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

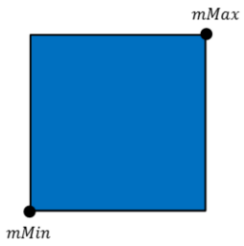
```
struct Triangle
{
  Vector3 mP0;
  Vector3 mP1;
  Vector3 mP2;
};
```

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 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;
};
```

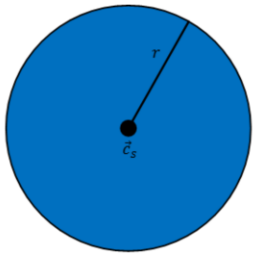


An aabb is a box that is aligned with the cardinal axes, hence we only need to store the min and max on each axis. The other common representation for an Aabb is a center plus a half extent. There are several occasions where this representation is better than min and max but it's easy to convert between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

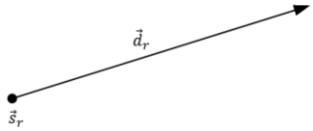


A sphere's representation is fairly straightforward. Just the center of the sphere and the radius. The equation of the sphere's surface is defined as: $(\vec{c}_s - \vec{p})^2 - r^2 = 0$ which simply states that all points that are distance r away from the center are part of the sphere's surface.

Ray

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

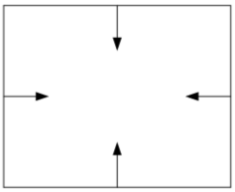
```
struct Ray
{
  Vector3 mStart;
  Vector3 mDirection;
};
```



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 planes as all pointing in or out, but the convention I chose is all of the plane normal pointing 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

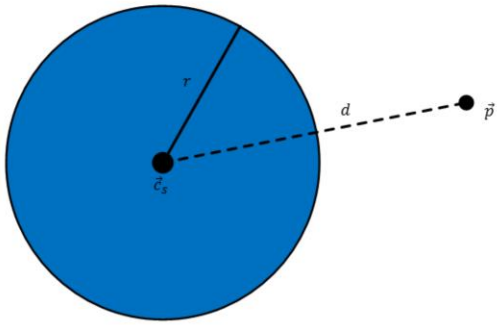
- Coplanar, Outside, Inside, Overlap

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.

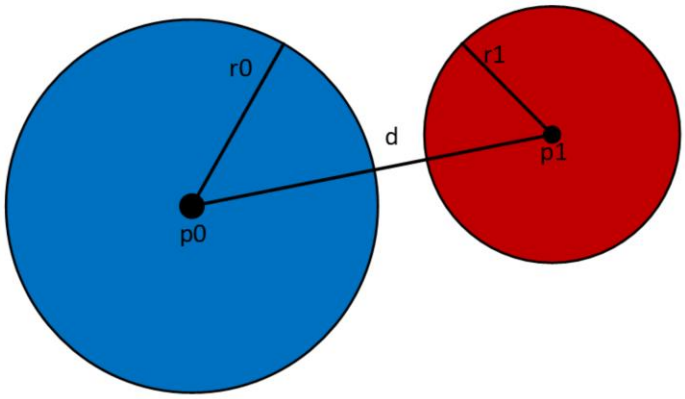
Point vs. Sphere



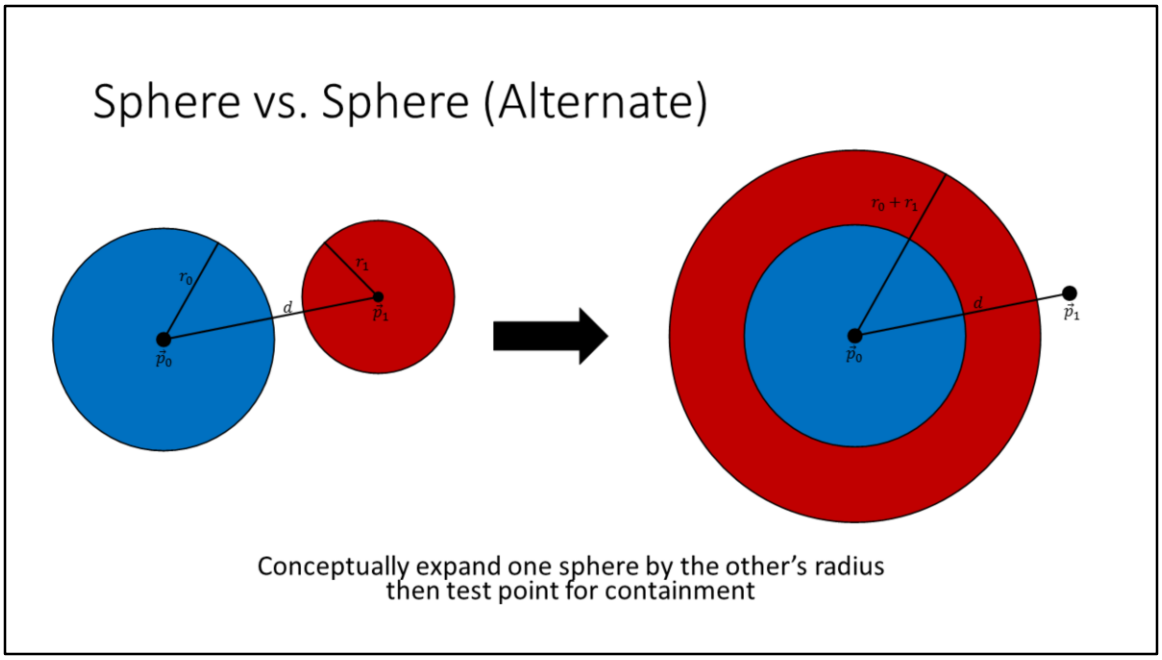
If $d \leq r$ then the point is contained

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: $(\vec{p}_1 - \vec{p}_0)^2 - (\vec{r}_1 + \vec{r}_0)^2 \leq 0$.



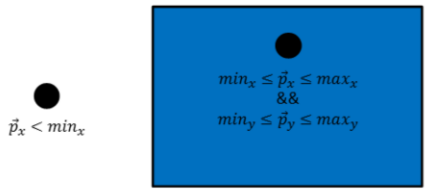
One thing you'll see throughout this class is that there's many different ways to view the same problem. Sometimes viewing a problem in a different way helps to understand/grasp difficult concepts. While sphere vs. sphere isn't a hard concept to grasp I want to present an alternate way to view this problem.

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.

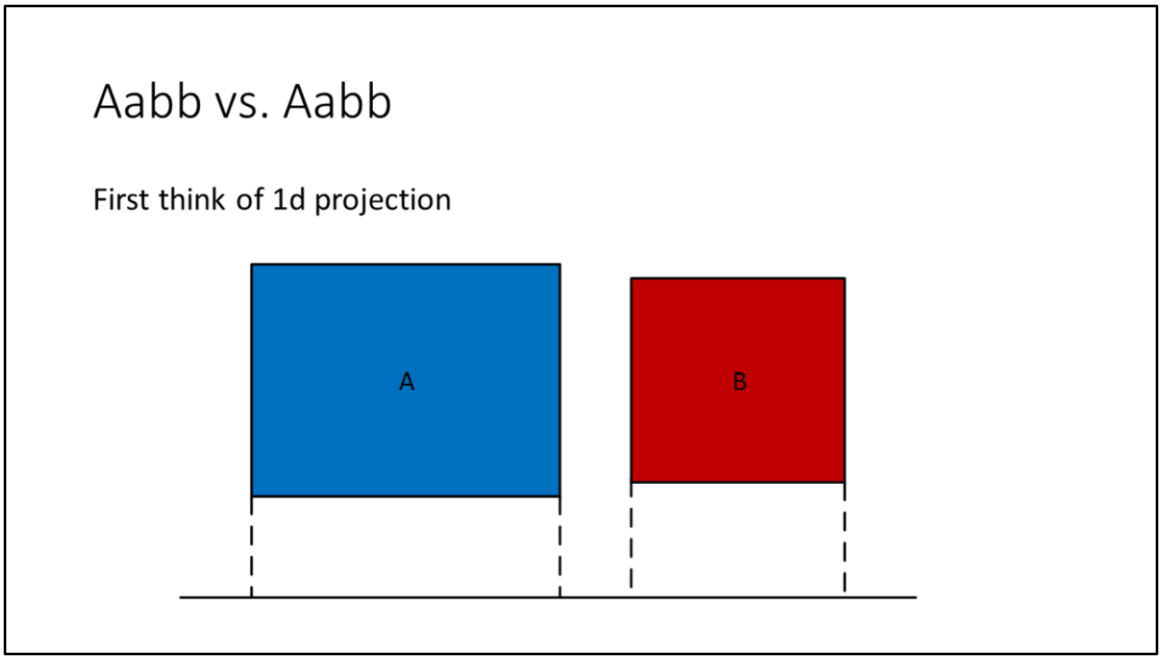
Point vs. Aabb

If the point is between min and max on all axes then it is contained

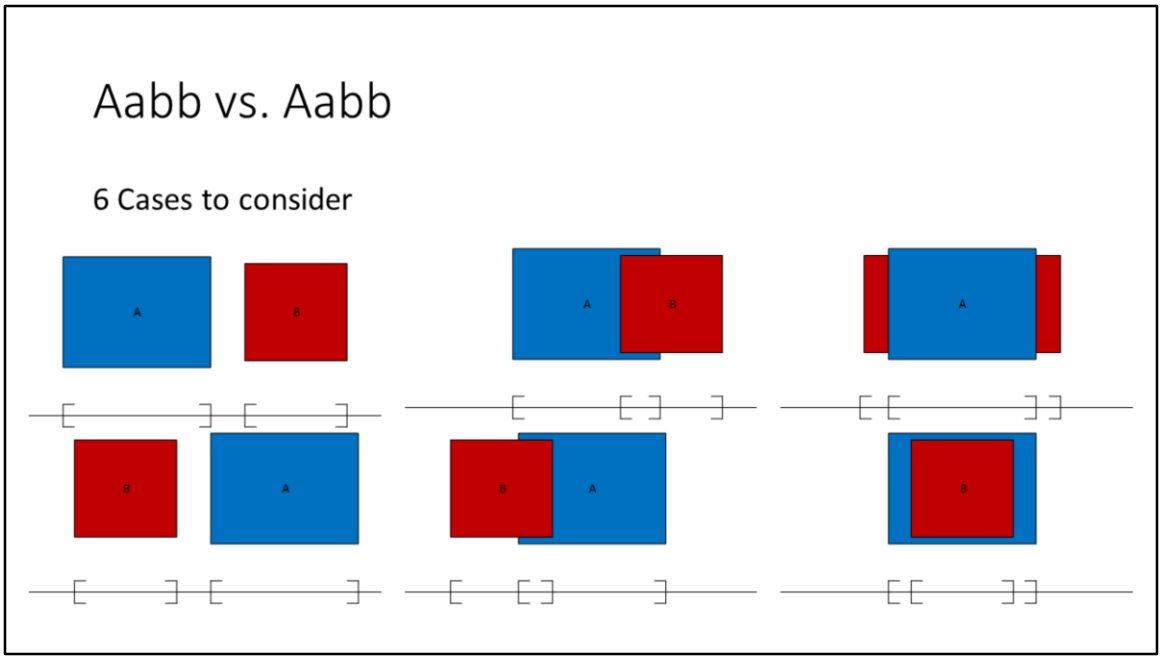


When working with Aabbs there's two important things to keep in mind. As an aabb is not an analytic shape (there's no "equation") most of it's intersection tests are broken down into testing each axis separately and then combining the results. This makes aabbs slightly more difficult conceptually. The other concept is less important for understanding and more for practical usage, the idea of testing for non-intersection instead of intersection.

For a point to be contained in an aabb it needs to be between the min and max of the aabb on all axes. Instead of writing this test, it is much easier and a bit more efficient to write the test for non-intersection. If the point is outside either the min or the max on any axis then the point is not contained in the aabb.



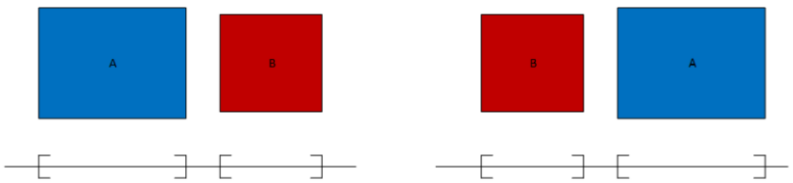
Aabb vs. Aabb is also fairly straightforward to write (with a few tricks to make it easier). It is easiest to first look at this test as testing 2 ranges in 1d. After you understand one axis the test trivially extends.



There are 6 cases to consider for this test. If we wanted to check if they were overlapping we'd need to see if we're in any of the 4 cases on the right. Verifying these cases is annoying and more work than necessary.

Aabb vs. Aabb

Instead of testing for intersection test for non-intersection (SAT, more later)



if($b_{min} > a_{max}$ *or* $a_{min} > b_{max}$)

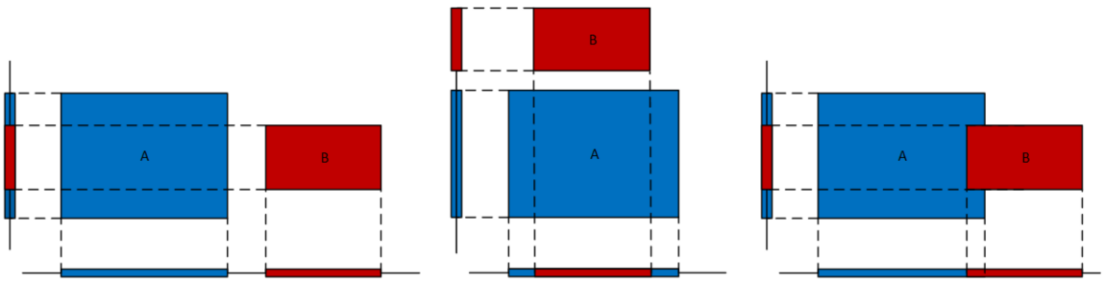
no intersection

Instead of testing for intersection, we can test for non-intersection. If we test all necessary cases for non-intersection and we can't determine non-intersection then they must overlap. This is just two simple checks. This practice of testing for non-intersection instead of testing for intersection is a common technique and the basis of the Separating Axis Test (SAT) that will be explained at the end of the semeseter.

Aabb vs. Aabb

To extend dimensions just check each axis

If any has separation then no overlap



Extending this to properly check in 2 or 3d is trivial. Just perform each axis' non-intersection test. If any axis has a separation then there is no overlap, otherwise the aabb's intersect.

Ray vs. Plane

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

Substitute:
$\vec{n} \cdot (\vec{s}_r + \vec{d}_r t - \vec{p}_0) = 0$
And solve for t

For several intersection tests it's easiest to look at the analytic equation for the shapes involved and substitute. Using the ray equation: $\vec{p}_r(t) = \vec{s}_r + \vec{d}_r t$ and the plane equation: $\vec{n} \cdot (\vec{p} - \vec{p}_0) = 0$, we can substituted and solve for t.
This is done by simply substituting the ray's point into the plane equation to give the equation: $\vec{n} \cdot (\vec{s}_r + \vec{d}_r t - \vec{p}_0) = 0$
After this it is simple to solve for t.

Ray vs. Triangle

First perform Ray vs. Plane

Check if intersection point is inside the triangle

Point inside Triangle?

Barycentric coordinates

Ray vs. triangle is very similar to ray vs. plane, as the triangle defines a plane. Once we determine where the ray hits the plane of the triangle we just need to figure out if the ray hit the triangle itself by determining if the point is within the triangle. The simplest way to go about this is with barycentric coordinates (which are also needed later in the class).

Barycentric coordinates

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

1 coordinate is redundant:

$$w = 1 - u - v$$

How do we solve for u and v?

Barycentric coordinates parametrize space represented by a set of points. You can think of them as a weighted combination of the points. The most 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, where $0 \leq u \leq 1$, $0 \leq v \leq 1$, and $u + v \leq 1$. The redundant 3rd coordinate can thus be express as $w = 1 - u - v$. Computing a point from its barycentric coordinates is simply: $\vec{P} = u\vec{A} + v\vec{B} + w\vec{C}$.

Barycentric Coordinates (2D)

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

$$\begin{bmatrix} P_x - C_x \\ P_y - C_y \end{bmatrix} = \begin{bmatrix} A_x - C_x & B_x - C_x \\ A_y - C_y & B_y - C_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

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

First let's start by looking at computing the barycentric coordinates in 2d. This is quite simple if we use the identity of $w = 1 - u - v$ and then re-arrange. From here we can write it this out as a system of equations, or rather as the vector form of $\vec{b} = \boldsymbol{A}\vec{x}$. From here it is a simple 2d matrix inverse to solve.

Barycentric coordinates

Method 1: Solve 3x3 system

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

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

Simple but not robust:

- Wasted calculations
- Breaks in many cases
- **Do Not Use!!**

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

We can try to do the same thing in 3d as we just did in 2d. With this we get a system of 3 equations and 3 unknowns. Unfortunately, this system is overly constrained as we only have 2 unique variables (remember $w = 1 - u - v$).

We could try to solve this overly constrained system but there's a few major problems. The first problem is that we're wasting computational time by computing w when we could just compute it from u and v. The second problem is tied to the first, we can introduce numerical drift by computing w independently. While solving this should give $u + v + w = 1$ there might be some error. If we instead compute w from u and v then we're guaranteed that the equality will hold.

The final, and most important reason is numerical robustness. There are many situations when the inverse calculation will break even though there is a correct answer. When does an inverse calculation break? When the determinant is zero. The questions is what cases can this happen? The most obvious ones are when the triangle is degenerate, but let's ignore these as they're exceptional cases. To determine more cases it's useful to inspect a property of determinants.

Barycentric coordinates

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?

A very useful formula is the scalar triple product. In particular one nice thing about it is that the result is equivalent to the determinant of the matrix whose row (or column) vectors are the 3 vectors from the product.

So what does this give us? Well it's much easier (in my opinion) to determine when this determinant will be zero by investigating the scalar triple product. One of the simplest cases when the determinant is zero is if any of the vectors is the zero vector. Another case is if all of the points are in one of the cardinal planes. This would cause the cross product to produce a vector perpendicular to $\vec{A}$. Another way to look at this determinant is finding the volume of the tetrahedron centered at the origin with the points $\vec{A}$, $\vec{B}$, and $\vec{C}$.

Because of these reasons we cannot use this formulation for a proper barycentric coordinate calculation. The root problem is that we have 3 equations but only 2 unknowns, to solve this we need to have only 2 equations.

Barycentric coordinates

Method 2: Substitute $w = 1 - u - v$

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

let

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

Project onto $\vec{v}_1$ and $\vec{v}_2$

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

Solve using Cramer's rule

The first usable method to compute the barycentric coordinates is to take this equation, put it in terms of two variables and solve with a few interesting substitutions (just like we did in 2d).

First we substitute $w = 1 - u - v$ into $\vec{P} = u\vec{A} + v\vec{B} + w\vec{C}$ and re-arrange to get $\vec{P} - \vec{C} = u(\vec{A} - \vec{C}) + v(\vec{B} - \vec{C})$.

We can then do a simple re-labelling to make it easier to work with these equations:

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

At this point we have 3 equations and 2 unknowns. We need to turn this into 2 equations to easily solve. We can do this by projecting onto the vectors $\vec{v}_1$ and $\vec{v}_2$. This system can then be solved using any method to solve a system of equations, such as 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:

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

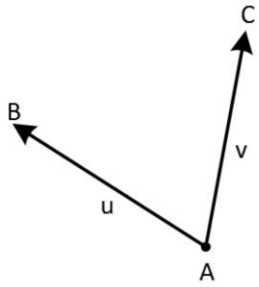
We can then subtract the two equations to get:

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

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

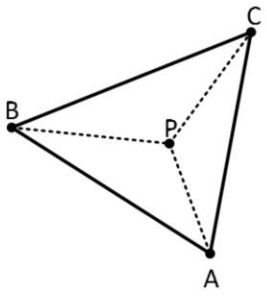


u and v are a ratios on the edges $(\vec{B} - \vec{A})$ and $(\vec{C} - \vec{A})$

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

Method 3: Triangle area ratio



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

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

$$SignedArea(PBC) = \vec{N}_{PBC} \cdot \frac{\vec{N}_{ABC}}{\left|\vec{N}_{ABC}\right|}$$

Another interesting method of computing barycentric coordinates is with any ratio in regards to triangle areas. In particular, the ratio of the normals of the triangles ABC, PBC, PCA, and PAB can be used. Any ratio that is related to the triangle areas will work, even if the result isn't the exact area itself as constant terms will disappear.

For instance, the triangle area can be represented as $\frac{1}{2}\left|\vec{\mathrm{N}}_{ABC}\right| = \frac{1}{2}\left|\left(\vec{B} - \vec{A}, \vec{C} - \vec{A}\right)\right|$ but we can drop the $\frac{1}{2}$ as constants will cancel.

Hence, we can write u as the ratio of PBC and ABC. We could just represent the area as $\left|\vec{N}_{PBC}\right|$ but that will always give a positive sign and we're after the signed area. Instead we can compute the signed area as $\vec{N}_{PBC} \cdot \frac{\vec{N}_{ABC}}{\left|\vec{N}_{ABC}\right|}$. As we normalize the normal of ABC we should not be affecting the length, hence we're just using it to flip the sign of PBC if it is the opposite winding order of ABC.

Now if you look at the signed area of ABC you'd get the equation $\vec{N}_{ABC} \cdot \frac{\vec{N}_{ABC}}{\left|\vec{N}_{ABC}\right|}$ (basically just the length of ABC) but if you then use the above equation for u you get

the equation: $u = \frac{\vec{N}_{PBC} \cdot \vec{N}_{ABC}}{\vec{N}_{ABC} \cdot \vec{N}_{ABC}}$ since the $\left|\vec{N}_{ABC}\right|$ terms cancel out.

The same approach can be used for v, and w is just $w = 1 - u - v$.

Barycentric coordinates - Line

Can use barycentric coordinates for a lot of shapes such as lines

2 main approaches that give the same result

- 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)

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

Substitute: $v = 1 - u$

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

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 should already be familiar with how to analytically compute the barycentric coordinates of a triangle now. Unfortunately, just as with a triangle we have too many equations. The barycentric equation of a line is:

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

which is unfortunately 3 equations with 2 unknowns (3 equations because of x, y, and z). Even worse, if you recall there's actually only 1 unknown as one coordinate is redundant.

To properly solve we have to reduce to one equation. To solve we have to first substitute $v = 1 - u$ to get everything in terms of 1 variable. Then to reduce we need to multiply both sides by the same thing, in this case a dot-product. The reason for which is more properly explained in the geometric version, but you can take a reasonable guess here. There's really only 2 options: $(\vec{P} - \vec{B})$ and $(\vec{A} - \vec{B})$. The equation can be zero and still have valid coordinates, so instead the second vector is better (it's only zero if we have a degenerate line). Now it's a simple matter of re-arranging to solve for u.

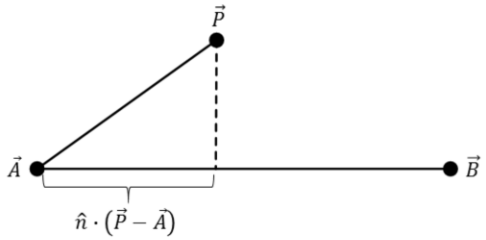
Barycentric coordinates – Line (Geometric)

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

Normalize $\vec{n}$: $\hat{n} = \frac{\vec{B}-\vec{A}}{|\vec{B}-\vec{A}|}$

Project $\vec{P}$ onto the line using the dot-product 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 geometric approach will give us the exact same answer but with a little more intuition for how we got there.

The main idea is to compute the projection of our query point onto the line segment. To do that we can use the dot-product. Remembering the identity $\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}|\cos(\theta)$ we should make sure to not let the length of $\vec{n}$ affect this so we want to normalize it first.

Now we can solve for one of the coordinates as we know the length projected onto the line is described by $(\vec{P} - \vec{A}) \cdot \vec{n}$. 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

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

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

Substitute:

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

Solve for t

We can just plug the equation for a ray into the equation for a sphere to get:

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

To make life a little easier, we can substitute $\vec{m} = \vec{c}_s - \vec{s}_r$ to get:

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

Which just expands to:

$$\vec{m}^2 - 2\vec{m} \cdot \vec{d}_r t + \vec{d}_r^2 t^2 - r^2 = 0$$

Ray vs. Sphere – Quadratic Equation

To find t we need to solve the quadratic equation: $at^2 + bt + c = 0$

Solve using the quadratic formula:

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

Consider the 3 cases of the discriminant (Δ):

$$\Delta < 0$$
$$\Delta > 0$$
$$\Delta = 0$$

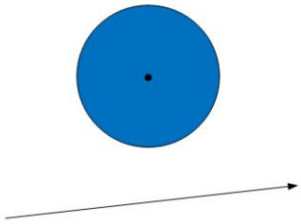
This is a quadratic equation which can be solved using the quadratic formula. Before blindly applying this we need to know what can cause this to fail. The first case is if $a = 0$ which luckily only happens if the ray direction is the zero vector.

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)



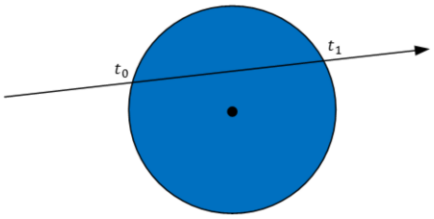
The line doesn't hit the sphere!

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

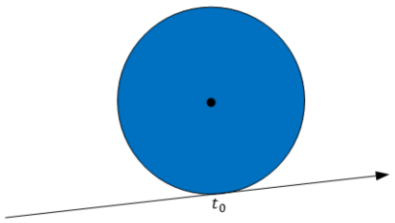
The second case is when the discriminant is positive. As the quadratic equation is $\pm\Delta$ that means we have 2 answers. What does this mean in terms of our ray vs. sphere? We have 2 intersection times as the line hits the sphere in 2 places.

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

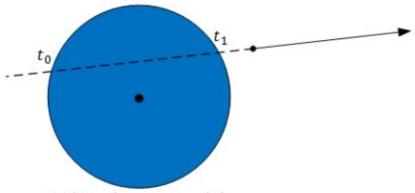


The line is tangent to the sphere

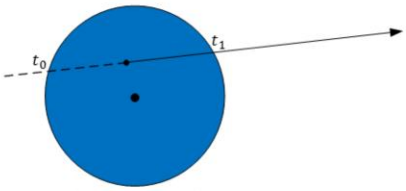
Finally, if $\Delta = 0$ then there is exactly 1 solution (± 0 gives only one result). This means that there is exactly 1 place the line hits the sphere, or rather that the line is tangent to the sphere.

Ray vs. Sphere – Invalid t-values

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



Both t-values are invalid: no intersection



The ray starts inside the sphere. T should be 0.

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).

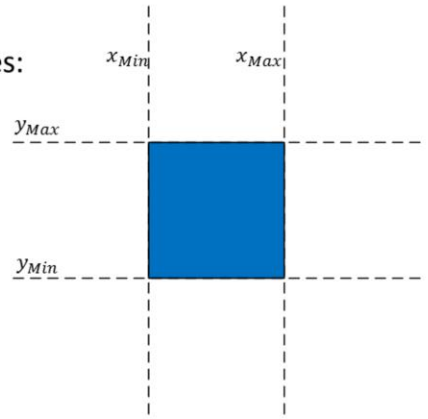
These cases can be determined by further inspecting the values of b and c in the quadratic formula.

Ray vs. Aabb

No analytic equation for an aabb

Conceptually think of an aabb as 6 planes:

- 3 axes (x, y, z)
- Min and max on each axis



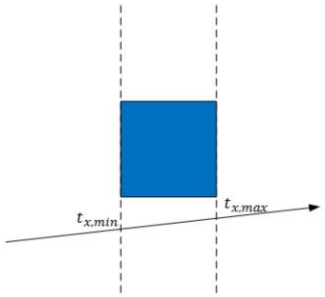
Unlike a sphere, there's no analytic form for an aabb. Instead, the intersection is computed as the intersection of 3 slabs (one for each axis). A slab is essentially a tMin and tMax for the ray on that axis. Another way to look at it is that an aabb consists of 2 "planes" on each axis: the min and the max. These slab values can then be combined to determine if there's actually an intersection. Before we can see how we combine them let's look at how to compute them.

Ray vs. Aabb

Computing a slab intersection range for x-axis:

Ray vs. Plane:

$$\vec{n} \cdot \left(\vec{s}_r + \vec{d}_r t - \vec{p}_{min}\right) = 0 \qquad \vec{n} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$



Expand constant dot-product:

$$s_x + d_x t - p_x = 0$$

$$t = \frac{\vec{p}_x - \vec{s}_x}{\vec{d}_x}$$

It's important to realize that each axis slab can be computed independently. The x-axis doesn't affect the y-axis and so on. This makes it possible to just check one axis at a time then combine the values.

How do we compute the t-values for each slab then? Well the simple way is to just compute the min and max plane for that axis then call PlaneRay to get the t-values, but this is inefficient. Instead by inspecting the equation for each axis we can get a simplified equation.

Without loss of generality, let's look at the x-axis and the min point: $\vec{p}$. If we write out the RayPlane equation we end up with:

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

The important thing to realize here is that the normal of this plane is $\hat{\mathrm{n}} = (1, 0, 0)$ which allows us to perform some optimizations (the y and z terms will drop to 0). We can now write this as:

$$\vec{s}_x + \vec{d}_x t - \vec{p}_x = 0$$

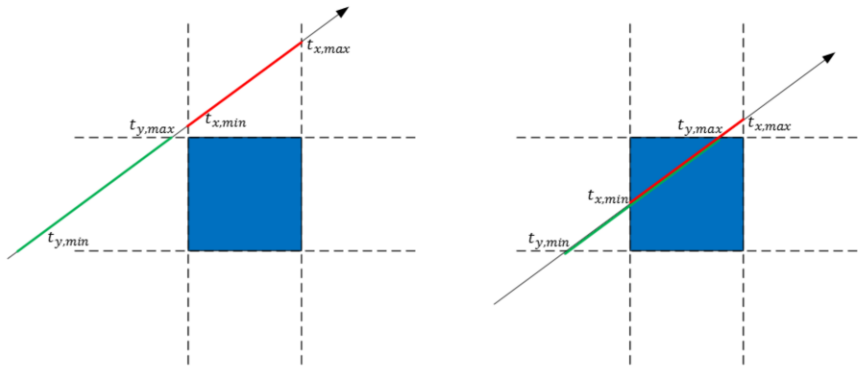
We can then easily solve for t:

$$t = \frac{\vec{p}_x - \vec{s}_x}{\vec{d}_x}$$

To find the min and max values we'd simply just use the aabb's min and max points.

Ray vs. Aabb

The ray hits the aabb if the intersection of all axis t-values is non-empty



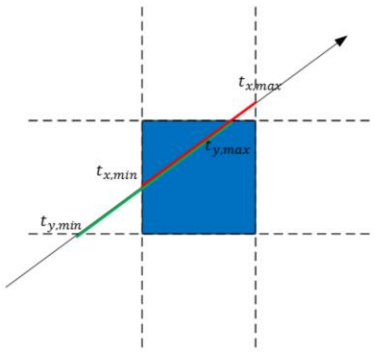
After computing each axis' min/max t-values it's a simple matter of combining them to determine if the ray actually hits the aabb.
The main observation is that if you take the intersection of the t-value range and it's non-empty then there is an overlap.

Ray vs. Aabb

Find the actual intersection range:

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

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



So how do we efficiently compute the intersection of the t-values? Also how do we compute the actual t min/max of the aabb? Both of these can be done at the same time.

To start with it's important to realize that if the ray hits the aabb then which t-value will be tMin? It's not too hard to show that'll be the largest (or last) min t-value. This is simply because the ray can't hit the aabb until it has passed all of the min planes. Similarly, tMax is the smallest of all the max t-values.

With the above picture it's clear to see that t_{min} should be $t_{x,min}$ and t_{max} should be $t_{y,max}$.

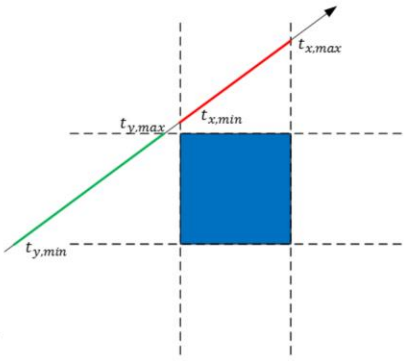
Ray vs. Aabb

Find the actual intersection range:

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

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

$$if(t_{min} > t_{max})\ no\ intersection$$



So what happens when the aabb and ray don't intersect? Well if we compute the min/max as described before we'll notice something interesting happen. In this case t_{min} will become greater than t_{max}. This range doesn't make any sense, and if you look at the picture you'll see why: the ray doesn't actually hit the aabb.

Now it's clear how we both compute t_{min} and t_{max} from each axis and use that to determine if there's an intersection or not. If the resultant range is invalid then there is no intersection.

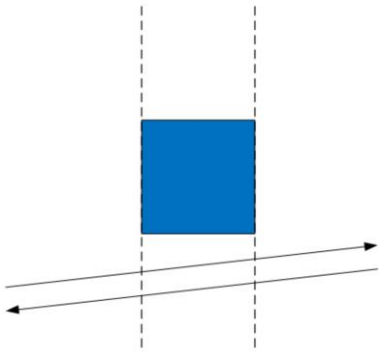
Ray vs. Aabb – Edge Cases

Have to take care of ray directionality

The sign of d_i determines ray directionality

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

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



There are 2 edge cases we have to be careful of with Ray vs. Aabb. The first is pretty easy to see. Does t_{min} always come from the aabb's min point and t_{max} from the max point? Unfortunately it doesn't and this is based upon which direction the ray is traveling.

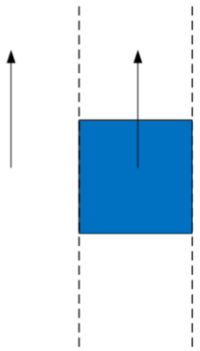
Luckily this is an easy fix that we can determine just from the sign of the ray's direction vector on the axis. If $\vec{d}_i$ is positive then the ray is travelling left to right and will hit the min point before the max. However, if $\vec{d}_i$ is negative then the ray is travelling right to left. In this case we only have to swap which value we considered t_{min} and t_{max} for this axis.

Ray vs. Aabb – Edge Cases

What if $\vec{d}_i = 0$?
Ray is parallel to plane

Could be inside or outside the slab
Test: $min_i \leq s_i \leq max_i$

If s_i is between then ignore this axis



Finally there's 1 more edge case to be careful of. What if $\vec{d}_i$ is zero? That means the ray is travelling parallel to that axis and doesn't hit the min or max plane. This is important to guard against not only because it breaks the previous logic but it will also be a zero division.

So how do we deal with this case? Well there's 2 different states we care about based upon the ray's start, it can either be inside the aabb's min and max or it can be outside. If the ray's start is outside then the ray can't intersect the aabb at all. However, if the ray's start is between the min and max then it's unknown if the ray actually hits the aabb, we have to defer to the other axes. So what do we do with this axis? Simply ignore it! Just don't use this axis when trying to compute the final t_{min} and t_{max} values.

Plane vs. Point

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

$$\begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} \cdot \begin{bmatrix} p_x \\ p_y \\ p_z \\ -1 \end{bmatrix} = \mathrm{w}$$

To classify a point against a plane we simply plug it into the plane equation. If the result is positive then the point is in-front of the plane, if negative then behind the plane, if zero then it's on the plane. Typically we want to test a thick plane, that is test to see if $abs(w) < \epsilon$.

Project Point On Plane

Compute the distance from the plane

Move point in opposite normal direction

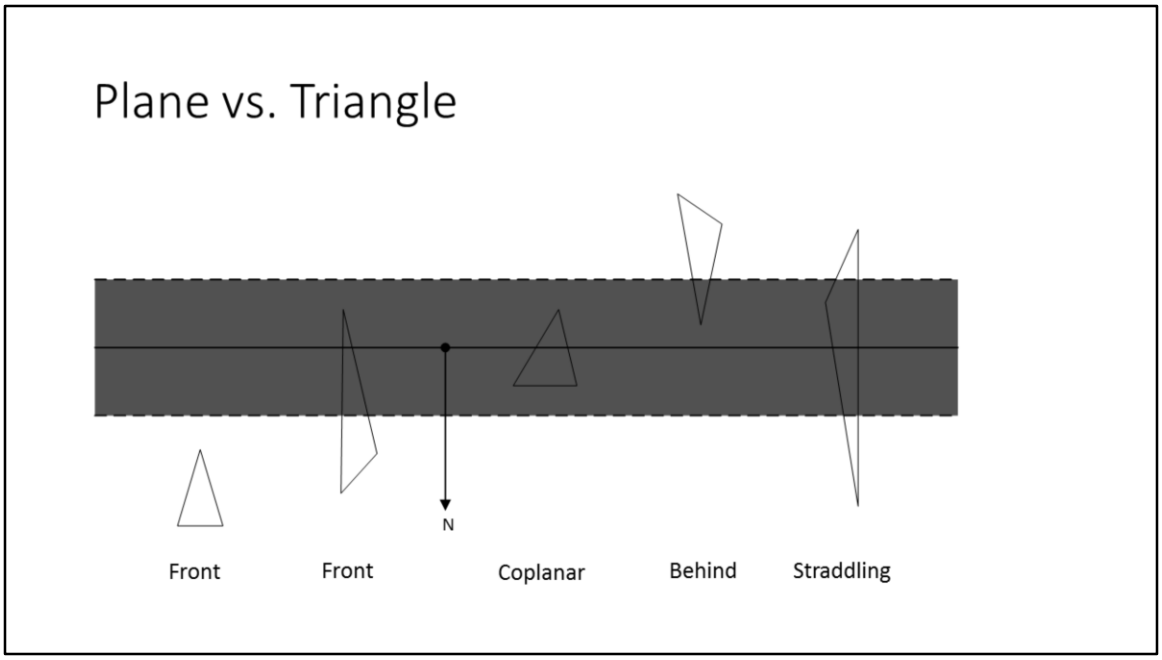
To project a point onto a plane you first must compute the distance from the plane (as shown in plane vs. point). Afterwards you only need to move the point towards the plane by this amount.

Plane vs. Triangle

Classify all 3 points

What about epsilon?

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

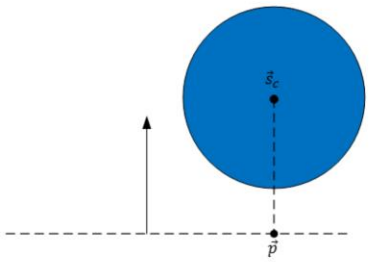


When dealing with a thick plane things get trickier, mostly due to coplanar points. The simple thing to keep in mind is that coplanar points don't affect the outcome of any other state.

Plane vs. Sphere

Project sphere center on plane

Test if that point is inside sphere



To test plane vs. sphere we just project the sphere center onto the plane. Then you can do a distance check between that point and the sphere center, if it's less than or equal to the sphere radius then there is an intersection. Otherwise, we can just classify the sphere's center against the plane to determine if it's in front or behind.

Plane vs. Aabb

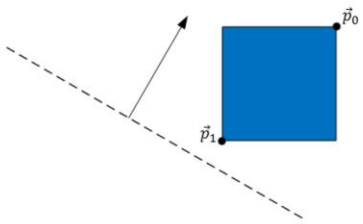
Method 1: Classify all 8 points

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 straddles the plane. This however is very inefficient and we can do a lot better.

Plane vs. Aabb

Method 2: Classify 2 extremal points



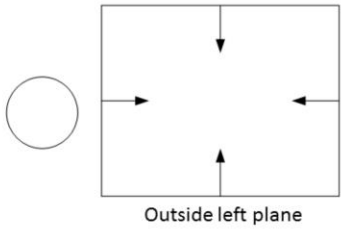
*easy to compute from center and half-extent

Instead if we can find 2 points on the aabb, 1 furthest in the direction of the normal and one furthest in the negative direction of the normal we'd only need to classify 2 points. This is easily done with the center and half extents representation of an aabb. For each axis, we should pick whether to add or subtract the half extent of that axis based upon the sign of the normal on that axis.
The most negative point is simply found by doing the opposite (or adding the negative offset vector computed from part 1).

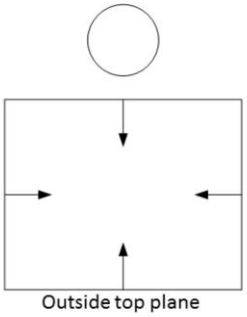
Do note that if we don't care about the cases of in front or behind but only a Boolean result we can optimize this a bit further (using some properties of SAT covered later). We can just compute the "radius" of the aabb with respect to the plane normal (just the length of the furthest point). Then we can perform the exact same test as plane vs. sphere using the "radius" of the aabb.

Frustum vs. Sphere Culling

Test all 6 planes:



Outside left plane



Outside top plane

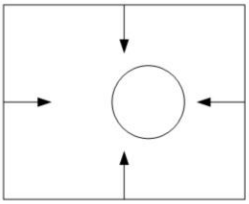
If the sphere is outside any plane then it is outside the frustum

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:

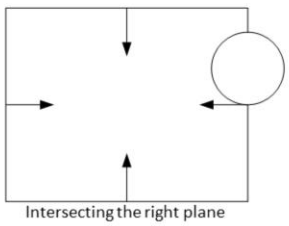


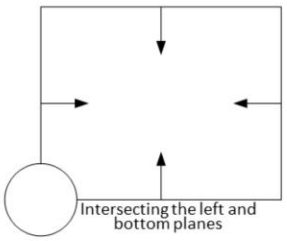
If the sphere is inside all planes then it is inside the frustum

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:



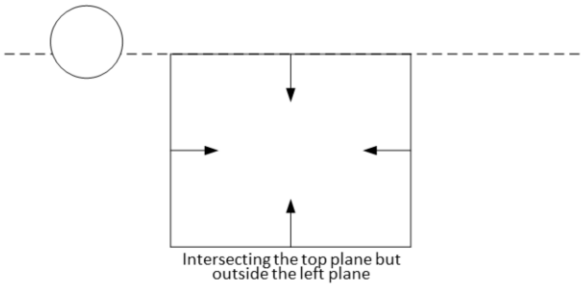


Otherwise if the sphere overlaps any plane then it overlaps the frustum

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

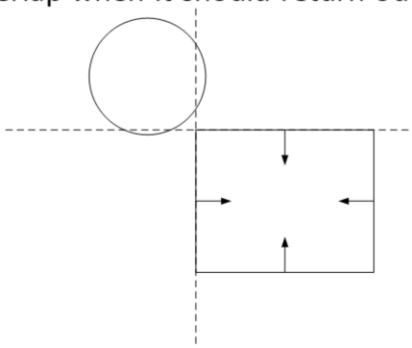
Frustum Culling is an approximation, it gives false positives

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.

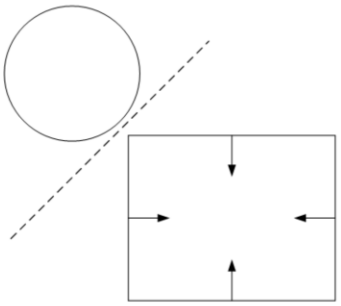
What's missing? Quick look at SAT

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.