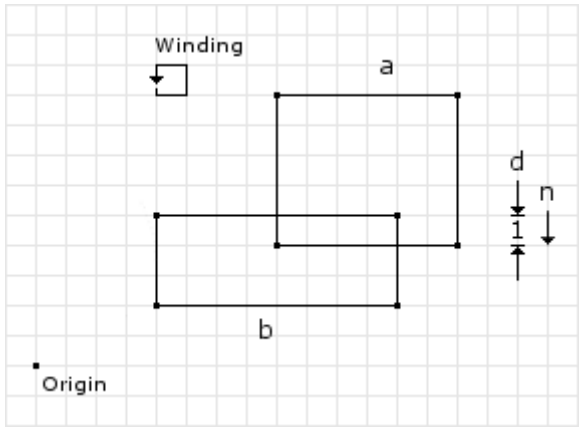


Figure 1: A simple box-box collision

```
// from the collision detector
// separation normal and depth
normal = (0, -1)
depth = 1
```

The first step is to get the “best” edges, or the edges that are involved in the collision:

```
//                    ---------+---------+-------
Edge e1 = A.best(n)  = ( 8, 4) | ( 8, 4) | (14, 4)
Edge e2 = B.best(-n) = (12, 5) | (12, 5) | ( 4, 5)
```

Figure 2 highlights the “best” edges on the shapes. Once we have found the edges, we need

to determine which edge is the reference edge and which is the incident edge:

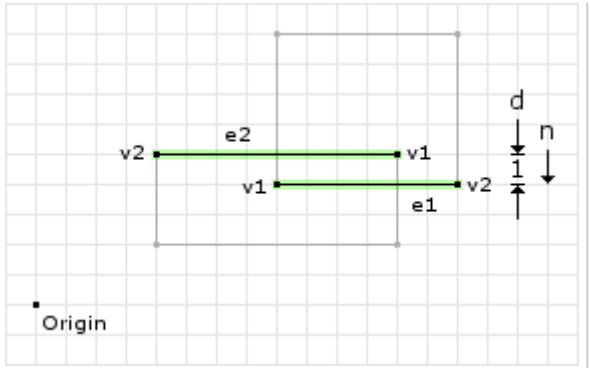


Figure 2: The "best" edges of figure 1

```
e1 = (8, 4) to (14, 4) = (14, 4) - (8, 4) = (6, 0)
e2 = (12, 5) to (4, 5) = (4, 5) - (12, 5) = (-8, 0)
e1Dotn = (6, 0) · (0, -1) = 0
e2Dotn = (-8, 0) · (0, -1) = 0
// since the dot product is the same we can choose either one
// using the first edge as the reference will let this example
// be slightly simpler
ref = e1;
inc = e2;
```

Now that we have identified the reference and incident edges we perform the first clipping operation:

```
ref.normalize() = (1, 0)
o1 = (1, 0) · (8, 4) = 8
// now we call clip with
// v1 = inc.v1 = (12, 5)
// v2 = inc.v2 = (4, 5)
// n  = ref    = (1, 0)
// o  = o1     = 8
d1 = (1, 0) · (12, 5) - 8 = 4
d2 = (1, 0) · (4, 5)  - 8 = -4
// we only add v1 to the clipped points since
// its the only one that is greater than or
// equal to zero
cp.add(v1);
// since d1 * d2 = -16 we go into the if block
e = (4, 5) - (12, 5) = (-8, 0)
u = 4 / (4 - -4) = 1/2
e * u + v1 = (-8 * 1/2, 0 * 1/2) + (12, 5) = (8, 5)
// then we add this point to the clipped points
cp.add(8, 5);
```

The first clipping operation removed one point that was outside the clipping plane (i.e. past the offset). But since there was another point on the opposite side of the clipping plane, we compute a new point on the edge and use it as the second point of the result. See figure 3 for an illustration.
Since we still have two points in the ClippedPoints object we can continue and perform the second clipping operation:

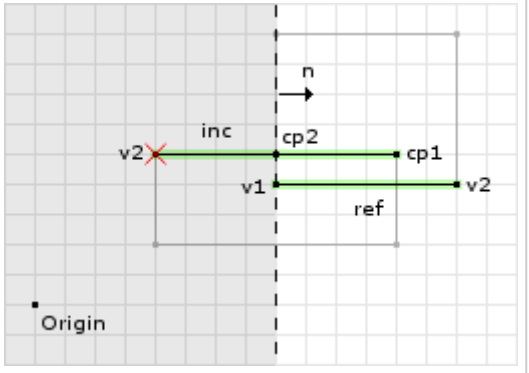


Figure 3: The first clip of example 1

```
o2 = (1, 0) · (14, 4) = 14
// now we call clip with
// v1 = cp[0] = (12, 5)
// v2 = cp[1] = (8, 5)
// n  = -ref  = (-1, 0)
// o  = -o1   = -14
d1 = (-1, 0) · (12, 5) - -14 = 2
d2 = (-1, 0) · (8, 5)  - -14 = 6
// since both are greater than or equal
// to zero we add both to the clipped
// points object
cp.add(v1);
cp.add(v2);
// since both are positive then we skip
// the if block and return
```

The second clipping operation did not remove any points. Figure 4 shows the clipping plane and the valid and invalid regions. Both points were found to be inside the valid region of the clipping plane. Now we continue to the last clipping operation:

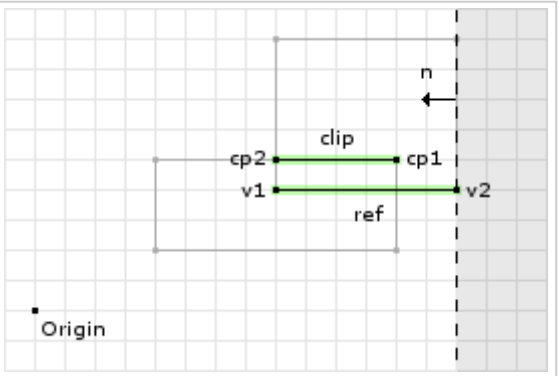


Figure 4: The second clip of example 1

```
// compute the reference edge's normal
refNorm = (0, 1)
// we didnt have to flip the reference and incident
// edges so refNorm stays the same
// compute the offset for this clipping operation
max = (0, 1) · (8, 4) = 4
// now we clip the points about this clipping plane, where:
// cp[0] = (12, 5)
// cp[1] = (8, 5)
(0, 1) · (12, 5) - 4 = 1
(0, 1) · (8, 5)  - 4 = 1
// since both points are greater than
// or equal to zero we keep them both
```

On the final clipping operation we keep both of the points. Figure 5 shows the final clipping operation and the valid region for the points. This ends the clipping operation returning a contact manifold of two points.

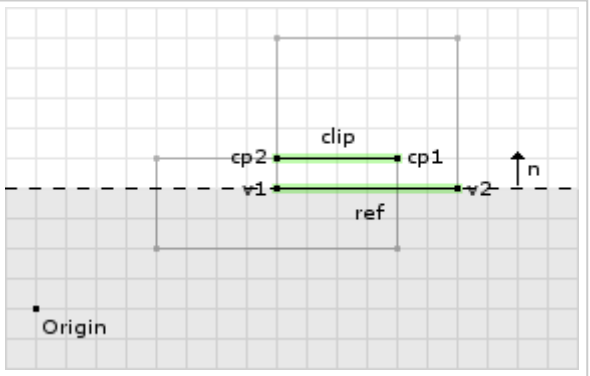


Figure 5: The final clip of example 1

```
// the collision manifold for example 1
cp[0] = (12, 5)
cp[1] = (8, 5)
```

Example 2
The first example was, by far, the simplest. In this example we will see how the last clipping operation is used. Figure 6 shows two boxes in collision, but in a slightly different configuration. We have following data to begin with:

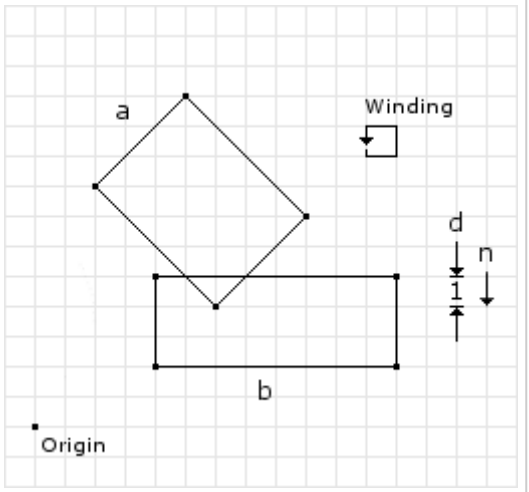


Figure 6: A more common box-box collision

```
// from the collision detector
// separation normal and depth
normal = (0, -1)
depth = 1
```

The first step is to get the “best” edges (the edges that are involved in the collision):

```
// the "best" edges    ( max ) | (  v1 ) | ( v2 )
//                    ---------+---------+-------
Edge e1 = A.best(n)  = ( 6, 4) | ( 2, 8) | (6, 4)
Edge e2 = B.best(-n) = (12, 5) | (12, 5) | (4, 5)
```

Figure 7 highlights the “best” edges on the shapes. Once we have found the edges we need to determine which edge is the reference edge and which is the incident edge:

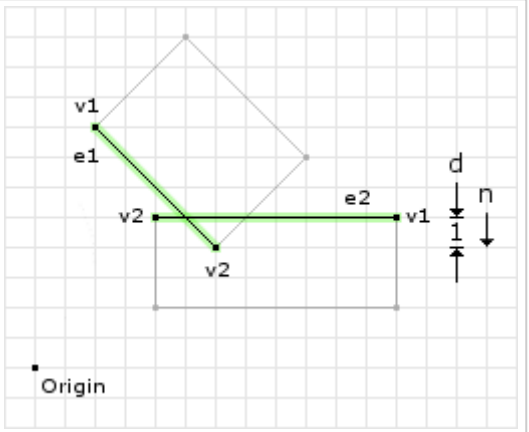


Figure 7: The “best” edges of figure 6

```
e1 = (2, 8) to (6, 4)  = (6, 4) - (2, 8)  = (4, -4)
e2 = (12, 5) to (4, 5) = (4, 5) - (12, 5) = (-8, 0)
e1Dotn = (4, -4) · (0, -1) = 4
e2Dotn = (-8, 0) · (0, -1) = 0
// since the dot product is greater for e1 we will use
// e2 as the reference edge and set the flip variable
// to true
ref = e2;
inc = e1;
flip = true;
```

Now that we have identified the reference and incident edges we perform the first clipping operation:

```
ref.normalize() = (-1, 0)
o1 = (-1, 0) · (12, 5) = -12
// now we call clip with
// v1 = inc.v1 = (2, 8)
// v2 = inc.v2 = (6, 4)
// n  = ref    = (-1, 0)
// o  = o1     = -12
d1 = (-1, 0) · (2, 8) - -12 = 10
d2 = (-1, 0) · (6, 4) - -12 = 6
// since both are greater than or equal
// to zero we add both to the clipped
// points object
cp.add(v1);
cp.add(v2);
// since both are positive then we skip
// the if block and return
```

The first clipping operation did not remove any points. Figure 8 shows the clipping plane and the valid and invalid regions. Both points were found to be inside the valid region of the clipping plane. Now for the second clipping operation:

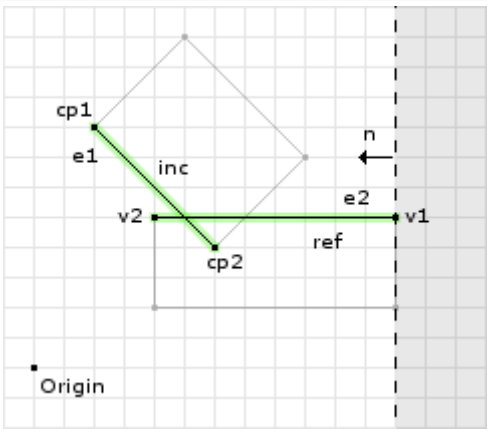


Figure 8: The first clip of example 2

```
o1 = (-1, 0) · (4, 5) = -4
// now we call clip with
// v1 = cp[0] = (2, 8)
// v2 = cp[1] = (6, 4)
// n  = ref   = (1, 0)
// o  = o1    = 4
d1 = (1, 0) · (2, 8) - 4 = -2
d2 = (1, 0) · (6, 4) - 4 = 2
// we only add v2 to the clipped points since
// its the only one that is greater than or
// equal to zero
cp.add(v2);
// since d1 * d2 = -4 we go into the if block
e = (6, 4) - (2, 8) = (4, -4)
u = -2 / (-2 - 2) = 1/2
e * u + v1 = (4 * 1/2, -4 * 1/2) + (2, 8) = (4, 6)
// then we add this point to the clipped points
cp.add(4, 6);
```

The second clipping operation removed one point that was outside the clipping plane (i.e. past the offset). But since there was another point on the opposite side of the clipping plane, we compute a new point on the edge and use it as the second point of the result. See figure 9 for an illustration. Now we continue to the last clipping operation:

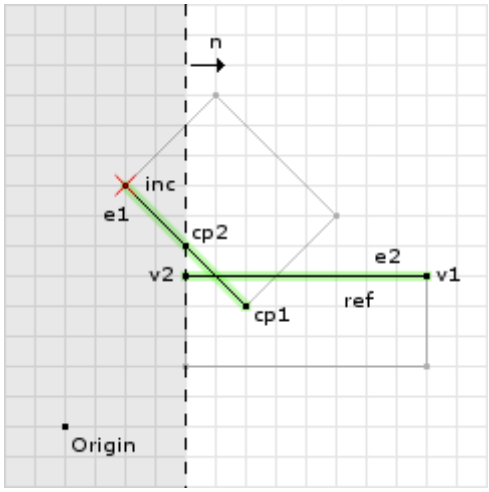


Figure 9: The second clip of example 2

```
// compute the reference edge's normal
refNorm = (0, 1)
// since we flipped the reference and incident
// edges we need to negate refNorm
refNorm = (0, -1)
max = (0, -1) · (12, 5) = -5
// now we clip the points about this clipping plane, where:
// cp[0] = (6, 4)
// cp[1] = (4, 6)
(0, -1) · (6, 4) - -5 = 1
(0, -1) · (4, 6) - -5 = -1
// since the second point is negative we remove the point
// from the final list of contact points
```

On the final clipping operation we remove one point. Figure 10 shows the final clipping operation and the valid region for the points. This ends the clipping operation returning a contact manifold of only one point.

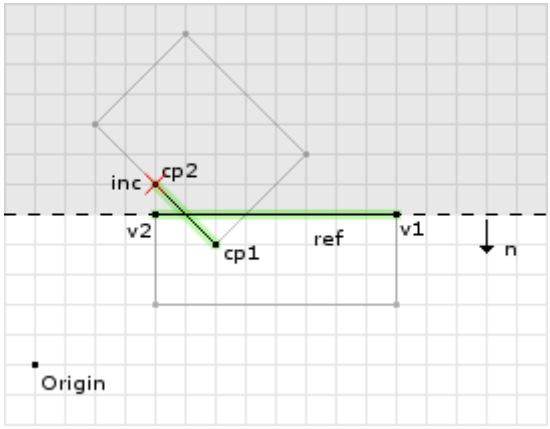


Figure 10: The final clip of example 2

```
// the collision manifold for example 2
cp[0] = (6, 4)
// removed because it was in the invalid region
cp[1] = null
```

Example 3

The last example will show the case where the contact point's depth must be adjusted. In the previous two examples, the depth of the contact point has remained valid at 1 unit. For this example we will need to modify the psuedo code slightly. See figure 11.

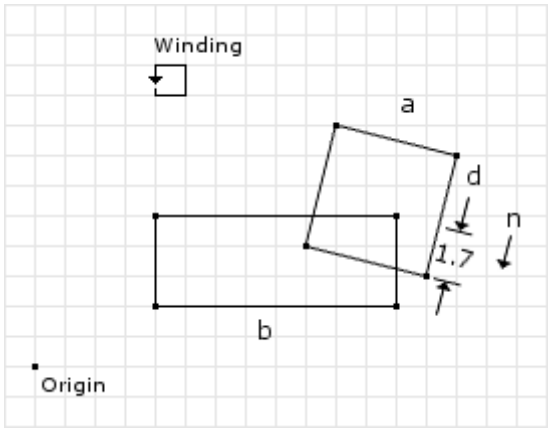


Figure 11: A very common box-box collision

```
// from the collision detector
// separation normal and depth
normal = (-0.19, -0.98)
depth = 1.7
```

The first step is to get the "best" edges (the edges that are involved in the collision):

```
// the "best" edges    ( max ) | (  v1 ) | ( v2 )
//                    ---------+---------+-------
Edge e1 = A.best(n)  = ( 9, 4) | ( 9, 4) | (13, 3)
Edge e2 = B.best(-n) = (12, 5) | (12, 5) | (4, 5)
```

Figure 12 highlights the "best" edges on the shapes. Once we have found the edges we need to determine which edge is the reference edge and which is the incident edge:

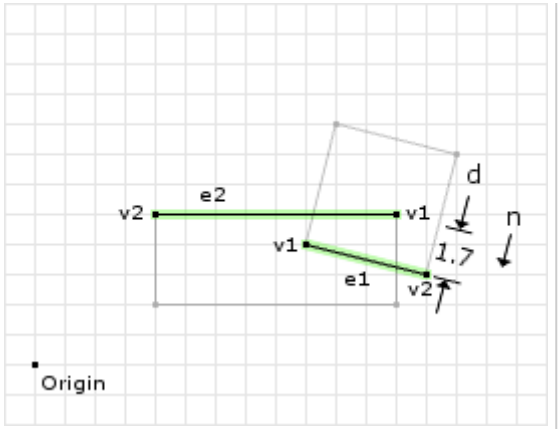


Figure 12: The “best” edges of figure 11

```
e1 = (9, 4) to (13, 3)  = (13, 3) - (9, 4)  = (4, -1)
e2 = (12, 5) to (4, 5) = (4, 5) - (12, 5) = (-8, 0)
e1Dotn = (4, -1) · (-0.19, -0.98) = -0.22
e2Dotn = (-8, 0) · (-0.19, -0.98) =  1.52
// since the dot product is greater for e2 we will use
// e1 as the reference edge and set the flip variable
// to true
ref = e1;
inc = e2;
```

Now that we have identified the reference and incident edges we perform the first clipping operation:

```
ref.normalize() = (0.97, -0.24)
o1 = (0.97, -0.24) · (9, 4) = 7.77
// now we call clip with
// v1 = inc.v1 = (12, 5)
// v2 = inc.v2 = (4, 5)
// n  = ref    = (0.97, -0.24)
// o  = o1     = 7.77
d1 = (0.97, -0.24) · (12, 5) - 7.77 = 2.67
d2 = (0.97, -0.24) · (4, 5)  - 7.77 = -5.09
// we only add v1 to the clipped points since
// its the only one that is greater than or
// equal to zero
cp.add(v1);
// since d1 * d2 = -13.5903 we go into the if block
e = (4, 5) - (12, 5) = (-8, 0)
u = 2.67 / (2.67 - -5.09) = 2.67/7.76
e * u + v1 = (-8 * 0.34, 0 * 0.34) + (12, 5) = (9.28, 5)
// then we add this point to the clipped points
cp.add(9.28, 5);
```

The first clipping operation removed one point that was outside the clipping plane (i.e. past the offset). But since there was another point on the opposite side of the clipping plane, compute a new point on the edge and use it as the second point of the result. See figure 13 for an illustration.
Since we still have two points in the ClippedPoints object we can continue and perform the second clipping operation:

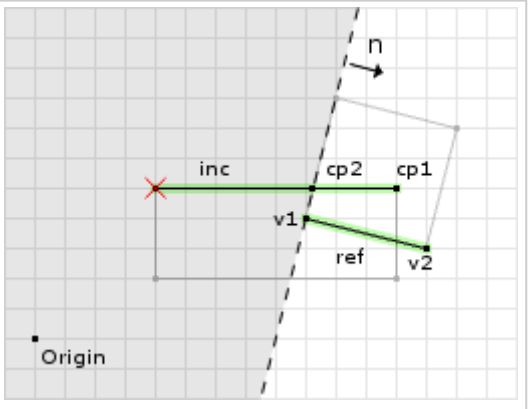


Figure 13: The first clip of example 3

```
o2 = (0.97, -0.24) · (13, 3) = 11.89
// now we call clip with
// v1 = cp[0] = (12, 5)
// v2 = cp[1] = (9.28, 5)
// n  = -ref  = (-0.97, 0.24)
// o  = -o1   = -11.89
d1 = (-0.97, 0.24) · (12, 5)   - -11.89 = 1.45
d2 = (-0.97, 0.24) · (9.28, 5) - -11.89 = 4.09
// since both are greater than or equal
// to zero we add both to the clipped
// points object
cp.add(v1);
cp.add(v2);
// since both are positive then we skip
// the if block and return
```

The second clipping operation did not remove any points. Figure 14 shows the clipping plane and the valid and invalid regions. Both points were found to be inside the valid region of the clipping plane. Now we continue to the last clipping operation:

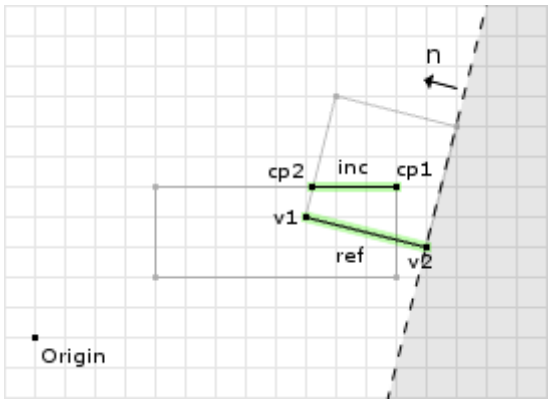


Figure 14: The second clip of example 3

```
// compute the reference edge's normal
refNorm = (0.24, 0.97)
// we didn't flip the reference and incident
// edges, so dont flip the reference edge normal
max = (0.24, 0.97) · (9, 4) = 6.04
// now we clip the points about this clipping plane, where:
// cp[0] = (12, 5)
// cp[1] = (9.28, 5)
(0.24, 0.97) · (12, 5)   - 6.04 = 1.69
(0.24, 0.97) · (9.28, 5) - 6.04 = 1.04
// both points are in the valid region so we keep them both
```

On the final clipping operation we keep both of the points. Figure 15 shows the final clipping operation and the valid region for the points. This ends the clipping operation returning a contact manifold of two points.

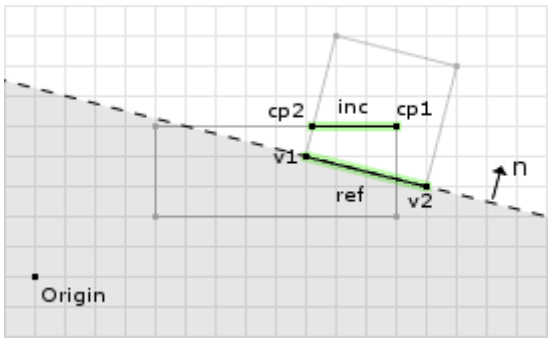


Figure 15: The final clip of example 3

```
// the collision manifold for example 3
cp[0] = (12, 5)
cp[1] = (9.28, 5)
```

The tricky bit here is the collision depth. The original depth of 1.7 that was computed by the collision detector is only valid for one of the points. If you were to use 1.7 for cp[1], you would over compensate the collision. So, because we may produce a new collision point, which is not a vertex on either shape, we must compute the depth of each of the points that we return. Thankfully, we have already done this when we test if the points are valid in the last clipping operation. The depth for the first point is 1.7, as originally found by the collision detector, and 1.04 for the second point.

```
// previous psuedo code
//if (refNorm.dot(cp[0]) - max < 0.0) {
// cp[0].valid = false;
//}
//if (refNorm.dot(cp[1]) - max < 0.0) {
// cp[1].valid = false;
//}

// new code, just save the depth
cp[0].depth = refNorm.dot(cp[0]) - max;
cp[1].depth = refNorm.dot(cp[1]) - max;
if (cp[0].depth < 0.0) {
  cp.remove(cp[0]);
}
if (cp[1].depth < 0.0) {
  cp.remove(cp[1]);
}
```

```
// the revised collision manifold for example 3
// point 1
cp[0].point = (12, 5)
cp[0].depth = 1.69
// point 2
cp[1].point = (9.28, 5)
cp[1].depth = 1.04
```

Curved Shapes

It's apparent by now that this method relies heavily on edge features. This poses a problem for curved shapes like circles since their edge(s) aren't represented by vertices. Handling circles can be achieved by simply get the farthest point in the circle, instead of the farthest edge, and using the single point returned as the contact manifold. Capsule shapes can do something similar when the farthest feature is inside the circular regions of the shape, and return an edge when its not.

Alternative Methods

An alternative method to clipping is to opt for the expanded shapes route that was discussed in the GJK/EPA posts. The original shapes are expaned/shrunk so that the GJK distance method can be used to detect collisions and obtain the MTV. Since GJK is being used, you can also get the closest points. The closest points can be used to obtain one collision point (see this).

Since GJK only gives one collision point per detection, and usually more than one is required (especially for physics), we need to do something else to obtain the other point(s). The following two methods are the most popular:

1. Cache the points from this iteration and subsequent ones until you have enough points to make a acceptable contact manifold.
2. Perturb the shapes slightly to obtain more points.

Caching is used by the popular Bullet physics engine and entails saving contact points over multiple iterations and then applying a reduction algorithm once a certain number of points has been reached. The reduction algorithm will typically keep the point of maximum depth and a fixed number of points. The points retained, other than the maximum depth point, will be the combination of points that maximize the contact area.

Perturbing the shapes slightly allows you to obtain all the contact points necessary on every iteration. This causes the shapes to be collision detected many times each iteration instead of once per iteration.

This entry was posted in *Blog*, *Collision Detection*, *Game Development*, *Physics* and tagged *Collision Detection*, *Contacts*, *EPA*, *Game Development*, *GJK*, *Physics*, *SAT*. Bookmark the *permalink*.

← Version 3.0.1 Version 3.0.2 →

52 thoughts on "Contact Points Using Clipping"

Konrad says:

February 20, 2012 at 6:47 PM

So f*****g(!!!) great tutorial! Thans a lot!

Konrad says:

February 20, 2012 at 6:54 PM

Got one question. By "normal" you mean collision normal or what?

William says:

February 21, 2012 at 8:38 AM

Typically I'm referring to the vector of minimum penetration of the two bodies as the normal (called the separation or collision normal). Later I use the term in the Clipping section to identify a vector perpendicular to an edge.

I qualified each time I used the word "normal" with either "separation" or "edge" to help clarify what I talking about. I hope that takes care of the confusion (thanks for the comment).

William

Konrad says:
February 22, 2012 at 10:24 AM
Once again thanks a lot :] I was looking for article like this for few days ;)

Konrad says:
March 14, 2012 at 11:48 AM
Hi!

When I started writing code, there were problems.

I have working SAT algorithm and there is collision information:
– MTD
– Depth

It's working very well, but I have problems with finding features.
Post to me on my e-mail and I show you my code.

I haven't any idea what I'm doing wrong…

Konrad says:
March 15, 2012 at 10:19 AM
Thanx for help via email ;)

Keep up the good work!

Albert says:
March 16, 2012 at 1:36 PM
Hi William, I have a doubt

How I calculate:

19 // get the reference edge normal
20 Vector2 refNorm = ref.cross(-1.0);

.cross is cross product?
cross product is defined for two vectors in R3, so..
How do I calculate ref.cross(-1.0) ?

Sorry my bad english..

Konrad says:

March 16, 2012 at 1:44 PM

Cross product for 2D vector with scalar looks like this:

Vec = (Vec.Y * Scalar, Vec.X * – Scalar)

So this code:
20 Vector2 refNorm = ref.cross(-1.0);
is:
Vector2 refNorm = Vector2 (ref.Y * -1.0, ref.X * -(-1.0));

Albert says:

March 16, 2012 at 2:46 PM

Thank you so much Konrad!!

Haubna says:

March 24, 2012 at 2:42 PM

First of all thank you for great tutorial!

```
1 Vector2 refNorm = ref.cross(-1.0);
2 // if we had to flip the incident and reference edges
3 // then we need to flip the reference edge normal to
4 // clip properly
5 if (flip) refNorm.negate();
```

I tried to implement it like that but it wasn't working right. Then i tried it with that peace of code (without the flip and a positive cross product):

```
1 Vector2d refNorm = ref.edge.cross(1.0);
```

and everything worked fine. Is that peace of code from you wrong or am i confused?

Haubna says:

March 25, 2012 at 11:21 AM

Oh sry i was wrong, fixed it xD

Peter says:

March 31, 2012 at 3:40 PM

Mind sharing what you did to fix your issue Haubna? I'm having an issue with example number 2. Everything matches up until the cross product and then it differs.

In the example it says:

```
ref.normalize() = (-1, 0)

// 2 clips...

refNorm = (0, 1)
```

How does that work? According to an above comment the 2D cross product we want is:

```
Vector2 refNorm = Vector2 (ref.Y * -1.0, ref.X * -(-1.0));
```

Thus this would mean refNorm = (-1.0 * 0.0, -1.0 * -(-1.0)) = (0.0, -1.0).

If just get an orthogonal vector using refNorm = (ref.Y, -ref.X) (which I've seen refered to as another form of the 2d cross product elsewhere), example 2 works but the other two fail.

If anyone has any advice to offer it would be greatly appreciated!

Also, thanks for the wonderful tutorial William!

William says:

April 2, 2012 at 7:44 AM

The definition of the 2D cross product I use is:

```
Vector2 cross(Vector2 v, double z) {
  return new Vector2(-1.0 * v.y * z, v.x * z);
}
```

Which when used will be:

```
ref = (-1, 0)
//                (-1.0 * v.y *    z,  v.x *    z)
cross(ref, -1.0) = (-1.0 * 0.0 * -1.0, -1.0 * -1.0) = (0.0, 1.0)
```

The cross product can yield two different vectors depending on the handedness of the system. I use the right-hand-rule for my projects. However, in two dimensions there exists 2 orthogonal vectors from which to choose from. Which do we use is the problem with this approach. In addition, three dimensions have an infinite number of orthogonal vectors to a given vector. This is why the cross product is used vs. choosing a orthogonal vector.

I hope this clears up some confusion.

Dandi says:

August 23, 2012 at 2:19 PM

SAT, GJK, EPA and now this. Great job! Really useful. I hope you will write an article about how to find the Time of Impact, and about something solver, for example erin catto's sequential impulses.

Dirk Gregorius says:

September 24, 2012 at 2:50 PM

Hi William,

awesome tutorial! May I ask what you use for the sketches? They look really nice!

William says:

September 25, 2012 at 7:30 AM

Thanks man!

Sadly, I still haven't found a *fast* tool to make good sketches. These were all made in Gimp just using a bunch of layers and time (at least there's an easy way to make a grid pattern). If you'd like any of the gimp files for these just let me know,

William

Peter says:

September 27, 2012 at 5:54 PM

Just wanted to say that your clarification did help! Thanks again for the awesome tutorial!

John says:

September 26, 2013 at 2:52 PM

Great tutorial!

I do have one question to William, though. Is there anything significantly different or anything to watch out for in implementing this contact manifold method in 3D? I'm asking, becasue I followed the tutorial and implemented this method, but it does not

seems to behave right in 3 dimensional space. Is there something I'm missing? The only part which strikes my suspicion is:

```
1 // get the reference edge normal
2 Vector2 refNorm = ref.cross(-1.0);
```

The way I'm doing it in 3d is pretty much this:

```
1 // get the reference edge normal
2 Vector3 refNorm = ref.cross(Vector3(-1.0, 0.0, 0.0));
```

Everything else is identical to the tutorial.
Thanks in advance, and once again, amazing tutorial!

William says:

September 26, 2013 at 9:15 PM

The cross product of a vector and 1, in this case, is the same as the cross product of a vector and the z-axis (0, 0, 1). It turns out that doing this, with the negative z-axis, gives us the counter-clockwise normal of the vector. So all you need is some way to get the edge's normal. In the 3d case, you might be able to use the face's normal that the edge belongs to.

Other than that, I can see this method needed some additional logic to get working properly (handling all the collision cases). I think that the cases are narrowed down to two (face-edge and edge-edge?)

To learn more, search on Sutherland-Hodgman clipping.

Jamie says:

September 29, 2013 at 6:27 PM

First, I would like to thank you for this tutorial. Been a great help implementing my physics engine.
But I was a bit confused at the Edge. How is the edge structured?
Does it contain 2 Vector3 that holds the data of the 2 vertices that forms the edge? or is it a vector from A to B?

Jamie says:

September 29, 2013 at 11:45 PM

Sorry I didnt read your tutorial till the end. :P

Jamie says:

September 30, 2013 at 2:27 AM

On 2nd thought, when you call the constructor Edge(v,v,v1) taken from above.
What does each value represent?

Jamie says:

September 30, 2013 at 2:46 AM

Think you have a typo at Example 2
The max for the first one should be (6,4)

William says:

September 30, 2013 at 7:12 AM

Yep, that was a typo. Thanks for pointing that out. I have fixed it in the post.

Of course, you can structure your Edge class however you want, but in this post an Edge is represented by the start and end vertex (the winding direction is important) and the maximum vertex.

Specifically, the first parameter is the maximum vertex, the second parameter is the first vertex of the edge, and the last parameter is the second vertex of the edge.

William

Karl says:

January 22, 2014 at 2:22 AM

Thank you for this awesome tutorial. I had a question on the initial value of 'max' at the finding the feature part.
Are you initializing it to a really big negative number (-100000.0f)? Or are you setting it to 0?

William says:

January 22, 2014 at 12:42 PM

@Karl

Yeah, you'd want to initialize it to the largest negative number possible (-Double.MAX_VALUE in Java). That way if the projections were negative, imagine the shapes in the 3rd quadrant, you'd still get the maximum.

William

Dylan says:

March 9, 2014 at 6:13 AM

Great tutorial but I'm having a bit of a doubt on one part.

Edge e1 = A.best(n)
Edge e2 = B.best(-n);

You do the above to find the most perpendicular edge for each polygon, which makes sense. Then later on to find the reference edge you compare (e1.dot(n) <= e2.dot(n)). Shouldn't you be comparing e1.dot(n) <= e2.dot(-n) instead?

Thanks.

William says:

March 10, 2014 at 6:49 PM

@Dylan

Good question. Actually, it shouldn't matter since both n and -n should be equally perpendicular. However, what does need to be done is compare the absolute value of the dot products, not just the dot products. I've updated the post to reflect this.

William

Dylan says:

March 11, 2014 at 1:10 PM

That makes more sense. I think you also need to compare absolute dot products in your findBestEdge() method, when looking for the secondary vertex of the edge. Cheers.

William says:

March 11, 2014 at 6:47 PM

@Dylan

You can certainly do that as a precaution, but since we have both edge vectors (l and r) pointing towards the farthest point and pointing in the same direction as n, then the projections will always be positive or zero.

William

Dylan says:

March 11, 2014 at 2:12 PM

I finally got my version of this working. All of my code seems to be on par with yours with one small difference however. I had to change my code to compare things differently if the reference angle is flipped/not flipped as follows:

```
    float max = referenceNormal.dotProduct(referenceEdge->vertex);

    if (flip) {
        if (referenceNormal.dotProduct(clippedPoints[1]) > max)
            clippedPoints.erase(clippedPoints.begin() + 1);

        if (referenceNormal.dotProduct(clippedPoints[0]) > max)
            clippedPoints.erase(clippedPoints.begin());
    } else {
        if (referenceNormal.dotProduct(clippedPoints[1]) < max)
            clippedPoints.erase(clippedPoints.begin() + 1);

        if (referenceNormal.dotProduct(clippedPoints[0]) < max)
            clippedPoints.erase(clippedPoints.begin());
    }
```

Dylan says:

March 11, 2014 at 3:36 PM

Actually it turns out I get the same correct results if I just always use the normal from referenceEdge.crossProduct(-1.0f) and forget about the whole flipping thing. Not sure why mine works like this.

William says:

March 11, 2014 at 6:49 PM

@Dylan

This may depend on what you are using the results for. For example, in my collision detection code I have a check at the end that makes sure the normal is always pointing from convex A to convex B. The same applies here in that, if the reference edge was B, then I need to negate the front normal so that the normal is still pointing from A to B.

William

Branden says:

April 2, 2014 at 9:47 PM

This is an excellent tutorial! And I appreciate the effort that went into it.

I have one question though.
When using the variable max, to later clamp our points, what is ref.max referring to?

```
1 double max = refNorm.dot(ref.max);
```

Is that the furthest vertex in the reference edge using the reference edge's direction?

William says:

April 3, 2014 at 5:09 PM

@Branden

That's correct. ref.max is the vertex that is farthest along the normal (v in the first code sample).

William

Nithin says:

July 23, 2014 at 10:10 AM

Hello William,

What do you mean when you take the dot product of the edge object and normal?

Nithin

William says:

July 24, 2014 at 7:26 AM

@Nithin

I think you are asking what does Edge.dot(Vector) do? It takes the edge vector of the Edge object and returns the dot product:

```
1 // the edge vector is the vector from the first
2 // vertex to the second vertex
3 edgeVector = (edge.p2 - edge.p1);
4 return edgeVector.dot(n);
```

William

Nithin says:

July 24, 2014 at 9:15 AM

Thank you very much! It makes more sense now!

I have one more question. When we use the if statement to check if there are less than two clipped points, you said we have failed if true. Are we really returning nothing in that case or did I miss something?

Nithin

William says:

July 25, 2014 at 1:30 PM

@Nithin

Given valid input from the collision detector, it should never hit that condition. The only way to return less than 2 points from the Clip method would be if both points were behind the clipping plane. For example, in Figure 3, imagine if both of the incident edge's vertices were behind the clipping plane. This would mean that the two edges don't intersect and we have a problem at some earlier stage in the collision processing pipeline.

That said, it is possible that you still get here in some cases (like touching contact) due to finite precision floating point.

William

Andrei says:

October 22, 2014 at 3:43 PM

William,

// the edge that is most perpendicular
// to n will have a dot product closer to zero
if (r.dot(n) <= l.dot(n)) {
When finding the most perpendicular edge in the code above, shouldn't you first normalize r and l? The length of the vectors should not matter if you looking at angles only. And also shouldn't you be looking at absolute values of dot products? You want closest to 0 irrelevant of the sign

William says:

October 23, 2014 at 12:46 PM

@Andrei

Good catch, those should be normalized. I've fixed it within the post. In my implementations I use the normals (pre-normalized in local space) of the edges rather than the edges themselves to do this test so I can avoid the normalization every iteration.

We don't need absolute value there because both the r and l vectors are pointing at the farthest point and because n is pointing in the same direction. This will always yield a non-negative value.

William

Bas says:

December 21, 2014 at 3:33 PM

Hey William,

How do you handle cases where one box is completely inside another?
Or actually maybe that would work, but in my case I have a triangle inside a box (in 3d actually). And the clipping ends up without any vertices.
It's a bit like this:

———-
| |
| |> |
| |
———-

Thanks!

Bas says:

December 21, 2014 at 3:34 PM

Oof, it messed up my drawing. Does this work?

```
--------
|       |
|    |> |
|       |
--------
```

William says:

December 23, 2014 at 8:09 AM

@Bas

It's hard to say what could be going wrong without seeing the code. It should work for containment cases as well. Are you doing any additional clipping operations for the faces? The best thing would be to put the shapes in the failing configuration and then step through your code to see where/why they are missing.

William

Jeremy says:

January 29, 2015 at 4:24 PM

Hi William,
I have trouble understanding the use of the ref vector. I thought it was an edge made from a maximum and two vectors representing the extremities, obtained from the best() method. However in the next code snippet you seem to be using it as a vector, as opposed to the incident edge, where you use it's v1 and v2 properties instead of the edge itself.
Could you please shed some light on my comprehension issue?
Thanks!

William says:

January 29, 2015 at 8:56 PM

@Jeremy

Good catch. I've updated the post to help clarify. I added a new variable called refv which is the edge vector (the vector from the first point to the second of the edge).

William

Mike says:

February 9, 2015 at 1:16 PM

Hi William,

What exactly happens inside the A.best(n), and B.best(-n) functions to the edges of A and B respectively?

William Bittle says:

February 9, 2015 at 10:21 PM

@Mike

The best method should return the edge on the shape (A or B) that is most perpendicular to the separation normal. An example implementation is given in the Finding the Features section (the code sample just before the one you reference). Call that for both A and B to get their respective “best” edges.

William

Chase says:

May 10, 2015 at 8:48 PM

Hi William,

Thanks for writing up this article, it’s helped me a ton.

One thing I’m a little confused about is in the first block of code for finding the closest edge. The return value of the function is of the form Edge(v0, v1, v2). What is the purpose of v0? It doesn’t look to me like it’s ever used in the succeeding code when accessing various properties of the edges.

William Bittle says:

May 14, 2015 at 9:14 AM

@Chase

I should update that code block to be more clear. The constructor should look like Edge(maxVertex, edgeVertex1, edgeVertex2). The max vertex is used in the 4th code block, line 28 (ref.max), to do the last clip operation.

William

Mike says:

June 24, 2015 at 12:28 PM

Hi,

Can you please tell me what you are crossing in:

Vector2 refNorm = ref.cross(-1.0);

ref.max?
ref.v1?
ref.v2?
ref.v2-ref.v1?

Thanks,

Mike

William Bittle says:
June 24, 2015 at 5:14 PM

@Mike

I'm doing the cross product of the edge (ref.v2 – ref.v1) and the z-axis (0, 0, 1).

Thanks,
William

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Search …

TAGS

Bible Collision Detection Constrained Dynamics Contacts dyn4j EPA Equality Constraints Game Development GJK Java One God OpenGL Physics SAT

CATEGORIES

- Bible (1)
- Blog (19)
- Game Development (17)
  - Collision Detection (7)
  - Constrained Dynamics (10)
  - Physics (11)
- News (33)
  - Release (29)

RECENT POSTS

- Version 3.2.3
- Version 3.2.2
- Version 3.2.1
- Version 3.2.0
- dyn4j Moved to GitHub

RECENT COMMENTS

- geometry dash hack coins on How A Differential Gear Works

- fifa ultimate team coin generator online on EPA (Expanding Polytope Algorithm)
- http://www.rocktownyouth.org/ on How A Differential Gear Works
- See Also – DubiouSoft on GJK (Gilbert–Johnson–Keerthi)
- David on SAT (Separating Axis Theorem)