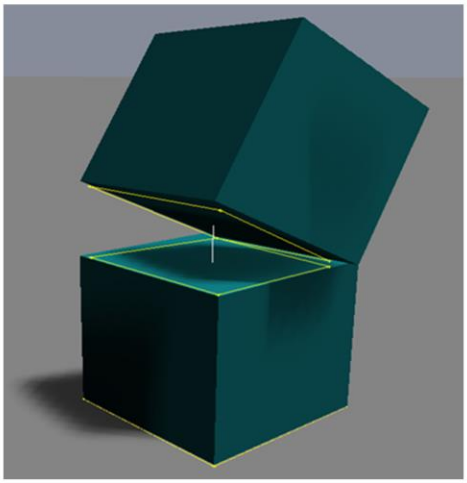
The Separating Axis Test between Convex Polyhedra

Dirk Gregorius – Valve Software

- Welcome! My name is Dirk Gregorius and I am software engineer at Valve working on collision detection and rigid body simulation.
- The talk today will be about the SAT between convex polyhedra

Collision Detection between Convex Polyhedra



- Why shall we use the SAT?

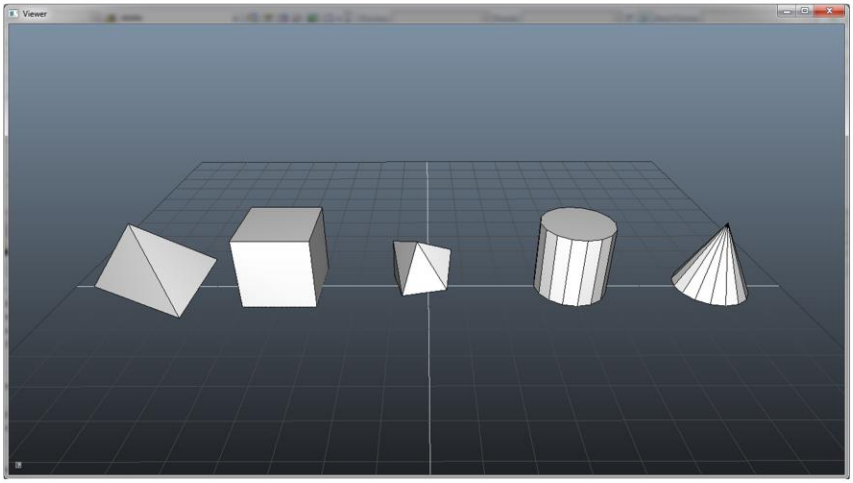
- The SAT is versatile algorithm and can tell us things like :
- Whether two convex polyhedra overlap
- The touching features
- The penetration distance
- The direction in which we need to resolve the penetration (axis of minimum penetration)

Talk Outline

- Show examples of convex polyhedra and define convexity
- Use simple sphere-sphere collision detection to introduce concept Minkowski difference
- Develop a **Separating Axis Test (SAT)** between convex polygons/polyhedra (2D and 3D)
- Use Gauss Maps of convex polyhedra to optimize the SAT in 3D
- Show an efficient implementation

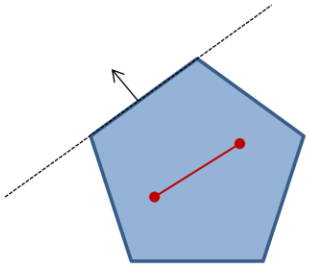
- Before we start I will give you a quick talk outline

Examples of Convex Polyhedra

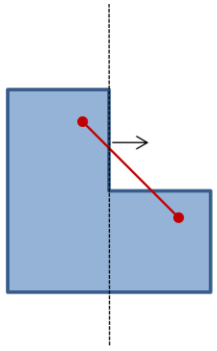


- Before we start some examples of convex polyhedra (tetrahedron, box and a convex hull)
- Notice cylinder and cone as approximation for quadric shapes!

Convexity



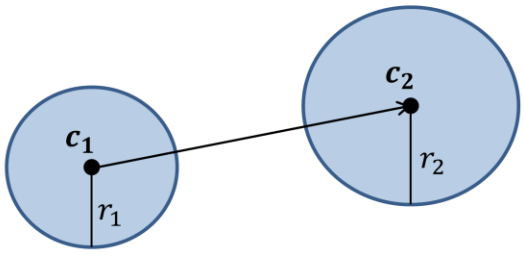
Convex



Concave

The line between any two points inside a convex shape must be completely contained
➔ The shape is completely behind each face plane if convex

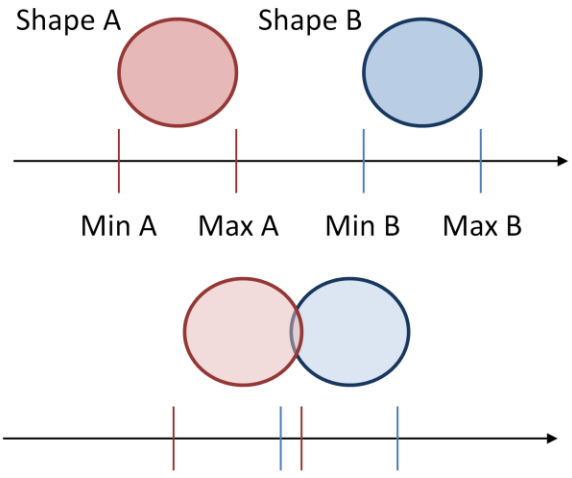
Overlap between Spheres



$$d = |\boldsymbol{c_1} - \boldsymbol{c_2}| - (r_1 + r_2) \Rightarrow \begin{cases} d > 0, separated \\ d \leq 0, overlapping \end{cases}$$

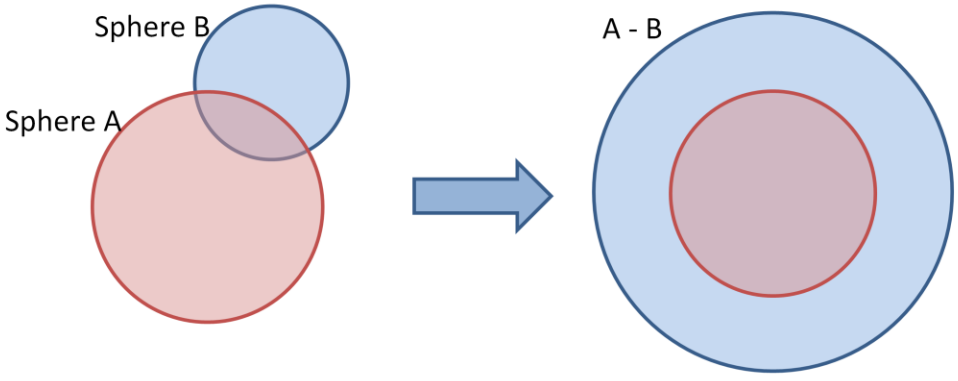
- We will start with a simple overlap test between two spheres
- Two spheres overlap if the distance between the centers is less then the sum of the radii.

The Separating Axis



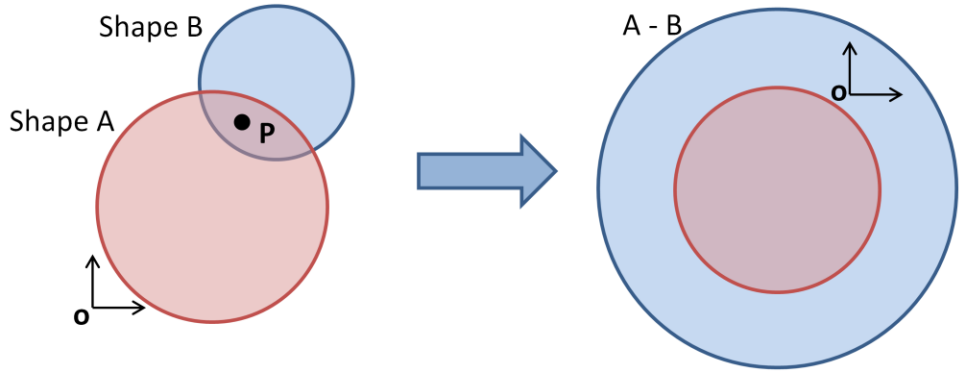
- Project both spheres onto axis through center points
- Then test intervals for overlap
- If interval not overlapping, we say we detected a separating axis
- The axis through the center points is the only possible separating axis because of the sphere symmetry

Minkowski Difference between two Spheres



- I would also like to use this example to introduce the concept of Minkowski difference
- What is the Minkowski Difference between two spheres?
- Handwaveingly: Inflate A up by the radius of B
- Shrink B to a point

Minkowski Difference between two Spheres (2)

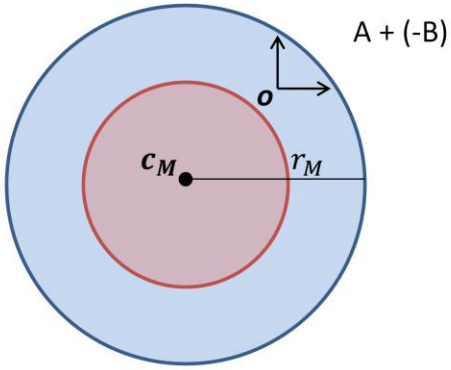


If two convex shapes overlap the Minkowski difference must contain the origin!

- Why is this a good idea?
- Mathematically: Subtract all points of B from all points of A
- It can be shown: **If two convex shapes overlap the Minkowski difference must contain the origin!**
- Explanation: If you subtract all points in B from all points in A and they share a common point one of the differences must be the zero vector
- Another way of thinking about this is: By subtracting each point in B from each point in A and there are no two points with zero distance, then the shapes must be disjoint

Overlap between Point and Sphere

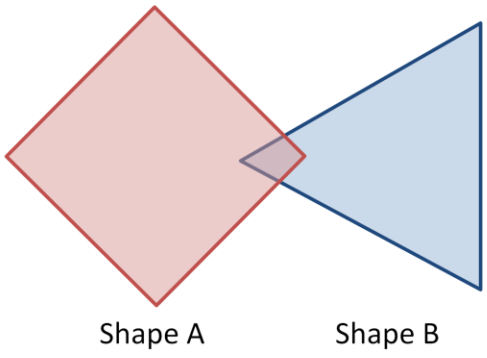
$s = |\boldsymbol{c_M} - \boldsymbol{o}| - r_M$



The original problem is transformed into a point inside sphere problem.

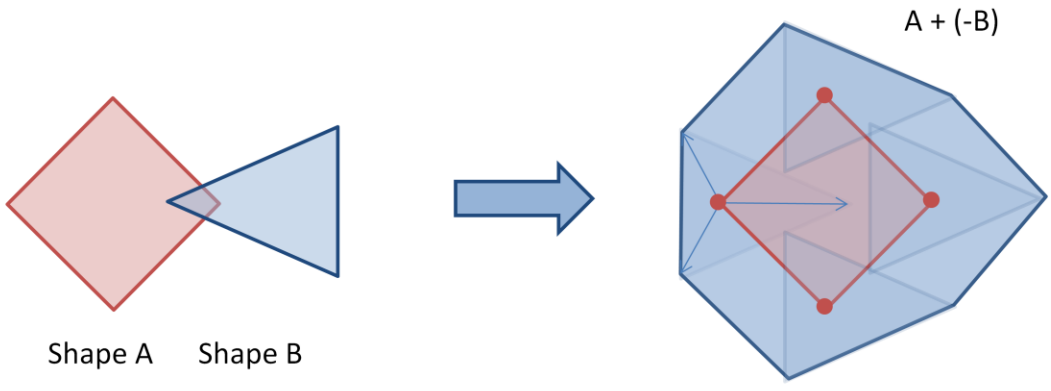
- Why is this a good idea?
- The Minkowski difference between two spheres transforms **the original problem into a point inside sphere problem.**
- Not big win here, but will become useful for more complex problems later

Overlap between Convex Polygons (2D)



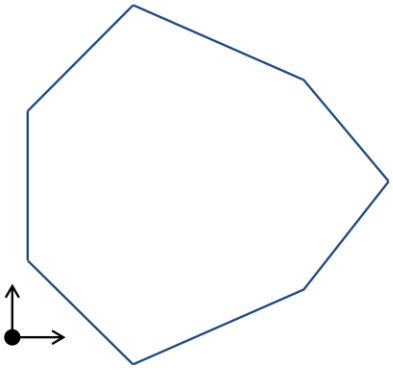
- Now that we understand what separating axes and Minkowski differences are, lets move on to SAT between convex polygons
- Examine the problem in Minkowski space since we cannot write down the solution immediately.

Minkowski Difference between Convex Polygons (2D)

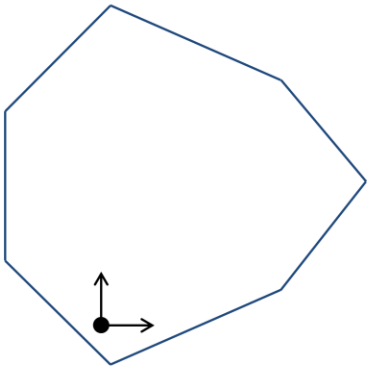


- How do we build the Minkowski difference between two polygons?
- Inflate quad by area of flipped triangle
- Shrink triangle to point
- Need to flip triangle to account for Minkowski **difference**
- The Minkowski difference between two convex polygons is also convex polygon itself

Point inside Convex Polygon



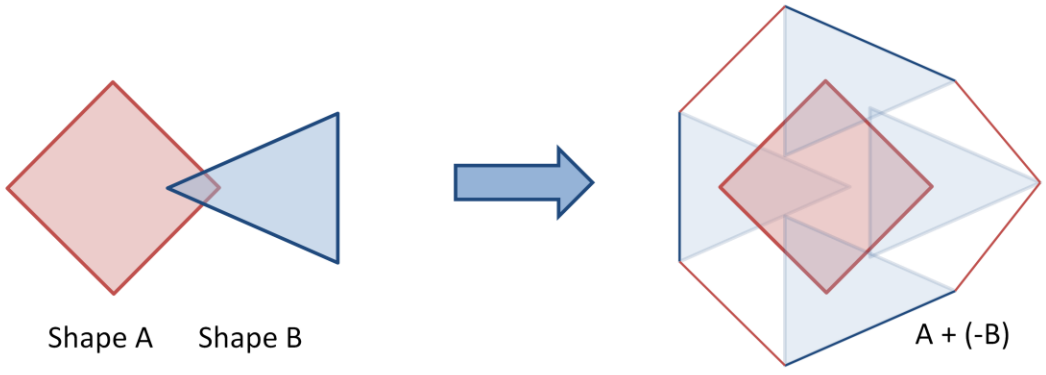
Origin outside -> Separation



Origin inside -> Overlap

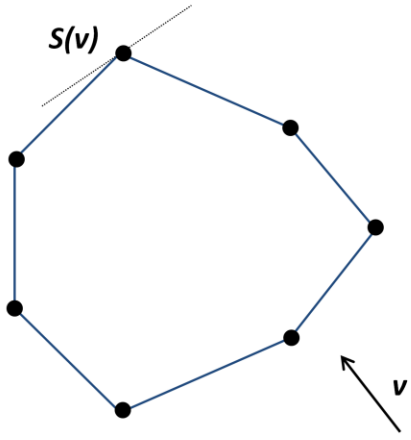
- This transforms the original problem into a point inside convex polygon problem
- Simple to solve: If origin is inside the Minkowski difference the original shapes overlap. If it outside, the shapes are separated
- This is a possible solution, but we don't want to build the Minkowski difference explicitly

Face Normals of Minkowski Difference define possible Separating Axes



- It can be shown that the possible separating axes between two convex polygons are the face normals of the Minkowski difference
- The faces of the Minkowski difference are the faces of original shapes, but pushed out
- So we know the face normals of the Minkowski difference and therefore the possible separating axes
- We could now project both shapes onto each possible separating axis and compare the intervals, but I'd like to show you a slightly more efficient and intuitive way

Finding support points



- We will need to understand support points to test possible separating axes
- A support point is simply the most extreme point into a given direction.
- And if the result is not unique any of those points will be fine

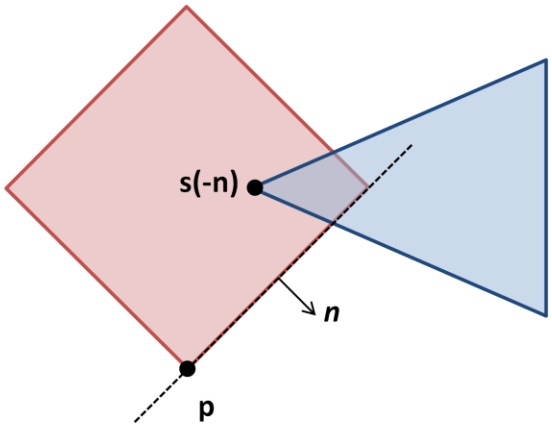
SAT: Support Function (2D)

```
Vector2 Polygon::GetSupport( const Vector2& direction ) const
{
   for ( int index = 0; index < m_VertexCount;  ++index )
      Vector2D vertex = m_Vertices[ index ];
      float projection = Dot( vertex , direction );

      if ( projection > bestProjection )
         bestProjection = projection,
         bestVertex = vertex;

   return bestVertex;
};
```

- I'd like to quickly show a possible implementation to give you an idea how this works (if you are not already familiar with this concept)
- Iterate all vertices and project each vertex onto the search direction using the dot product.
- Done inside this loop and we keep track of the vertex with the largest projection
- Finally we return the best vertex

Testing a Separating Axis (2D)

$d = \mathbf{n} \cdot (\mathbf{s} - \mathbf{p})$



- We will now use support points to test a possible separating axis between two convex polygons
- We build a plane for each edge on one polygon and find the support point in the opposite normal direction on the other polygon
- The distance of that support point to the plane is the separation or penetration for this axis
- This is a bit more efficient since we only touch half of the vertices per axis as compared to projecting both shapes and then comparing intervals

SAT: Face Directions (2D)

```
Query QueryFaceDirections( const Polygon& polygonA, const Polygon& polygonB )
{
    for ( int index = 0; index < polygonA .FaceCount; ++index )
      Plane planeA = polygonA .GetPlane( index );
      Vector vertexB = polygonB .GetSupport( -planeA.Normal );
      float distance= Distance( planeA, VertexB );

      if (distance > bestDistance )
         bestDistance = distance;
         bestIndex = index;

   return largest distance and associated index of face;
};
```

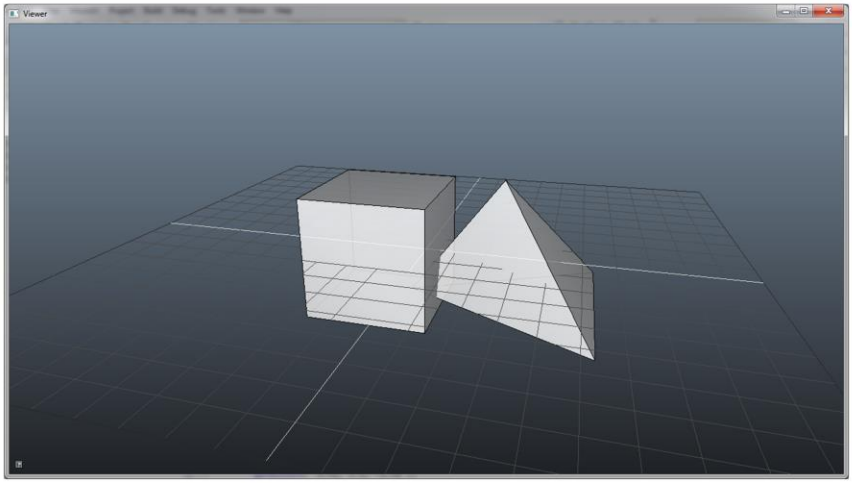
- Let's write a function that tests all possible separating axes of a polygon
- Iterate edges of A and build plane
- Find support point in opposite normal direction
- The searched distance is then simply the distance of the support point to that plane
- This is a signed distance and if it is negative it is actually a penetration
- Finally return vertex with largest distance

Example 1: Overlap Test (2D)

```
bool Overlap( const Polygon& polygonA, const Polygon& polygonB )
{
   Query queryA = QueryFaceDirections( polygonA, polygonB ); // Face normals of A
   if ( queryA.m_Separation > 0.0f )
      return false;

   Query queryB = QueryFaceDirections( polygonB, polygonA ); // Face normals of B
   if ( queryB.m_Separation > 0.0f )
      return false;

   // No separating axis found, the polygons must overlap!
   return true;
};
```

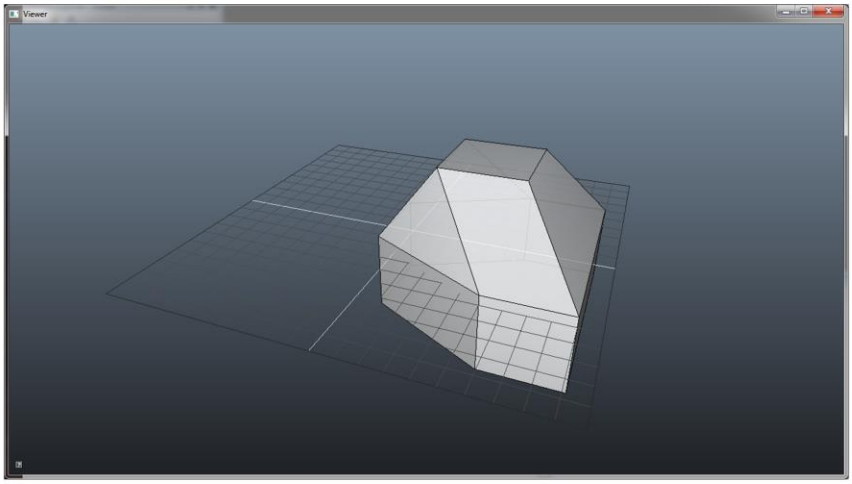
- How do we use this function?
- This a simple example which tests the overlap between convex polygons
- Possible separating axes are face normals of A and B
- Obviously need to use it twice (once for A and once for B) by simply switching arguments
- Can exit early if we find separating axis
- If no separating axis was found the polygons must overlap

Overlap between two Convex Polyhedra (3D)



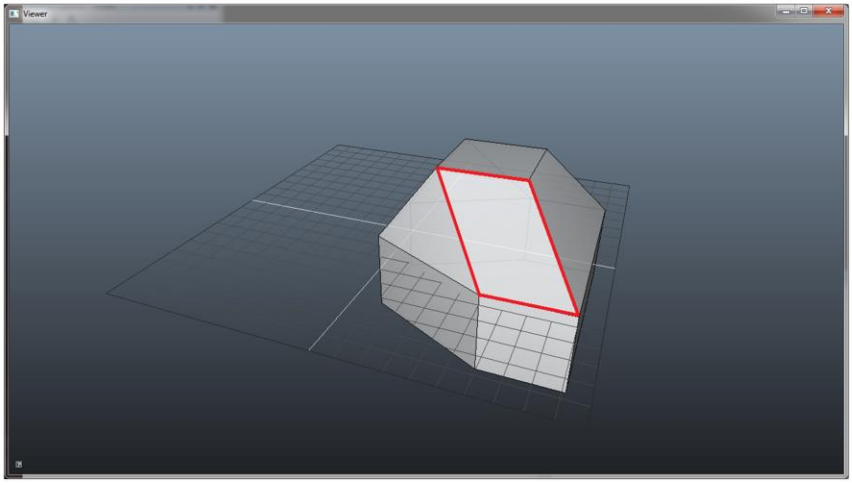
- After some preparation we arrived at actual problem we like to solve today
- As in the 2D case we start to investigate problem in Minkowski space

Minkowski Difference between two Convex Polyhedra (3D)



- In order to build the Minkowski difference between two convex polyhedra I'd like to encourage you to use you imagination from 2D
- Inflate polyhedron A by volume of the flipped polyhedron B
- B is shrunken to a point
- The Minkowski difference between two convex polyhedra is a convex polyhedron itself

Minkowski Difference: Faces (3D)



- Remember: The face normals of the Minkowski difference define the possible separating axes
- If we inspect the Minkowski difference we can detect conceptually the face planes of polyhedron A and polyhedron B – pushed out (as in 2D)
- In 3D we also identify additional faces from sweeping an edge of A along and edge of B. And the normal of such a face is the cross product between the edges
- These edge face build parallelograms

Separating Axes between Convex Polyhedra (3D)

- The possible separating axes between two convex polyhedra are:
  - The face normals of polyhedron A (2D & 3D)
  - The face normals of polyhedron B (2D & 3D)
  - **The cross product between <u>all</u> edge combinations of A and B (3D only)**

Let's summarize the possible separating axes between two convex polyhedra:

- The face normals of A
- The face normals of B
- The cross products between all edges of A and all edges of B

Brute-Force Separating Axis Test (3D)

```
EdgeQuery QueryEdgeDirection( Polyhedron* pPolyA, Polyhedron* pPolyB )
{
   for ( int index1; index1 < pPolyA->EdgeCount; ++index1 )
      Edge* pEdge1 = pPoly1->GetEdge( index1 );
      for ( int index2; index2 < pPolyB->EdgeCount; ++index2 )
         Edge* pEdge2 = pPoly2->GetEdge( index1 );
         Vector axis = Cross( pEdge1->GetDirection(), pEdge2->GetDirection() )

         Interval interval1 = pPolyA->Project( axis );
         Interval interval2 = pPolyB->Project( axis );
         float separation = Compare( interval1 , interval2 );
};
```

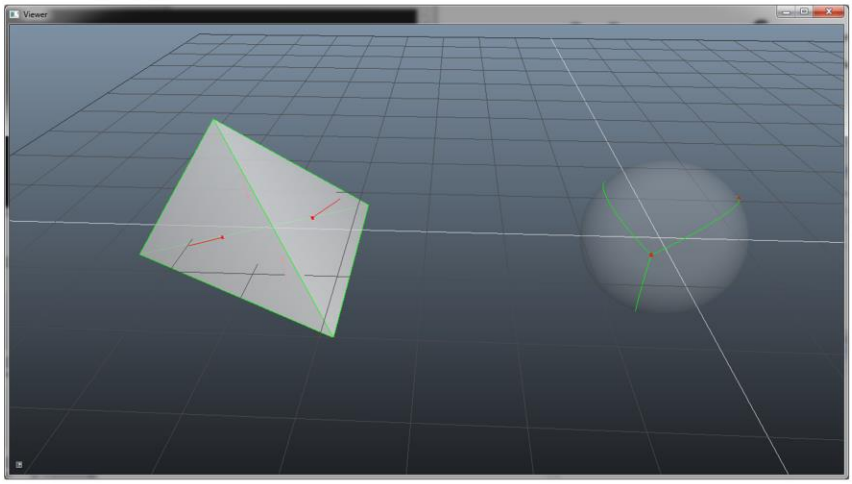
- Let's have a look at a possible brute force implementation
- O(n^2) in the number of edges
- Build cross product to get the possible separating axis
- Then we project both polyhedra onto that axis
- Need to touch all vertices to do so
- Makes the whole test conceptually O(n^3) in he complexity of the polyhedra
- Not practical -> too slow

Optimizing Edge Tests (3D)

- Define the Gauss-Map of a convex polyhedron
- Show how we can use the Gauss-Map to quickly prune edge tests
- Show how to compute the edge-edge axis separation without computing support points

- Remember that the faces on the Minkowski difference define the possible separating axes
- You might be able to imagine that not all edge pairs actually build a face on the Minkowski difference

The Gauss Map of a Convex Polyhedron

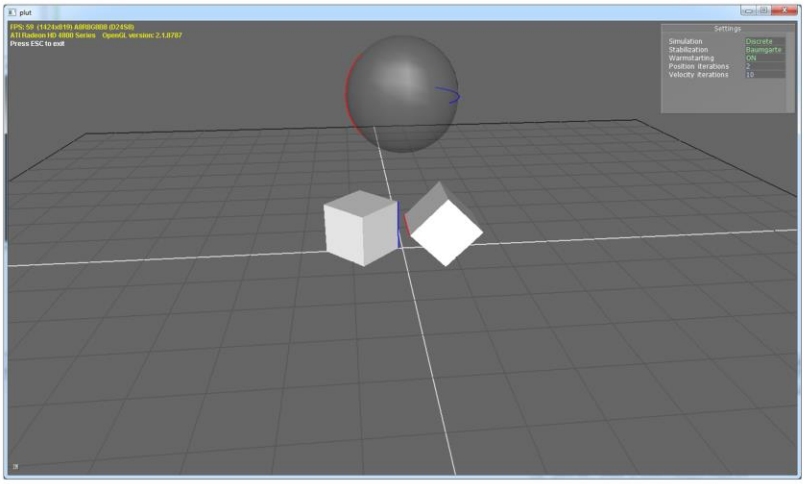


Wikipedia: The Gauss map maps the surface of a convex polyhedron onto the unit sphere:

Let's have a look how we build a Gauss Map:

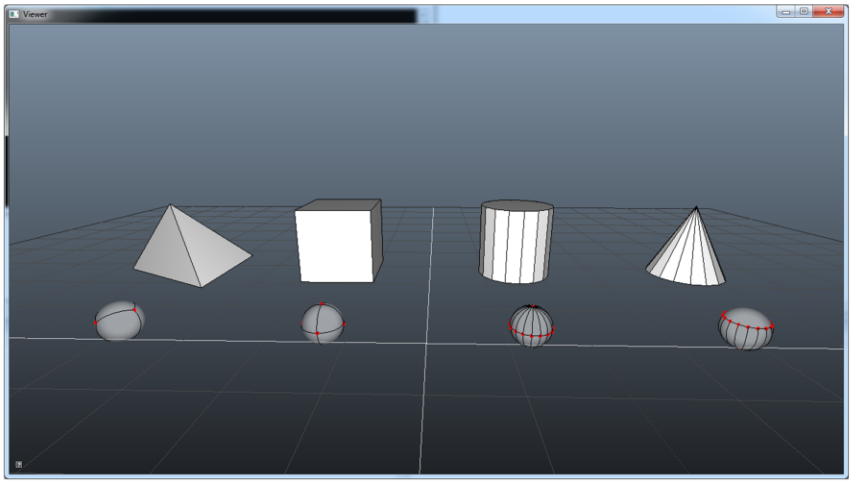
1) We make all face normals **n** of the polyhedron vertices V(f) on the unit sphere
2) For each edge adjacent to two faces we connect the associated vertices V(f1) and V(f2) of these faces to a great arc A(e)

Edges and Arcs



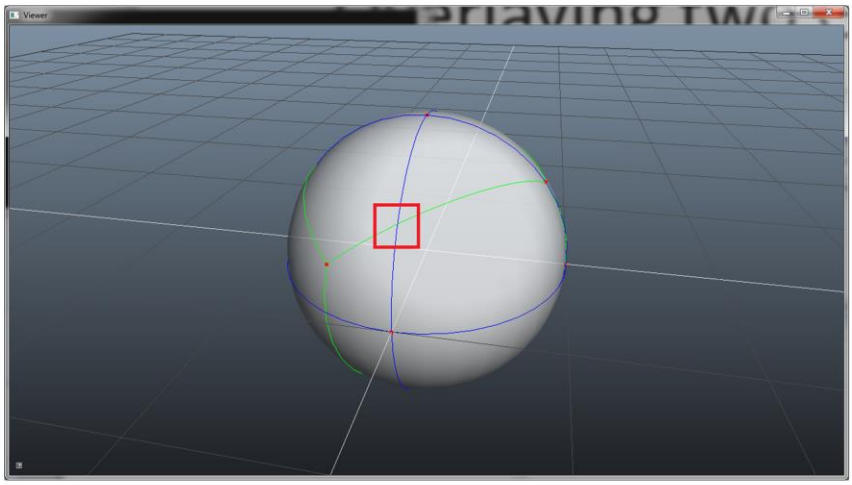
- Very abstract, but essentially two critical points here to remember:
- Faces become vertices
- Edges become great arcs

Example Gauss Maps



I compiled a bunch of Gauss Maps to get a feeling for this: Try to match some vertices and faces to edges and arcs!

Overlaying Gauss Maps



Two edges build a face on the Minkwoski difference if their two associated arcs intersect!

SLOW NOW:

- We can build the Gauss-Map of a convex polyhedron so lets overlay two of them on the unit sphere

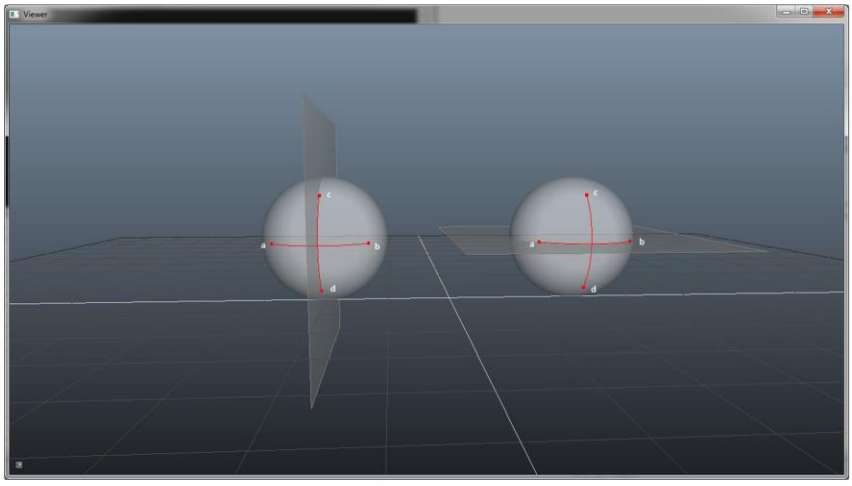
Remember that vertices on the Gauss Map and faces on the convex poyhedron correspond
Identifying the vertices of the super-positioned Gauss-Maps actually tells us faces on the Minkowski difference
And the faces on the Minkowski difference are the possible separating axes

Let's Inspect the picture and see what we find :

1) Face normals of polyhedron A
2) Face normals of polyhedron B
3) Intersections of arcs from polyhedron A with arcs of polyhedron B

This leads to a nice conclusion: Two edges only build a face on the Minkwoski difference if their two associated arcs intersect!
Important: We need actually one more step to account for Minkowski difference. I will show this some slides later.

Overlap Test



The vertices A, B and C, D of the arcs are the normals of the adjacent faces of the edges!

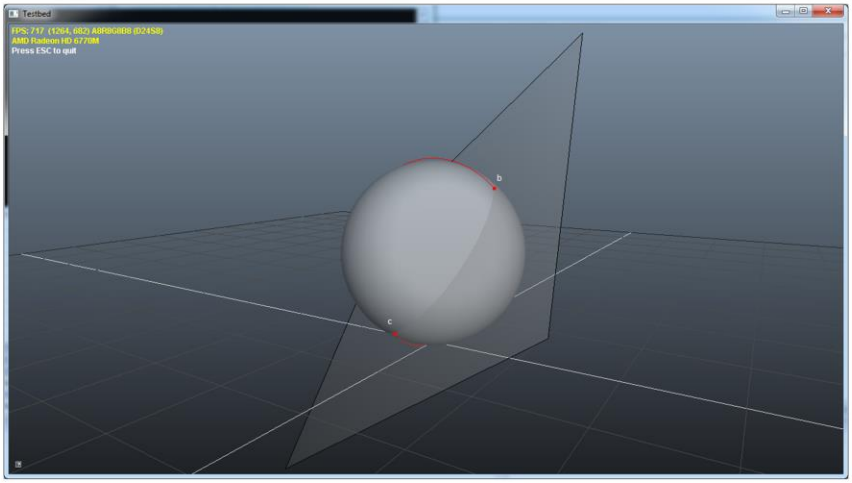
How do we detect the overlap between two arcs on a sphere"
Funnily: This is a separating axis test itself

1) The vertices A and B of arc 1 are on **different** sides of the plane through arc 2
2) The vertices C and D of arc 2 are on **different** sides of the plane through arc 1

Important:
- The vertices A and B of arc 1 are simply the normals of the faces that share edge 1
- The vertices C and D of arc 2 are simply the normals of the faces that share edge 2

So this boils down to very simple plane test, but there is one final gotcha here.

Hemisphere Test



Can fail if arcs are on different hemispheres ➔The two arcs must be in the same hemisphere.
To test for the same hemisphere we build a plane through vertex B of arc 1 and vertex C of arc 2.
If the other two vertices (A and D) are on the same side of this plane this arcs are in the same hemisphere
Note: This condition is not satisfied in this example!

Final Overlap Test: Formulas

- **Intersection tests:**
  - $[\boldsymbol{c} \cdot (\boldsymbol{b} \times \boldsymbol{a})] \cdot [\boldsymbol{d} \cdot (\boldsymbol{b} \times \boldsymbol{a})] < \mathbf{0}$
  - $[\boldsymbol{a} \cdot (\boldsymbol{d} \times \boldsymbol{c})] \cdot [\boldsymbol{b} \cdot (\boldsymbol{d} \times \boldsymbol{c})] < \mathbf{0}$
- **Hemisphere test:**
  - $[\boldsymbol{a} \cdot (\boldsymbol{c} \times \boldsymbol{b})] \cdot [\boldsymbol{d} \cdot (\boldsymbol{c} \times \boldsymbol{b})] > \mathbf{0}$

Finally some ugly math.

- Remember: A, B, C, and D are the vertices of the two arcs.
- Use simple scalar triple product for plane tests
- No need for normalization – just checking signs!

Optimizing Plane Tests

- **Identities:**
  - $\boldsymbol{a} \cdot (\boldsymbol{c} \times \boldsymbol{b}) = \boldsymbol{c} \cdot (\boldsymbol{b} \times \boldsymbol{a})$
  - $\boldsymbol{d} \cdot (\boldsymbol{c} \times \boldsymbol{b}) = \boldsymbol{b} \cdot (\boldsymbol{d} \times \boldsymbol{c})$

- **Final minimal test:**
  - $|\boldsymbol{CBA}| \cdot |\boldsymbol{DBA}| < \boldsymbol{0}$ $\boldsymbol{and}$
  - $|\boldsymbol{ADC}| \cdot |\boldsymbol{BDC}| < \boldsymbol{0}$ $\boldsymbol{and}$
  - $|\boldsymbol{CBA}| \cdot |\boldsymbol{BDC}| > \boldsymbol{0}$

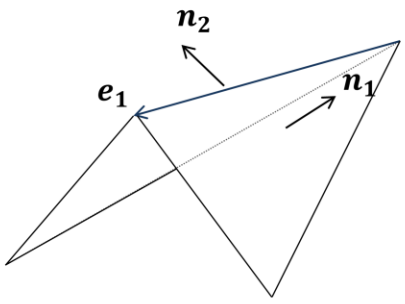
- We can optimize this a bit using the scalar triple product identity
- Reuse terms of first two intersection test in hemisphere test
- This makes the final test pretty compact

Pseudo Code: Edge Pruning

```
bool IsMinkowskiFace( Vector3 A, Vector3 B, Vector3 C, Vector3 D )
{
   // Test if arcs AB and CD intersect on the unit sphere
   Vector3 B_x_A = Cross( B, A );
   Vector3 D_x_C = Cross( D, C );

   float CBA = Dot( C, B_x_A );
   float DBA = Dot( D, B_x_A );
   float ADC = Dot( A, D_x_C );
   float BDC = Dot( B, D_x_C );

   return CBA * DBA < 0 && ADC * BDC < 0 && CBA * BDC > 0;
}
```

- We pass the vertices A and B of arc 1 and C and D of arc 2
- Remember that the vertices are simply the normals of the **adjacent faces** of both edges
- Build cross and dot products and the finally check the signs of the plane tests

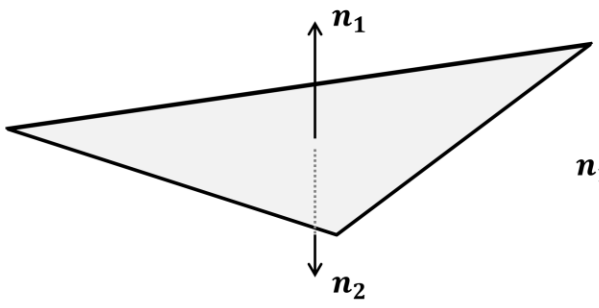
Avoiding Cross Products



$$e_1 \parallel (n_1 \times n_2)$$

- Inspect a shared edge on a convex polyhedron
- The edge has same direction as the cross product between the face normals
- We can avoid the cross products and use a simple subtraction of the edge vertices instead
- No need to normalize

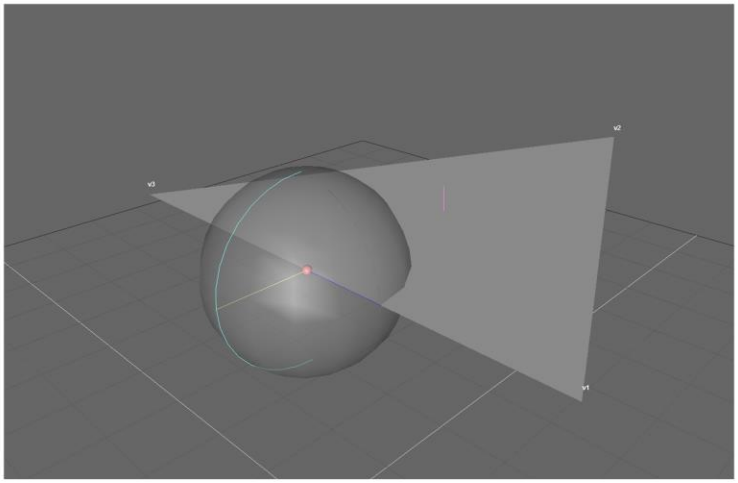
Double Sided Triangle



$$n_1 \parallel n_2 \Rightarrow n_1 \times n_2 = 0$$

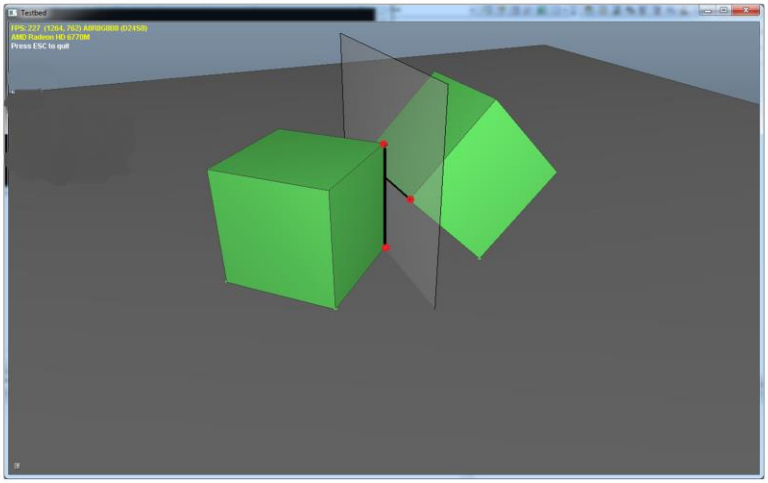
- Extreme case:
- Cross product is zero vector for double sided triangle
- This would break our plane tests
- When using edge directions this works again

Triangle Gauss Map



Here is another picture to help understanding this:
Even though the cross product is not defined in this configuration, we still can setup the plane through the associated arc of a triangle edge.
If you look at the picture you should see that the edge is orthogonal to a plane through the corresponding arc and defines the plane normal that we seek

Optimized Edge-Edge Distance



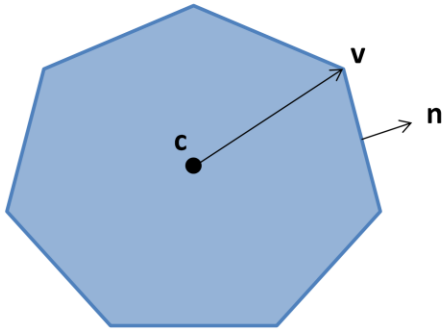
- Now that we can prune edge pairs that don't build a face on the Minkowski difference we need a method to compute the distance between the remaining edge combinations
- The Gauss Map essentially defines the directions that the corresponding feature might get in contact with
- Obviously if two arcs overlap and build a Minkowski face the colliding features must be supporting edges

The Gauss map does not only allow us to prune edge combinations. It also allows us to compute the distance between the edges efficiently.

- Build the plane through one edge (with the edge cross product as the normal)
- Compute the distance of any vertex on the other edge to that plane
- As opposed to the brute force version this is now O(1) – no need for support points!

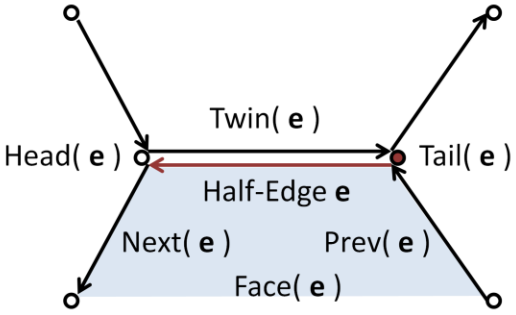
Consistent Normal Orientation

if *(dot*(**n**, **v** – **c**) < 0) **n** = **-n**



- Small gotcha here: We need a well defined normal direction to get the sign of the distance right
- E.g. from A toward B
- Simple compare the normal against the vector from centroid to any of the two edge vertices and flip if necessary

Half-Edge Mesh



So after quite some theory lets investigate one possible implementation
There are endless possibilities to describe a convex polyhedron.
Personally I use the half-edge mesh which is an edge centric mesh representation.
Allows to efficiently access the two associated faces of a shared edge.

SAT: Edge Directions (3D)

```
EdgeQuery QueryEdgeDirection( Polyhedron* pPolyA, Polyhedron* pPolyB )
{
   for ( int indexA; indexA < pPolyA->EdgeCount; indexA += 2 )
      HalfEdge* pEdgeA = pPolyA->GetEdge( indexA );
      for ( int indexB; indexB < pPolyB->EdgeCount; indexB += 2 )
         HalfEdge* pEdgeB = pPolyB->GetEdge( indexB );

         if ( BuildMinkowskiFace( pEdgeA, pEdgeB )  )
            float separation = Distance( pEdgeA, pEdgeB, pPolyA );
            // Keep track of largest separation and associated edges

   return indices of edge pair with minimum separation and distance
};
```

- Finally let's look at optimized for version of the SAT using the Gauss Map
- We iterate over all edge combination and discard all combinations that don't build a face on the Minkowski difference
- If we detect an edge combination that realizes a potential separating axis we can now compute the distance between these two edges in constant time
- Remember that this was the bottleneck in the brute force version

- **Hint**: If you use a half-edge data structure store two half edges (edge and twin) after each other. This allows us to iterate **unique** edges efficiently! In the slide every second edge is skipped for this reason!

SAT: Edge Pruning (3D)

```
bool BuildMinkowskiFace( HalfEdge* pEdgeA, HalfEdge* pEdgeB )
{
  // Extract face normals which define the vertices of the two arcs!
   Vector3 a = pEdgeA->GetNormal1();
   Vector3 b = pEdgeA->GetNormal2();
   Vector3 c = pEdgeB->GetNormal1();
   Vector3 d = pEdgeB->GetNormal2();

   // Negate normals c and d to account for Minkowski difference!
   return IsMinkowskiFace( a, b, -c, -d );
};
```

We extract the associated normals of each edge and call the IsMinkowskiFace() function.
In order to account for the Minkowski difference we need to negate one pair of normals.

SAT: Edge-Edge Distance (3D)

```
float Distance( HalfEdge* pEdgeA, HalfEdge* pEdgeB, Polyhedron* pPolyA )
{
   Vector3 edgeA = pEdgeA->Direction();
   Vector3 pointA = pEdgeA->Head();
   Vector3 edgeB = pEdgeB-> Direction();
   Vector3 pointB = pEdgeB->Head();
   if ( AreEdgesParallel( edgeA , edgeB ) ) return –FLT_MAX;   // Skip parallel edges

   Vector3 normal = NormalizedCross( edgeA , edgeB );
   if ( Dot( normal, pointA - pPolyA->Centroid() );
      normal = -normal; // Assure normal points from A -> B

   return Dot( normal, pointB – pointA ); // No need to compute support points: O(1)
};
```

- Finally we have two edges that define a possible separating
- Skip parallel edges, because they can also not build a face on the Minkowski difference
- Assure consistent normal orientation to get a correct sign for distance calculation
- Finally build plane through edge A and compute distance of a vertex on edge B to that plane

Overlap Test (3D)

```
bool Overlap( Polyhedron* pPolyA, Polyhedron* pPolyB )
{
   FaceQuery faceQueryA = QueryFaceDirections( pPolyA, pPolyB  ); // Faces A (as 2D)
   if ( faceQueryA > 0.0f ) return false;

   FaceQuery faceQueryB = QueryFaceDirections( pPolyB , pPolyA ); // Faces B (as 2D)
   if ( faceQueryB > 0.0f ) return false;

   EdgeQuery edgeQuery = QueryEdgeDirections( pPolyA, pPolyB ); // Edges A & B
   if ( edgeQuery > 0.0f ) return false;

   return true;
};
```

- As we did in 2D a simple example of the overlap test in 3D
- Face directions same as in 2D
- Also need to test edge combinations in 3D – otherwise test is not complete which might result in false positives since we would skip possible separating axes

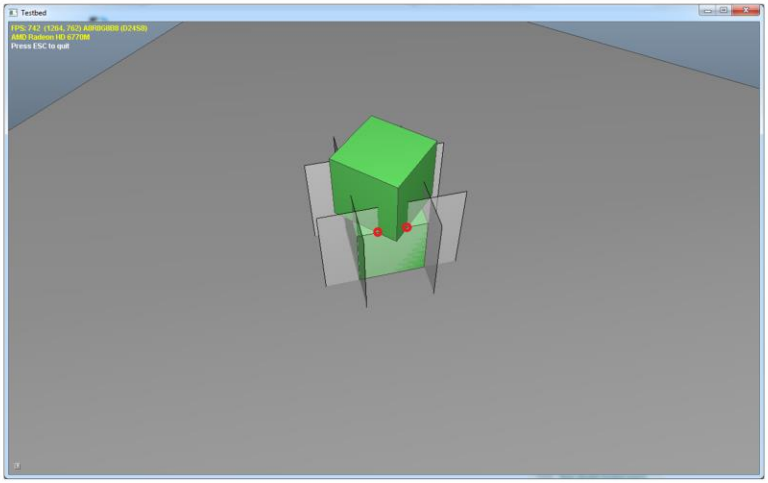
SAT Comparison: Brute-Force vs. Gauss Map

- Convex hulls of random points on sphere:
  - 4 Vertices: 2x speed-up
  - 8 Vertices: 6x speed-up
  - 16 Vertices: 20x speed-up
  - 32 Vertices: 40x speed-up
- No extra memory needed

Speed-up is essentially what you would expect when optimizing an algorithm from O(n^3) to O(n^2)
This makes this test actually a candidate for collision detection between small convex hulls

This algorithm is heavily used in production code and not just theory!

Face Contact (3D)



- Since this is the physics tutorial I'd also like to talk about contact creation using the SAT:

- Create a contact between the features that define the axis of minimum penetration.
- If axis of minimum penetration comes from face
- Clip one face against the side planes of the other.

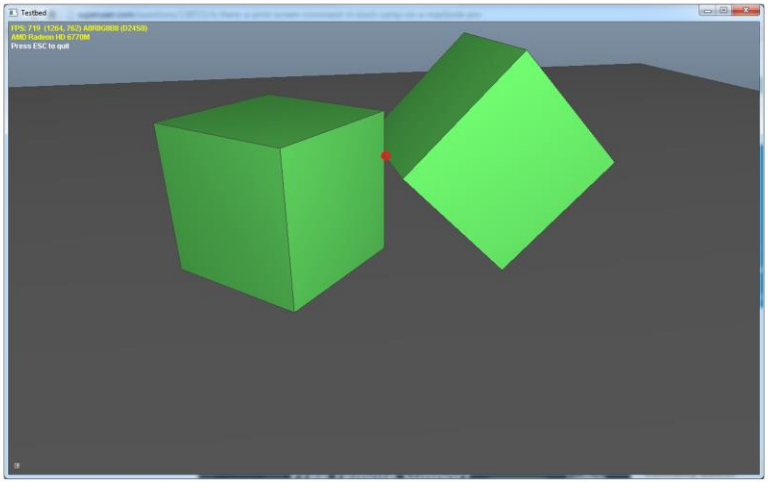
Face Contact (3D) - Overview

- Identify axis of minimum penetration using the SAT (this defines the reference face)
- Find the most anti-parallel face the on other shape (this defines the incident face)
- Clip incident face against side planes of reference face
- Keep all vertices below reference face

Let's see how this works in a little bit more detail:

- I recommend Sutherland-Hodgman clipping in the third step.
- Some engines also clip against the reference face in the end, but I cannot recommend this step. This adds additional contact points that don't add to the manifold stability

Edge Contact



If the axis of minimum penetration is realized by an edge pair compute the closest points between the two edge segments and are done
So you see that the SAT makes contact point creation pretty straight forward

Thanks!

- Thanks to Valve!
- Thanks to Ted, Sergiy, Ali, Oliver, Erwin and especially Anoush for exhaustive rehearsing!
- Special thanks to Erin Catto for sharing these ideas on the Bullet forum!

Thinks closes the talk.

Questions?

This closes the talk and hopefully there is some time left for questions! Thank you very much…

References

- C. Ericson: “Real-Time Collision Detection”
- G. v. d. Bergen: “Collision Detection in Interactive 3D Environments”
- S. Migdalsky: “SAT in Narrow Phase and Contact Manifold Construction” (in Game Physics Pearls)
- Y. Choi: “Collision Detection of Convex Polyhedra Based on Duality Transformation” (Minkowski Sum and Gauss Map)
- S. Iserloh: “Reframing the Problem” (Minkowski Sum)
- G. Rhodes: “Computational Geometry” (Half-Edge Mesh)
- M. McGuire: “The Half-Edge Data Structure” (FlipCode)
- E. Catto: “Contact Manifolds” (GDC 2007)

Here are some papers that might help you to better understand this presentation.