realtimecollisiondetection.net - the blog

Optimizing a sphere-triangle intersection test

December 30, 2010 at 10:31 pm · Christer Ericson · Filed under Math, Code

It's been quite a while since I posted on the blog (a) at all, and (b) about a topic remotely related to the name of my domain and subsequently **my book**. Thanks to a gentle nudge from my friend and colleague Pål-Kristian Engstad I'll try to rectify the situation with this post about how one could go about optimizing a sphere-triangle intersection test, suitable for a SIMD-implementation (such as on the PS3 SPUs).

This article isn't so much about producing useful code, but more about the steps that you should be able to identify and perform when optimizing vector-heavy code. I also briefly discuss how to deal with separating axis tests when one of the objects is a sphere (which doesn't have any obvious features).

Testing sphere vs. triangle

There are several different methods we can use to determine the intersection between a sphere and a triangle.

One is to "inflate" or "grow" the triangle by the sphere (forming the Minkowski sum of them) while simultaneously shrinking the sphere to a point. The test then becomes that of testing if the sphere center (the point) lies inside the spherically inflated triangle. In practical terms, this resolves into testing containment of the sphere center in one of 3 spheres (at the triangle vertices), 3 cylinders (aligned to the triangle edges), or a triangular prism (centered around the triangle itself).

Another option is to use the method of testing for a separating axis. Since this method is usually applied to (convex) polygons and polytopes only, and not for spheres, I thought it is a more interesting solution to elaborate on here, since I get to touch on two things at once.

Before continuing, let's briefly review the separating axis test, to see where it comes from and how it works.

Justifying separating-axis testing

If two convex objects, `C1` and `C2`, are not intersecting there must exist a pair of points `P1` and `P2`, one from each object, such that no other pair of points are closer than these points. Given such a pair, we can insert a separating plane between `P1` and `P2`, with normal `P2 - P1`. We call `P2 - P1` a **separating axis**.

Since there is an infinite number of points on either object, which is not practical for testing, we classify points into a finite number sets we call **features**. We do this in such a way that all points of a feature share the same test axis for separation. This way, it is sufficient to form candidate axes for all pairs of features from each object, and see if we can find object separation on any of those axes.

If, after examining all candidate axes, we cannot find separation, we must conclude that `C1` and `C2` are intersecting.

For convex polyhedra (aka polytopes) features are the vertices, edges, and faces of the polytopes.

Separating axis testing for a sphere

From the above we conclude that a triangle has three kinds of features that are involved in testing: vertices (3), edges (3), and a face (1). But what about a sphere? Well, if you followed the justification of the test, it should be clear a sphere has only one feature, but it varies from test to test: the point on the sphere surface closest to each given triangle feature!

If we define the problem as testing the intersection between a triangle `T`, defined by the vertices `A`, `B`, and `C`, and a sphere `S`, given by its center point `P` and a radius `r`, the axes we need to test are therefore the following:

1. normal to the triangle plane
2. perpendicular to AB through P
3. perpendicular to BC through P
4. perpendicular to CA through P
5. through A and P
6. through B and P
7. through C and P

We can now proceed with the actual testing, below.

Testing if sphere lies outside the triangle plane

The sphere `S` cannot intersect the triangle `T` if the sphere center `P` lies further away from the plane of `T` than `r` units. This can be expressed as follows:

```
Translate problem so sphere is centered at origin
A = A - P
B = B - P
C = C - P
Compute a vector normal to triangle plane (V), normalize it (N)
V = cross(B - A, C - A)
N = V / sqrt(dot(V, V))
Compute distance d of sphere center to triangle plane
d = abs(dot(A, N))
separated = d > r
```

Since both `d` and `r` are positive, we can square both sides of the test; this allows us to remove the `abs()` too. `N` is only used once, so we can substitute the expression in the calculation of `d`, and move out the constant `sqrt()` expression from within the dot product of `A` and `N`:

```
A = A - P
B = B - P
C = C - P
V = cross(B - A, C - A)
d = dot(A, V) / sqrt(dot(V, V))
separated = d^2 > r^2
```

`d^2` is equivalent to `dot(A, V)^2 / dot(V, V)` and since `dot(V, V)` is always positive we can multiply by `dot(V, V)` on both sides of the inequality, giving the final form:

```
A = A - P
B = B - P
C = C - P
V = cross(B - A, C - A)
d = dot(A, V)
e = dot(V, V)
separated = d * d > r * r * e
```

Testing if sphere lies outside a triangle vertex

For the axis through the sphere center `P` and the vertex `A` we have separation if:

```
Translate problem so sphere is centered at origin
A = A - P
B = B - P
C = C - P
Compute distance between sphere center and vertex A
d = sqrt(dot(A, A))
The plane through A with normal A ("A - P") separates sphere iff:
(1) A lies outside the sphere, and
separated1 = d > r
(2) if B and C lie on the opposite side of the plane w.r.t. the sphere center
separared2 = dot(B - A, A) > 0
separared3 = dot(C - A, A) > 0
separated = separated1 & separated2 & separated3
```

(Here we assume that `A` is the feature that separates `T` and `S`. As such, if `B` or `C` are closer, in the projection, then `A` cannot possibly be the separating feature. `B` or `C` might be, but we will find that out when testing for `B` and `C` as the potential separating feature, so we do not need to test that case here. The idea is to not do any early outs in this code.)

As before we square terms to get rid of the `sqrt()` and also use the linearity (or distributive properties) of the dot product to remove some vector subtractions, to give the optimized pseudocode:

```
A = A - P
B = B - P
C = C - P
aa = dot(A, A)
ab = dot(A, B)
```

```
ac = dot(A, C)
separated = (aa > r * r) & (ab > aa) & (ac > aa)
```

If we do the same thing for the remaining two axes and combine the tests we arrive at the final form for testing all vertices:

```
A = A - P
B = B - P
C = C - P
aa = dot(A, A)
ab = dot(A, B)
ac = dot(A, C)
bb = dot(B, B)
bc = dot(B, C)
cc = dot(C, C)
rr = r * r
separateda = (aa > rr) & (ab > aa) & (ac > aa)
separatedb = (bb > rr) & (ab > bb) & (bc > bb)
separatedc = (cc > rr) & (ac > cc) & (bc > cc)
separated = separateda | separatedb | separatedc
```

Testing if sphere lies outside a triangle edge

To determine if there is a separating plane through the triangle edge `AB` with a normal parallel to the axis through the sphere center `P` and perpendicular to `AB`, we first need to compute the point `Q` on the line `AB` to which `P` projects. This `Q` is really just the projection of `P` onto the plane and must, by construction, fall onto the line `AB` (which is why we project `P` onto `AB`). We compute `Q` as follows:

```
AB = B - A
t = dot(P - A, AB) / dot(AB, AB)
Q = A + t * AB
```

(For a full justification of why Q is computed like this, see Section 5.1.2 of RTCD.)

To test separation we now need to see if `Q` is further than `r` from `P` and if `C` lies on the opposite side of the plane through `AB` with normal `PQ`.

```
Compute Q
AB = B - A
t = dot(P - A, AB) / dot(AB, AB)
Q = A + t * AB
Test separation
QP = P - Q
QC = C - Q
separated1 = sqrt(dot(QP, QP)) > r
separated2 = dot(QP, QC) < 0
separated = separated1 & separated2
```

We can get rid of the `sqrt()` by squaring both sides of the first inequality (as both sides are positive) and we can prepare to get rid of the division by multiplying both sides of the inequalities by the positive number `e^2`:

```
AB = B - A
d = dot(P - A, AB)
```

```
e = dot(AB, AB)
Q = A + (d / e) * AB
QP = P - Q
QC = C - Q
separated1 = dot(QP * e, QP * e) > r * r * e * e
separated2 = dot(QP * e, QC * e) < 0
separated = separated1 & separated2
```

We finalize getting rid of the division through the following transformation:

```
AB = B - A
d = dot(P - A, AB)
e = dot(AB, AB)
Q = A * e + d * AB
QP = P * e - Q
QC = C * e - Q
separated = [dot(QP, QP) > r * r * e * e] & [dot(QP, QC) < 0]
```

Doing all three edge tests together results in the following pseudocode:

```
AB = B - A
d1 = dot(P - A, AB)
e1 = dot(AB, AB)
Q1 = A * e1 + d1 * AB
QP1 = P * e1 - Q1
QC = C * e1 - Q1
separated1 = [dot(QP1, QP1) > r * r * e1 * e1] & [dot(QP1, QC) < 0]

BC = C - B
d2 = dot(P - B, BC)
e2 = dot(BC, BC)
Q2 = B * e2 + d2 * BC
QP2 = P * e2 - Q2
QA = A * e2 - Q2
separated2 = [dot(QP2, QP2) > r * r * e2 * e2] & [dot(QP2, QA) < 0]

CA = A - C
d3 = dot(P - C, CA)
e3 = dot(CA, CA)
Q3 = C * e3 + d3 * CA
QP3 = P * e3 - Q3
QB = B * e3 - Q3
separated3 = [dot(QP3, QP3) > r * r * e3 * e3] & [dot(QP3, QB) < 0]

separated = separated1 | separated2 | separated3
```

Yet again we translate there problem so the sphere is at the origin (`P` = **0**), which, after some shuffling to remove unary minuses and making all comparisons greater-than, turns the pseudocode into the final version:

```
A = A - P
B = B - P
C = C - P
AB = B - A
BC = C - B
CA = A - C
d1 = dot(A, AB)
e1 = dot(AB, AB)
```

```
d2 = dot(B, BC)
e2 = dot(BC, BC)
d3 = dot(C, CA)
e3 = dot(CA, CA)
Q1 = A * e1 - d1 * AB
QC = C * e1 - Q1
Q2 = B * e2 - d2 * BC
QA = A * e2 - Q2
Q3 = C * e3 - d3 * CA
QB = B * e3 - Q3
separated1 = [dot(Q1, Q1) > r * r * e1 * e1] & [dot(Q1, QC) > 0]
separated2 = [dot(Q2, Q2) > r * r * e2 * e2] & [dot(Q2, QA) > 0]
separated3 = [dot(Q3, Q3) > r * r * e3 * e3] & [dot(Q3, QB) > 0]
separated = separated1 | separated2 | separated3
```

All together now

Finally, let’s combine all three (or 7) tests together in one final block of pseudocode:

```
A = A - P
B = B - P
C = C - P
rr = r * r
V = cross(B - A, C - A)
d = dot(A, V)
e = dot(V, V)
sep1 = d * d > rr * e
aa = dot(A, A)
ab = dot(A, B)
ac = dot(A, C)
bb = dot(B, B)
bc = dot(B, C)
cc = dot(C, C)
sep2 = (aa > rr) & (ab > aa) & (ac > aa)
sep3 = (bb > rr) & (ab > bb) & (bc > bb)
sep4 = (cc > rr) & (ac > cc) & (bc > cc)
AB = B - A
BC = C - B
CA = A - C
d1 = ab - aa
d2 = bc - bb
d3 = ac - cc
e1 = dot(AB, AB)
e2 = dot(BC, BC)
e3 = dot(CA, CA)
Q1 = A * e1 - d1 * AB
Q2 = B * e2 - d2 * BC
Q3 = C * e3 - d3 * CA
QC = C * e1 - Q1
QA = A * e2 - Q2
QB = B * e3 - Q3
sep5 = [dot(Q1, Q1) > rr * e1 * e1] & [dot(Q1, QC) > 0]
sep6 = [dot(Q2, Q2) > rr * e2 * e2] & [dot(Q2, QA) > 0]
sep7 = [dot(Q3, Q3) > rr * e3 * e3] & [dot(Q3, QB) > 0]
separated = sep1 | sep2 | sep3 | sep4 | sep5 | sep6 | sep7
```

It should be noted that throughout I’ve been deliberately sloppy with the separation criteria (i.e. greater-than vs. greater-equal, etc) to allow for the final version of the code to perform the same comparison for several tests in parallel. (Using SIMD instructions, the sixteen scalar greater-than comparisons can be packed into four SIMD comparisons.) Also, if implemented as derived, the resulting code is not the greatest example of floating-point robustness since we have cross-multiplied and squared terms to get rid of divisions and square roots in our quest for faster code. Finally, like Knuth, I have only "proved" the pseudocode to be correct, I haven't actually tested it, so beware bugs. Caveat emptor, in other words!

Aside: there is no "separating-axis theorem"

As a side note, I feel compelled to point out that there is no such thing as a "separating-axis theorem" despite people talking about it in academic papers and even writing Wikipedia articles about it! Why? Because it is not a theorem and no one has ever proven it in print, nor would anyone attempt to, because it is a trivial consequence of the separating hyperplane theorem, which is a *real* proven and fundamental theorem in convex analysis! The correct vernacular is **"separating-axis test."** Get it right.

Similar Posts:

- Minimum bounding circle (sphere) for a triangle (tetrahedron)
- Coplanarity is teH eVil!!!11!1!!
- Triangle-triangle tests, plus the art of benchmarking
- Robustly computing the centroid for a point set
- My recommended books

Share and Enjoy:

Permalink

5 Comments »

1. **Pierre said,**

   January 21, 2011 @ 11:01 am

   Hey, nice to see new posts from you, Christer! Unfortunately the final code doesn't work. There is indeed a bug here:

QC = C * e1 + Q1
QA = A * e2 + Q2
QB = B * e3 + Q3

Should be:

QC = C * e1 - Q1
QA = A * e2 - Q2
QB = B * e3 - Q3

As for the separating axis theorem, I suppose we all have to blame Stefan Gottschalk, who was the first AFAIK to talk about it in his thesis. I don't remember if he "proved it in print" though :)

2. **christer said,**

   January 21, 2011 @ 9:11 pm

   Thanks Pierre! I fixed the sign error in the article.

   It is historically interesting to note (as I did in my book) that the whole idea was actually presented to Gottschalk by Mikki Larcombe on comp.graphics.algorithms in 1995. Google groups have the two postings archived (as do I in my personal archive of interesting posts from back then). It's worth noting two things: 1) Larcombe clearly presents the whole idea behind separating-axis testing to Gottschalk, and 2) there's no mentioning of "separating axis theorems" anywhere. That's an unfortunate later construction.

   Gottschalk 1995 comp.graphics.algorithms inquiry:

   > From: gotts...@cs.unc.edu (Stefan Gottschalk)
   > Subject: arbitrary bounding box intersection
   > Date: 1995/10/09
   > Message-ID: <45bup9$c9k@zworykin.cs.unc.edu>#1/1
   > X-Deja-AN: 117294500
   > organization: The University of North Carolina
   > newsgroups: comp.graphics.algorithms
   >
   > I have two right rectangular prisms (I think that's right - they're 3d rectangles, anyway) arbitrarily positioned and oriented in 3-space, and I want to know if they touch or intersect. I don't need to know WHERE they touch, but only WHETHER they touch.
   >
   > Does anyone know an efficient way to do this?
   >
   > Currently I test whether any of the edges of box A intersects any of the faces of box B and vice versa, for a total of 144 edge-face intersection tests. Yuck. I want to do better.
   >
   > -stefan

Mikki Larcombe 1995 comp.graphics.algorithms answer:

From: Mikki.Larco...@dcs.warwick.ac.uk (Mikki Larcombe)
Subject: Re: arbitrary bounding box intersection
Date: 1995/10/11
Message-ID: <1995Oct11.130545.22985@dcs.warwick.ac.uk>#1/1
X-Deja-AN: 117385584
sender: mhel@granite (Mikki Larcombe)
x-nntp-posting-host: granite
references: <45bup9$c9k@zworykin.cs.unc.edu>
<813274427snz@jjavp.demon.co.uk>
organization: Department of Computer Science, Warwick University,
England
newsgroups: comp.graphics.algorithms
originator: Mikk@granite

Given two convex polyhedra these are separable (i.e. have no point in
common) if there is some axis such that the points from each polygon
project
onto this axis in two separable sets. This is equivalent to the notion of a
plane of separation but has the advantage of giving a direct measure for
comparison purposes: given the two convex polyhedra A and B and
letting
these project onto some axis as p(A),p(B) then separability exists if
max p(a)<min p(B) or max p(B)<min p(A).

This leaves us to choose the appropriate directions. There is a finite set
of
such directions for polyhedra: the facet normals from both polyhedra
and the
directions formed from the vector cross-products of an edge direction
taken from
each polyhedra. For the rectangular boxes this gives 15 directions, but
the tests are rejective… it is sufficient to find separability in one of these
directions, intersection requires overlap of the projected sets in all of
these
directions.

In making these tests it is useful to have the formula for the extent of a
rectangular box in any direction to hand (extent is the range of the
projected
points. Given a mid-point m, orthonormal
basis vectors b0,b1,b2, and semi-axis s0,s1,s2 the extent in a direction
d
relative to a test point q is:
d.(m-q)+/-(s0|d.b0|+s1|d.b1|+s2|d.b2|)
where |d.b0| means the absolute value of the dot product.
–

Mikki Larcombe
–
Mikki Larcombe

3. **Pierre said,**

   January 22, 2011 @ 4:05 am

   >That’s an unfortunate later construction

   Well yeah, a later construction that appears in Gottschalk’s thesis, section 4.2:
   http://www.mechcore.net/files/docs/alg/gottschalk00collision.pdf

   But this is from 2000 apparently, I suppose somebody else may have used the term first, between 1995 and 2000. Oh well. Fascinating stuff anyway :)

4. **Pierre said,**

   January 23, 2011 @ 3:51 am

   Ok, some more thoughts… :)

   Since this is a SAT, it should be possible to use the same axes in the “dynamic” version of SAT - the thing popularized by Ron Levine on GDA a long time ago -, to get a moving-sphere-vs-triangle test. Right? Except it doesn’t work, and I think it’s simply because contrary to what happens for, say, box-vs-box, here the axes change over time. For boxes the axes remain the same when a box gets translated. For spheres, the axes need recomputing.

   But then, going back to how the axes are built, there is something not very satisfying there. For the “edge axes” (number 2, 3 and 4 in the article) we effectively already do some distance computations (Q = closest point from P to line AB). Ok, but if we can do that, why not simply compute Q = closest point from P to *segment* AB, which is basically the same computation, and it lets us drop the “sphere axes” (number 5, 6, 7) completely. And if we continue in that direction, computing Q = closest point from P to the triangle gives us, in a way, a single axis to test (at which point we don’t really have a SAT anymore of course, just a distance-based test).

   I found it interesting to see how the SAT version can gradually turn into the distance test. I also wonder now what are the proper “rules” to choose the candidate axes. You wrote:

   “it should be clear a sphere has only one feature, but it varies from test to test: the point on the sphere surface closest to each given triangle feature!”

   and at the same time:

   “a triangle has three kinds of features that are involved in testing: vertices (3), edges (3), and a face (1)"

But that's edges right there, not lines. So why not computing the closest point from P to the edges indeed? Is it an arbitrary choice? A way to avoid the extra tests in the segment case, to make it more SIMD friendly?

And more importantly, how do we make this work for the moving-sphere case? Any idea? :)

5. **christer said,**

   January 23, 2011 @ 2:26 pm

   I look at it as follows. Computing distance is a superset of testing for separation, as the former directly determines the latter. As such, additional computation is generally (but not necessarily) needed to compute distance over separation.

   As you note, there's also often overlap between the calculations for distance vs. separation. Some of this follows because in both types of test we have to perform the testing by cases that correspond to regions of space. Sometimes these regions are the same or at least sufficiently close that the math becomes the same or very similar.

   I haven't tried, nor really thought about it much, but to handle the moving-sphere case SAT-style, you would have to parameterize the sphere-axis movement and then track the projections onto this parameterized axis, same as you would for a stationary axis. Sounds messy.

   For moving sphere vs. triangle, a distance-based approach intuitively sounds more appealing. Dave Eberly describes one such approach in Intersection of Moving Sphere and Triangle.

RSS feed for comments on this post · TrackBack URI

Leave a Comment

You must be logged in to post a comment.

- **About me**

  Hi, I'm Christer Ericson, Director of Tools and Technology at Sony Santa Monica (the God of War team). This is my blog.

  Linked in profile

  + Google™

- **The book I wrote:**

  See the RTCD web page, or see my recommended books page.

- **Recent Comments**
  - christer on Range reduction (for trig functions)
  - DavidGalloway on Range reduction (for trig functions)
  - Post-Mortem – ColorIt | Fatal Abstraction on Order your graphics draw calls around!
  - latimerius on Order your graphics draw calls around!
  - christer on Order your graphics draw calls around!

- **Recent Posts**
  - Game developer salaries revisited
  - Stopping Wordpress spammer registrations
  - Optimizing a sphere-triangle intersection test
  - ACM commits more evil - please act!
  - ACM censors linking!

- **Categories**
  - AI
  - Code
  - Design
  - From hell
  - Games industry
  - Graphics
  - Links
  - Math
  - Miscellaneous
  - Robustness

- **Archives**
  - February 2011
  - January 2011

- December 2010
- December 2009
- November 2009
- August 2009
- June 2009
- May 2009
- January 2009
- November 2008
- October 2008
- September 2008
- August 2008
- July 2008
- June 2008
- May 2008
- April 2008
- March 2008
- February 2008
- December 2007
- November 2007
- October 2007
- September 2007
- August 2007
- July 2007
- June 2007

Search

Links

- .mischief.mayhem.soap.
- A neighborhood of infinity
- another bottle at sea
- Aurora
- Bright shiny object syndrome
- C0DE517E
- Code Fortress
- code monk
- CodeItNow
- Coder Corner
- Deano’s Home From Home
- Gaffer on games
- Game Creator
- gamedev.net
- Graphics Runner
- highly professional scums
- Ignacio Castaño

- Jakob's blog
- Level of detail
- Little Tutorials
- Lucille development diary
- Math Blog
- meshula.net blog
- Molly Rocket forums
- ompf.org forums
- Pete Shirley's Graphics Blog
- Physics simulation forum
- pixels too many
- Pushing buttons...
- Rakkar's Blog
- RenderWonk
- Senzee 5
- Simon Brown's page
- smallcode
- Systematic Gaming
- The Atom Project
- The Daily DIP Count
- The Unapologetic Mathematician
- The Universe of Discourse
- The Workbench
- TomF's Tech Blog
- What your mother never told you about graphics development

Meta

- Register
- Login
- Entries RSS
- Comments RSS
- WordPress.org

Design by Beccary and Weblogs.us · XHTML · CSS