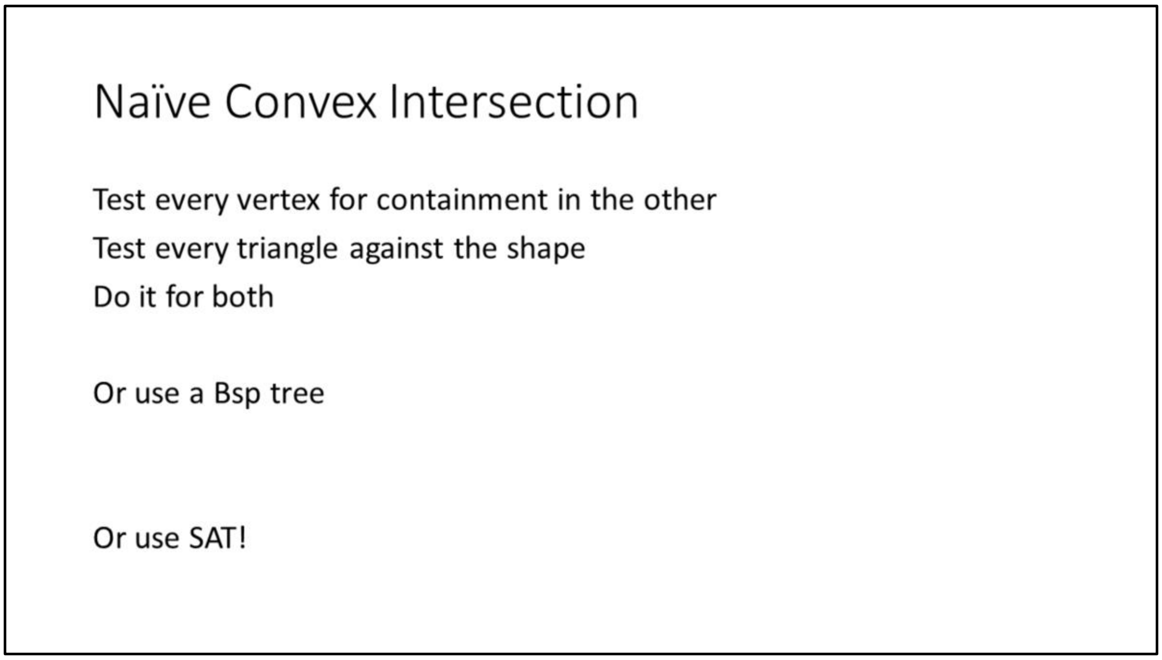
SAT

jodavis42@gmail.com



Naïve Convex Intersection

Test every vertex for containment in the other

Test every triangle against the shape

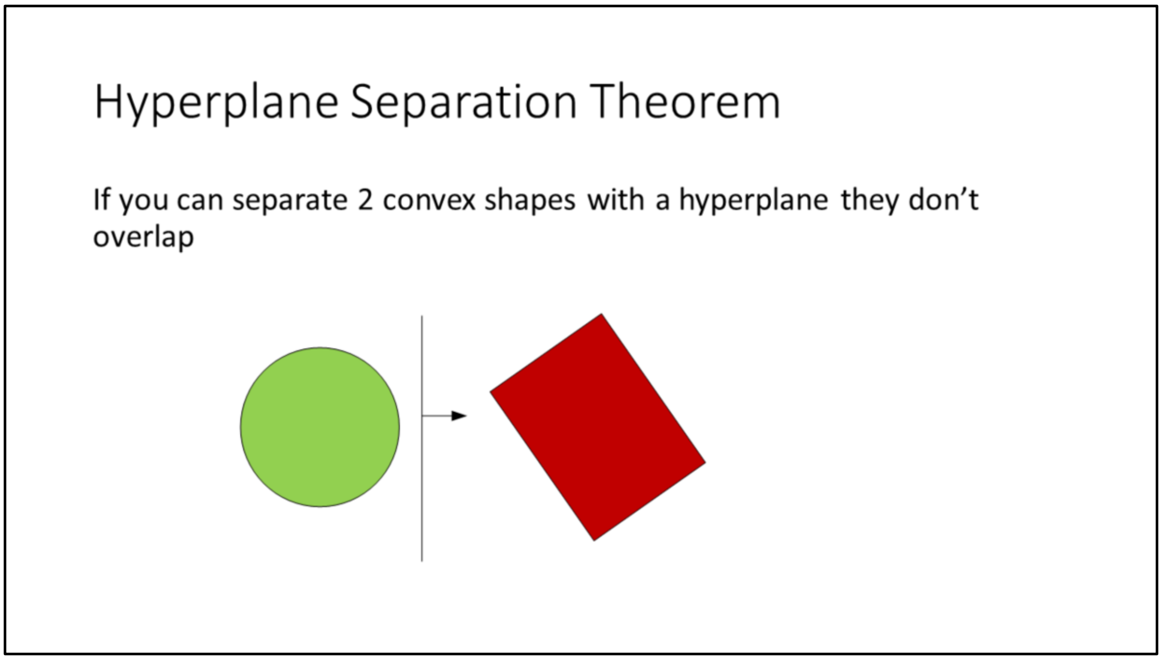
Do it for both

Or use a Bsp tree

Or use SAT!

If we want to determine if two convex meshes are intersecting the naïve approach would be to test all vertices in mesh A for containment in mesh B. Likewise we'd have to test all of B for containment in A. Even this isn't enough though as it won't catch cases of edge intersection, hence we have to check each triangle in A against each triangle in B. One method to make this better would be to use bsp-trees as discussed before, but there are much better ways.

Convex shapes in particular can be optimized quite extensively for collision detection. The simplest and easiest to understand algorithm for this is known as the Separating Axis Theorem or SAT.

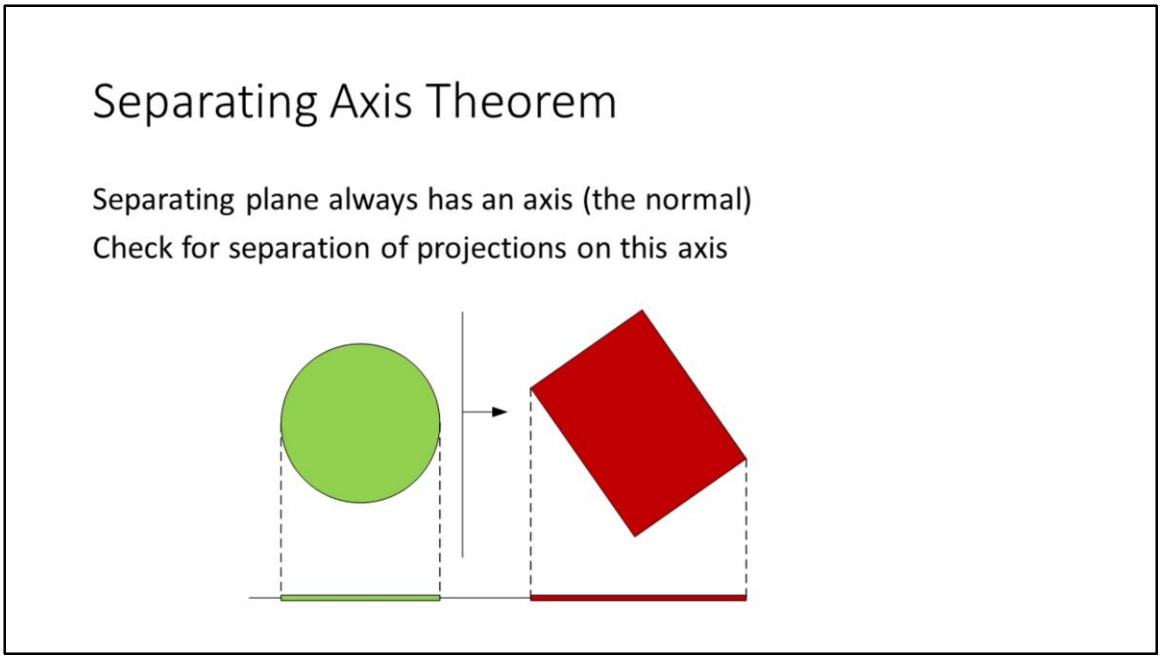


Hyperplane Separation Theorem

If you can separate 2 convex shapes with a hyperplane they don't overlap

SAT comes from the Hyperplane Separation Theorem which roughly states that two convex objects are separating if a hyperplane can be drawn between them. Once again, a hyperplane is just a plane of dimensionality d-1, so in 3d it's just a regular plane and in 2d it's a line. If we can draw a line between two convex shapes that doesn't intersect the two objects then we know they don't collide.

We could perform this test in 3d by classifying all points on each shape and if their classifications overlap then the shapes "overlap" on this axis.

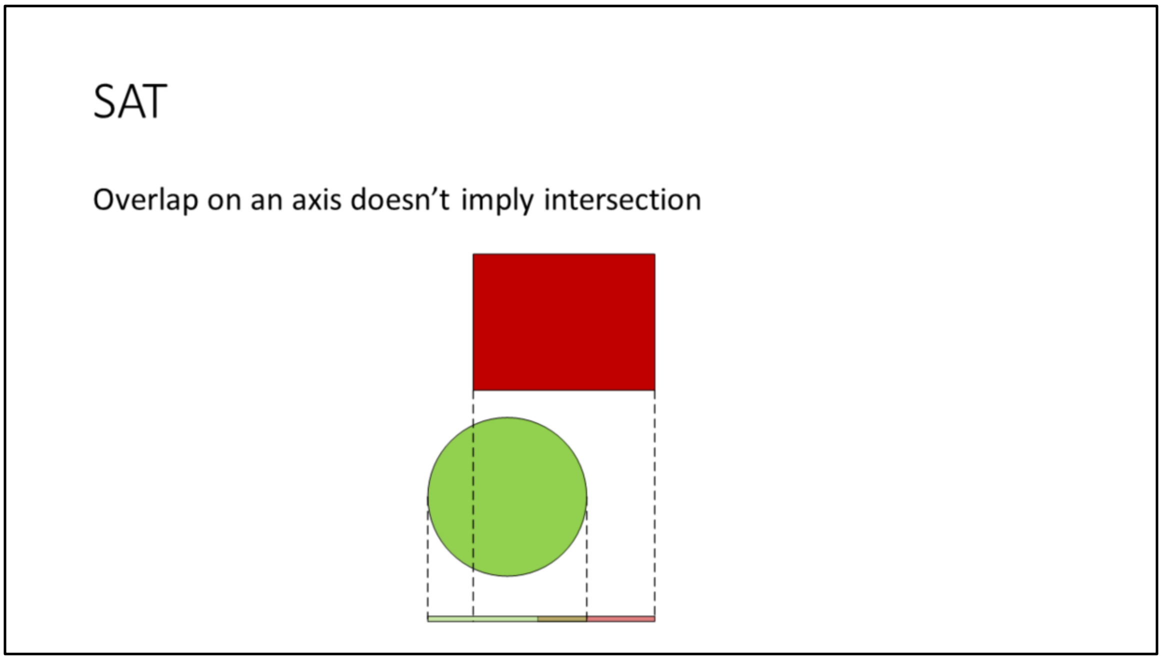


Separating Axis Theorem

Separating plane always has an axis (the normal)

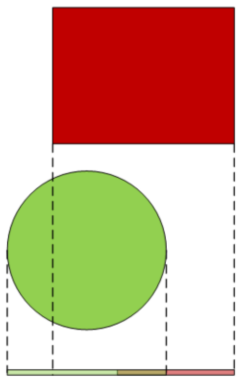
Check for separation of projections on this axis

Instead of using the hyperplane we can use an axis. No matter what dimension we're in, the separating hyperplane defines an axis (or the hyperplane's normal). SAT then states that if two shape projections onto this axis don't overlap then there is no overlap between the objects.

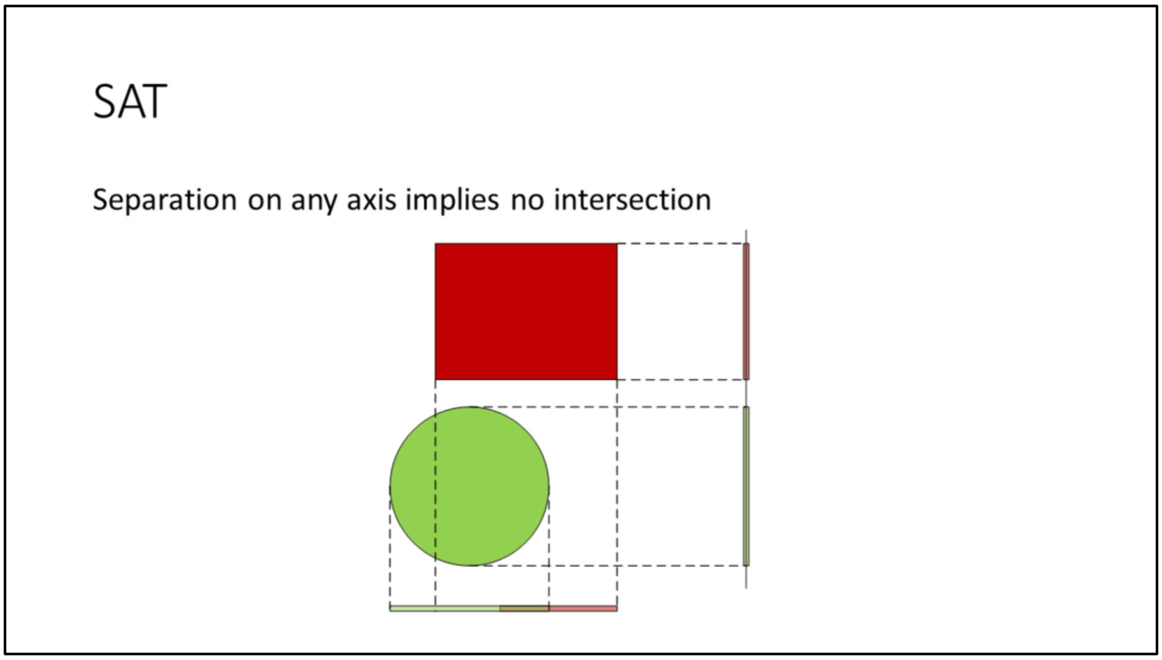


SAT

Overlap on an axis doesn't imply intersection

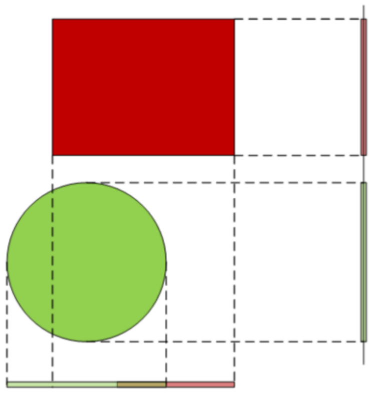


Note that if we test an axis using SAT and the projection intervals overlap it does not mean the objects overlap as is obvious from this picture.

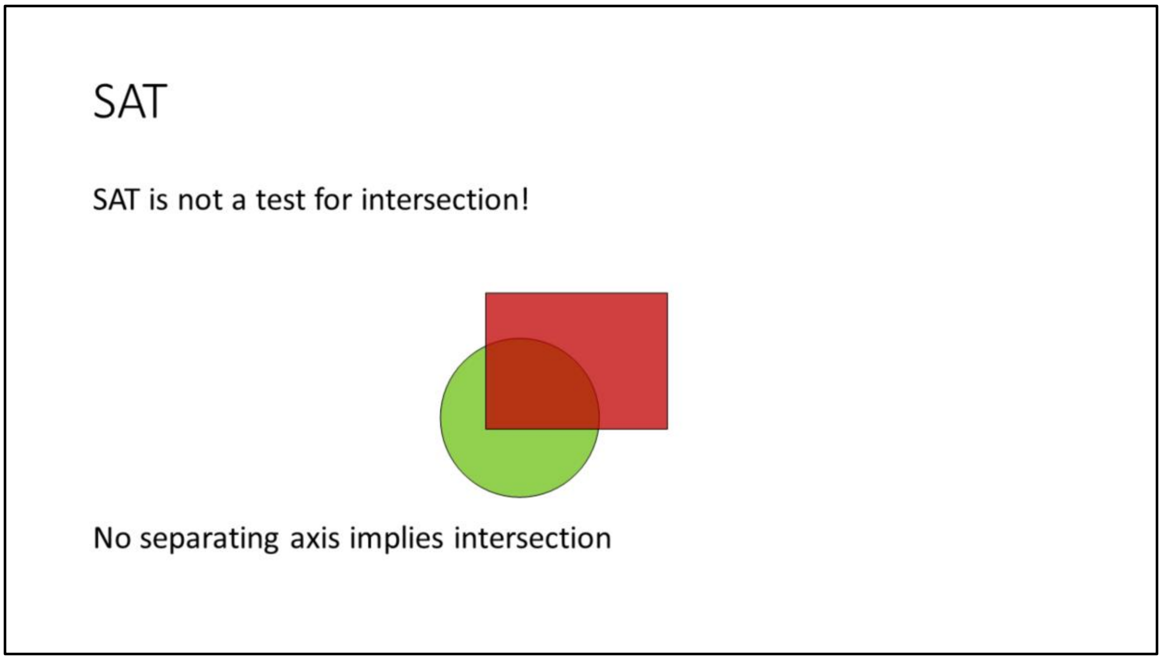


SAT

Separation on any axis implies no intersection

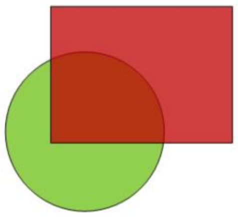


Do note though that no matter how many axes do overlap, if there is even 1 axis of separation then the objects don’t overlap.



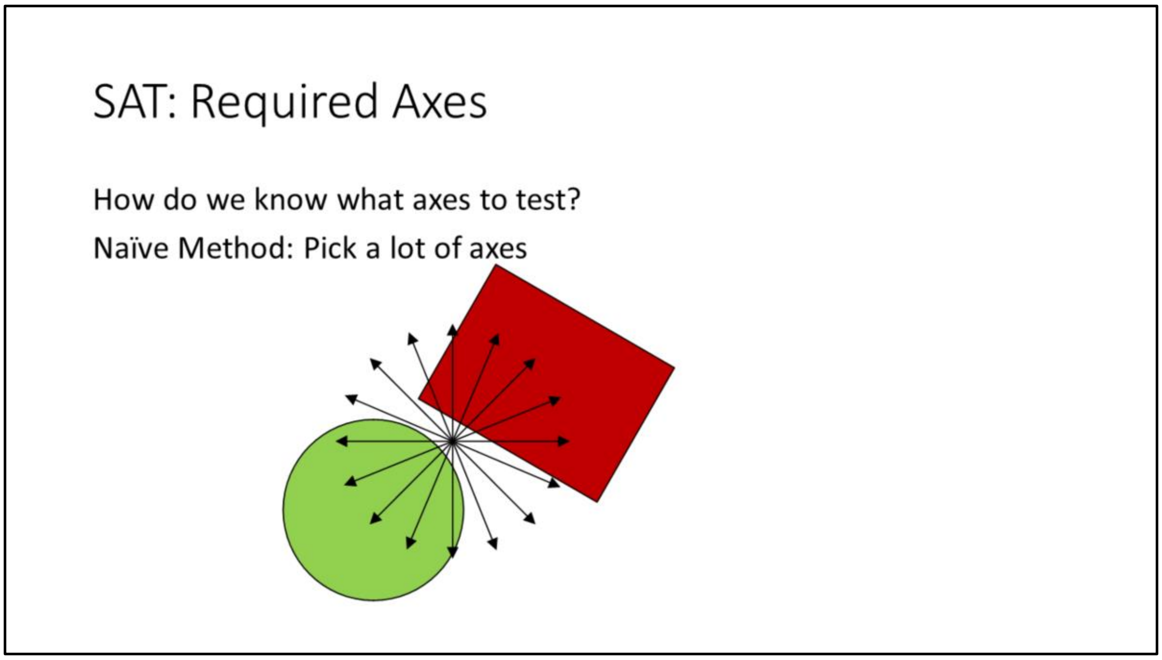
SAT

SAT is not a test for intersection!



No separating axis implies intersection

This leads to the important point that SAT is not a test for intersection. It is a test for non-intersection. Only if we can't find an axis of separation can we conclude that the objects overlap.



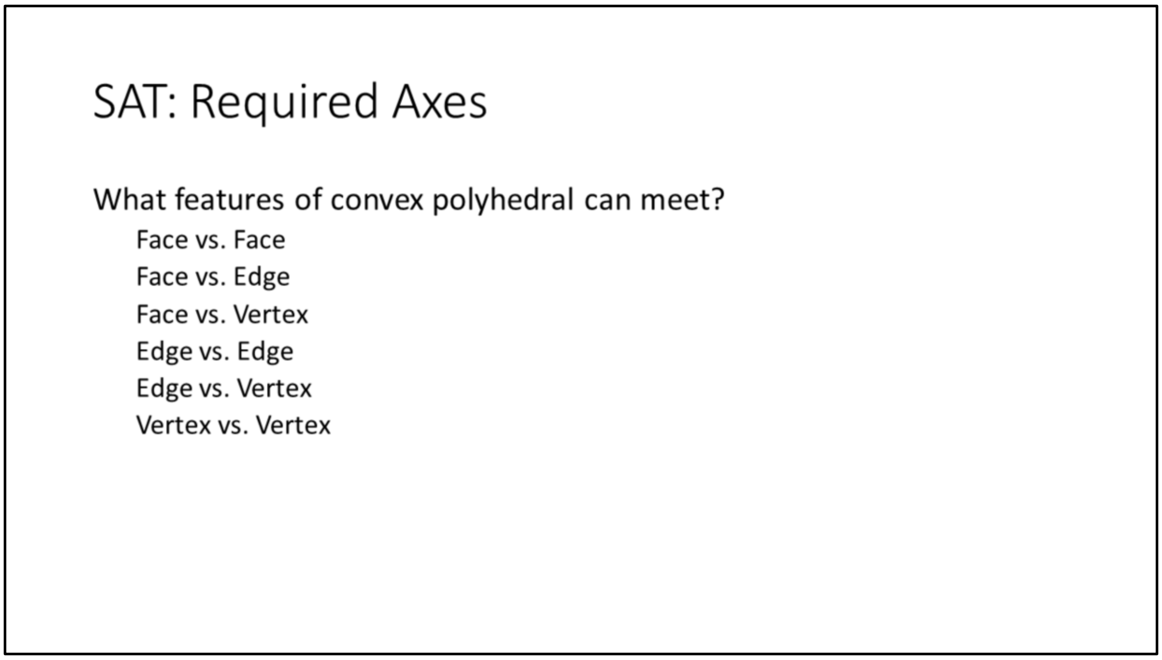
SAT: Required Axes

How do we know what axes to test?

Naïve Method: Pick a lot of axes

So SAT states if we can't find an axis of separation then the objects overlap, but how do we know what axes have to be tested? We obviously can't test every axis so how many are enough?

One method would be to spread out axes every 5 degree and test each of them. This starts to get very expensive and is not even guaranteed to find separation if it exists. There has to be some proper method of choosing what axes must be tested.

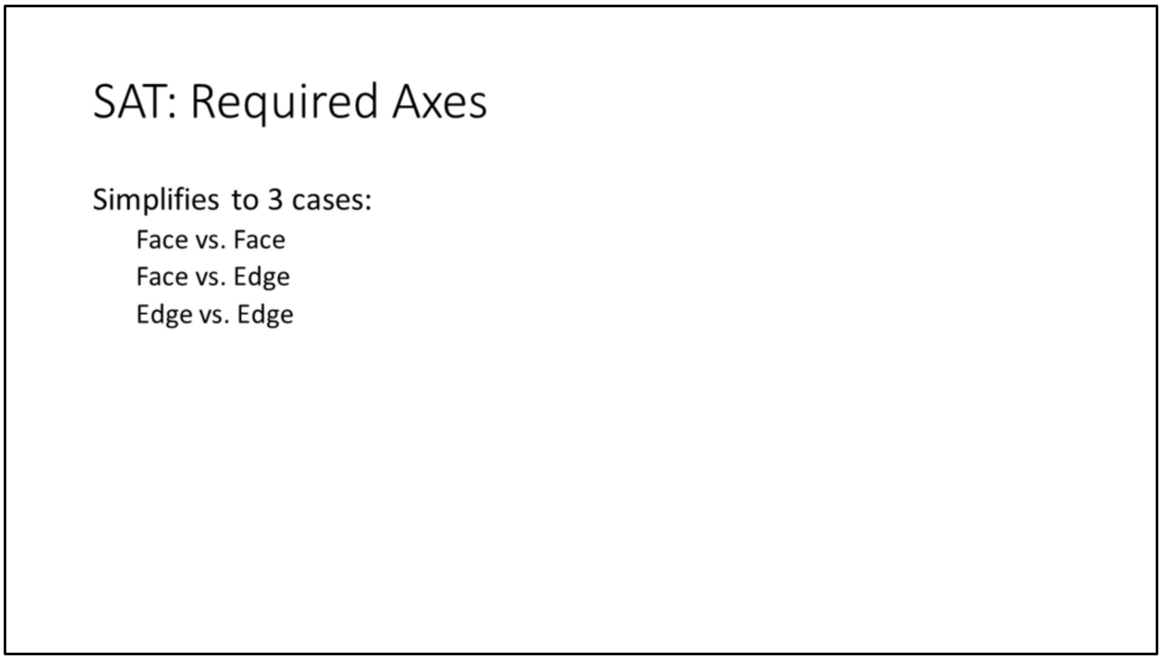


SAT: Required Axes

What features of convex polyhedral can meet?

- Face vs. Face
- Face vs. Edge
- Face vs. Vertex
- Edge vs. Edge
- Edge vs. Vertex
- Vertex vs. Vertex

To figure out what axes have to be tested we should first look at what kind of features can meet for collision between convex polyhedral. There are 6 different feature sets that can meet.

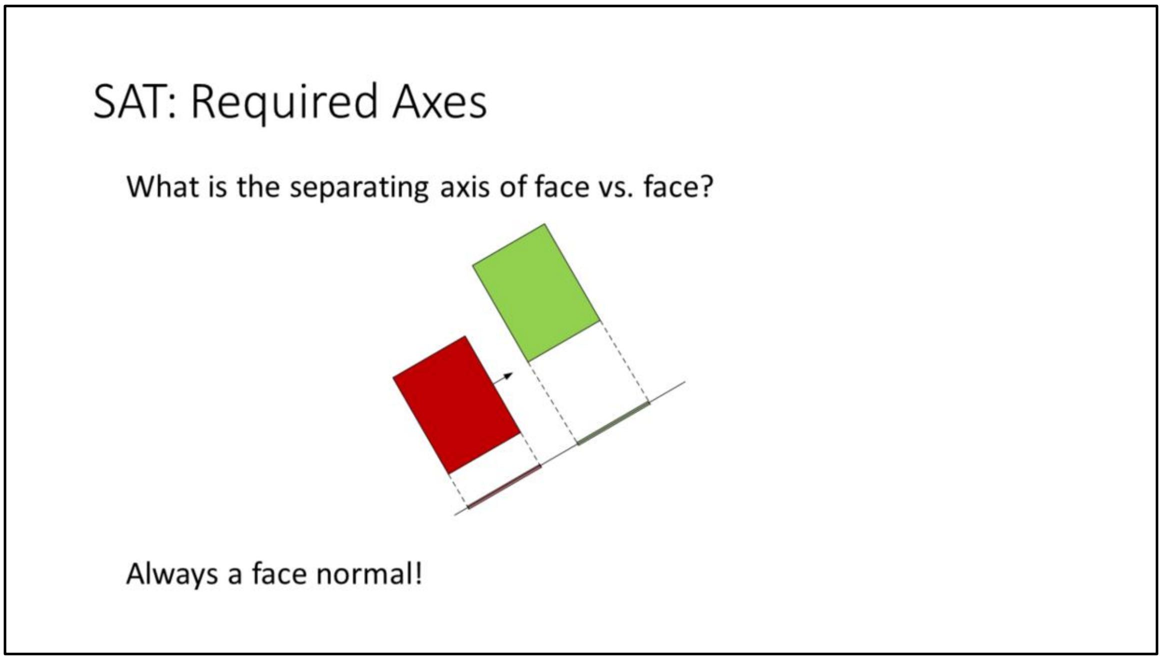


SAT: Required Axes

Simplifies to 3 cases:

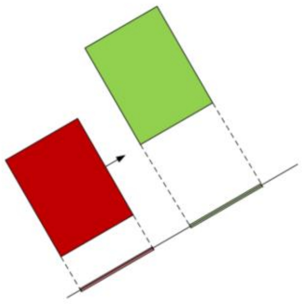
- Face vs. Face
- Face vs. Edge
- Edge vs. Edge

Several of these tests are redundant though. As vertices are part of edges, all tests that involve edges can replace the ones with vertices. This leaves us with only 3 kinds of feature pairings to consider.



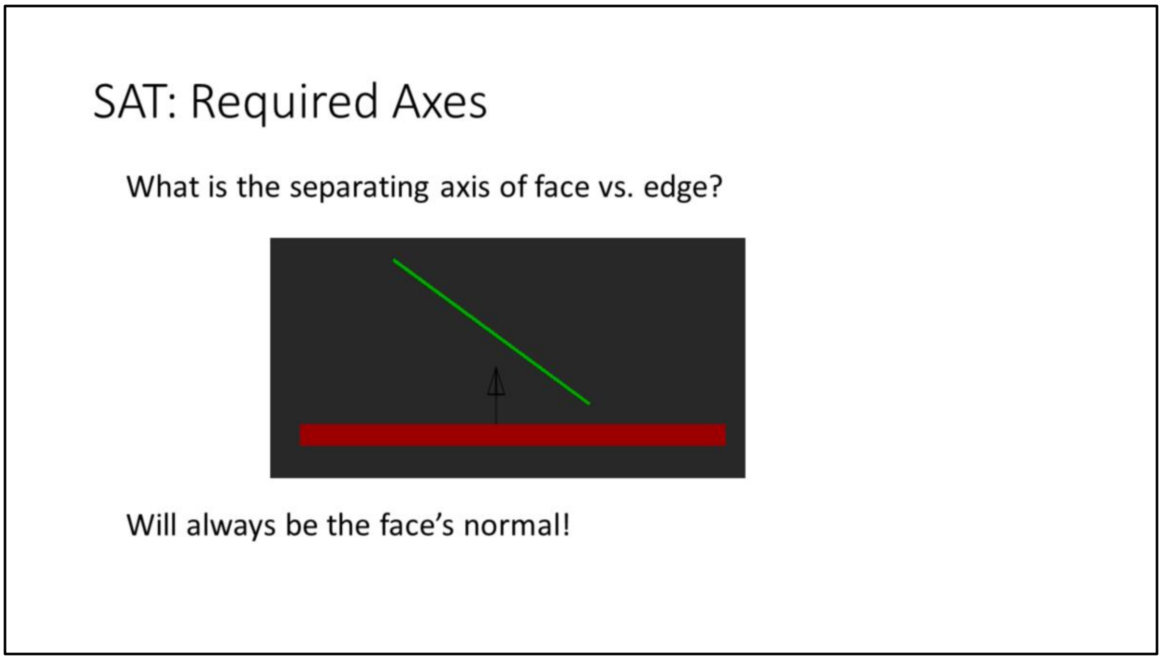
SAT: Required Axes

What is the separating axis of face vs. face?



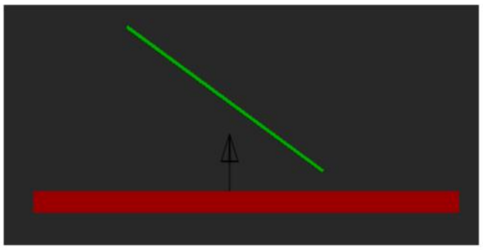
Always a face normal!

When two faces come to meet the obvious separation axis is one of their face normals.



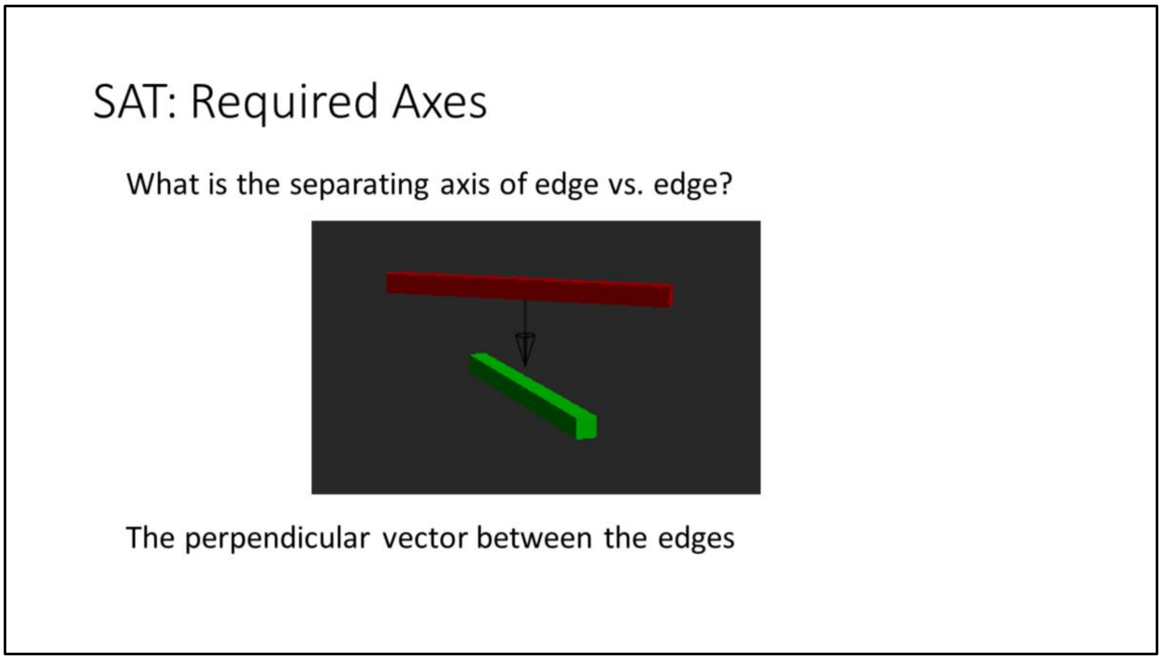
SAT: Required Axes

What is the separating axis of face vs. edge?



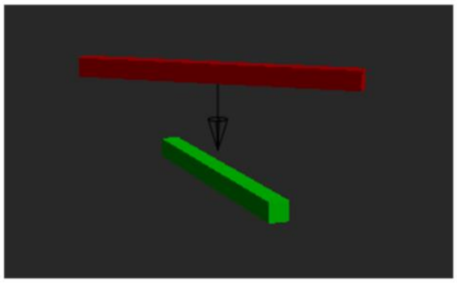
Will always be the face’s normal!

For the face vs. edge case it is also sufficient to just test the face’s normal. No matter how the features of face vs. edge meet the face’s normal will be separating them. If this is not the case then we’ve moved on to the next case.



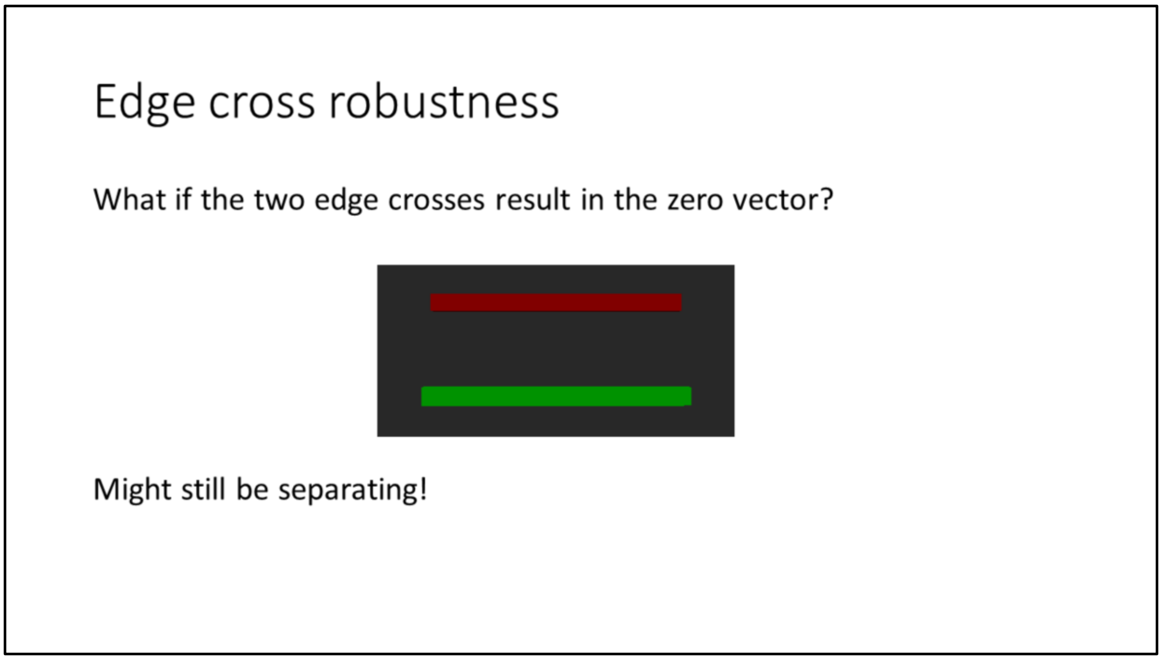
SAT: Required Axes

What is the separating axis of edge vs. edge?



The perpendicular vector between the edges

Edge vs. edge is the hardest case to visualize because it only happens in 3d. If we imaging two edges coming towards each other in 3d the axis that would always separate them (if possible) is the perpendicular vector between them. This vector can be computed from the cross product of the two edges.



Edge cross robustness

What if the two edge crosses result in the zero vector?

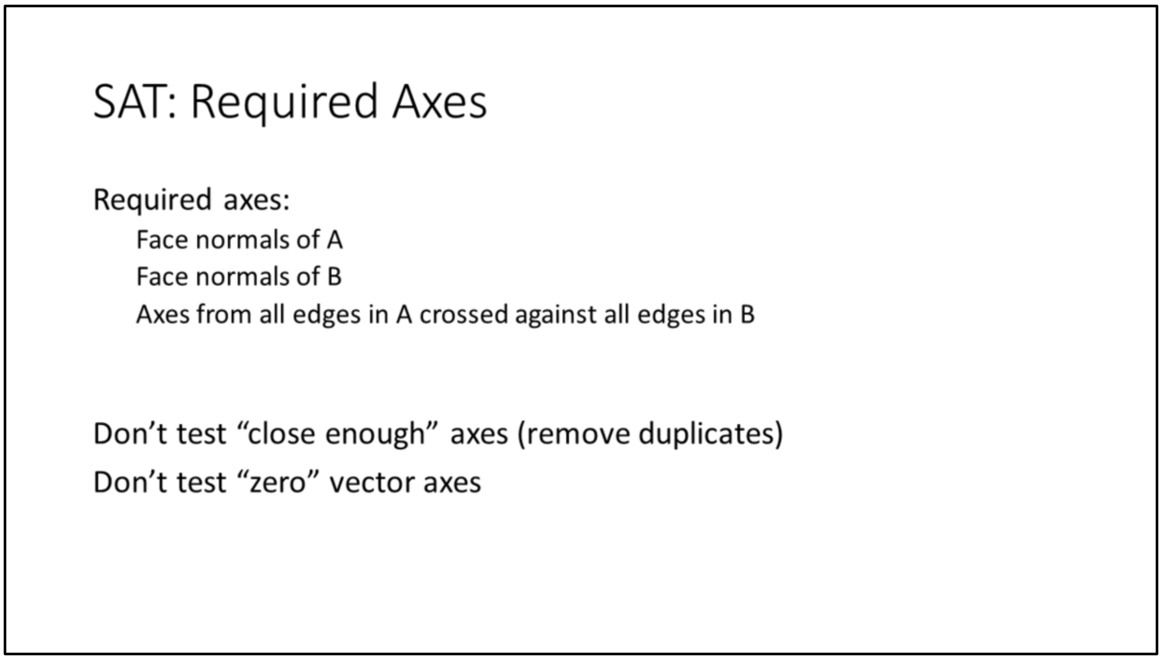
Might still be separating!

If the edge's cross product is zero does this mean there's no axis to test? We can choose to ignore the axis and hope it's not the only separating one. This is only likely to cause small overlaps to get reported as intersecting so this is a viable solution, but we do need to make sure to not test a zero axis (or one close enough to zero).

To more correctly deal with this we need to find another way to compute this axis of separation. To do this we will assume that the two lines are contained on a plane. We can compute the plane's normal by forming a triangle with one of the edges and a vertex from the other edge. The normal of this triangle should be perpendicular to both edges. We can then take the cross with the original edge and the plane's normal to produce a vector perpendicular to the two edges. That is, given the two edges represented from points AB, and CD:

$$\vec{p}_n = Cross(\vec{b} - \vec{a}, \vec{c} - \vec{d})$$
$$\vec{n} = Cross(\vec{b} - \vec{a}, \vec{p}_n)$$

If this still fails then the edges are on a line and we can safely ignore them for testing as they cannot be the separating feature.



SAT: Required Axes

Required axes:
- Face normals of A
- Face normals of B
- Axes from all edges in A crossed against all edges in B

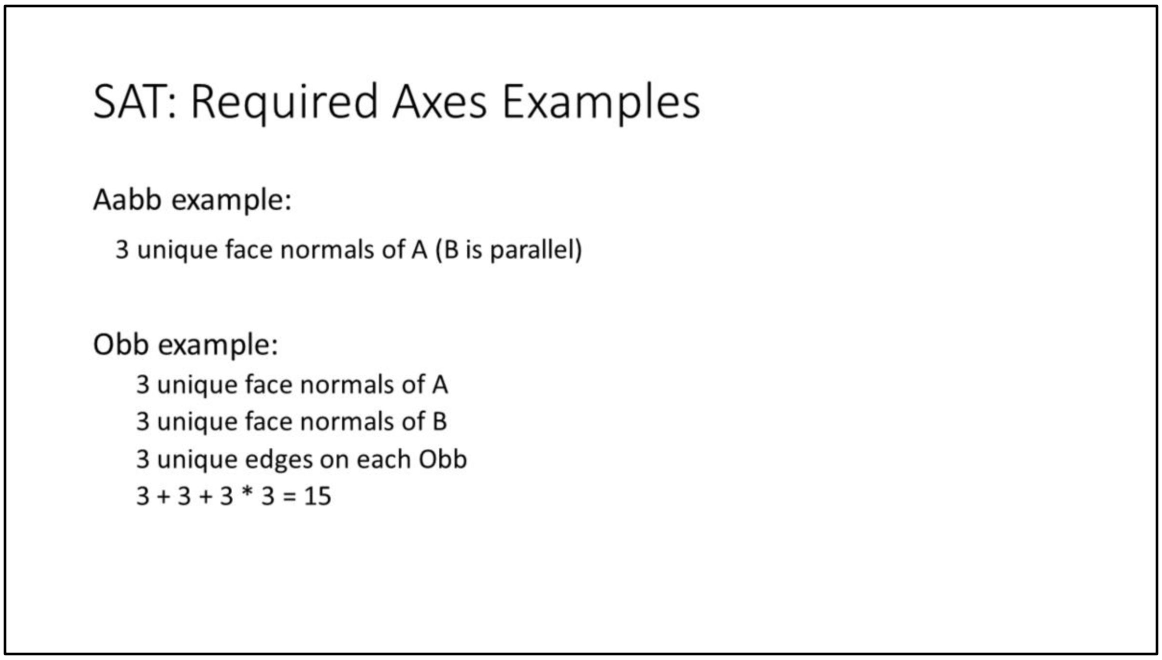
Don't test "close enough" axes (remove duplicates)

Don't test "zero" vector axes

So at the end of the day we know what axes have to be tested. To cover the face vs. face and face vs. edge cases we need to test all face normals of each shape. To deal with the edge vs. edge cases we have to test all combinations of Cross(edgeA, edgeB).

This is likely to produce many redundant axes. Any axes that are close enough to each other should be merged together. This can be done with a simple dot product on normalized edges. Likewise we can and should remove vectors that are close enough to zero. This can be tested by just checking the vector's magnitude.

What epsilons to use is a much harder question and tends to be application specific.



SAT: Required Axes Examples

Aabb example:

3 unique face normals of A (B is parallel)

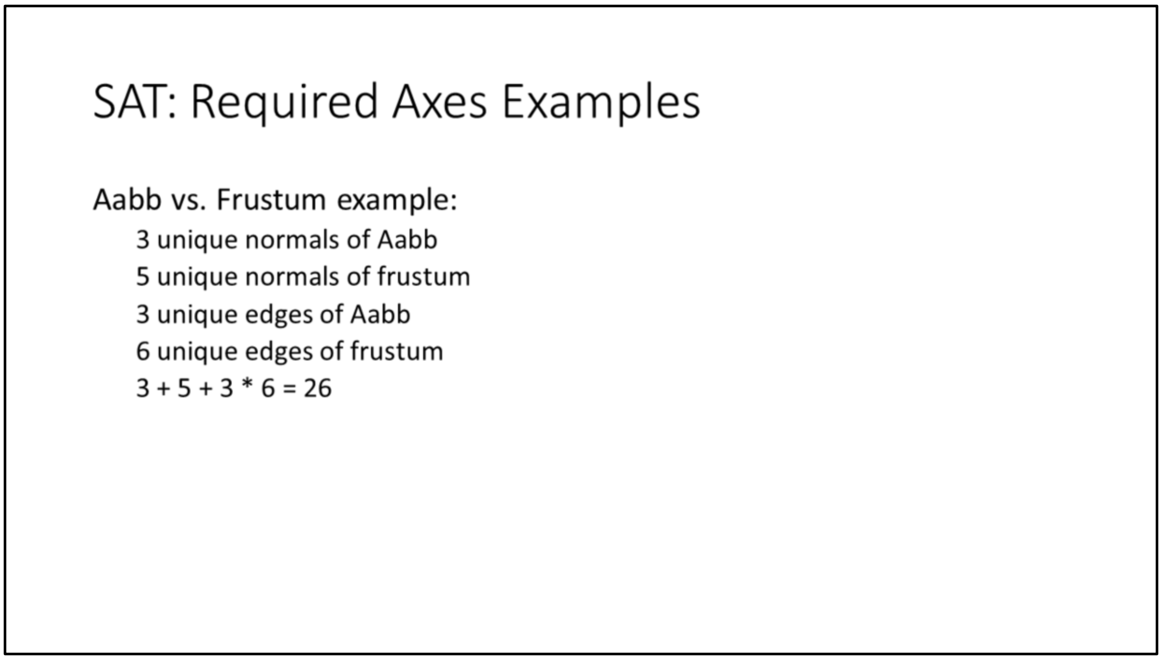
Obb example:

3 unique face normals of A
3 unique face normals of B
3 unique edges on each Obb
3 + 3 + 3 * 3 = 15

So the simple formula for how many axes we have to test is: $F_a + F_b + E_a * E_b$; where F and E are the number of faces and edges respectively for each object.

If we look at the simple intersection test of Aabb vs. Aabb from before we can see that we have 3 faces from A and B, but by definition they're parallel. Likewise every edge cross term will produce another axis that is parallel to the faces of A. Hence only 3 axes need to be tested.

A slightly more complicated example is Obb vs. Obb. In this case we have no guarantee that the faces of A and B are parallel. Each obb has 3 unique edges (the local space x, y, and z axes), however Obb A and Obb B are likely to have a different set of 3 unique edges. So in this case we get 15 total axes that have to be tested.



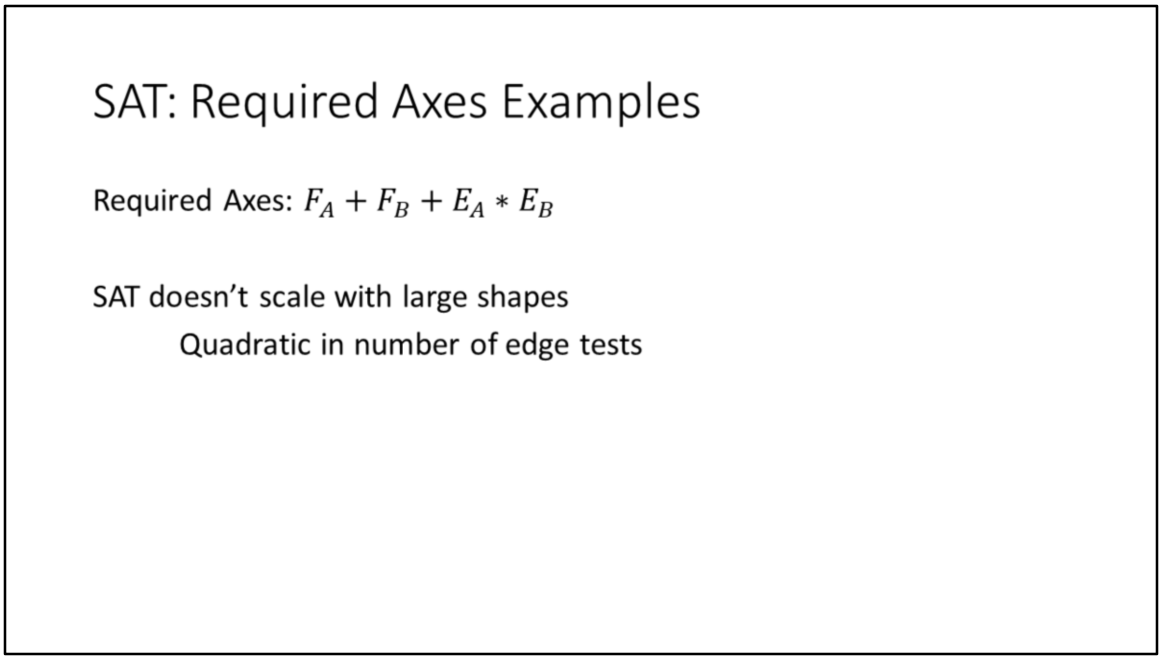
SAT: Required Axes Examples

Aabb vs. Frustum example:

- 3 unique normals of Aabb
- 5 unique normals of frustum
- 3 unique edges of Aabb
- 6 unique edges of frustum
- 3 + 5 + 3 * 6 = 26

Now for a much more fun example, and one that I glossed over before. If you remember I talked about how our Aabb vs. Frustum test was an approximation typically used for culling. The test we used wasn't sufficient to actually detect overlap, but it would often detect separation. If we use our new knowledge of required axes we can find out what exactly would have to be tested.

An aabb has 3 unique faced normal and 3 unique edges. A frustum has 5 unique faces (the near and far planes are the same). It also has 6 unique edges (x and y of the near plane and the 4 axes along the z axis connecting the near and far plane). That means we get a whopping total of 24 axes we'd have to test.



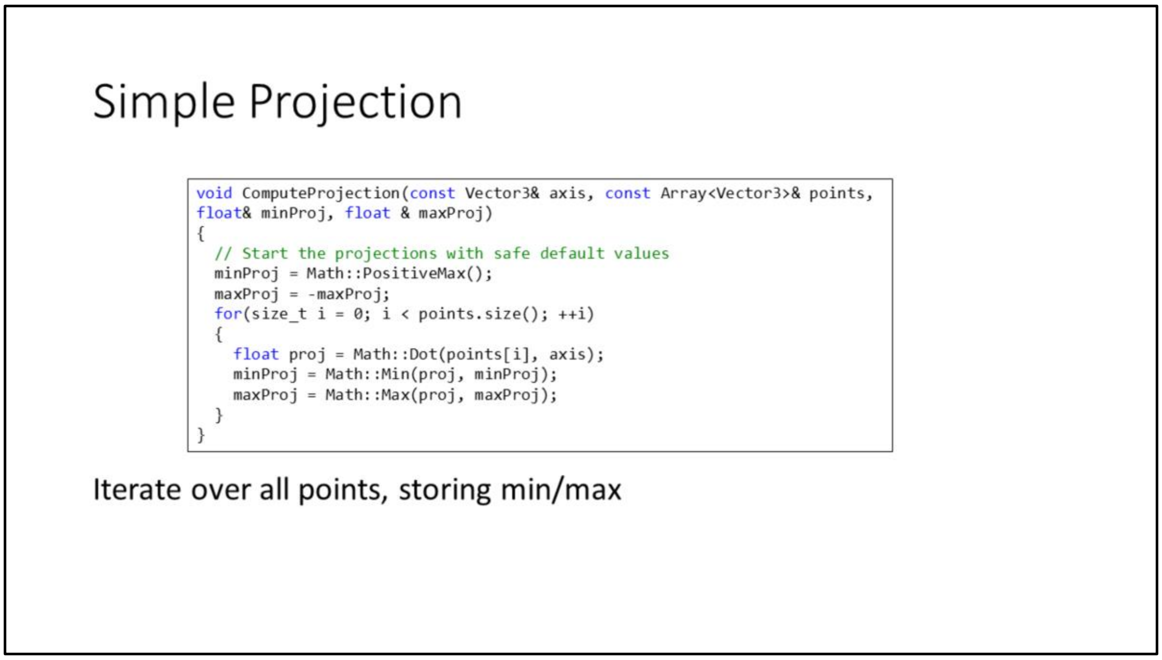
SAT: Required Axes Examples

Required Axes: $F_A + F_B + E_A * E_B$

SAT doesn't scale with large shapes

Quadratic in number of edge tests

So if we look at the equation for the required number of intersection tests it becomes clear that SAT scales quadratically with edges. So SAT isn't the most practical for large meshes in it's raw form. We'll look at some other methods of convex shape intersection that help alleviate this as well as some extra SAT tricks to minimize required tests later. It is useful to note though that SAT performs very well with a low number of faces, in particular with boxes and triangles.

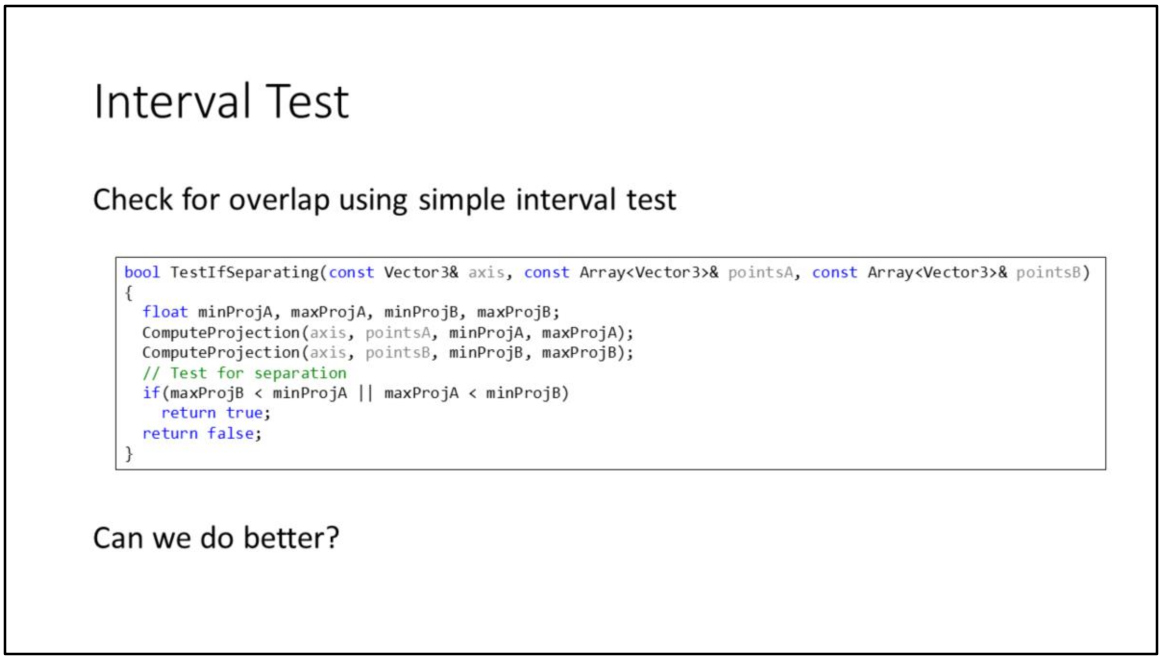


Simple Projection

```
void ComputeProjection(const Vector3& axis, const Array<Vector3>& points,
float& minProj, float & maxProj)
{
  // Start the projections with safe default values
  minProj = Math::PositiveMax();
  maxProj = -maxProj;
  for(size_t i = 0; i < points.size(); ++i)
  {
    float proj = Math::Dot(points[i], axis);
    minProj = Math::Min(proj, minProj);
    maxProj = Math::Max(proj, maxProj);
  }
}
```

Iterate over all points, storing min/max

So now that we know what axes have to be tested, how do we actually test an axis? First we have to find the projection interval on an axis for a shape. The simple method to do this is to just iterate over all points and project them on the axis, storing the min and max projection value.



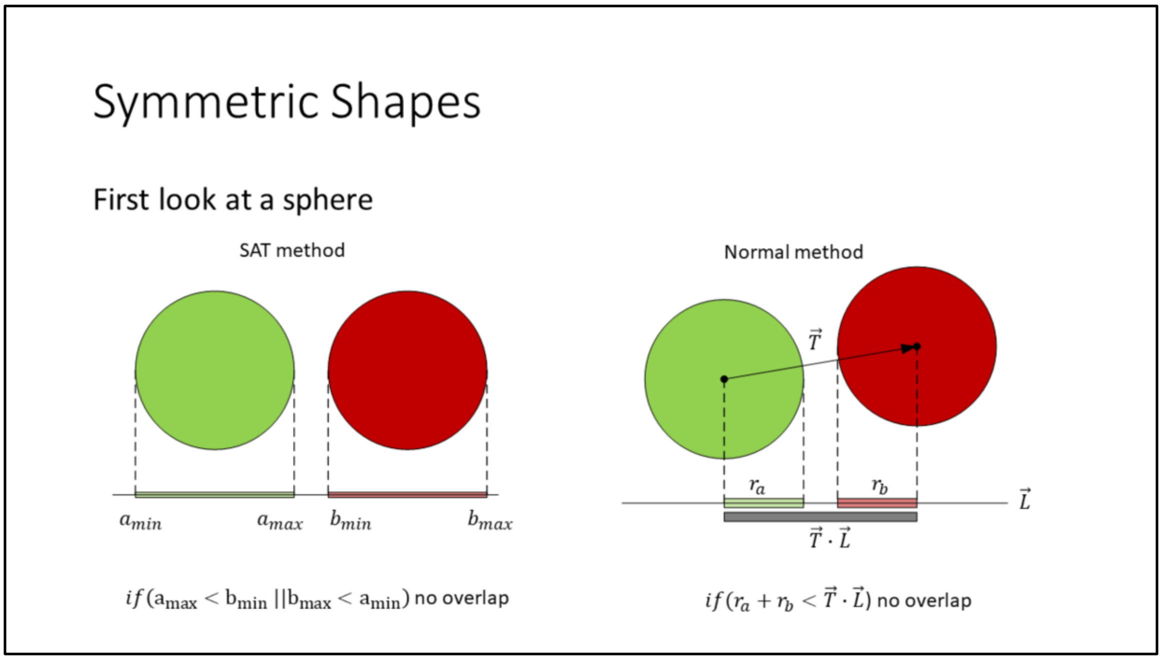
Interval Test

Check for overlap using simple interval test

```
bool TestIfSeparating(const Vector3& axis, const Array<Vector3>& pointsA, const Array<Vector3>& pointsB)
{
  float minProjA, maxProjA, minProjB, maxProjB;
  ComputeProjection(axis, pointsA, minProjA, maxProjA);
  ComputeProjection(axis, pointsB, minProjB, maxProjB);
  // Test for separation
  if(maxProjB < minProjA || maxProjA < minProjB)
    return true;
  return false;
}
```

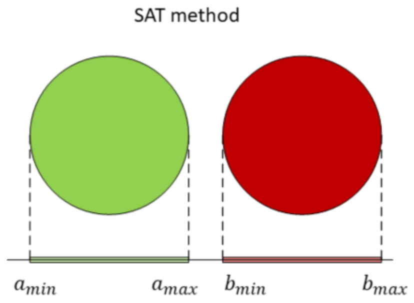
Can we do better?

We can then do the same separation check we did for an aabb on this axis to see if the projection intervals overlap. The question is can we do better than this?

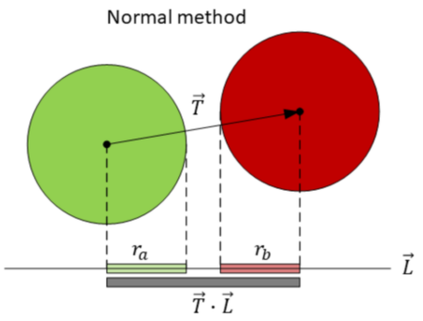


Symmetric Shapes

First look at a sphere



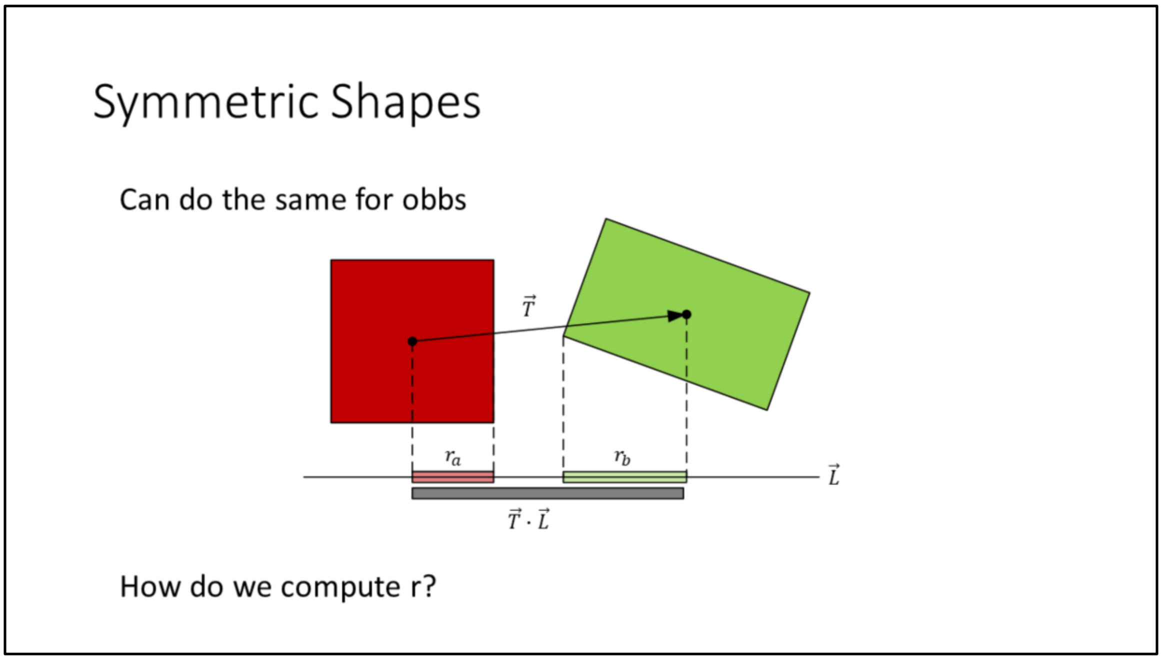
$if\,(\mathrm{a_{max}} < \mathrm{b_{min}}\ ||\mathrm{b_{max}} < \mathrm{a_{min}})$ no overlap



$if\,(r_a + r_b < \vec{T}\cdot\vec{L})$ no overlap

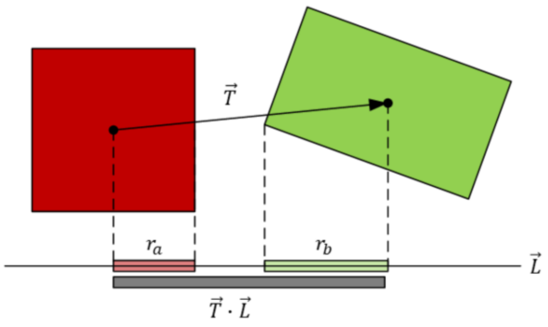
To start with lets look at a sphere. We could represent a sphere as the min and max projection on the axis and perform an overlap range, but if you recall that's not how we write there sphere vs. sphere test. Instead we check the distance between their centers and the sum of their radii.

We can do the same thing using SAT on two spheres. We can project the vector between their centers and their radii onto the test axis. We can then do a 1d test for separation with the radius'.



Symmetric Shapes

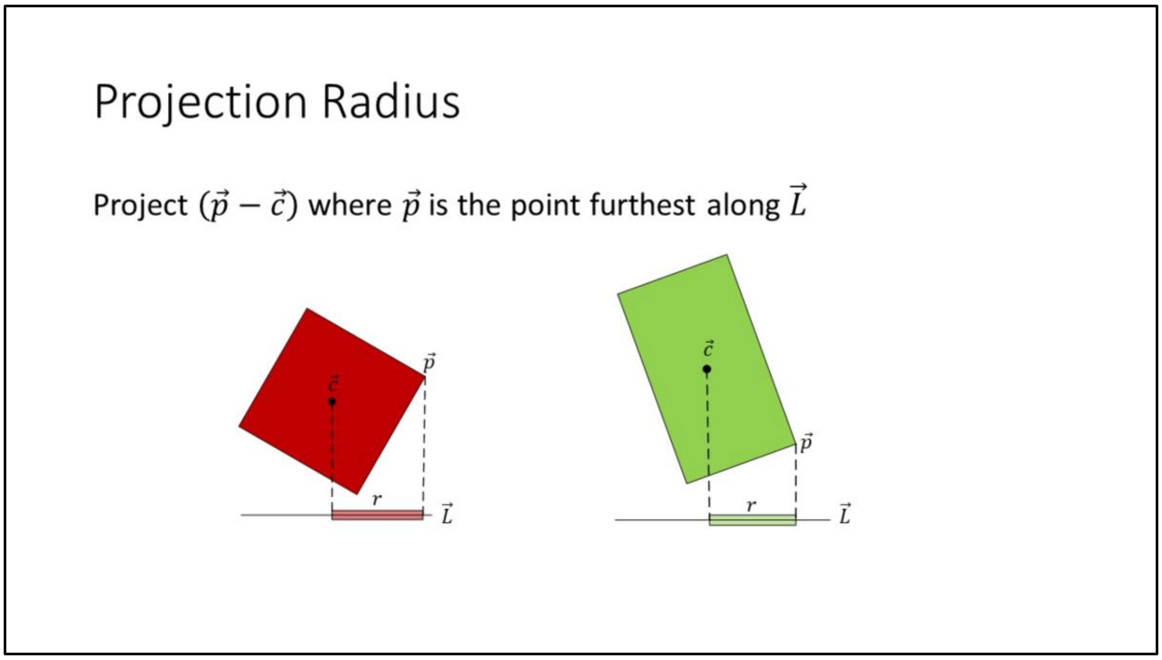
Can do the same for obbs



How do we compute r?

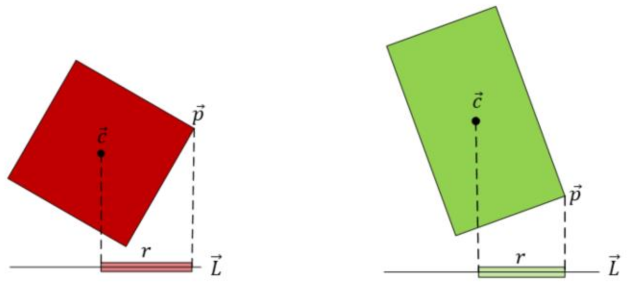
We can do the same thing with an obb. In fact, we can do this with any object that is symmetric about its center.

For any object it should be obvious how to compute $\vec{T}$ as it's just the projection of the vector between the two object's centers onto the test axis.
How do we compute the projection radius though?

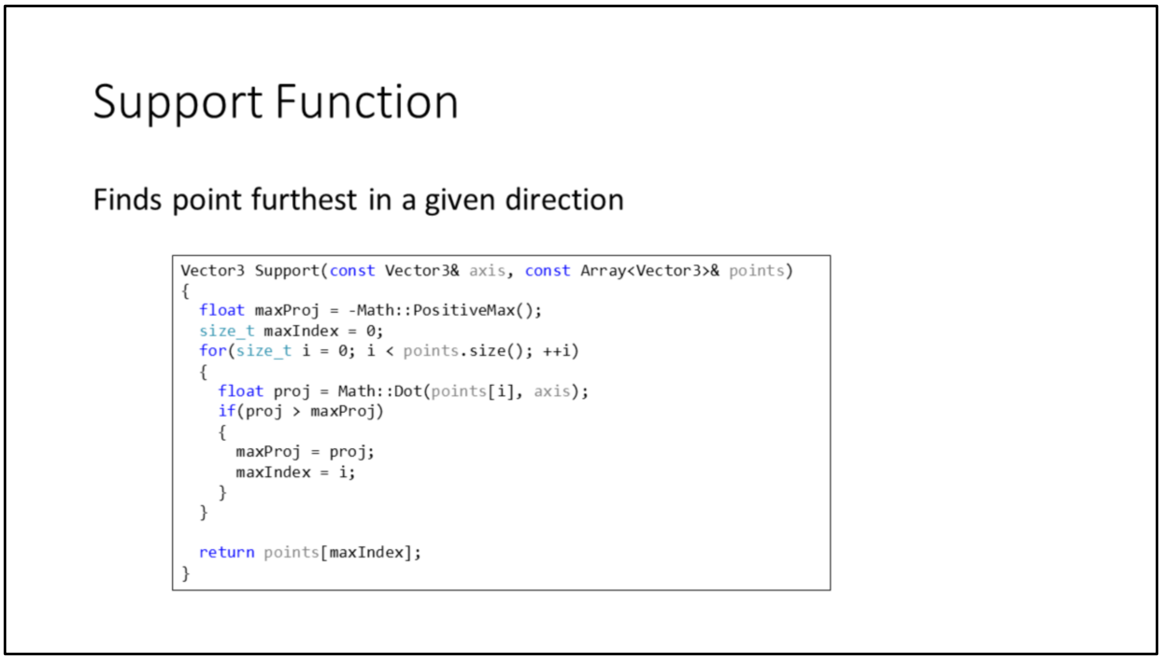


Projection Radius

Project $(\vec{p} - \vec{c})$ where $\vec{p}$ is the point furthest along $\vec{L}$



If we look carefully it should become apparent that the projection of the vector from the center of the shape to the point furthest along the axis will produce the projection radius.



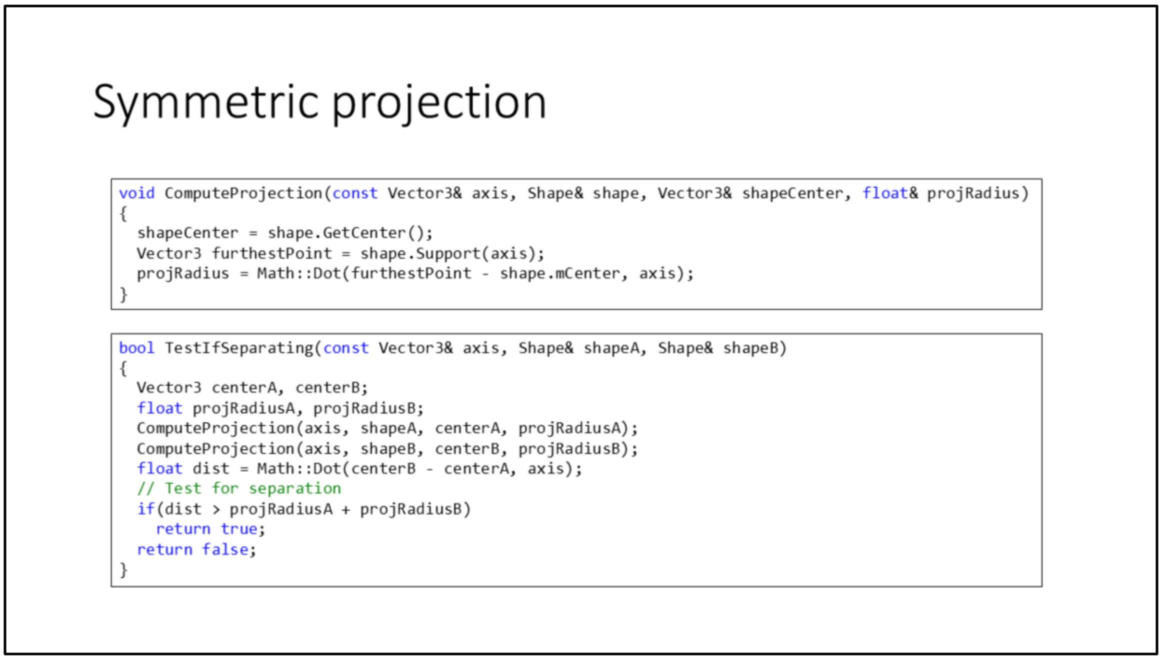
Support Function

Finds point furthest in a given direction

```
Vector3 Support(const Vector3& axis, const Array<Vector3>& points)
{
  float maxProj = -Math::PositiveMax();
  size_t maxIndex = 0;
  for(size_t i = 0; i < points.size(); ++i)
  {
    float proj = Math::Dot(points[i], axis);
    if(proj > maxProj)
    {
      maxProj = proj;
      maxIndex = i;
    }
  }

  return points[maxIndex];
}
```

This leads nicely to the concept of a support function. A support function is simply a function that finds the point on the object furthest in the given direction. Support functions are not commonly referenced in SAT literature. Instead they come up in things like GJK (next topic). However, I'm bringing them up here as they are still useful in understanding SAT and it's important to get used to them early on.



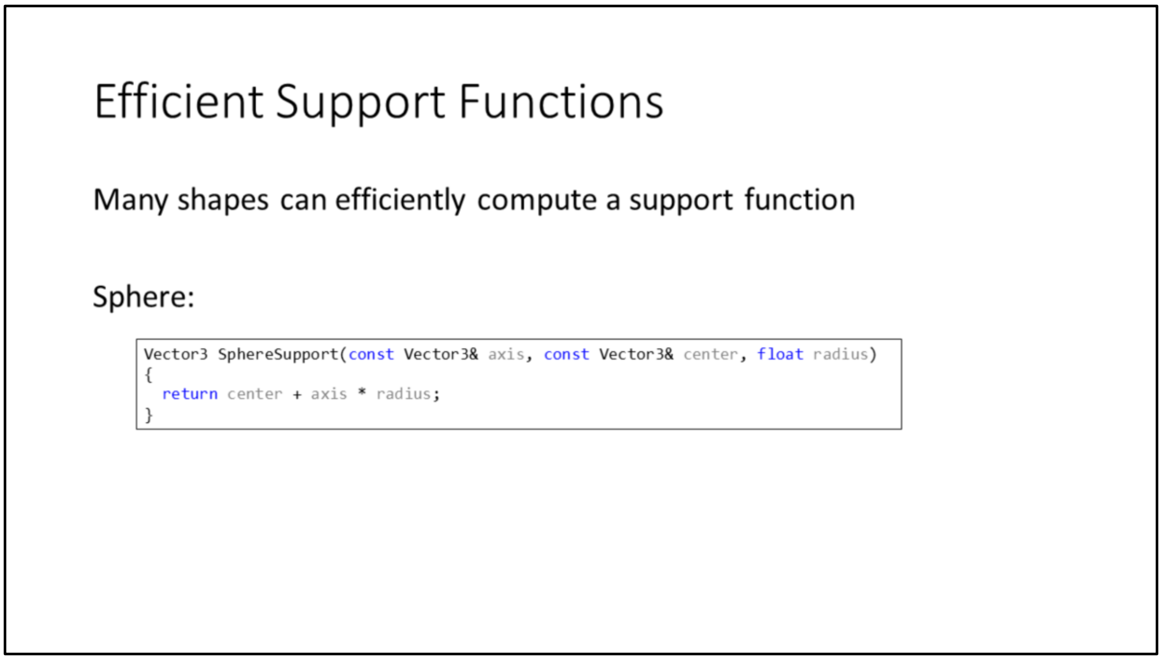
Symmetric projection

```
void ComputeProjection(const Vector3& axis, Shape& shape, Vector3& shapeCenter, float& projRadius)
{
  shapeCenter = shape.GetCenter();
  Vector3 furthestPoint = shape.Support(axis);
  projRadius = Math::Dot(furthestPoint - shape.mCenter, axis);
}
```

```
bool TestIfSeparating(const Vector3& axis, Shape& shapeA, Shape& shapeB)
{
  Vector3 centerA, centerB;
  float projRadiusA, projRadiusB;
  ComputeProjection(axis, shapeA, centerA, projRadiusA);
  ComputeProjection(axis, shapeB, centerB, projRadiusB);
  float dist = Math::Dot(centerB - centerA, axis);
  // Test for separation
  if(dist > projRadiusA + projRadiusB)
    return true;
  return false;
}
```

Now we can re-write our symmetric projection in terms of a support function and re-write testing an axis for symmetric objects to use the radius test.

We can even re-write the non-symmetric version using support functions but this might be less efficient as it has to perform 2 support function calls.



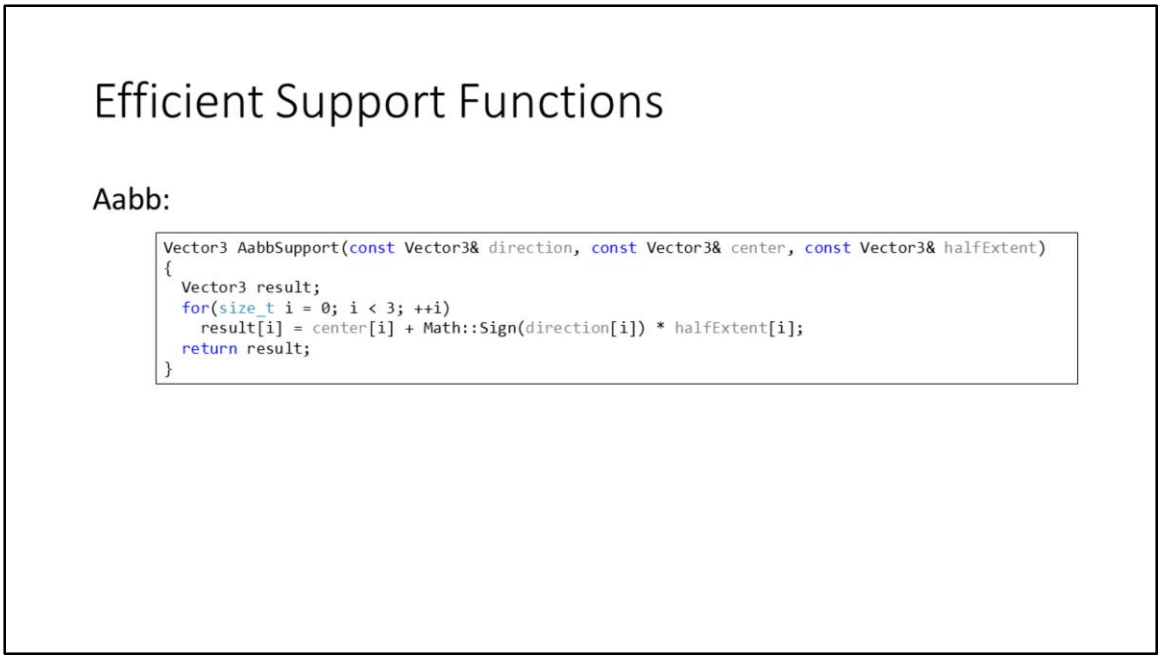
Efficient Support Functions

Many shapes can efficiently compute a support function

Sphere:

```
Vector3 SphereSupport(const Vector3& axis, const Vector3& center, float radius)
{
  return center + axis * radius;
}
```

So now that we've gone through a few of these hoops, what has this saved us? With symmetric objects we've gained just a little bit of time (avoiding one "if" in the loop). But here's where things get interesting. We can uniquely define our support shape per object. Many objects can efficiently compute a support. The simplest example is a sphere. Instead of iterating through all points on its surface it can simply multiply the direction vector by its radius.

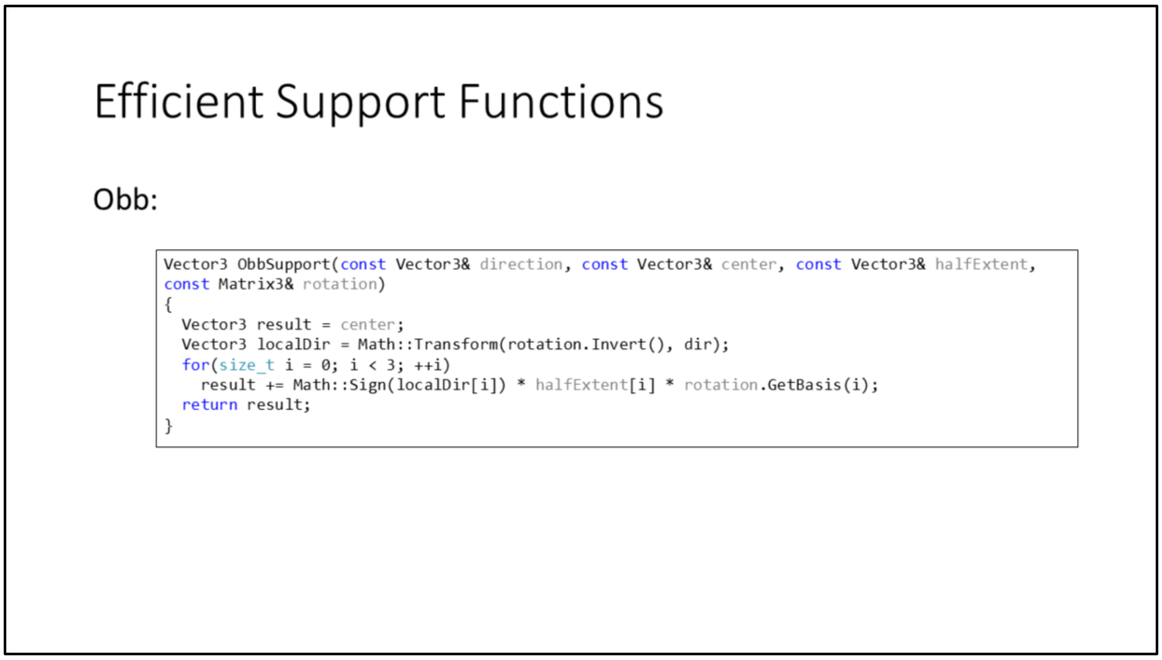


Efficient Support Functions

Aabb:

```
Vector3 AabbSupport(const Vector3& direction, const Vector3& center, const Vector3& halfExtent)
{
  Vector3 result;
  for(size_t i = 0; i < 3; ++i)
    result[i] = center[i] + Math::Sign(direction[i]) * halfExtent[i];
  return result;
}
```

An aabb can perform a similar set of operations to more efficiently compute a support as mentioned during the simple intersections slides. Using center and half extents is the easiest aabb representation to implement the support function with. Any point on an aabb can be represented as a sum of positive or negative components of the half extents, so it’s just a matter of choosing the correct signs to add. As we want the point furthest in a direction we want either the positive or negative half extent depending on the sign of the direction vector.



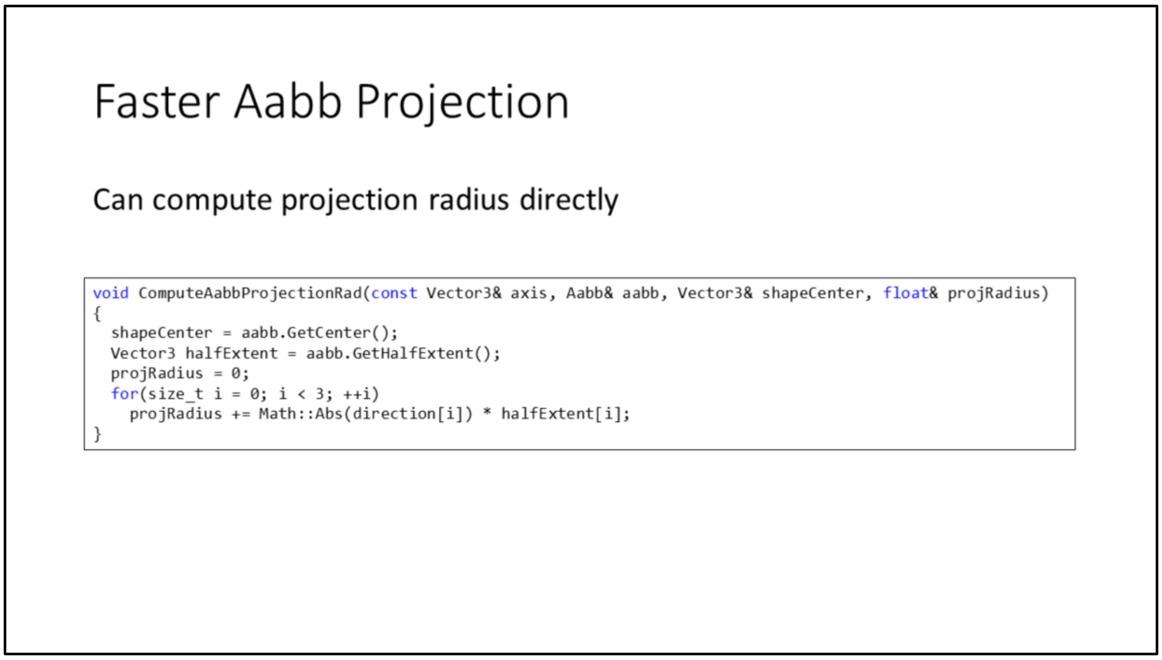
Efficient Support Functions

Obb:

```
Vector3 ObbSupport(const Vector3& direction, const Vector3& center, const Vector3& halfExtent,
const Matrix3& rotation)
{
  Vector3 result = center;
  Vector3 localDir = Math::Transform(rotation.Invert(), dir);
  for(size_t i = 0; i < 3; ++i)
    result += Math::Sign(localDir[i]) * halfExtent[i] * rotation.GetBasis(i);
  return result;
}
```

Obbs can perform a very similar operation. The only difference is that we can’t just add the half extents on the world axes. We need to sum on the local axes. Luckily an object’s rotation matrix contains the local basis vectors of the object in world space. That is the world space x axis of an obb is represented by the x column (assuming column basis matrices) of its rotation matrix. We also need to transform the search direction back into local space to properly check the sign value (think an obb rotated 180 degrees, the sign values would be flipped. Hence we can write obb support with just a minor modification.

Alternatively we could transform the direction vector into the local space of the obb, use the aabb support function, and then transform the support point back into world space.

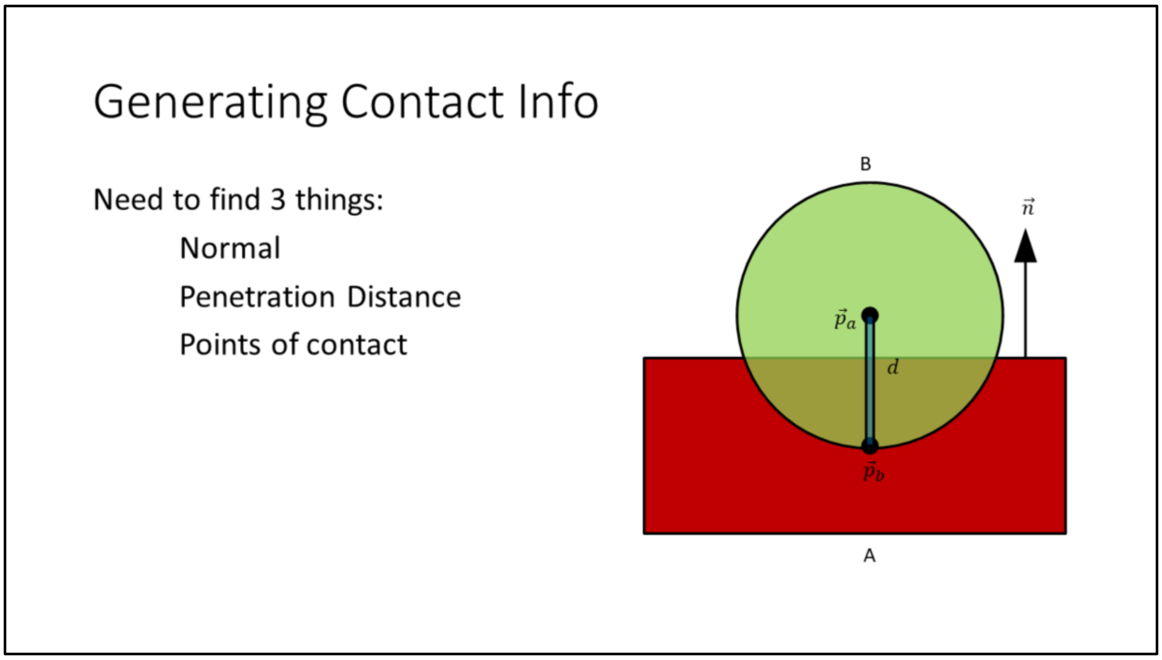


Faster Aabb Projection

Can compute projection radius directly

```
void ComputeAabbProjectionRad(const Vector3& axis, Aabb& aabb, Vector3& shapeCenter, float& projRadius)
{
  shapeCenter = aabb.GetCenter();
  Vector3 halfExtent = aabb.GetHalfExtent();
  projRadius = 0;
  for(size_t i = 0; i < 3; ++i)
    projRadius += Math::Abs(direction[i]) * halfExtent[i];
}
```

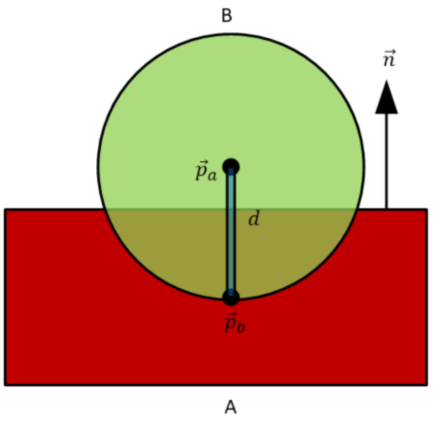
All this being said, we can write a slightly more efficient SAT by not using a support function. In particular we don't have to compute the furthest point and then project it to get a radius; we can compute the radius explicitly. This is the exact same procedure used in computing the rotated Aabb of an Aabb. The same operation can also be extended for Obbs.



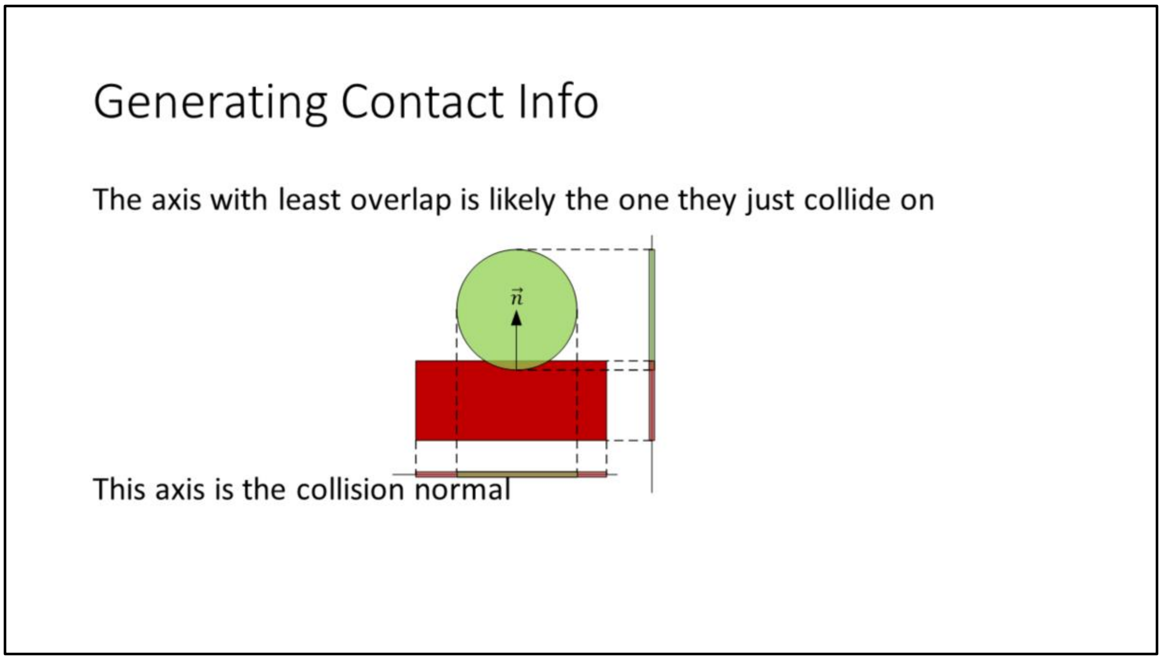
Generating Contact Info

Need to find 3 things:

- Normal
- Penetration Distance
- Points of contact

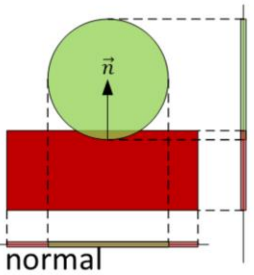


In the process of determining that objects overlap using SAT, we can determine more information about how the objects collided. This information can be used to generate contact information to resolve collisions. For resolving collisions we need 3 pieces of information: normal, penetration distance and points of contact.



Generating Contact Info

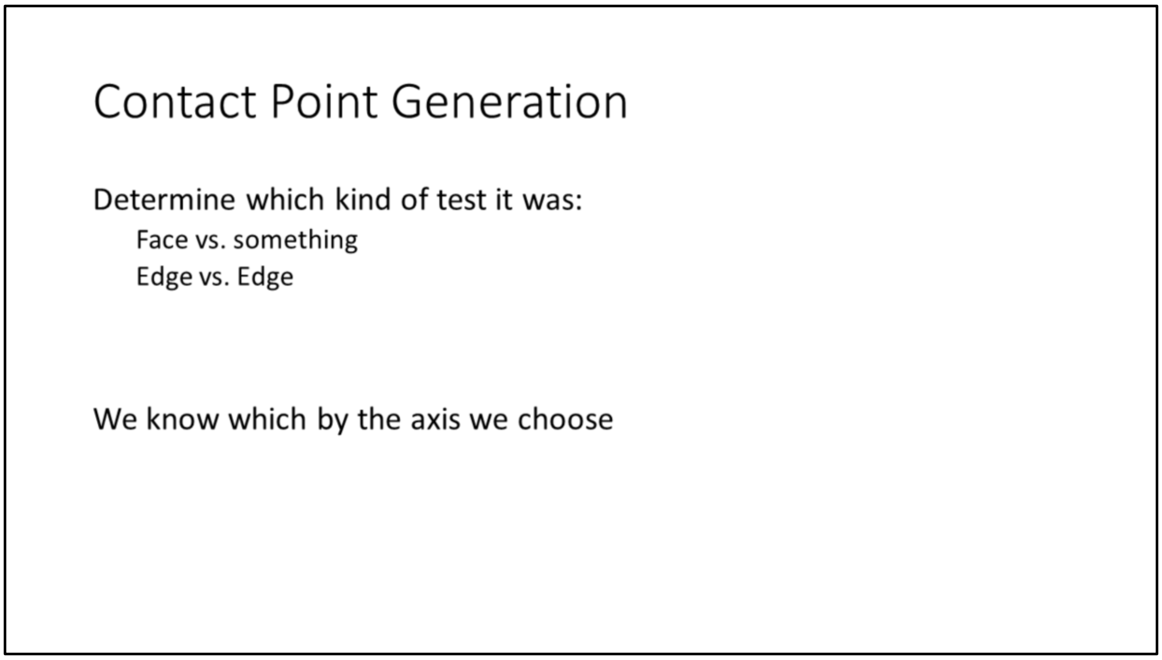
The axis with least overlap is likely the one they just collide on



This axis is the collision normal

So what do we have from SAT that can be used to compute contact information? Well we have a bunch of overlap values on a bunch of axes. How is this useful? Well the first assumption we make is that object's haven't moved a whole bunch in a frame (temporal coherence). This implies that the axis with the least overlap is the one that the objects started colliding on. So when we test all axes in SAT we need to record information about which one produced the smallest overlap. This axis is the collision normal used to resolve objects because it requires the least amount of work to push the objects apart.

Do note that when comparing the smallest overlaps on an axis that the axis must be normalized otherwise we'll have incorrect results. Much of the literature on SAT only talks about detecting collision, not generating contact info. Because of this they tend to do tests without normalized vectors.



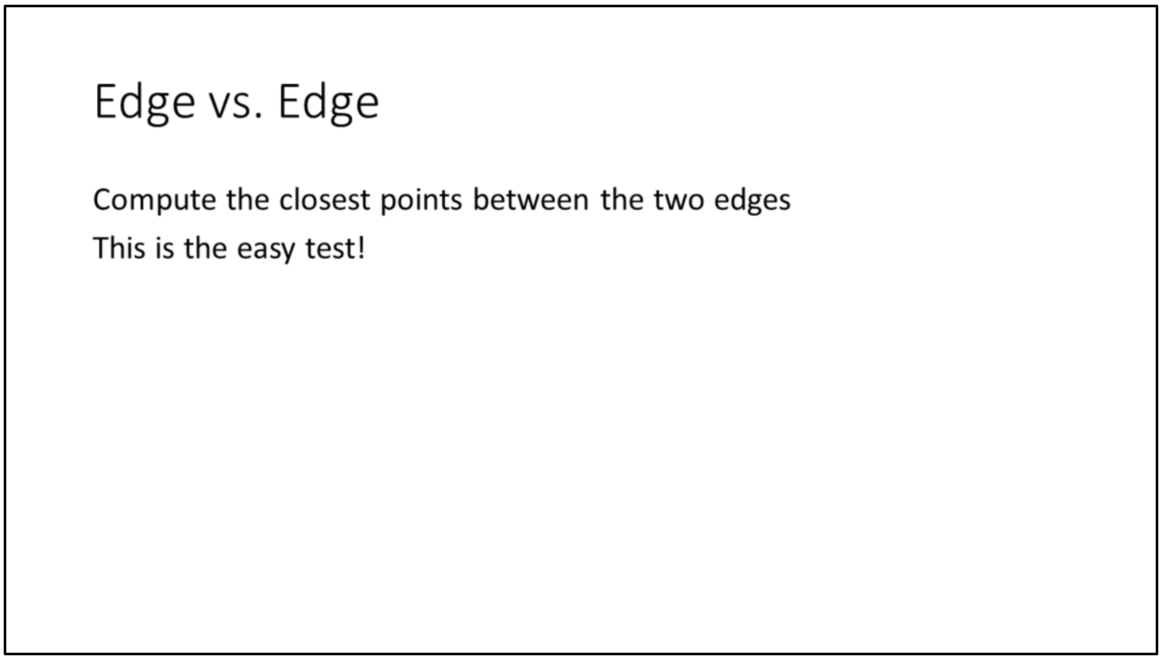
Contact Point Generation

Determine which kind of test it was:

- Face vs. something
- Edge vs. Edge

We know which by the axis we choose

To find the contact points we want the overlapping features closest to each other on the minimum axis. The first thing we need to do is determine which kinds of features were intersecting. We should be able to determine part of this from which axis was overlapping. If this was an edge vs. edge normal then we know we have edge vs. edge. Otherwise we had a face vs. something test.

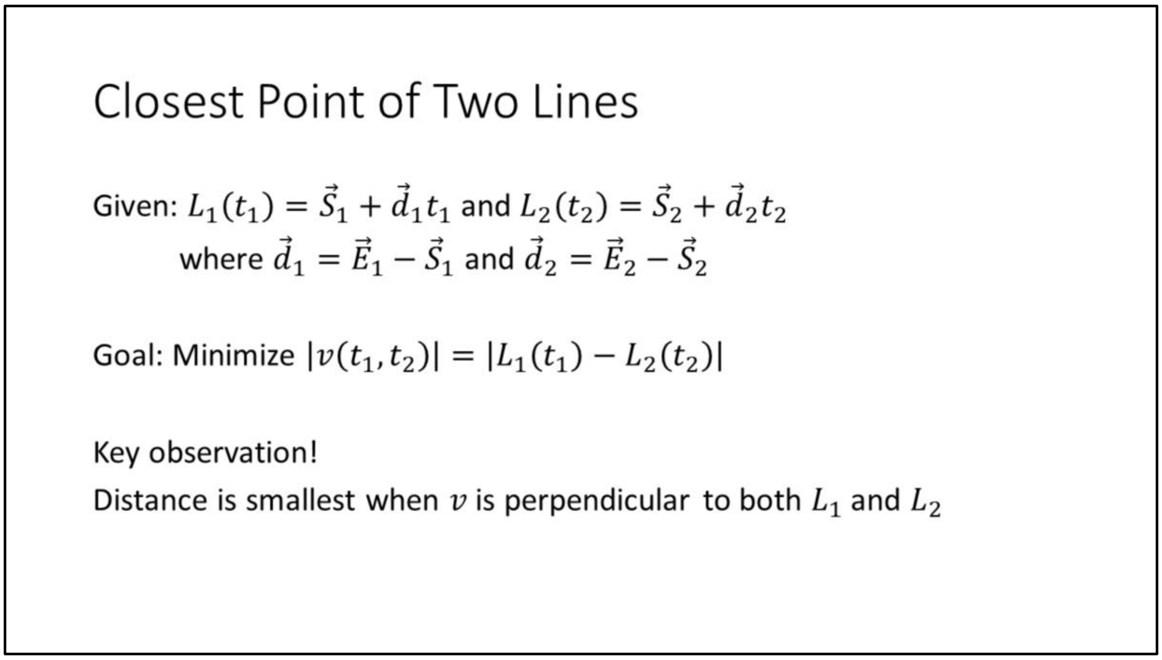


Edge vs. Edge

Compute the closest points between the two edges

This is the easy test!

In the case of edge vs. edge we want to find the closest features between the two edges. This amounts to finding the closest point between two line segments. For the fine details of this test see the section on closest features in Real-Time Collision Detection.



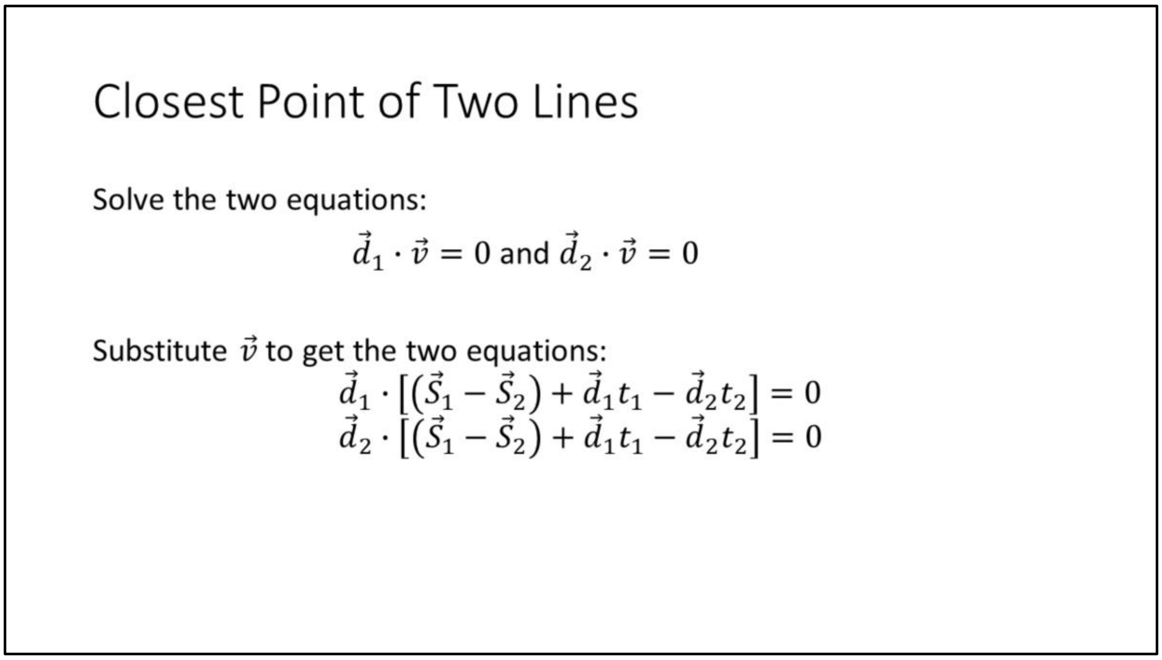
Closest Point of Two Lines

Given: $L_1(t_1) = \vec{S}_1 + \vec{d}_1 t_1$ and $L_2(t_2) = \vec{S}_2 + \vec{d}_2 t_2$
where $\vec{d}_1 = \vec{E}_1 - \vec{S}_1$ and $\vec{d}_2 = \vec{E}_2 - \vec{S}_2$

Goal: Minimize $|v(t_1, t_2)| = |L_1(t_1) - L_2(t_2)|$

Key observation!
Distance is smallest when v is perpendicular to both L_1 and L_2

To find the closest point between two edges we first need to look at the closest point between two infinite lines.
The closest point between two lines will be one where we minimize the length of the vector formed by a point on line A and a point on line B.
The key observation here is that the closest points between the two objects will form a line that is perpendicular to both L_1 and L_2.
This is easiest to see if we just look at closest point between a point and a line. In this case the closest point will be the orthogonal projection of the test point on the line.



Closest Point of Two Lines

Solve the two equations:

$$\vec{d}_1 \cdot \vec{v} = 0 \text{ and } \vec{d}_2 \cdot \vec{v} = 0$$

Substitute $\vec{v}$ to get the two equations:

$$\vec{d}_1 \cdot \left[\left(\vec{S}_1 - \vec{S}_2\right) + \vec{d}_1 t_1 - \vec{d}_2 t_2\right] = 0$$
$$\vec{d}_2 \cdot \left[\left(\vec{S}_1 - \vec{S}_2\right) + \vec{d}_1 t_1 - \vec{d}_2 t_2\right] = 0$$

To find the 2 t-values we can plug in v and solve.

Closest Point of Two Lines

Solution:

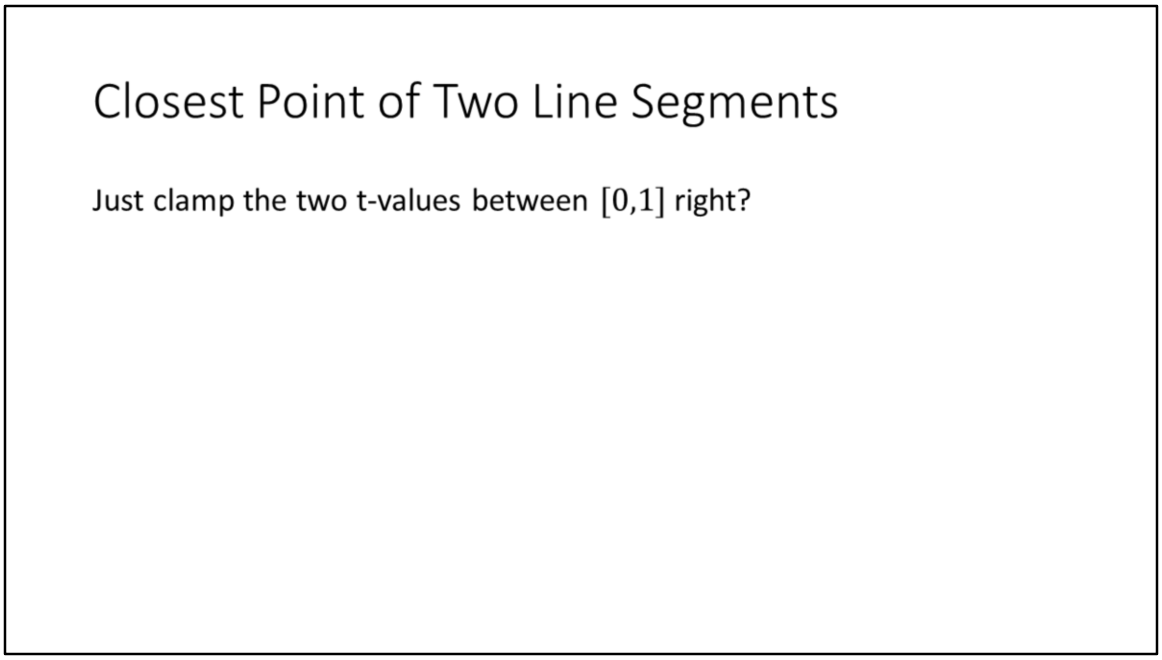
$$\begin{bmatrix} \vec{d}_1 \cdot \vec{d}_1 & -\vec{d}_1 \cdot \vec{d}_2 \\ \vec{d}_1 \cdot \vec{d}_2 & -\vec{d}_2 \cdot \vec{d}_2 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} -\vec{d}_1 \cdot \vec{S}_{12} \\ -\vec{d}_2 \cdot \vec{S}_{12} \end{bmatrix}$$

Where $\vec{S}_{12} = \vec{S}_1 - \vec{S}_2$

Solve using Cramer's rule.

If we sit down we can solve to get a 2x2 system of equations.
We can then label this as $\begin{bmatrix} a & -b \\ b & -e \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} -c \\ -f \end{bmatrix}$
Where $a = \vec{d}_1 \cdot \vec{d}_1$, $b = \vec{d}_1 \cdot \vec{d}_2$, $c = \vec{d}_1 \cdot \vec{S}_{12}$, $e = \vec{d}_2 \cdot \vec{d}_2$, and $f = -\vec{d}_2 \cdot \vec{S}_{12}$.
When solving with Cramer's rule we can then solve for $t_1 = (fb - ce)/d$ and $t_2 = (af - cb)/d$
Where $d = ae - b^2$. Note: this re-arranged some terms to remove some minus signs.

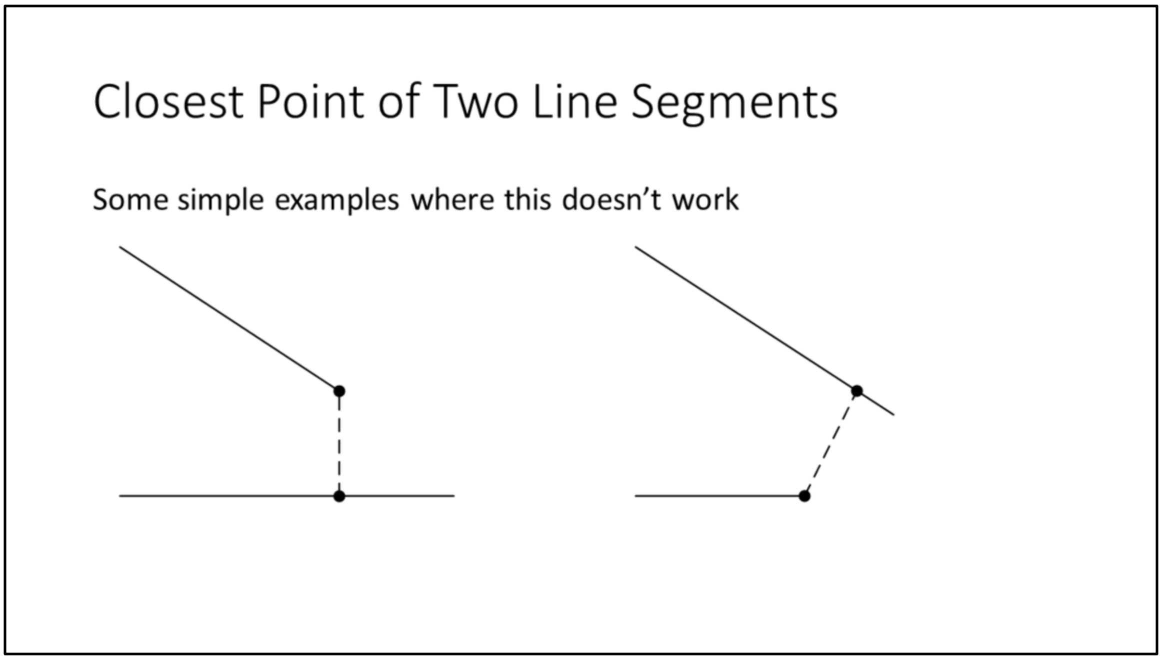
From a quick look we can see some interesting things about d. First note that we can expand to get $d = \vec{\mathrm{d}}_1^2 \vec{d}_2^2 - \left(\vec{d}_1 \cdot \vec{d}_2\right)^2$. First remember our dot-product identities. So we can expand to get: $d = \vec{\mathrm{d}}_1^2 \vec{d}_2^2 - \left(\left|\vec{d}_1\right|\left|\vec{d}_2\right|\cos(\theta)\right)^2$. From here we can do some more manipulation to get: $d = \vec{\mathrm{d}}_1^2 \vec{d}_2^2 \left(1 - \cos(\theta)\right)^2 = \vec{\mathrm{d}}_1^2 \vec{d}_2^2 \sin^2(\theta)$. Note now that $d \geq 0$ and if d is zero then the lines are parallel.



Closest Point of Two Line Segments

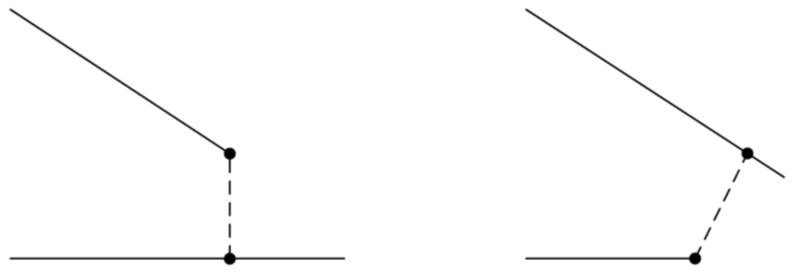
Just clamp the two t-values between $[0,1]$ right?

So now we can move onto closest point of two line segments. This should only be clamping the t-values on each line segment right?

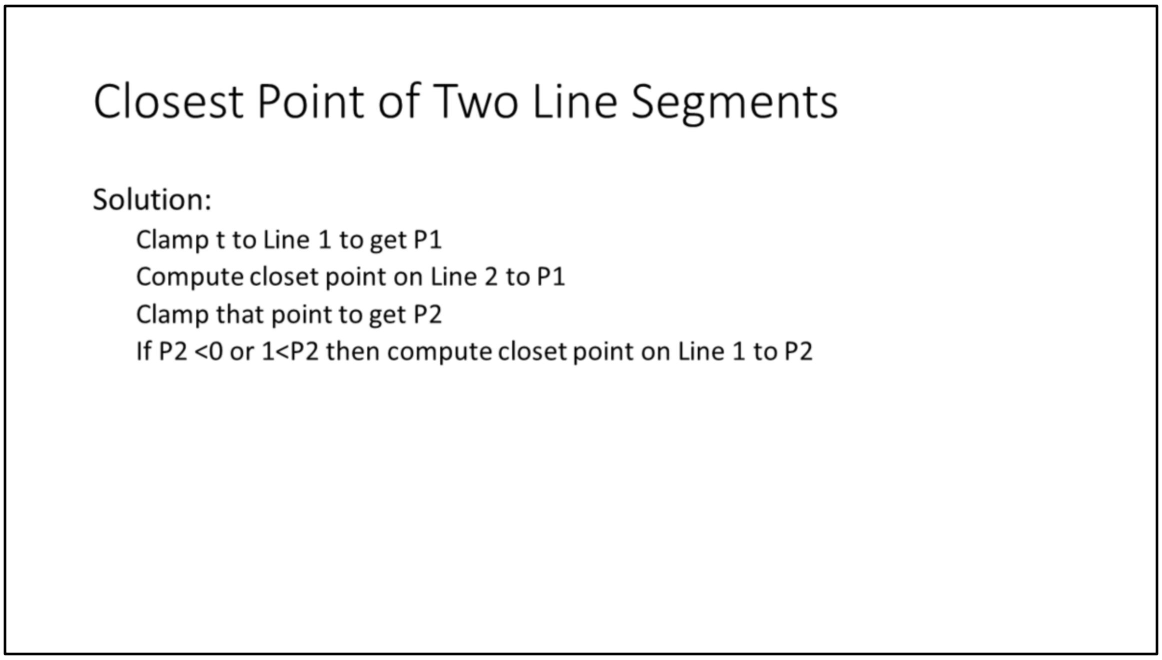


Closest Point of Two Line Segments

Some simple examples where this doesn't work



Unfortunately things are a bit more complicated than that. It's not too hard to construct an example where clamping one each line segment will not work.



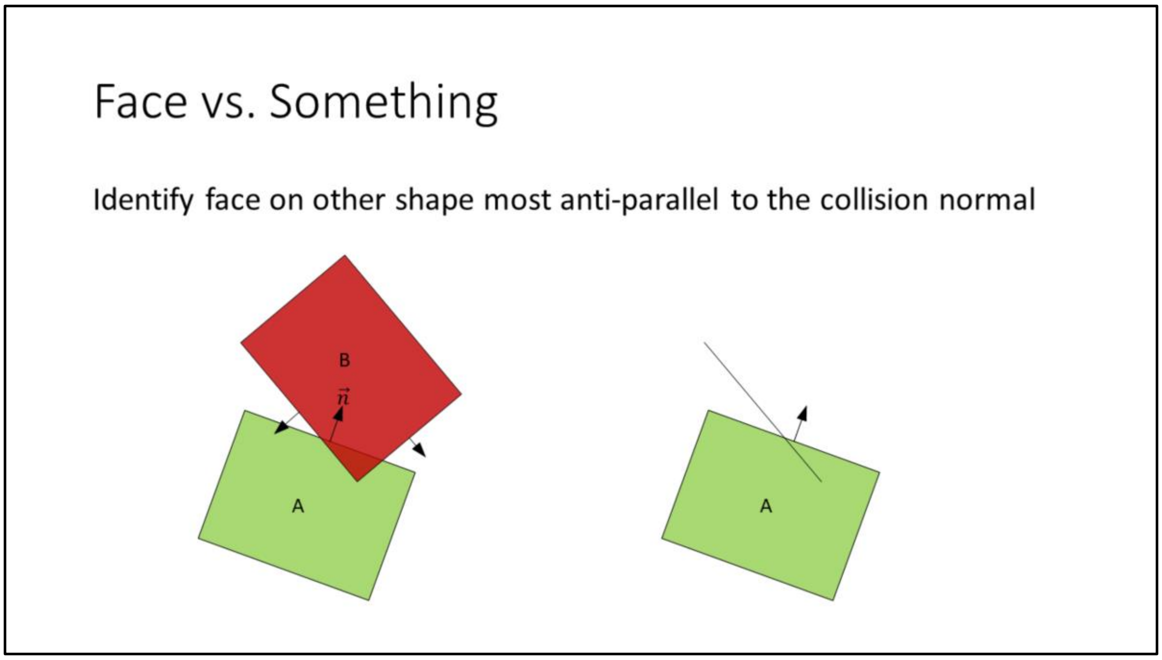
Closest Point of Two Line Segments

Solution:

```
    Clamp t to Line 1 to get P1
    Compute closet point on Line 2 to P1
    Clamp that point to get P2
    If P2 <0 or 1<P2 then compute closet point on Line 1 to P2
```

The solution is to just clamp multiple times using closest point to line segment from point.

Some extra optimizations can be made by computing the solution for t_2 given t_1 and vice-versa. This is left as an exercise for the reader (also see the orange book).



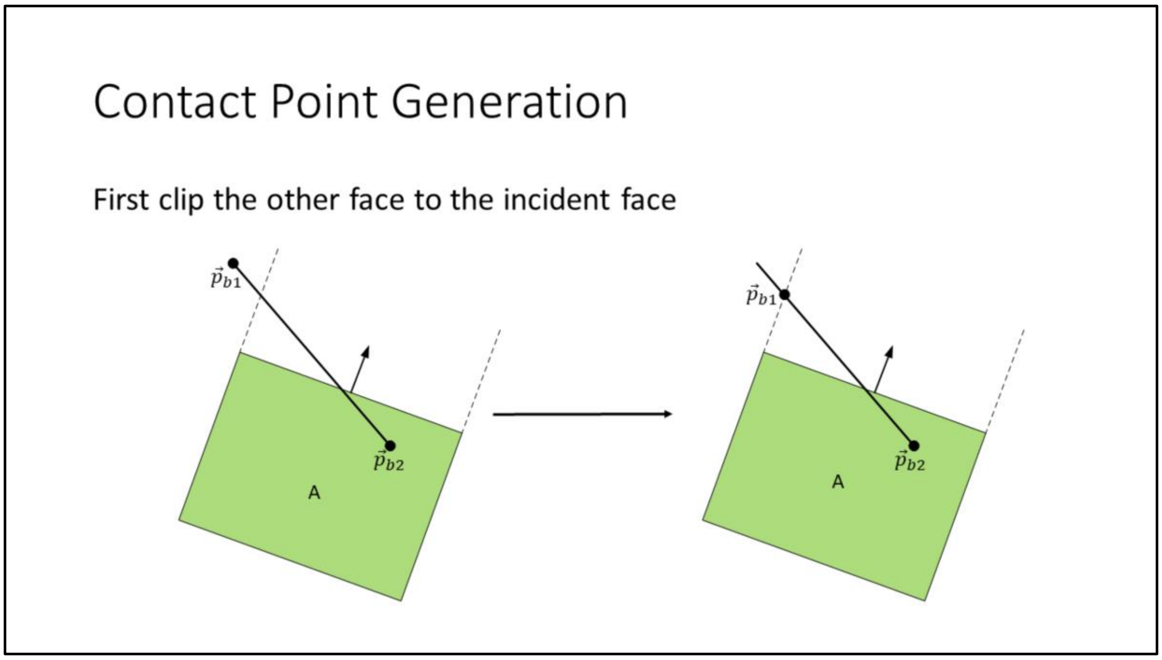
Face vs. Something

Identify face on other shape most anti-parallel to the collision normal

Face vs. something is much harder. The basic idea is to find the overlapping area and what points identify that area. This is a bit difficult and we can do with a little worse information. Instead we want to find the features most offending to the collision.

To make life easier, I will reference to shape A as being the one that the normal is coming from and shape B as the other shape.

To start with we need to find which feature on shape B is most anti-parallel to the contact normal.



Contact Point Generation

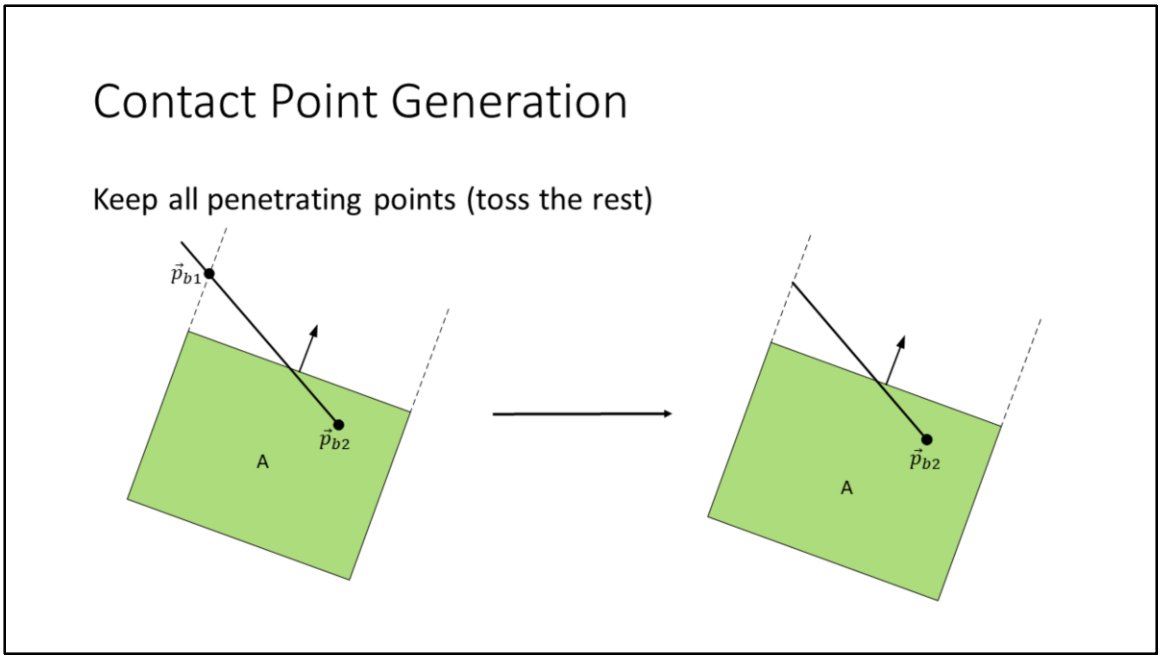
First clip the other face to the incident face

$\vec{p}_{b1}$
$\vec{p}_{b2}$
A
$\vec{p}_{b1}$
$\vec{p}_{b2}$
A

Once we've identified the offending face we need to generate contact points to represent the intersecting region. To do this we need to clip face B against face A.

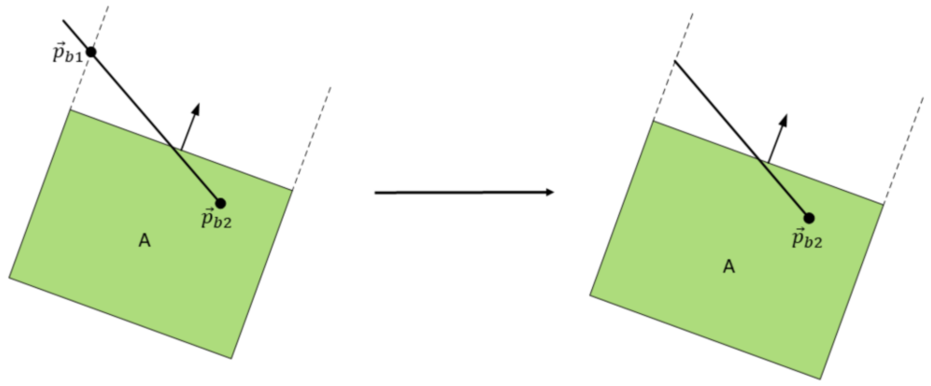
In 3d clipping a face is a bit tricky. One good method to go about this is to transform into A's space and then project B's face onto A's. Now clipping is a 2d problem, the only trick is how to un-project the clipped points. Barycentric coordinates can be used to do this.

Another thing to consider in 3d is how to reduce the number of contact points down to a manageable level. We might only store 4 contact points but generate 8 during clipping. Ideally we will compute the widest spread of points. One method to do this is to divide up our projection space into as many sections as we will create points for. Then we can find a point most in that area.

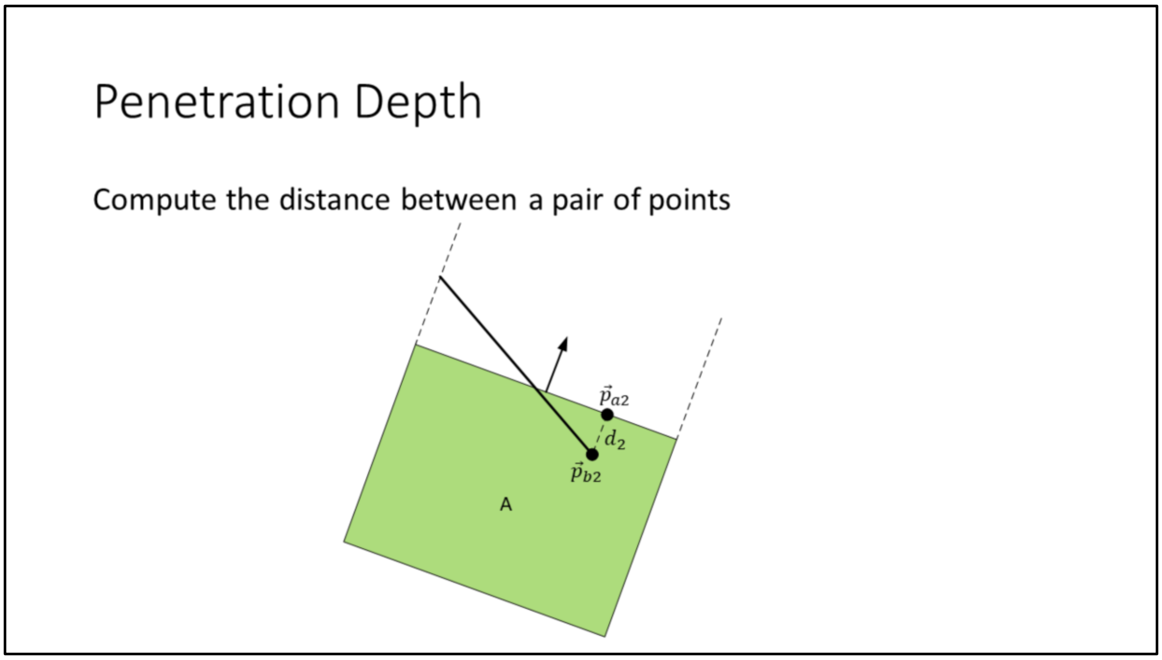


Contact Point Generation

Keep all penetrating points (toss the rest)

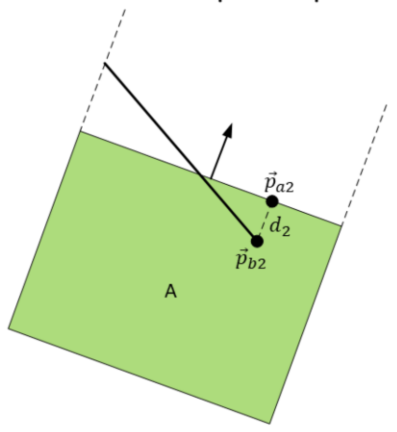


After clipping we want to only keep points that are actually penetrating into face A. In the simple case like pictured above we're fine with only 1 point of contact as the object won't really need to come to rest.



Penetration Depth

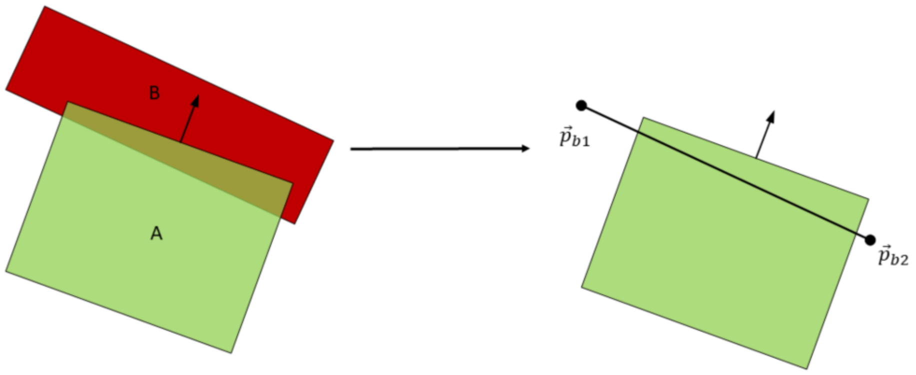
Compute the distance between a pair of points



To compute the penetration depth is easy now. For a pair of points the penetration depth is the distance between the point on A and the point on B in the direction of the normal ($d = Dot(\vec{p}_b - \vec{p}_a, \vec{n})$).

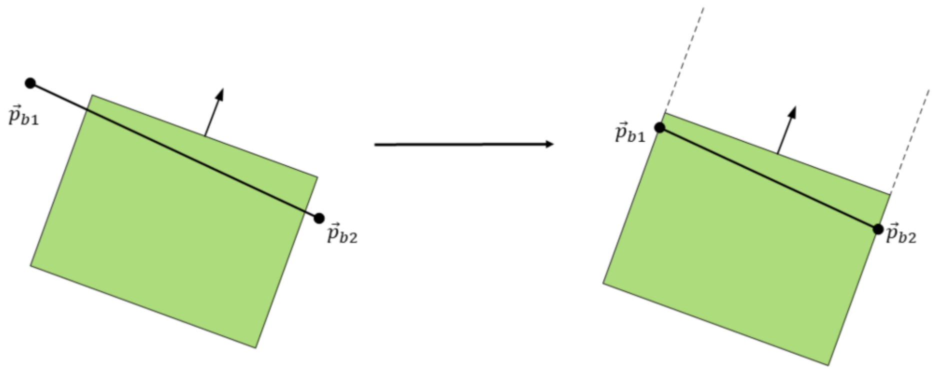
More complicated example

First identify the two faces



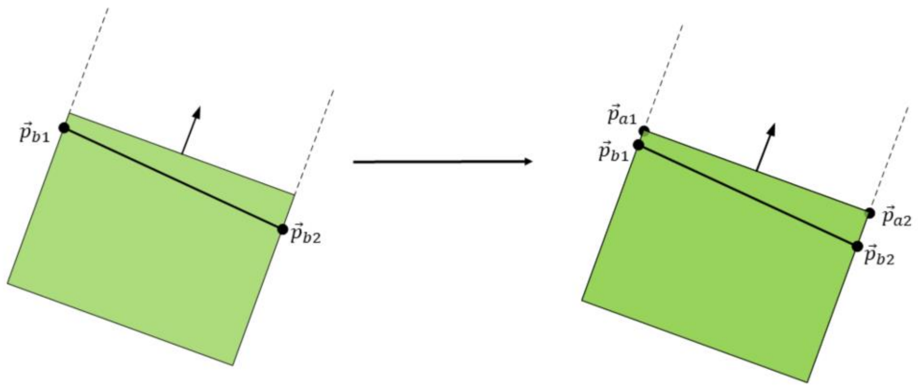
More complicated example

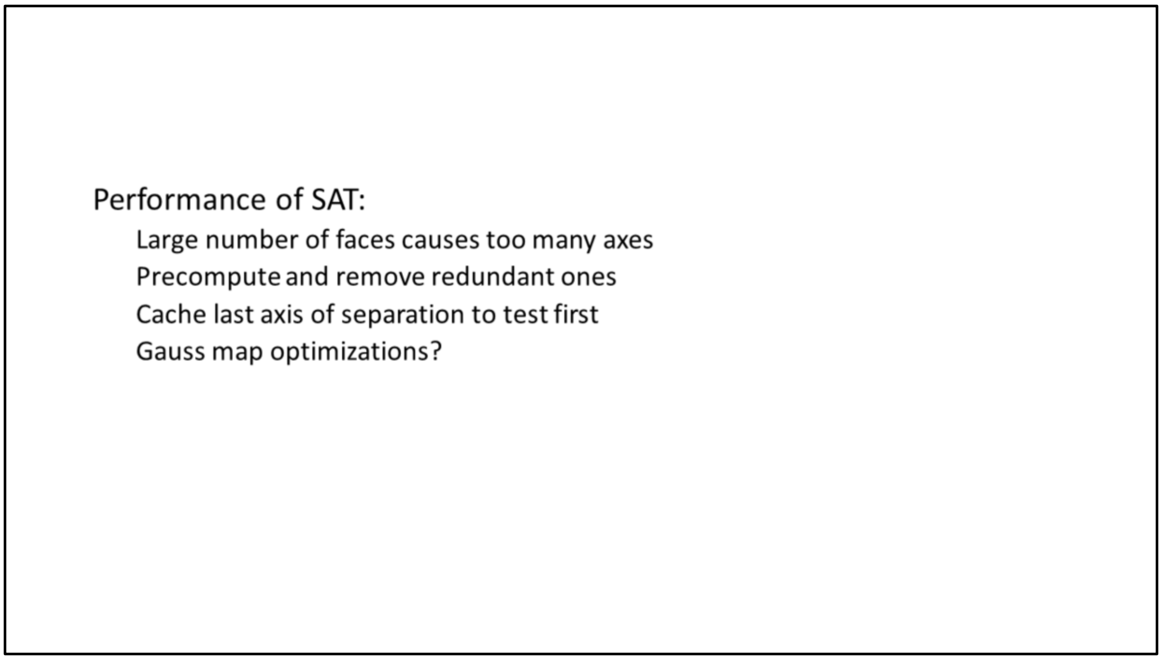
Clip face B to face A



More complicated example

Project the points onto face A to compute the point pairs





Performance of SAT:

- Large number of faces causes too many axes
- Precompute and remove redundant ones
- Cache last axis of separation to test first
- Gauss map optimizations?

As mentioned briefly before, SAT doesn't scale with a large number of axes. There's a few methods to try and deal with this. One of them was mentioned briefly before: removing any axes that are close enough. Another common method is to keep a cache of what axes caused a pair of objects to be separating last time. By testing this axis first we may early out after only 1 test.

The final major optimization I won't talk about it much detail as it's outside the scope of what I can easily cover here. The idea is to efficiently reduce the number of edges cross terms to test. This can be performed using Gauss Maps. For details see Dirk Gregorius' 2013 GDC presentation.