Simple Intersection

jodavis42@gmail.com

Simple shape review

Primitives:

- Point
- Plane
- Triangle
- Aabb
- Sphere
- Ray
- Frustum

Before getting into any intersection algorithms, it's first important to cover the basic representation of the shapes being used. For the most part, I will only cover the representations used in the framework with some more detailed variations mentioned later. Most of this is expected to be review.

Plane

Point + Normal

$$\vec{n} \cdot (\vec{p} - \vec{p}_0) = 0$$

Requires 6 floats

Expand to $\vec{n} \cdot \vec{p} = d$

```
struct Plane
{
  // (n.x, n.y, n.z, d)
  Vector4 mData;
};
```

A plane is actually one of the more complicated shapes to represent because there's several good methods to represent them. The most intuitive method to represent a plane is with a point and a normal which gives the equation: $\vec{n} \cdot (\vec{p} - \vec{p}_0) = 0$. However, this representation requires 6 floats which is more than ideal. Instead, if you plug and chug you can get the equation: $\vec{n} \cdot \vec{p} = d$ which only requires 4 floats (3 for the normal and 1 for d).

Triangle

Nothing special

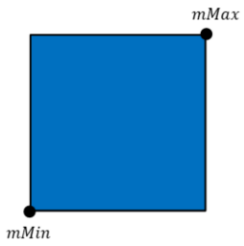
```
struct Triangle
{
  Vector3 mP0;
  Vector3 mP1;
  Vector3 mP2;
};
```

A triangle is very simple to represent, just 3 points. Note that you can easily compute a plane from a triangle, all you need is the normal. Assuming the triangle is defined counter-clockwise, the normal is just $Cross(\vec{p}_1 - \vec{p}_0, \vec{p}_2 - \vec{p}_0)$

Axis Aligned Bounding Box (Aabb)

Min and max on each axis

```
struct Aabb
{
  Vector3 mMin;
  Vector3 mMax;
};
```

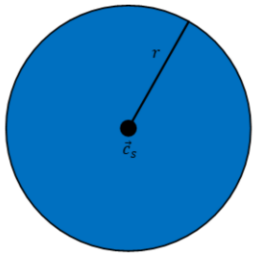


An aabb is a box that is aligned with the cardinal axes, hence we only need to store the min and max on each axis. The other common representation for an Aabb is a center plus a half extent. There are several occasions where this representation is better than min and max but it's easy to convert between and min/max is more useful for intersection tests.

Sphere

Sphere equation: $(\vec{c}_s - \vec{p})^2 - r^2 = 0$

```
struct Sphere
{
  Vector3 mPosition;
  float mRadius;
};
```

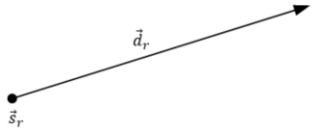


A sphere's representation is fairly straightforward. Just the center of the sphere and the radius. The equation of the sphere's surface is defined as: $(\vec{c}_s - \vec{p})^2 - r^2 = 0$ which simply states that all points that are distance r away from the center are part of the sphere's surface.

Ray

Ray equation: $\vec{p}_r(t) = \vec{s}_r + \vec{d}_r t$

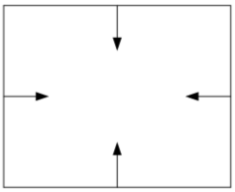
```
struct Ray
{
  Vector3 mStart;
  Vector3 mDirection;
};
```



A ray is represented by a starting point and a direction but can also be thought of as a half-infinite line segment. For simplicity, it is generally assumed that the direction is normalized (but be careful with this assumption, if it isn't true you can compute wrong answers). A ray can mathematically be represented with the equation $\vec{p}_r(t) = \vec{s}_r + \vec{d}_r t$ where I use the subscript r to denote values that are for the ray (to avoid confusion later). Note: a ray is only valid for the t values $[0, \infty]$.

Frustum

```
struct Frustum
{
  Plane mPlanes[6];
  Vector3 mPoints[8];
};
```



Normals point inwards

Frustums are particularly useful for anything regarding a camera. This includes frustum culling and multi-selection. For most computations the 6 planes of the frustum are sufficient. Occasionally it's also nice to have the 8 points (which could be computed from the planes) so for this class I store both.

You can represent the frustum with all planes pointing inwards or outwards. For this class I've chosen to have all the plane normal point inward (or towards the centroid of the frustum). The reasoning for this will become clearer later, but the basic idea is that in order to be inside the frustum you must be inside all 6 planes.

Intersection Test Types

Boolean

Containment

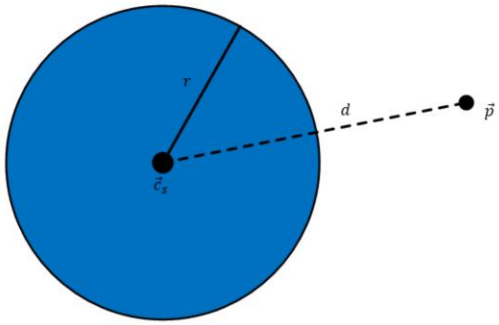
- Coplanar, Outside, Inside, Overlap

Intersection

- Same as containment but typically with a t-value

There's three main kinds of intersection tests we will perform: boolean, containment and intersection. A boolean test just determines true/false if the shapes overlap. A containment test typically is a 3 state boolean test, returning some form of inside/overlap/outside result (occasionally we also classify coplanar). Finally an intersection test can be either boolean or containment, but also returns some form of where the objects overlap. For the most part we will not be doing intersection tests with the main exception being ray vs. shape.

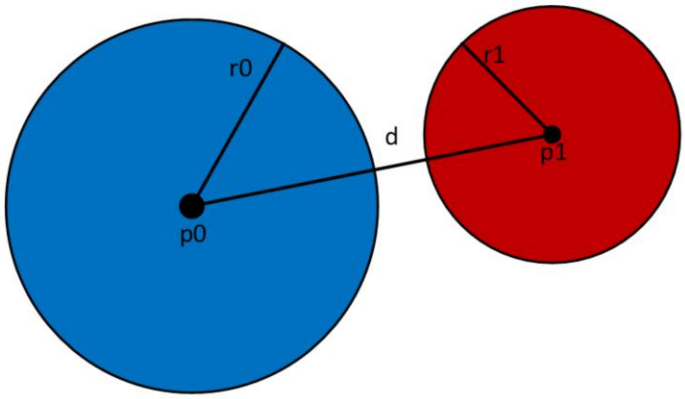
Point vs. Sphere



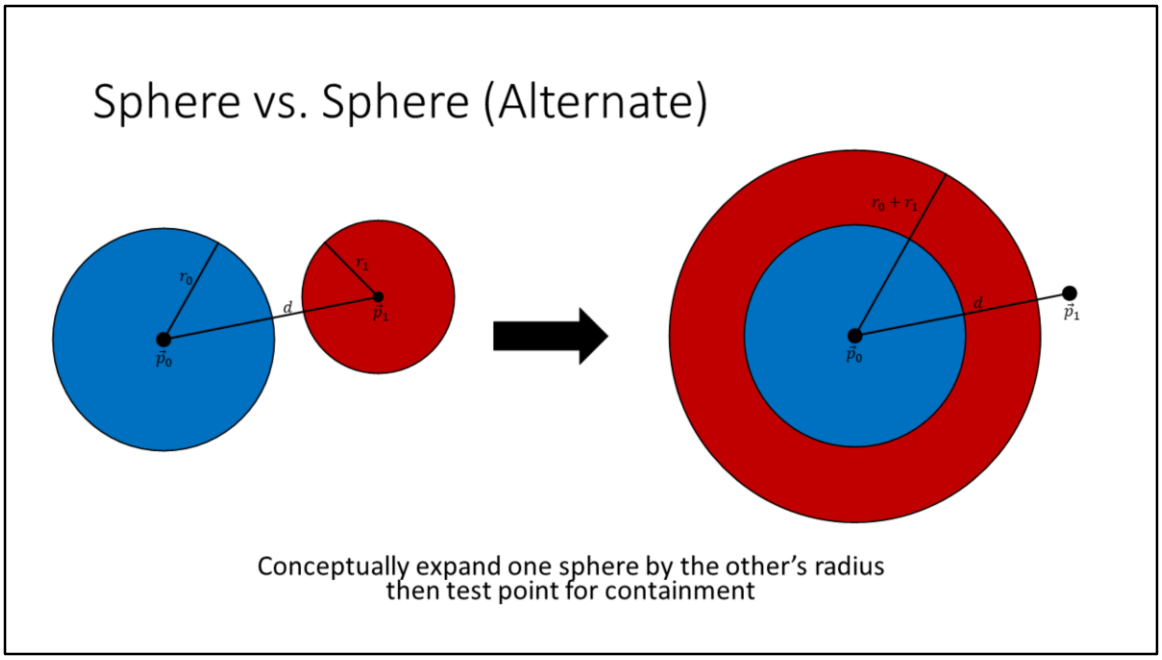
If $d \leq r$ then the point is contained

Perhaps the easiest test to write is point vs. sphere. The point $\vec{p}$ is in a sphere if the distance between it and the sphere's center is less than the sphere's radius. This yields the equation for containment: $(\vec{c}_s - \vec{p})^2 - r^2 \leq 0$. Note that in the above picture d is the distance between the sphere's center and the query point, which is the equation: $(\vec{c}_s - \vec{p}_s)$.

Sphere vs. Sphere



Sphere vs. sphere is another very easy test to write. Two spheres overlap when the distance between them is less than or equal to the sum of their radii. Typically this equation is squared to avoid a square root calculation. That is, we have intersection if: $(\vec{p}_1 - \vec{p}_0)^2 - (\vec{r}_1 + \vec{r}_0)^2 \leq 0$.



One thing you'll see throughout this class is that there's many different ways to view the same problem. Sometimes viewing a problem in a different way helps to understand difficult concepts. While sphere vs. sphere isn't a hard concept to grasp I want to present an alternate way to view this problem.

Instead of viewing this problem as two distinct spheres we can turn this into point vs. sphere. We can do this by expanding one of the sphere's by the other's radius. It should be easy to see that these are mathematically equivalent.

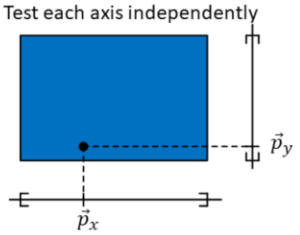
This is easy to do since our shape is a sphere, however we'll visit a topic called Minkowski sums at the end of the semester which will shed a bit more light on this.

Point vs. Aabb

Each axis of an aabb is independent.

Instead of Point vs. Aabb

Test each axis independently



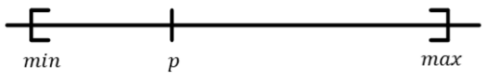
Defining a test for one dimension is easy.

We can extend to n dimensions later.

Since an aabb is axis aligned, it's typically possible to test each axis separately. This means that if we figure out how to do a test on one axis, which is easier, then we know how to do a test on all axes. Afterwards, we only have to figure out how to combine the axis results together to get the total aabb test. This also allows extending a 2d aabb to 3d very easily.

Point vs. Aabb – 1 Dimension Test

How do we test one axis?



Two main ways:

Intersection Test: $min \leq p \leq max$

Non-Intersection Test: $p < min \; or \; p > max$

So how do we test if a value p is in-between a $\min$ and a $\max$ in one dimension? Well there's two ways to write the test.

Method 1 is to see if p is contained in the interval between min and max, this is writing a test to see if there is intersection.

Method 2 is to see if p is outside the interval between min and max, this is writing a test to see if there is non-intersection.

This distinction will come up a lot later as some tests are much easier to write for non-intersection than for intersection.

Point vs. Aabb – n Dimensions

How do we combine the 1-dimensions test to get n-dimensions?

Intersection Test: If all axes are contained

Non-Intersection Test: If any axis isn't contained

*Some tests will be much easier to write for non-intersection

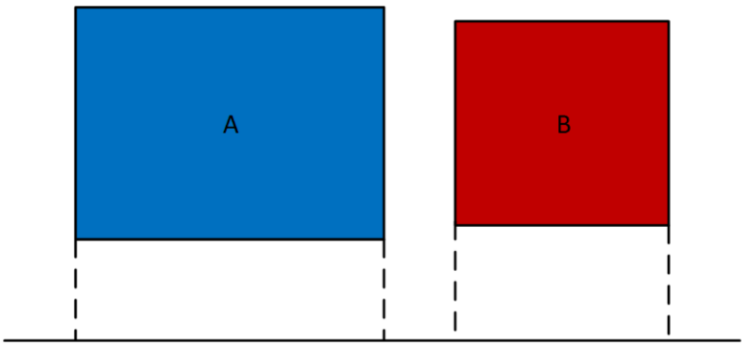
So we now need to combine each individual axis together to get the full aabb result.

If we use the intersection test previously described then how would we combine the results together? If the point is overlapping on all axes then the point is in the aabb. This can be written as: if all axis intersection tests returned true then the point is in the aabb.

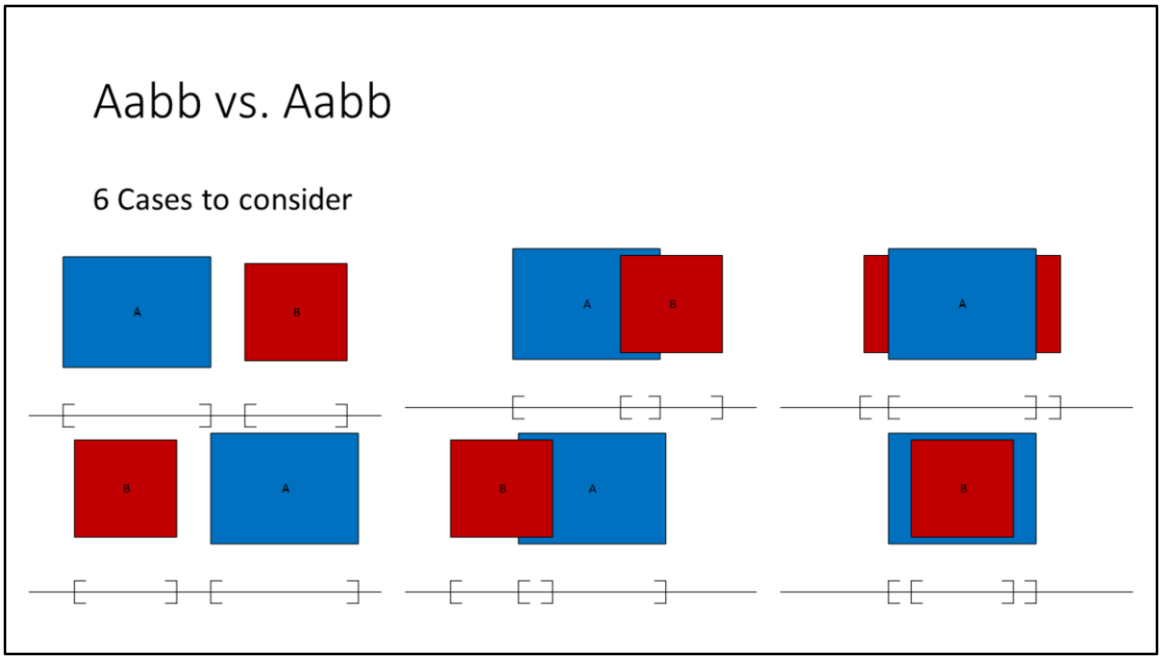
If we use the non-intersection test then how would they be combined? Well if any axis doesn't contain the point then it doesn't matter if any other axis does, the point is not in the aabb. This can therefore be written as: if any axis has non-intersection then the point is not in the aabb.

Aabb vs. Aabb

First look at one axis.



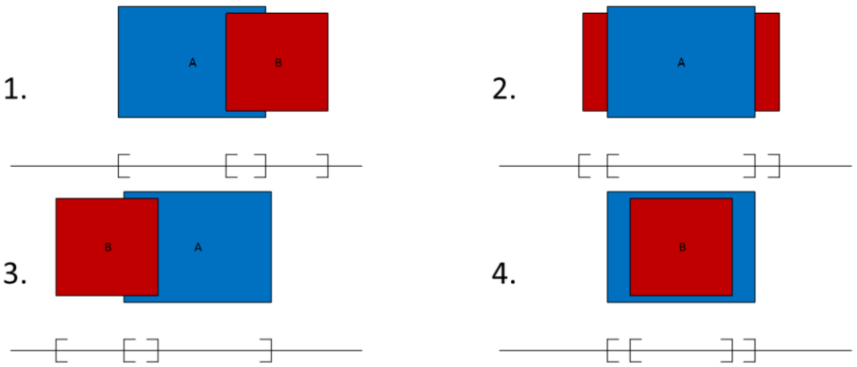
Like Point vs. Aabb, we'll start by looking at one axis first and then combine all of the axis results together.



Of the 6 possible cases, 4 have to be considered for intersection.

Aabb vs. Aabb

How can we write a test for intersection from these 4 cases?



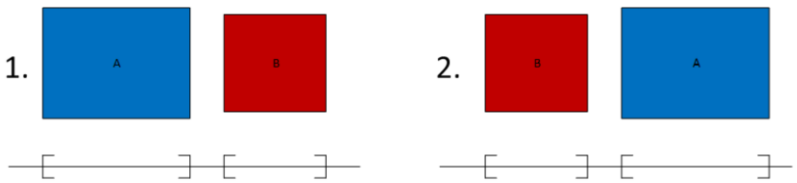
If we wanted to write a test for intersection how would we do so from these 4 cases?

Well case 1 can be checked by testing if B's min is contained in A's range. Additionally test 3 can be checked by seeing if B's max is in A's range. Case 4 is covered by case 1 and 3. The only problem is case 2. To verify case 2 we'd have to see if A's min is within B's range (alternatively we could check A's max).

What about writing a test for non-intersection instead?

Aabb vs. Aabb

How can we write a test for non-intersection from these 2 cases?



How about a test for non-intersection? Well first it should be obvious that we only have 2 cases to consider instead of 4 so this might be better.

Initially it would seem like case 1 would have to check if B's min and max are greater than A's max. However we can make one assumption that'll make this easier: Any aabb's max must be greater than or equal to its min.

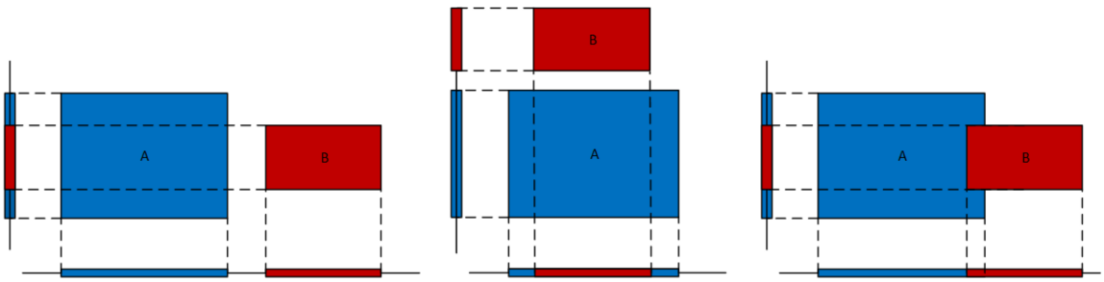
With this in mind we can test case 1 by checking if b's max is greater than a's min.

What about case 2? This can be written just as easily by check if a's max is greater than b's min.

Aabb vs. Aabb

How do we combine tests for non-intersection?

If an axis is separating then there's no intersection



Now we can revisit combining the individual axis results together. This is the exact same as with Point vs. Aabb. If any axis is non-intersecting then there's no intersection between the two objects. It's only if all axes intersect (or all axes are not not-intersecting) that there is intersection between the aabbs.

Ray vs. Plane

Given:

Ray: $\vec{p}_r(t) = \vec{s}_r + \vec{d}_r t$

Plane: $\vec{n} \cdot (\vec{p} - \vec{p}_0) = 0$

How do we solve?

What are we solving for?

To solve ray vs. plane we first have to look at the 2 equations we have.

Given these two equations we have to first figure out what we know, what we don't know, and what we want to find. In the ray equation we don't know $\vec{p}_r(t)$ or t. In the plane equation we don't know $\vec{p}$. So this would seem like we have 2 equations with 3 unknowns.

Ray vs. Plane

We had a third equation we forgot about.

Given:

$$\vec{p}_r(t) = \vec{s}_r + \vec{d}_r$$
$$\vec{n} \cdot (\vec{p} - \vec{p}_0) = 0$$
$$\vec{p} = \vec{p}_r(t)$$

Substitute:

$$\vec{n} \cdot (\vec{p} - \vec{p}_0) = 0$$
$$\vec{n} \cdot (\vec{p}_r(t) - \vec{p}_0) = 0$$
$$\vec{n} \cdot (\vec{s}_r + \vec{d}_r t - \vec{p}_0) = 0$$

Solve for t

In actuality we don't have 2 equations with 3 unknowns, we forgot one equation. We're trying to solve for the point of intersection between the ray and plane. Another way to look at this is we're trying to solve for the spot where $\vec{p} = \vec{\mathrm{p}}_r(t)$. With this third equation we can now substitute and get 1 equation with 1 unknown: t.

Ray vs. Plane

What do we have to consider before finishing?

1. When can this fail to give a t-value?

$$t = \frac{\vec{n} \cdot (\vec{p}_0 - \vec{s}_r)}{\vec{n} \cdot \vec{d}_r}$$

It's easy to solve the previous equation by first distributing the dot product through then simply re-arranging.

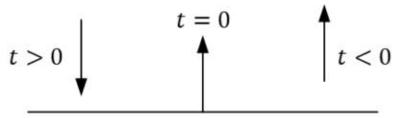
This now gives a t-value which tells us where the ray intersects. Before finishing there's 2 things we have to consider.

1. Does this ever fail to give a t-value?
Logically this always works unless the denominator $\vec{n} \cdot \vec{d}_r$ is zero. So when is this value zero and what does this mean geometrically? Remembering that the dot product formula is $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}|\cos(\theta)$ we can see that this is only zero if $\cos(\theta)$ is zero (assuming non-zero vectors). This can only happen when the two vectors are perpendicular. So if $\vec{n}$ and $\vec{d}_r$ are perpendicular then this means the plane normal is perpendicular to the ray direction, or rather the ray is parallel to the plane. In this case we should return no intersection.

Ray vs. Plane

What do we have to consider before finishing?

2. Are all values of t valid?



2. Are all t-value valid?
Remember that a ray is only defined for the range of $[0, \infty]$. This means the ray doesn't intersect the plane when $t < 0$ as the plane is behind the ray.

Ray vs. Triangle

A triangle defines a plane

We know how to test Ray vs. Plane

If we can define Point vs. Triangle we know Ray vs. Triangle

How do we test Point vs. Triangle?

There's a few different methods of computing Ray vs. Triangle, but I'll cover a simple one that overlaps the most with other useful topics.

Assuming we have a non-degenerate triangle, this triangle defines a plane. We already know how to test Ray vs. Plane and get an intersection point. Perhaps we can use this result to test Ray vs. Triangle? All we'd have to do is figure out how to test if a point is inside a triangle.

Barycentric Coordinates - Triangle

Barycentric coordinates are defined as:

$$\vec{P} = u\vec{A} + v\vec{B} + w\vec{C}$$
$$u + v + w = 1$$

1 coordinate is redundant:

$$w = 1 - u - v$$

If $0 \leq u, v, w \leq 1$ then $\vec{P}$ is inside the triangle

How do we compute u and v?

We'll use barycentric coordinates to determine if a point is inside a triangle.

Barycentric coordinates parametrize space represented by a set of points. You can think of them as a weighted combination of the points. An important rule of barycentric coordinates is that they must add up to 1, this means that one coordinate is redundant. In the case of a triangle, only 2 barycentric coordinates are needed as the 3rd can be computed from the other two.

An important property of barycentric coordinates is that if all 3 coordinate values are between 0 and 1, then the point they define is inside the triangle. So if we check the coordinates of our point then we can easily solve Point vs. Triangle. The problem is that we have the equation for $\vec{P}$ given $u, v,$ and w, how do we invert this equation?

Barycentric Coordinates - Triangle

How do we solve $\vec{P} = u\vec{A} + v\vec{B} + w\vec{C}$?

We have 3 equations and 3 unknowns!

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = u \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} + v \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} + w \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix}$$

To solve for the coordinates, we have to look again at what we know. We have 3 equations and 3 unknowns so we should be able to solve this easily right?

Barycentric Coordinates - Triangle

Re-arrange to make life easier:

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$

And now we can simply invert:

$$\begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{bmatrix}^{-1} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$

Any issues?

When is a matrix inverse not defined?

We can do a little re-arranging of the previous equation to make life easier. Instead of look at this as 3 vectors being scaled and added together, we can look at this as a matrix times vector. With this equation we have all of our knowns in the matrix and unknowns in the vector, so we can just invert to solve for our unknowns! Easy right?

Before we finish we should check and see if there are any problems? The only problem would be if this matrix is un-invertible. So this gives the question, when is a matrix not invertible?

Barycentric Coordinates - Triangle

A matrix $\boldsymbol{M}$ is invertible if and only if its determinant is non-zero.

Is this matrix's determinant always non-zero? $\begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{bmatrix}^{-1}$

Hopefully you should remember that a matrix is invertible if and only if its determinant is non-zero.

I can write out the standard equation of a determinant, but I'd be hard pressed to tell you why this matrix would ever be non-zero.

Barycentric coordinates - Triangle

Scalar Triple Product:

$$det\begin{pmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{pmatrix} = \vec{A} \cdot (\vec{B} \times \vec{C})$$

When is this zero?

What is wrong with this formula?

Fortunately, if you look around it's not hard to find a handy rule called the scalar triple product. This rule relates the determinant of a matrix to a dot and cross product of its column vectors.

Now I can actually intuit if this determinant is ever zero and give geometric examples why it is.

- If A, B, or C are zero
- If B and C are parallel
- If B and C are on a plane perpendicular to A

All of these are reasons why we cannot use this solution. So the question is, what's wrong with this formula? Why does it blow up?

Barycentric coordinates - Triangle

We thought we had 3 equations and 3 unknowns…

We actually have 4 equations and 3 unknowns

$$\vec{P} = u\vec{A} + v\vec{B} + w\vec{C}$$
$$u + v + w = 1$$

How do we solve now?

Well if we had 3 equations and 3 unknowns we should've been able to solve. Upon closer inspection we actually had a 4th equation we forgot about, the one specifying that all of the coordinates have to add up to 1. This means we had an overly constrained problem (https://en.wikipedia.org/wiki/Overdetermined_system). These types of systems almost always have some problem that causes the solution to be non-robust.

We should be able to fix this though.

Barycentric coordinates - Triangle

Knowing: $w = 1 - u - v$

$$\vec{P} = u\vec{A} + v\vec{B} + (1 - u - v)\vec{C}$$

Then re-arrange: $\vec{P} - \vec{C} = u(\vec{A} - \vec{C}) + v(\vec{B} - \vec{C})$

Now we have 3 equations and 2 unknowns…

Our 4th equation shared some unknowns with the 1st three so we can re-arrange and substitute. Without loss of generality, we'll choose to substitute in and replace w. From here we can distribute and re-arrange all knowns on the left and unknowns on the right.

Now we've successfully reduce the number of equations down to 3, but we also reduced the number of unknowns down to 2…

Barycentric coordinates - Triangle

First define:

$$\vec{v}_0 = \vec{P} - \vec{C}$$
$$\vec{v}_1 = \vec{A} - \vec{C}$$
$$\vec{v}_2 = \vec{B} - \vec{C}$$

Now we have: $\vec{v}_0 = u\vec{v}_1 + v\vec{v}_2$

Can turn this into 2 equations by projecting on $\vec{v}_1$ and $\vec{v}_2$

$$\vec{v}_0 \cdot \vec{v}_1 = u(\vec{v}_1 \cdot \vec{v}_1) + v(\vec{v}_2 \cdot \vec{v}_1)$$
$$\vec{v}_0 \cdot \vec{v}_2 = u(\vec{v}_1 \cdot \vec{v}_2) + v(\vec{v}_2 \cdot \vec{v}_2)$$

To solve this we'll first do some re-labeling to make life easier. We'll take our 3 vector subtractions and label them as one vector since they're constant. This gives the equation: $\vec{v}_0 = u\vec{v}_1 + v\vec{v}_2$.

Now we can pull a standard math trick of multiplying both sides by a constant. In this case we'll choose to multiply the equation by $\vec{v}_1$ (here multiply means the dot-product). Doing so will turn our 3 equations into 1 equation while still having 2 unknowns. We can make a second equation by doing the same thing but with $\vec{v}_2$. Now we finally have 2 equations and 2 unknowns.

There's a few different ways to solve this, but the simplest is to use Cramer's rule.

Cramer's Rule

$$ax + by = e$$
$$cx + dy = f$$

$$x = \frac{\begin{vmatrix} e & b \\ f & d \end{vmatrix}}{\begin{vmatrix} a & b \\ c & d \end{vmatrix}} \qquad y = \frac{\begin{vmatrix} a & e \\ c & f \end{vmatrix}}{\begin{vmatrix} a & b \\ c & d \end{vmatrix}}$$

Cramer's rule is a simple method to solve a system of equations. In particular, for a 2x2 system it's quite easy. To see how this solution is reached first we multiply equations 1 and 2 by d and b respectively:

$$adx + bdy = ed$$
$$bcx + bdy = bf$$

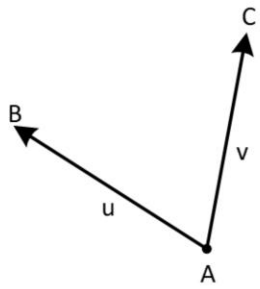
We can then subtract the two equations to get:

$$adx - bcx = ed - bf$$
$$(ad - bc)x = ed - bf$$

The same approach can be done for y.

This method can be extended for higher dimensions, but it is not recommended for use above 3 equations as a lot of calculations are wasted.

Barycentric coordinates - Triangle



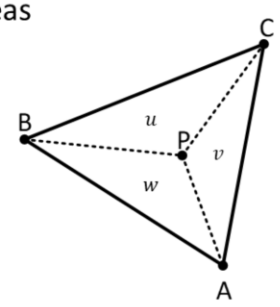
u and v are a ratios on the edges $(\vec{B} - \vec{A})$ and $(\vec{C} - \vec{A})$

It's worth noting that what we've done is represented a point on the triangle as a linear combination of two of the edges of the triangle. This information can be useful later if you want to determine when one of the coordinates is negative which side we're on the reverse side of.

Barycentric coordinates (areal coordinates)

Method 3: Signed triangle area ratio

Coordinates are proportional to signed ratio areas



How do we get the area of a triangle?

Barycentric coordinates are also known as areal coordinates. This is because a barycentric coordinate is proportional to the signed area of the sub-triangle it defines.

So if we can compute the ratio of a sub-triangle with respect to the total triangle then we can compute a barycentric coordinate.

Barycentric coordinates (areal coordinates)

Cross product defines the area of a parallelogram

Area of a triangle is: $A = \frac{1}{2}\left|(\vec{B} - \vec{A}) \times (\vec{C} - \vec{A})\right|$

What about the signed area?

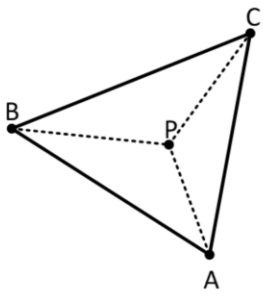
If you remember your cross product identities then it should be easy to compute the area of a triangle. The cross product of two vectors defines the area of a parallelogram with the two vectors for edges. The triangle area is just half of that. The problem is that we need signed areas so we can represent negative coordinates.

Barycentric coordinates (areal coordinates)

The sub-triangle PBC defines the normal $\vec{N}_{PBC}$

Now we can define signed area:

$$SA = \frac{1}{2}\vec{N}_{PBC} \cdot \frac{\vec{N}_{ABC}}{|\vec{N}_{ABC}|}$$



If the winding order flips, so does the sign

The signed area of a triangle can be used by using the dot product to compare it with a known direction. The cross product $\vec{N}_{PBC} = (\vec{B} - \vec{P}) \times (\vec{C} - \vec{P})$ gives the normal of this sub-triangle. We know what this sub-triangle's normal should face the same direction as the original triangle's normal $\vec{N}_{ABC}$. If it faces the other direction then the sub-triangle's winding order flipped relative the original triangle and we want a negative area.

So we can define a signed area by multiplying by 1 or -1 depending on the direction of the normal:

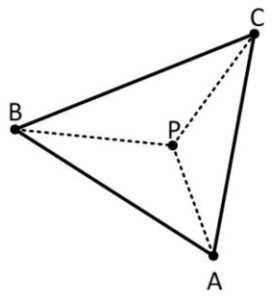
$$SA(PBC) = \frac{1}{2}\vec{N}_{PBC} \cdot \frac{\vec{N}_{ABC}}{|\vec{N}_{ABC}|}$$

Barycentric coordinates (areal coordinates)

Barycentric coordinates are a ratio of singed areas

$$u = \frac{SA(PBC)}{SA(ABC)}$$

$$v = \frac{SA(PCA)}{SA(ABC)}$$



These can be simplified: $u = \frac{\vec{N}_{PBC} \cdot \vec{N}_{ABC}}{\vec{N}_{ABC} \cdot \vec{N}_{ABC}}$

Now we can finally define a barycentric coordinate. As previously stated, a barycentric coordinate is a signed ratio of the sub-triangle it defines with respect to the original triangle.

As u is related to point A, it defines the sub-triangle of PBC (check the sub-triangles when u is 1). Therefore u is the ratio PBC's signed area with respect to the original triangle. This expands to:

$$u = \frac{\frac{1}{2}\vec{N}_{PBC} \cdot \frac{\vec{N}_{ABC}}{|\vec{N}_{ABC}|}}{\frac{1}{2}\vec{N}_{ABC} \cdot \frac{\vec{N}_{ABC}}{|\vec{N}_{ABC}|}}$$

Now we can simply cross out any constants to get $u = \frac{\vec{N}_{PBC} \cdot \vec{N}_{ABC}}{\vec{N}_{ABC} \cdot \vec{N}_{ABC}}$. Similar optimizations can be done for v.

Barycentric coordinates - Line

$$\vec{P} = u\vec{A} + v\vec{B}$$
$$u + v = 1$$

How do we compute the barycentric coordinates of a line?

2 main approaches like before:

- Analytic
- Geometric

Just as we can compute the barycentric coordinates of a point with respect to a triangle, we can also compute the coordinates for a line. In fact, this is likely to be something you already have done (maybe without knowing it) so that you can interpolate between two positions.

There's two main approaches to go about computing this: an analytic and a geometric approach.

Barycentric coordinates – Line (Analytic)

Solve like before:

$$\vec{P} = u\vec{A} + v\vec{B}$$
$$\vec{P} = u\vec{A} + (1-u)\vec{B}$$
$$\vec{P} - \vec{B} = u(\vec{A} - \vec{B})$$

Multiply both sides by $(\vec{A} - \vec{B})$:

$$(\vec{P} - \vec{B}) \cdot (\vec{A} - \vec{B}) = u(\vec{A} - \vec{B}) \cdot (\vec{A} - \vec{B})$$
$$\frac{(\vec{P} - \vec{B}) \cdot (\vec{A} - \vec{B})}{(\vec{A} - \vec{B}) \cdot (\vec{A} - \vec{B})} = \mathrm{u}$$

We can do the same thing for a line that we did for a triangle. First we should see that we have 4 equations and 2 unknowns. We can substitute just as before to get 3 equations and 1 unknown.

Now we can multiply both sides by a vector just like before. This time we'll multiply both side by $(\vec{A} - \vec{B})$. Now we have 1 equation and 1 unknown and we can easily solve for u.

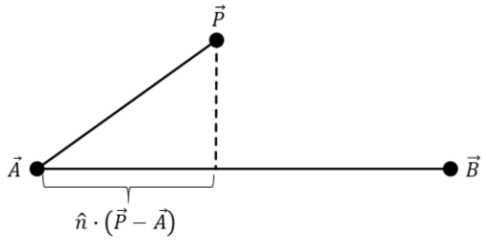
Barycentric coordinates – Line (Geometric)

First let $\vec{n} = (\vec{B} - \vec{A})$

Compute $\hat{n} = \frac{\vec{n}}{|\vec{n}|}$

Project $\vec{P}$ onto the and solve

$$v = \frac{\hat{n} \cdot (\vec{P} - \vec{A})}{|\vec{B} - \vec{A}|}$$



*Divide by $|\vec{B} - \vec{A}|$ to "normalize" v

The idea of the geometry approach is similar to using signed areas like we did for triangles, only instead of area it's signed length.

The main idea is to compute the projection of our query point onto the line segment using the dot-product. To do this, we need to remember the identity: $\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}|\cos(\theta)$. To correctly project $\vec{P}$ we have to normalize $\vec{n}$.

Now we can solve for one of the coordinates as we know the length projected onto the line is described by $\vec{n} \cdot (\vec{P} - \vec{A})$. This however gives us the length of the projection, we want an interpolant between 0 and 1. To get this we simply divide again by the length of the line segment to finally get the equation for v. Note that this is the equation for v not u.

Misc. Barycentric coordinates facts

Can map points between different shapes

Can map points between spaces (including projection)

Can interpolate values (actual triangle rasterization)

A few miscellaneous points about barycentric coordinates and why they're super useful (some of these we'll see later).

One of the most important properties of barycentric coordinates is that they allow us to map points between different shapes. Given one triangle and a point we can compute the "same" point on some other arbitrary triangle by using the barycentric coordinates. This can be useful to map anchor points, un-project points, etc… we'll see this application in particular when covering GJK.

Another useful property of barycentric coordinates is that they can be used to interpolate any value across the surface of a triangle. Not only this, but they're what the GPU actually uses to draw triangles. Back in the day you probably learned the scanline approach, but this isn't actually used in practice. There's a number of reasons for this, but see this blog post and the next few articles for details: https://fgiesen.wordpress.com/2013/02/06/the-barycentric-conspirac/
The basic idea is that barycentric coordinates allow high parallelization unlike scanline approaches.

Ray vs. Sphere

Given:

Ray: $\vec{p}_r(t) = \vec{s}_r + \vec{d}_r t$

Sphere: $(\vec{c}_s - \vec{p})^2 - r^2 = 0$

$\vec{p}_r(t) = \vec{p}$

Substitute:

$$\left(\vec{c}_s - \left(\vec{s}_r + \vec{d}_r t\right)\right)^2 - r^2 = 0$$

We can use the quadratic formula if we re-arrange to:

$$at^2 + bt + c = 0$$

Just as with Ray vs. Plane, we can look and see that we have 3 equations with 3 unknowns. From here we can plug the ray equation into the sphere equation. This gives us a quadratic equation to solve for t. This tends to be a blocking point for lots of students though, as they don't know how to solve this equation. If we can re-arrange this equation to a standard quadratic equation then we can use the quadratic formula to solve, but how?

Ray vs. Sphere

Given:

$$\left(\vec{c}_s - \left(\vec{s}_r + \vec{d}_r t\right)\right)^2 - r^2 = 0$$

How do we expand a 3-term square?

$$(a + b + c)^2 = ?$$

Alternatively, we can group knowns together:

$$\left(\vec{m} - \vec{d}_r t\right)^2 - r^2 = 0$$

Well there's 2 ways to solve, the first is to just expand a 3-term square. This isn't hard but most people don't know how. A regular 2-term quadratic just multiplies every term by every other: $a * a + a * b + b * a + b * b$. The same is true for an n-term versions.

There is an easier way that also reflects what we'll tend to write in code. In the squared term we have a value multiplied by t and then the other 2 terms. We can set $\vec{m} = \vec{c}_s - \vec{s}_r$ as both terms are constants. This allows us to expand using FOIL as normal.

Ray vs. Sphere – Quadratic Equation

Solve the quadratic equation $at^2 + bt + c = 0$
with the quadratic formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

What cases do we need to consider?

Now that we can get a quadratic equation we can solve using the quadratic formula as pictured above. Before blindly applying this we need to know what can cause this to fail.

Ray vs. Sphere – Quadratic Equation

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

When is the denominator 0?

What do the 3 cases of the discriminant (Δ) mean?

$\Delta < 0$

$\Delta > 0$

$\Delta = 0$

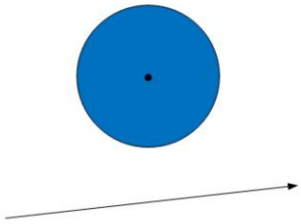
The first case to consider is what happens if the denominator is 0? This can only happen if $a = 0$. If you follow through on the math you would've seen that $a = \vec{d}^2$ which can only be zero if the ray direction is zero.

The second thing to consider is the discriminant. Remember, the discriminant is the portion under the square root, that is $\Delta = b^2 - 4ac$. There's actually 3 cases to consider here, the discriminant being negative, positive and zero. Each of these means something useful to us.

Ray vs. Sphere – Quadratic Equation

Case 1: $\Delta < 0$

There is no solution (in Euclidean space)



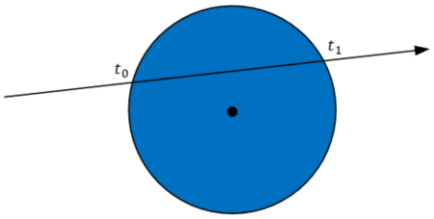
The line doesn't hit the sphere!

The first case is if the discriminant is negative. In this case we have to take the square root of a negative number. Well this would give us an imaginary number, technically meaning the solution is in complex space…but let's just stick to Euclidean space. This means that there is no solution and hence the ray's line doesn't hit the sphere.

Ray vs. Sphere – Quadratic Equation

Case 2: $\Delta > 0$

There are 2 solutions



The line hits the sphere in 2 spots

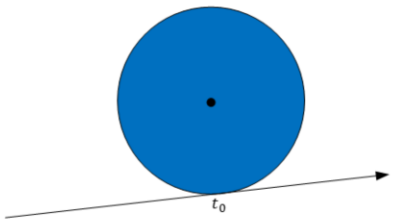
The second case is when the discriminant is positive. As the quadratic equation is $\pm\Delta$ that means we have 2 answers. What does this mean in terms of our ray vs. sphere? We have 2 intersection times as the line hits the sphere in 2 places.

Do note here that the smaller t value will always be defined by $-\Delta$ (although not necessarily the first t value as shown later).

Ray vs. Sphere – Quadratic Equation

Case 3: $\Delta = 0$

There is only 1 solution

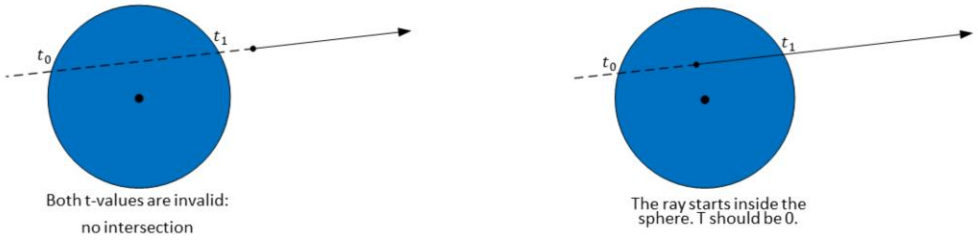


The line is tangent to the sphere

Finally, if $\Delta = 0$ then there is exactly 1 solution (± 0 gives only one result). This means that there is exactly 1 place the line hits the sphere, or rather that the line is tangent to the sphere.

Ray vs. Sphere – Invalid t-values

Important! $\Delta \geq 0$ does not guarantee a "correct" t-value!



A t-value can be behind the ray! All negative t-values are invalid!

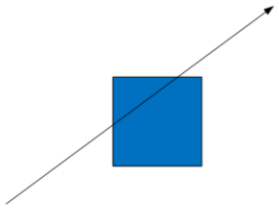
It's important to realize that just because $\Delta \geq 0$ there is no guarantee that either of the t-values are correct. The easiest case to think about is if the sphere is behind the ray. In this case there is no intersection with the ray (only the infinite line segment).

The other important case is if the ray's origin starts within the sphere. In this case one of the t-values will be positive and the other will be negative. While it's debatable what the t-value returned should be for this class the result should be 0 (negative is never correct and the first time the ray hit's the sphere is at t=0).

These cases can be determined by further inspecting the values of b and c in the quadratic formula.

Ray vs. Aabb

There's no equation for an Aabb



Perform each axis test independently

Combine the results afterwards

Unlike a sphere, there's no analytic form for an aabb. So instead we'll continue the usual pattern with an aabb: we'll test each axis independently and combine the results. By testing each axis independently I mean we'll find when we intersect the planes defined by each axis.

Ray vs. Aabb

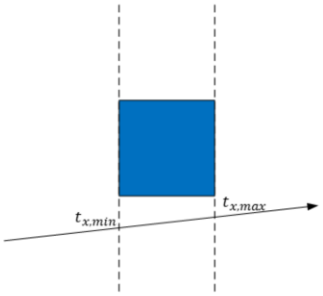
Each axis has 2 planes

Need to compute a min/max range for each axis

For the x-axis:

$$\vec{n} \cdot \left(\vec{s}_r + \vec{d}_r t_{x,max} - \vec{p}_{max}\right) = 0$$
$$\vec{n} \cdot \left(\vec{s}_r + \vec{d}_r t_{x,min} - \vec{p}_{min}\right) = 0$$

$$\vec{n} = \begin{bmatrix}1\\0\\0\end{bmatrix}$$



Don't do this!

As each axis has 2 plane values we'll have to solve for a t-min and a t-max on that axis.

The obvious way to do this is to re-use Ray vs. Plane where the plane is defined by the min/max point on the aabb and the plane's normal (implicit from the axis).

Without loss of generality, the we'd solve the plane equation $\vec{n} \cdot \left(\vec{s}_r + \vec{d}_r t - \vec{p}_0\right) = 0$ where the normal is the x-axis. As there's 2 t-values to solve for, one would set $\vec{p}_0$ to the aabb's min and the other would use the aabb's max.

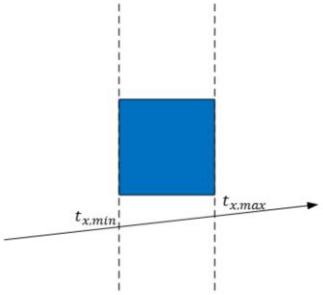
This is very easy to do but we shouldn't stop here, we can make this a lot better (and fix a few problems).

Ray vs. Aabb

Each axis is independent, why are we using the full vector equation?

Since $\vec{n} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

$\vec{n} \cdot (\vec{s} + \vec{d}t - \vec{p}) = 0$
becomes
$s_x + d_x t - p_x = 0$



If we look at this equation we'll see that a lot of wasted calculations are happening. This is because we know that the plane's normal is only along 1 axis. We can take advantage of this and avoid all of the extra zero multiplications.

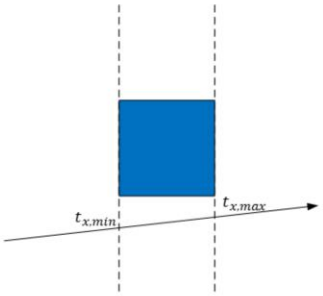
Ray vs. Aabb

Now any axis can define:

$$t_{i,min} = \frac{p_{i,min} - s_i}{d_i}$$

$$t_{i,max} = \frac{p_{i,max} - s_i}{d_i}$$

What problems do we have to consider?



After multiplying out through constants we can arrive at a very simple equation for t on an axis given the plane's start value on that axis.

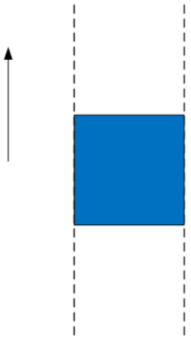
As always we have to stop and see what problems can arise.

Ray vs. Aabb – Edge Cases

Problem 1: What if $d_i = 0$?
The ray is parallel to the plane

Case 1: The ray might be outside the aabb

$s_i < min_i$ or $max_i < s_i$



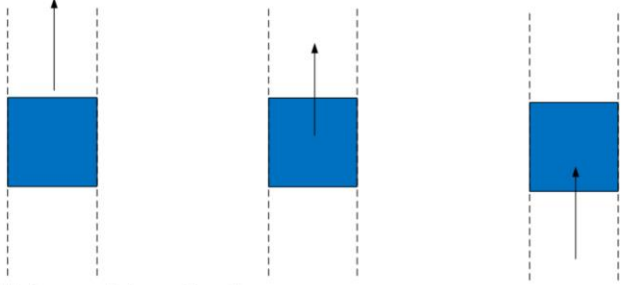
The first thing to consider is zero divisions. We divide by zero if d_i is zero. If d_i is zero then the ray is parallel to this axis and we can't compute an intersection point. In previous intersection tests this meant that the ray didn't intersect, but here we can't derive an exact meaning.

The ray can be outside the min or max plane in which case it can't intersect the aabb. We can determine this by checking the ray's start point against the aabb, effectively checking 1 axis for point in aabb.

Ray vs. Aabb – Edge Cases

Case 2: The ray is inside the aabb

$min_i \leq s_i \leq max_i$?



We can't tell from this axis alone

Skip this axis and defer to the remaining axes

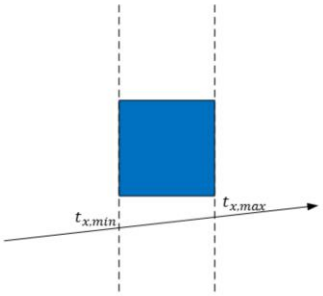
If the ray's start is inside the aabb we have 3 cases that could happen. We could try and determine which case we're in but we should take a step back and think about what we're after. We want to know if we're colliding on this axis and this axis alone. We could easily stop and correctly return true, but we return more than just a Boolean. We also have to consider the t-values for intersection on this axis which will be important for getting the full 3D t-values. The t-values are effectively $-\infty$ and ∞ which we could use, but more typically we'll just skip this axis and defer to the t-values from the remaining axes.

Ray vs. Aabb – Edge Cases

Problem 2: Are t_{min} and t_{max} always right?

$$t_{i,min} = \frac{p_{i,min} - s_i}{d_i}$$

$$t_{i,max} = \frac{p_{i,max} - s_i}{d_i}$$



Is there ever a case where this is wrong?

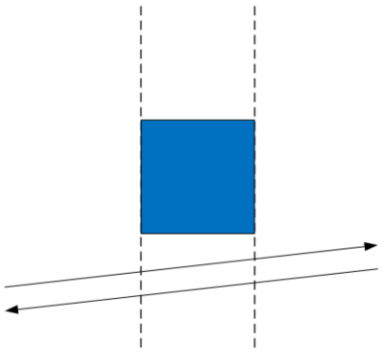
The second thing to consider is our t_{min} and t_{max} values. We've assumed that the min t-value will always come from the aabb's min point. Unfortunately this isn't true.

Ray vs. Aabb – Edge Cases

Consider the ray's direction

$$\vec{d}_i > 0 \rightarrow \begin{cases} t_{min} = t(min_i) \\ t_{max} = t(max_i) \end{cases}$$

$$\vec{d}_i < 0 \leftarrow \begin{cases} t_{min} = t(max_i) \\ t_{max} = t(min_i) \end{cases}$$

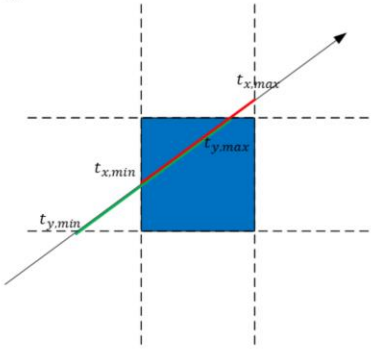


If the ray was travelling from left-to-right then we computed the correct t-values, but if the ray was right-to-left then we flipped min/max. Luckily this is easy to fix just by checking the ray's sign on the axis.

Ray vs. Aabb

Now we have all the axis results

How do we them together?



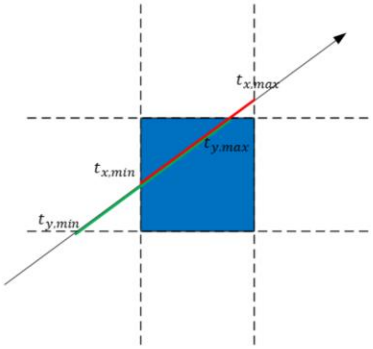
After we have each axis' min/max t-values we need to figure out how to combine them to get the actual min/max t-values.

Ray vs. Aabb

We want the last min and the first max t-values

$$t_{min} = \max(t_{i,min}) = t_{x,min}$$

$$t_{max} = \min(t_{i,max}) = t_{y,max}$$

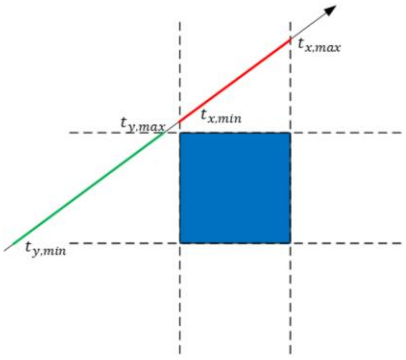


We have 3-different axes and we know for each when we start and stop intersecting. Logically, we can't actually start colliding with the final aabb until we're colliding on all axes simultaneously. This means we can take the last (or max) of the axis min t-values to find t_{min}. Similarly, once we stop intersecting on one axis we're no longer intersecting the aabb, so t_{max} is the min of the max t-values.

Ray vs. Aabb

What happens when there's no intersection?

$t_{min} = t_{x,min}$
$t_{max} = t_{y,max}$



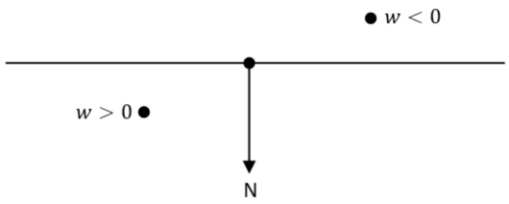
$t_{min} > t_{max} \Rightarrow$ no intersection

If we use compute t_{min} and t_{max} this way then how do we detect non-intersection? If we look at a picture of this case it should become clear. Here we'll compute a t_{min} that's greater than t_{max}, or an empty range. In this case we never intersected on all of the axes at the same time.

Plane vs. Point

Compute the distance from the plane:

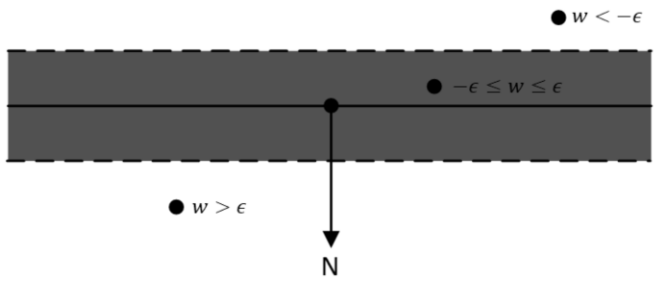
$$\vec{n} \cdot \vec{p} - d = w \qquad \text{or} \qquad \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} \cdot \begin{bmatrix} p_x \\ p_y \\ p_z \\ -1 \end{bmatrix} = \text{w}$$



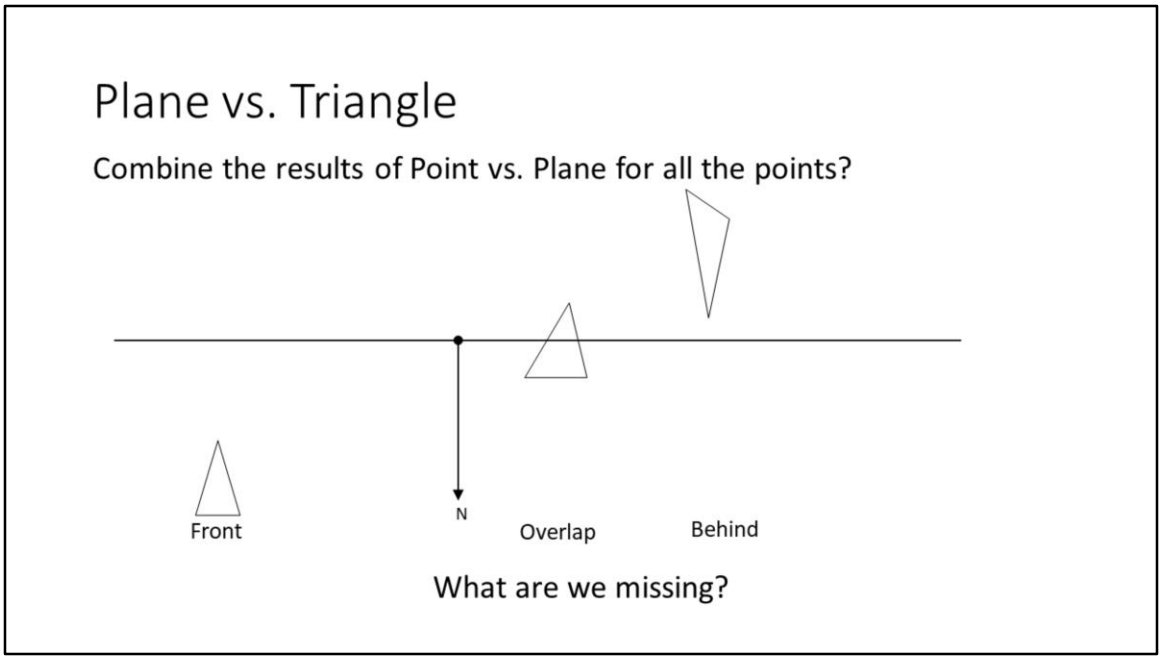
To classify a point against a plane we just plug the point into the plane equation. The result will give us the signed distance from the plane which I'm calling w here. If this value is positive then the point is on the positive side of the plane, otherwise its on the back side.

Plane vs. Point

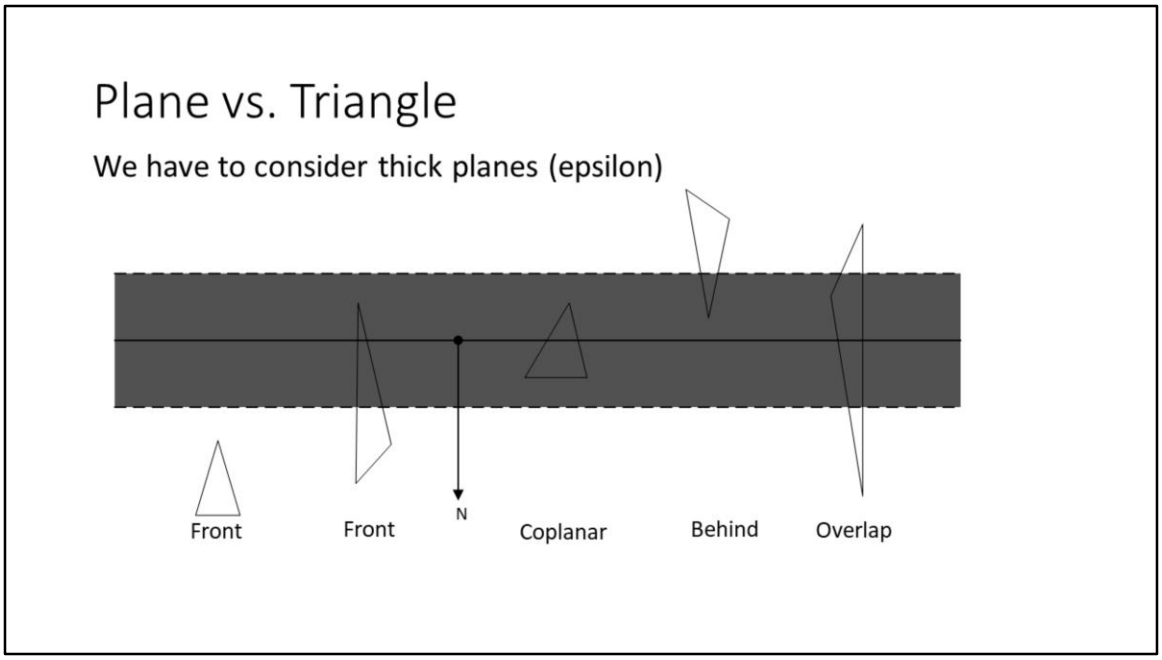
What about numerical robustness (thick planes)?



Unfortunately this won't be robust due to floating point numbers. Typically this is overcome by using a thick plane, i.e. adding a coplanar epsilon threshold.



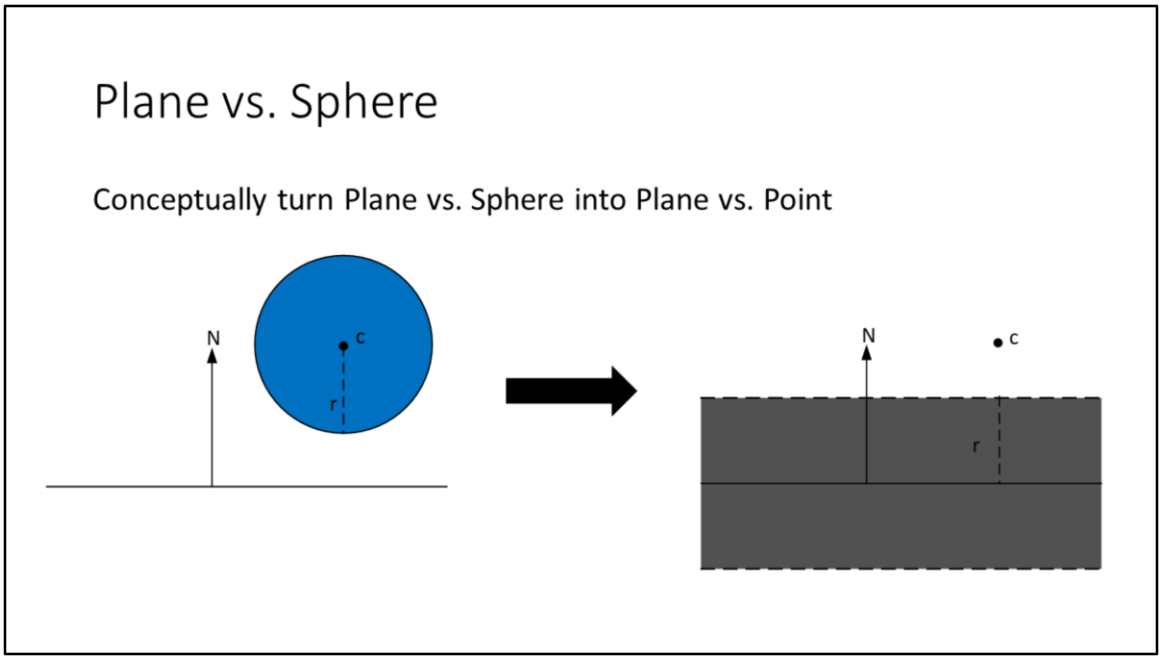
Plane vs. triangle is just classifying all 3 points and determining if all the points are on one side or a combination. Unfortunately things get a little more complicated when using a thick plane.



We have to use a thick plane to deal with robustness. This adds a coplanar state to a point to consider. Luckily this is easy to deal with still if we keep one thing in mind: a point being coplanar doesn't change the triangle's state. If all points are coplanar then the point is coplanar, otherwise the triangle isn't affected by the coplanar point.

It's important to note that points can still be technically behind the plane but within the coplanar epsilon and it won't affect the triangle's classification (as shown by the second front triangle).

There's a handy bit-field tricky to make life easier here. Take a look at the framework and see what the values of coplanar, front, behind, and overlaps are set to.



There's a few different ways to test Plane vs. Sphere, but the easiest is probably to turn the problem into one we already know how to solve. Just as we turned Sphere vs. Sphere into Point vs. Sphere, we can turn this into Plane vs. Point. We already know how to test a point vs. a thick plane so we can just define the thickness of the plane to be the radius of the sphere.

This is equivalent to computing the signed distance of the sphere's center from the plane and using this to determine front, behind, or overlapping.

Plane vs. Aabb

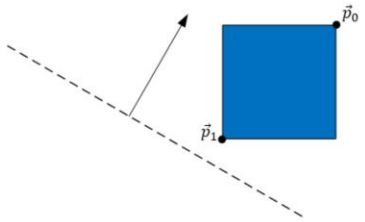
Method 1: Classify all points against the plane

All in-front: Aabb in front
All behind: Aabb behind
Otherwise: Overlaps plane

The simple (and naïve) method of classifying an aabb against a plane is to check all 8 points of the aabb. If all points are on one side or the other then the entire aabb is on that side, otherwise it overlaps the plane. This however is very inefficient and we can do a lot better.

Plane vs. Aabb

Method 2: Classify the extremal points



Only two points actually need to be tested

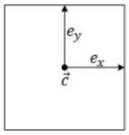
How do we compute these points?

The 2nd method I'll talk about is often called the near and far point method. The key realization here is that only two points actually matter, the two points furthest apart in the direction of the plane normal. These points are often called the near and far point. If we can compute these points more efficiently than projecting every point then we can save a lot.

Plane vs. Aabb

How can we find the point furthest in a direction?

All points can be computed from the center and half-extents



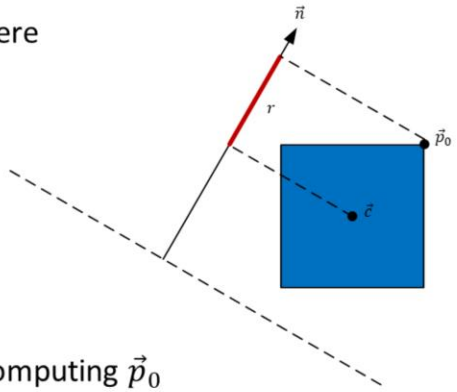
Can determine $+$ or $-$ based upon sign of the vector

To efficiently compute these points we need to look at the problem of finding the point furthest in a direction. This is easiest to do with the center and half-extent method. Any point on the aabb can be computed by adding or subtracting each half-extent.

In order to find the point furthest in a direction we just have to choose the signs for $\vec{c} + Vec3(\pm e_x, \pm e_y, \pm e_z)$. This only requires inspecting the sign of the direction vector $\vec{d}$. As each axis can be independently calculated, if d_i is positive then the point furthest in that direction will be $+e_i$, otherwise it'll be $-e_i$.

Plane vs. Aabb

Method 3: Turn into Plane vs. Sphere



Aabbs are symmetric

A "radius" can be defined

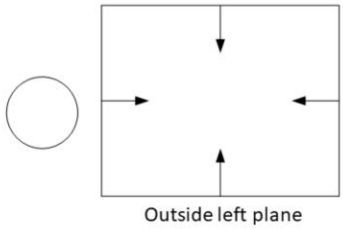
Can compute r directly without computing $\vec{p}_0$

The third method is to turn plane vs. aabb into plane vs. sphere. This can be done by realizing that an aabb is symmetric on any axis. Another way to look at this is to realize the if we compute the furthest point $\vec{p}_0 = \vec{c} + \vec{d}$ (where $\vec{d}$ is or half-extent vector with all the correct signs) then the "near" point is in the opposite direction, i.e. $\vec{p}_1 = \vec{c} - \vec{d}$. This implies that there's a "radius" we can compute for the aabb.

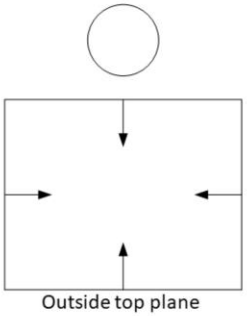
One way to compute this radius would be to compute the projection length of $\vec{p}_0 - \vec{c}$ onto $\vec{n}$. Instead we can compute the projection radius directly. Knowing that $\vec{p}_0 = \vec{c} + Vec3(\pm e_x, \pm e_y, \pm e_z)$ and $r = Dot(\vec{p}_0 - \vec{c}, \vec{n})$ we can simplify as $r = \sum e_i |\mathrm{n_i}|$.

Frustum vs. Sphere Culling

Test all 6 planes:



Outside left plane



Outside top plane

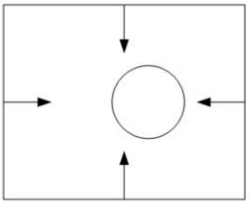
If the sphere is outside any plane then it is outside the frustum

The basic method of testing a frustum vs. a sphere is known as frustum culling. Simply test all 6 planes of the frustum against the sphere then use that to determine if sphere is outside, intersecting, or inside the frustum.

The first and easiest case is if the sphere is outside any plane. In this case the sphere is outside the frustum. Note: the classification of the sphere against the other 5 planes don't matter, the sphere is guaranteed to be outside the frustum.

Frustum vs. Sphere Culling

Test all 6 planes:

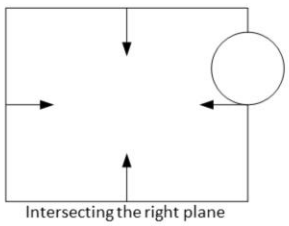


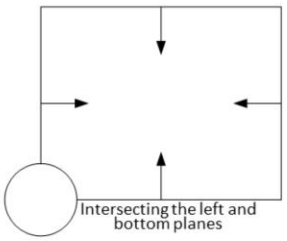
If the sphere is inside all planes then it is inside the frustum

The second case is if the sphere is inside all of the planes (not intersecting any). In this case the frustum completely contains the sphere.

Frustum vs. Sphere Culling

Test all 6 planes:



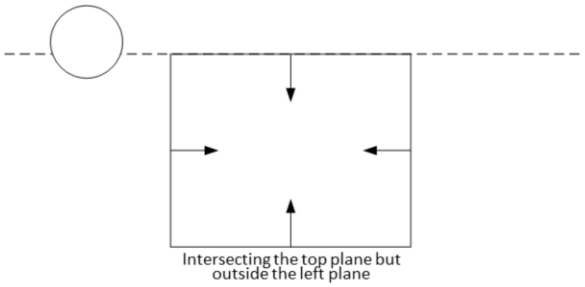


Otherwise if the sphere overlaps any plane then it overlaps the frustum

Finally, if we aren't outside any planes and we're not inside all of the planes then we're overlapping 1 or more planes. In this case the sphere should be classified as intersecting the frustum.

Frustum vs. Sphere Culling

Note: Overlap on one plane does not guarantee an Overlap!!



Do note that just because there is an overlap on one plane does not guarantee that the sphere intersects the frustum. Remember if the sphere is outside any planes then it is outside the frustum! This is an easy case to miss!

Frustum vs. Aabb Culling

Same as sphere, test all 6 planes:

- If outside any return outside
- If inside all return inside
- Otherwise return overlaps

Frustum Aabb is written the exact same as Frustum Sphere, only testing each plane against an aabb instead of a sphere. The exact same cases apply.

Frustum Culling vs. Frustum Intersection

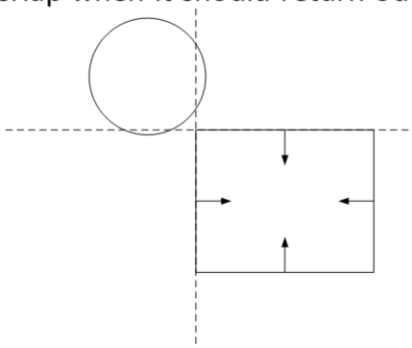
Frustum Culling is an approximation, it gives false positives

Can you think of a case where Frustum vs. Sphere returns the wrong answer?

Unfortunately, the described tests are insufficient for an actual intersection test. There are several scenarios where these tests will return overlap when the shape is actually outside.

Frustum Culling – False Positives

This case returns Overlap when it should return outside



Note: the sphere is not outside any plane!

The easiest case to draw (in 2d) is with a sphere intersecting 2 planes. If you look carefully you'll see that the sphere is not strictly outside any 1 plane even though it is outside the frustum. The same thing can happen to aabbs and any other shape, although they can't easily be drawn in 2d.

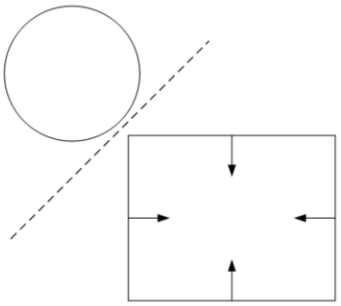
What's missing? Quick look at SAT

Some extra "planes" need to be tested for correctness

Which planes? Well it depends…

Proper solution is defined by SAT (more later)

Basically, if you can draw a line between them they don't intersect



Unfortunately there are more "planes" that would need to be tested to accurately classify the shape. Also, there's no easy answer to what planes, at least at this point in the class.

Looking ahead a bit, the answer is provided by a test called the Separating Axis Test or SAT. This will be defined in detail later, but the basic idea is that if you can draw a line between the two objects without overlapping either one then the shapes do not overlap. There's only an overlap if "all" axes are not separating. How to determine which axis is more complicated and will be talked about later.

Frustum Culling vs. Frustum Intersection

Why not define the proper intersection test?

- More complicated to write
- More computationally expensive
  - Sphere needs 1 more test
  - Aabb needs a total of 26 tests…yes…26

When only culling this is good enough (basic optimizations)

When to use the proper test?

When the exact answer matters! (Picking, etc…)

This begs the question, why even talk about culling? Well the base answer is because when culling false positives are ok. If we're only worrying about quick rejections for performance then it's often ok to incorrectly report true if it saves a lot of computation.

This leads to the next point which is the proper tests are a lot more expensive to write, especially Frustum vs. Aabb. Frustum vs. Aabb requires 26 plane tests to be correct!! How I came across this number is irrelevant for now and will be covered when we talk about SAT.

That being said, it is important to be able to write the actual intersection tests. Most notably, any application where incorrect results will be noticeable. The simplest one I personally have run across is with multi-selection (picking). A user will not find it acceptable to select an object that their selection clearly doesn't hit. In these cases you'll need to write the actual test. Unfortunately this can be very tedious to do per shape, which is why we'll cover some catch all collision detection methods at the end of the class.

Frustum Culling – Temporal Coherence

Once we hit a plane that is outside we can return
- Best case only 1 plane test
- Worst case 6 tests

Temporal Coherence: Objects don't move much from frame to frame
- We can test the planes in any order
- Start with the last plane that returned outside!

One extra implementation detail (not required for assignment 1, but it is for assignment 3). We can write a frustum test a few different ways, but an efficient one (single threaded, no simd, etc…) will early out as soon as the test shape is outside a plane. If the shape is outside one plane then the rest of the planes don't matter. This means in the best case scenario we can return with only one plane test. Our worst case scenario is still the same: testing all 6 planes. This will happen either when the shape is not outside the frustum (overlap or inside) or the last axis is the one we are outside of.

This leads to a simple optimization: Temporal Coherence. The basic idea with temporal coherence is that objects don't move too much from frame to frame, so if we can use a result from the previous frame as a starting guess then we can potentially early out.

So how do we use this? Well imagine the scenario where plane 6 was the plane we were outside of. We simply got unlucky and tested this plane last, but if we had tested it first we would've returned right away. It's important to realize that the order we test the planes doesn't matter as long as we test them all! So after frame 1 where we determine plane 6 was the offending plane we can "seed" frame 2 with this. By

simply storing and passing in this value we can change our plane visiting order to [6, 1, 2, 3, 4, 5] or something similar. The only downside with this is the memory required to store this somewhere (more later).

For the assignment, all you need to do is start with the passed in lastAxis and fill it out with the plane index that the shape was outside of. Note: If the shape wasn't outside any plane then it can be any value.