Bounding Volumes

jodavis42@gmail.com

This presentation is all about bounding volumes. Bounding volumes are one of the key foundations of computational geometry and reducing cost in a collision detection pipeline.

Bounding Volumes

What is a bounding volume?

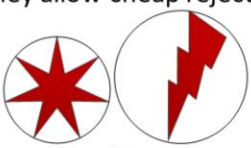
A shape that fully contains a (typically) more expensive underlying shape

To start off with we have to establish what a bounding volume is. There's only really 1 rule to a bounding volume: it's some shape that fully contains (bounds) some underlying set of shapes. These shapes are typically more expensive than the bounding volume for reasons that will be explored further in this presentation. No other rule is really required for a shape to be a bounding volume.

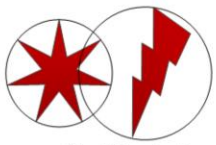
Bounding Volumes

Bounding Volumes don't change algorithmic complexity

- They allow cheap rejection tests



can't collide: skip



could collide: check

If the BVs collide then we check the underlying shapes

- Wait, we have to check both?
- Is this actually faster?

So what is the purpose of a bounding volume? The main purpose is to speed up the collision detection pipeline, but not through algorithmic complexity change. If a bounding volume doesn't intersect with some shape then neither can the underlying geometry that it contains. This allows bounding volume to provide cheap rejection tests for more complex shapes. If the bounding volumes do collide then the underlying shapes have to be checked.

This should lead to the obvious realization that sometimes we get away with a cheaper computation but sometimes we have to check the bounding volume and the underlying shapes, resulting in "double" the work. So is doing more work sometimes worth it or should we just check the underlying shapes.

Bounding Volumes

Bounding volumes reduce algorithmic constants

Given 100 shapes that are twice as expensive as spheres:

- How many can collide?
- How many could realistically collide?

Compare these costs. Are bounding volumes worth it?

*An average bounding volume is much cheaper than ½ cost

To help illustrate why we use bounding volumes it's important to first realize that they effectively reduce the algorithmic constants. Now we can perform a simple thought experiment. If we have 100 shapes that are twice as expensive to check for intersection than spheres then how much could we possibly save?

First we need to determine how many collisions could happen. To make life easy we'll just use a pure n^2 calculation and go with 10,000 possible collisions.

Now we need to determine how many shapes could possibly collide. With a sphere in 2d it's reasonable to assume that a sphere could intersect 8 other spheres at max. This leads to an average max of 400 intersections (800 / 2 because of a colliding with b and b colliding with a).

This gives us less than a 1% actual collision rate. Now how much does this actually save? Well the brute force method would cost: 10,000 * 2 or 20,000 to test. With bounding spheres we'd on average expect 99% to use the sphere cost and 1% use both: 9,900 * 1 + 100 * (1 + 2) = 9,900 + 300 = 1,200.

We save a lot by using a bounding volume! Realistically though, the underlying shape cost is significantly more than twice as expensive.

Bounding Volumes

Unfortunately, there is no best bounding volume

Considerations:

- Intersection cost
- Tight fitting
- Initial computation cost
- Update cost
- Memory overhead

Unfortunately, no “best” bounding volume exists. It’s typically a trade off of several characteristics:

1. Intersection cost: How expensive it is to test a bounding volume is very important as we saw earlier. This is effectively the constant term in front of all tests.
2. Tight fitting: How many tests will pass the bounding volume test and still fail the intersection test. This is basically how many false positives are returned.
3. Computation cost: How expensive is it to compute a bounding volume. This gets complicated as there are often several different methods to compute a bounding volume that produce differing levels of tight fitting. For example, we’ll several different sphere methods that produce tighter and tighter bounding spheres. Typically we have to choose to produce a less than perfectly fitting bounding volume due to how expensive it is to compute.
4. Update cost: As a shape moves, and primarily as it rotates, how hard is it to update the bounding volume. Some methods will maintain the tightness of the original computation while some may favor cheap approximations.
5. Memory overhead: To store a bounding volume requires extra memory per shape. Memory overhead is important not only because of total memory budget, but because of performance ramifications with memory and cache. Typically the

more tight fitting a bounding volume can be the more memory is required.

These metrics typically have to be weighed against each other for a specific application to choose a bounding volume.

Bounding Volumes

Common bounding volumes:

- Aabb
- Sphere
- Obb
- K-Dop
- Convex-Hull

The two simplest and probably most common bounding volumes are bounding sphere's and axis aligned bounding boxes (aabb). I will go into great depth on our considerations for these two bounding volumes as they are very useful. In particular, I will go into the previous 5 considerations for these two.

The remaining bounding volumes aren't as commonly used so I will only spend a small amount of time talking about them.

Personally, aabb's are my favorite.

Aabb

Two primary representations

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

```
struct Aabb
{
  Vector3 mCenter;
  Vector3 mHalfExtents;
};
```

Min/Max tends to be more commonly used

*both require 6 floats

An aabb is a min/max on each cardinal axis (x, y, z). Because of this only 6 floats are needed to store it. There are two primary formats for an aabb: Min + Max, and Center + Half-Extents. Both have their benefits but I'll stick to min/max for the majority of this class as I prefer it, especially for computation and intersection tests. Luckily, for the few operations where center and half-extents are preferred, it's very easy to convert between the two.

Aabb Computation

Computation from a point cloud is trivial:

- Simply find min/max on each axis

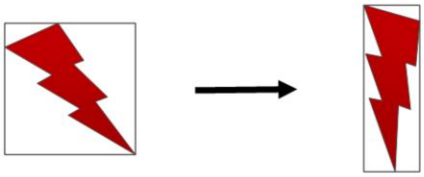
This always produces the tightest fitting aabb possible!

First we'll look at the initial computation cost. For an aabb this is a trivial operation of simply finding the min/max value on each axis.
It's worth noting that this will always produce the tightest fitting aabb possible.

Aabb Update

Method 1: Full re-computation

Transform the shape and then re-compute



The simplest method to refit an aabb is to just re-compute it from the new shape. This means taking each vertex and updating it with the current transform and then using those points to compute the new aabb. This is obviously slow, however it will produce the tightest fit aabb possible. That is the biggest thing to note about this method, it's slow but accurate.

This method is typically too slow to perform per object per frame, so we want to find a good trade-off between accuracy and speed.

Aabb Update

Method 2: From a bounding sphere

Cheap and easy, but inaccurate

From bounding sphere

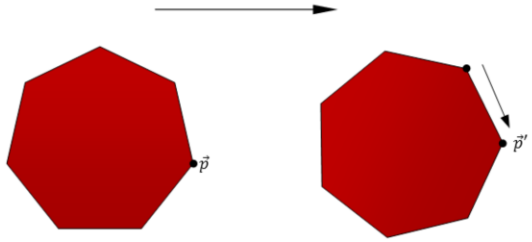
Best

If refitting the aabb every frame is all accuracy and no speed, then the opposite of that is to compute the aabb from a bounding sphere. Since the bounding sphere is guaranteed to contain the object at all rotations, then an aabb from that bounding sphere is guaranteed to contain the object at all rotations. That means only translation needs to be updated, which is just setting the aabb center/offset from the object's center.

The one interesting thing to note is that the center of the object is not always the translation position. Because of this updating the translation can be a little tricky for non-symmetric objects. Ignoring rotation, the center of the aabb needs to be the same relative center as the aabb from the local space (taking scale into account) model. Because of this, and other methods of computing the aabb later, it's common to store a local space aabb (potentially just on the mesh as it's unique to the mesh not to the component) and then update the world aabb from that.

Aabb Update

Method 3: Hill Climbing



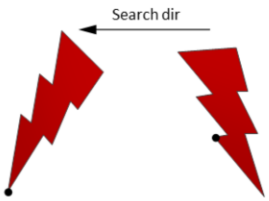
One method that seeks to gain both benefits is using hill climbing. The idea is that instead of storing the min/max on each axis, we store indices for the min/max into the vertex list. Using some data structure that stores adjacency info we can check each frame if the min/max is still the min/max by checking adjacent edges. If the point is not extremal then we walk edges until we find a more extremal point.

One speed performance gain that can be achieved is by realizing that when finding the min/max x we only need to transform the x coordinate into world space. This can cut expenses down quite a bit.

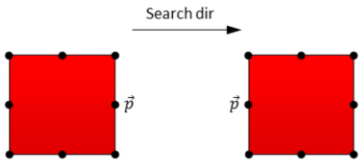
Aabb Update

Hill climbing breaks down in 2 areas

Concave shapes



Local minima



There are two major problems with this approach though, and they both stem from the fact that hill climbing only works when we're guaranteed to make progress at each step or be done, aka there's no local minima. The first problem arises when the mesh is not convex. One way to fix this is to pre-compute the convex hull and only update the aabb from the convex hull. This does require storing a bit of extra data per mesh, but the convex hull is commonly used so it's not too bad.

The other local minima arises when there's a plateau. This can happen if you imagine (pictured on the right) a highly tessellated cube that rotates 180 degrees. Any point in the center will be unable to find a point further in the direction and stop at a wrong extremal point. These sort of scenarios have to be special cased for hill-climbing and hence I don't' recommend it.

Aabb Update

Method 4: Aabb of rotated Aabb

Original

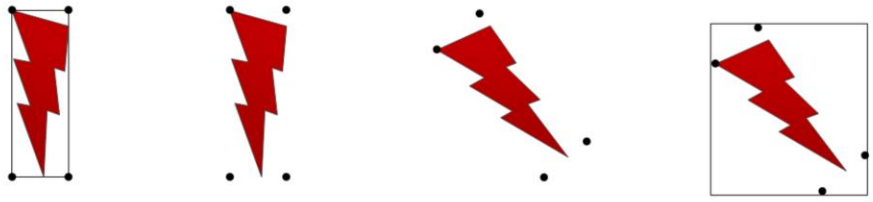
Aabb of aabb

Best

A good compromise between speed and accuracy is to compute an aabb from the rotated aabb.

Aabb Rotation

Simple way: Compute aabb of rotated aabb points (8)

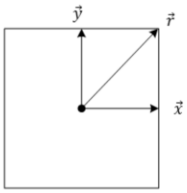


The simple way to think about doing this is to take the 8 points of the aabb and rotate them then build the new world aabb from these points. This is obviously not as tight fit as the refitted aabb, but is still much tighter than the sphere method.

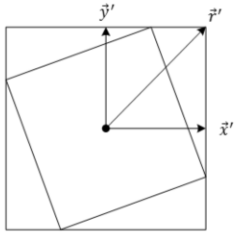
Rotating 8 points is still a little expensive. This method can be sped up quite a bit by examining the aabb calculation and the matrix properties.

Aabb Rotation - Optimized

Easiest with center and half-extent representation



$\vec{r} = \vec{x} + \vec{y} = \begin{bmatrix} x \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ y \end{bmatrix}$

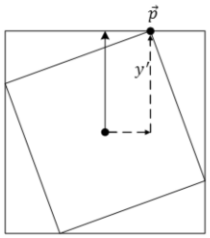


$\vec{r}' = \vec{x}' + \vec{y}' = \begin{bmatrix} x' \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ y' \end{bmatrix}$

To start with we want to use the center and half extent method. First note that the half extent vector $\vec{r}$ can be thought of as a compact form of the sum of the basis vectors: $\vec{r} = \vec{x} + \vec{y} + \vec{z}$. Another way to look at this is that the length of these vectors is the size of the aabb on each basis axis: $\vec{r} = \begin{bmatrix} x \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ y \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ z \end{bmatrix}$. With this in mind we can also express the final aabb's half extent the same way: $\vec{r}' = \vec{x}' + \vec{y}' + \vec{z}'$.

Aabb Rotation - Optimized

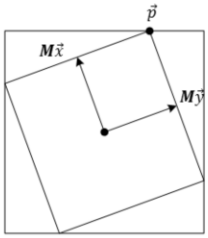
We get y' from the top-most point



We already know we can compute the rotated aabb from the rotated points, and we also know that y' will come from the top-most point's y-value.

Aabb Rotation - Optimized

We can compute any point from the bases

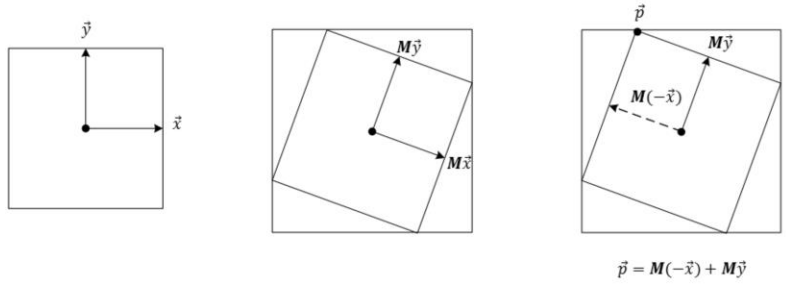


$$\vec{p} = M\vec{x} + M\vec{y}$$

We also know that any rotated point can be represented from the center and half extent representation. In this case $\vec{p} = M\vec{x} + M\vec{y}$.

Aabb Rotation - Optimized

How do we know whether to add or subtract a basis?

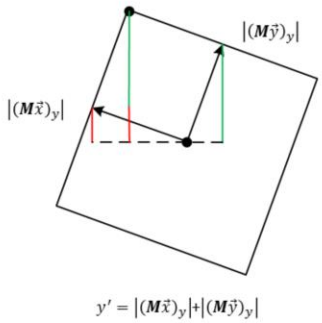


However, the points are created from the plus or minus of the basis vectors, how do we know whether to add or subtract a basis?

Aabb Rotation - Optimized

Instead of summing vectors, sum projections

We want most positive, so sum the abs(projection)



Instead of trying to compute the point directly, and then take its y-value we should just focus on computing the y-value. The y-value can be thought of as the sum of the projections of the rotated basis vectors onto the y-axis. When computing this sum, we're after the most positive value. Clearly we only want to sum positive values to get the most positive value, to achieve this we can just sum the absolute value of the projection: $y' = \left|\left(\boldsymbol{M}\vec{x}\right)_y\right| + \left|\left(\boldsymbol{M}\vec{y}\right)_y\right| + \left|\left(\boldsymbol{M}\vec{z}\right)_y\right|$.

The same operation can be performed for the x and z axes.

Aabb Rotation - Optimized

Still wasting computations as $\vec{x}, \vec{y}$, and $\vec{z}$ are mostly 0

$$r'_x = |(\boldsymbol{M}\vec{x})_x| + |(\boldsymbol{M}\vec{y})_x| + |(\boldsymbol{M}\vec{z})_x|$$

Give that $\vec{x}$, $\vec{y}$, and $\vec{z}$ are all positive:

$$r'_x = (|\boldsymbol{M}|\vec{x})_x + (|\boldsymbol{M}|\vec{y})_x + (|\boldsymbol{M}|\vec{z})_x$$
$$\vec{r}_x = [|\boldsymbol{M}|(\vec{x} + \vec{y} + \vec{z})]_x$$
$$\vec{r}_x = (|\boldsymbol{M}|\vec{r})_x$$

With further inspection we can see:

$$\vec{r}' = |\boldsymbol{M}|\vec{r}$$
$$= \begin{bmatrix} |m_{00}|r_x + |m_{01}|r_y + |m_{02}|r_z \\ |m_{10}|r_x + |m_{11}|r_y + |m_{12}|r_z \\ |m_{20}|r_x + |m_{21}|r_y + |m_{22}|r_z \end{bmatrix}$$

While this does save some computational over rotating all points, this can still be made a bit faster by inspecting the matrix multiplications, in particular, we are wasting a large amount of computation by multiplying with so many zero's from the basis vectors. There are a few spots where we rotate a vector that only contains 1 value and then take the x, y, or z axis. Instead we can inspect the matrix multiplication and only multiply what is needed.

Similar calculations can be done for min/max representation, however when including scale and translation it's quite a bit more cumbersome (as you have to scale about the center, not the origin).

Aabb Full Transform

How do we include scale and translation?

$\vec{r}$ is a direction vector:

$$\vec{r}' = |\boldsymbol{M}|(\vec{s}\vec{r})$$

The aabb center is a position vector:

$$\vec{c}' = \vec{t} + \boldsymbol{M}(\vec{s}\vec{c})$$

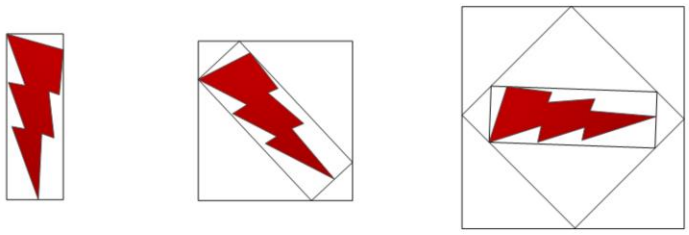
So how do we perform a full transformation including scale and rotation?
The aabb's $\vec{r}$ vector is a direction vector so it's unaffected by translation. We can simply scale it before applying the previous operation.

The aabb center is a position vector so it' transforms with the regular rotation matrix (not the absolute value one) and translation is added afterwards.

Note here that $\vec{s}\vec{r}$ is short-hand for $\begin{bmatrix} s_x * r_x \\ s_y * r_y \\ s_z * r_z \end{bmatrix}$.

Aabb Rotation

Problem: Aabb will grow forever with rotations



Solution: store local (original) aabb and transform to world space

There is one major problem with this method to consider, luckily it is easy to deal with. If we compute the new aabb by transforming the current aabb with a delta then it'll keep growing forever. This is actually easy to deal with as we almost never perform a delta transformation anyways. Instead we need to use a little more memory by storing the local space aabb and transforming that each frame into world space.

Sphere

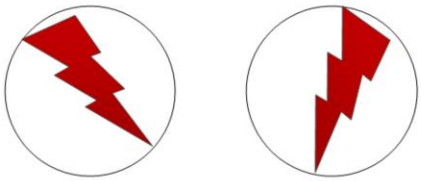
Typically only one storage format

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

Bounding spheres are typically only stored in one format: center and radius.
The computation of a sphere is quite hard. In fact, the process of computing the tightest fit sphere is very complicated so typically one opts for an approximation method.

Sphere Update

Updates are mostly trivial



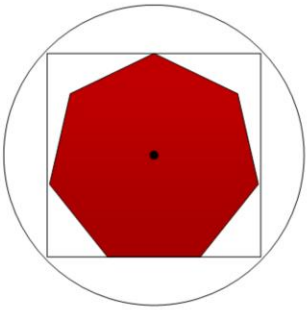
*Careful as some shapes don't rotate around their centroid.

Unlike aabb's, sphere computation is hard while updating is trivial. Do note though, that sphere's have to be careful of objects that don't rotate about their centroid. Often times in a game an object will rotate about a point that is not it's center (say a sword rotating about the handle). In this case you need to store an offset of the sphere's center from the pivot point and transform that as the object rotates.

Sphere Computation

Method 0: From an Aabb

Optimal aabb's are easy, compute the sphere of the aabb



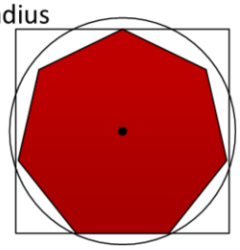
The first method I'll talk about is to utilize the ease of optimal aabb computation. Simply compute the aabb for the object then compute the bounding sphere of that aabb. This method should almost never be used though as it's easy to do better. Because of that I'm calling this method 0…

Sphere Computation

Method 1: Centroid

2 pass algorithm:

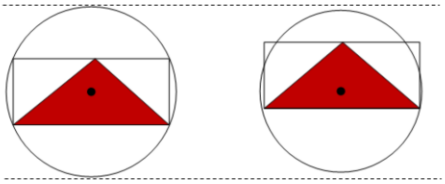
- Compute centroid (aabb center)
- Pick furthest away point as radius



To improve upon the aabb method we can look at the centroid method. We can pick a good center point for our sphere by first computing the centroid (which just so happens to be the aabb center). Now instead of taking the radius as the furthest aabb "radius" we can do a second pass to find the actual furthest away point. This will always give as good if not better of a sphere than the aabb method.

Sphere Computation

Observation 1: If we let the center move we can make a better sphere

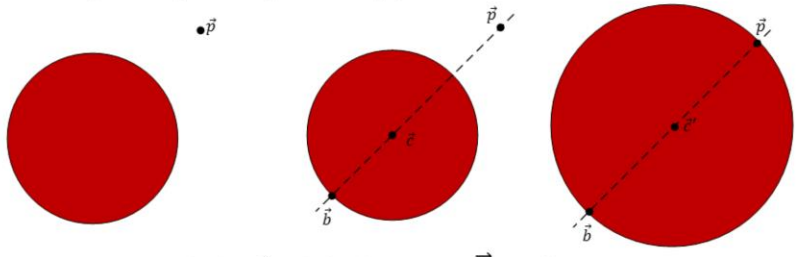


How could we expand with a point more efficiently?

One key observation to realize is that when we picked the sphere's starting center we never moved it. We basically assumed this was the best center. Instead we could expand our sphere more intelligently by a point.

Sphere Expansion

Minimal way to expand sphere by point



Move center to include $\vec{p}$ while keeping $\vec{b}$ in the same spot

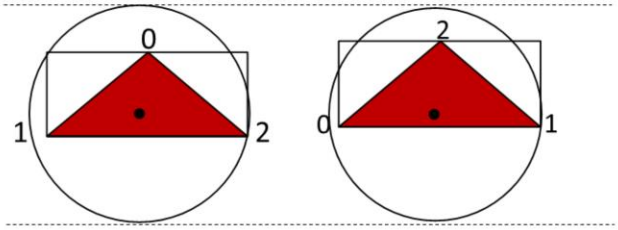
So how do we minimally expand a sphere by a point? The basics version is quite simple to implement. To minimally expand a sphere we the point on the opposite side of the sphere as the new point to stay in place and then expand the sphere to contain the new point.

The simplest way to do this is to compute the back point as $\vec{b} = \vec{c} - r\left(\frac{\vec{p}-\vec{c}}{|\vec{p}-\vec{c}|}\right)$. The center of the new sphere can then just be computed as the mid-point of $\vec{p}$ and $\vec{b}$ and the new radius is just $\frac{1}{2}\left|\vec{p} - \vec{b}\right|$.

With a little bit of work you can expand and re-arrange the equations for $\vec{c}'$ and r' to remove some square roots.

Sphere Computation

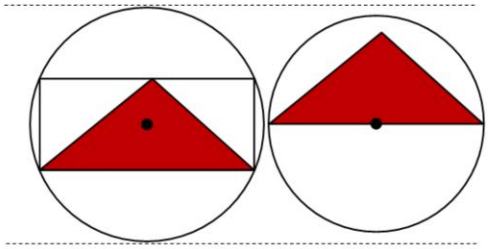
Observation 2: The order you visit points matters



Another important observation is the order of points you use to expand a sphere can produce a very different sphere.

Sphere Computation

Observation 3: Where the sphere center starts matters



*Minimal bounding spheres are non-trivial

Finally, we can observe that the first guess of the starting sphere center is very important.

This leads to several bounding sphere methods that are all about approximating the best sphere center to start with followed by adding incrementally all points outside the sphere.

It's also worth noting here that computing a minimal bounding sphere is a non-trivial operation.

Sphere Computation

This is the basis of most bounding sphere methods:
How can we find the best sphere center start?

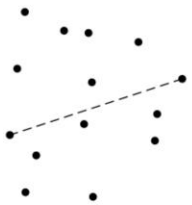
Where's a good heuristic for the center?
The center of the most spread axis

The next two bounding sphere methods both utilize these observations by minimally expanding a sphere after guessing at a good center. The question is what's a good guess for the starting center? The basic heuristic both user is to find an axis with the largest spread of points.

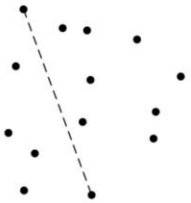
Sphere Computation

Method 2: Ritter Sphere

Check each cardinal axes for largest spread



Largest x-spread



Largest y-spread

Choose largest axis' spread

The idea behind Ritter sphere is that there's too many axes to logically check for the largest spread. To simplify the number we can check 3 axes, primarily the cardinal axes. Even better yet, these axes do not require a projection to compute the distance along the axes.

In one pass we can iterate through all points and compute the most spread points on each axis. We can then pick whichever pair of points were further apart as our starting line segment. Afterwards we can just expand the sphere as described before.

Sphere Computation

Method 3: PCA

Use statistical analysis to find the axis of max spread

Instead of checking only the cardinal axes, the axis of most spread can be computed using statistics. This is done with a technique called PCA, or Principle Component Analysis.

Covariance Matrix

Covariance measures how data varies together

	Summation notation	Vector form
mean	$\vec{u}_i = \frac{1}{n}\sum_{k=0}^{n-1} \vec{P}_i^k$	$\vec{u} = \frac{1}{n}\sum_{k=0}^{n-1} \vec{P}^k$
covariance	$\boldsymbol{c}_{ij} = \frac{1}{n}\sum_{k=0}^{n-1} (\vec{P}_i^k - \vec{u}_i)(\vec{P}_j^k - \vec{u}_j)$	$\boldsymbol{c} = \frac{1}{n}\sum_{k=0}^{n-1} \begin{bmatrix} \vec{v}_0\vec{v}_0 & \vec{v}_0\vec{v}_1 & \vec{v}_0\vec{v}_2 \\ \vec{v}_0\vec{v}_1 & \vec{v}_1\vec{v}_1 & \vec{v}_1\vec{v}_2 \\ \vec{v}_0\vec{v}_2 & \vec{v}_1\vec{v}_2 & \vec{v}_2\vec{v}_2 \end{bmatrix}$

$*\vec{v} = \vec{P}^k - \vec{u}$

To start with, we have to compute what's called the covariance matrix. For two variables, we can compute the covariance value which measures how the two variables vary together. For multiple variables we end up getting a matrix.

Shown above is the equations for computing the mean of a data set which is needed to compute the covariance matrix. Typically these equations are given in summation notation (as shown on the left), on the right is the expanded vector equations.
Note that for the vector form of the covariance matrix that $\vec{v} = \vec{P}^k - \vec{u}$
Also note that for the summation notation format that the superscript is the point index while the subscript is the vector index (x, y, or z value).

Covariance Matrix Properties

1. The covariance matrix is symmetric
2. The eigenvector with the largest eigenvalue is the axis of max spread

How do we find the eigenvectors?

The covariance matrix has the useful property that the eigenvector that has the largest eigenvalue is the axis of max spread of the data set. The question then is how to compute the eigenvectors and eigenvalues of the matrix. There are a number of different techniques, but the one we'll look at is the Jacobi rotation method.

Eigenvector and Eigenvalue

An eigenvector for a matrix is a vector that doesn't change direction under a linear transform

$$\boldsymbol{A}\vec{v} = \lambda\vec{v}$$

$\vec{v}$ is the eigenvector

λ is the eigenvalue

Just a quick refresher for what eigenvectors and eigenvalues are.
Basically we're after a vector that only gets scaled when being multiplied by the matrix.

Jacobi Rotation

Hard to compute eigenvectors for general matrices:

$$\boldsymbol{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Easy for a diagonal matrix:

$$\boldsymbol{D} = \begin{bmatrix} d_{00} & 0 & 0 \\ 0 & d_{11} & 0 \\ 0 & 0 & d_{22} \end{bmatrix}$$

Idea! Rotate $\boldsymbol{A}$ into a diagonal matrix via some rotation matrix $\boldsymbol{J}$:

$$\boldsymbol{D} = \boldsymbol{J}^{-1}\boldsymbol{A}\boldsymbol{J}$$

The idea of Jacobi rotation is fairly straightforward. It's difficult to compute the eigenvectors and eigenvalues of a general matrix, however computing these is really easy for a diagonal matrix. So what we can do is build a series of rotation matrices to transform A.

Jacobi Rotation

Let a_{pq} be the largest off-diagonal term

We want to rotate a_{pq} and a_{qp} to 0:

$$\begin{bmatrix} * & & \cdots & * \\ & \ddots & & \\ & a_{pp} \ \cdots \ a_{pq} & & \\ \vdots & \vdots \ \ddots \ \vdots & & \vdots \\ & a_{qp} \ \cdots \ a_{qq} & & \\ & & \ddots & \\ * & \cdots & & * \end{bmatrix} \rightarrow \begin{bmatrix} * & & \cdots & * \\ & \ddots & & \\ & a'_{pp} \ \cdots \ 0 & & \\ \vdots & \vdots \ \ddots \ \vdots & & \vdots \\ & 0 \ \cdots \ a'_{qq} & & \\ & & \ddots & \\ * & \cdots & & * \end{bmatrix}$$

Build the rotation matrix $\boldsymbol{J}$:

$$\begin{bmatrix} 1 & & \cdots & 0 \\ & \ddots & & \\ & c \ \cdots \ s & & \\ \vdots & \vdots \ \ddots \ \vdots & & \vdots \\ & -s \ \cdots \ c & & \\ & & \ddots & \\ 0 & \cdots & & 1 \end{bmatrix}$$

The easiest way to think about the Jacobi rotation algorithm is that we build rotation matrices one at a time to transform $\boldsymbol{A}$ such that the largest off-diagonal term (and it's symmetric pair) becomes zero. The idea behind picking the largest term is to make the most progress at each step.

Since we're only worrying about 2 terms each rotation matrix we can envision the rotation matrix as a very sparse matrix that is the identity except for the rotational sin and cosine terms that are at p and q indices.

Jacobi Rotation

Inspect the rotational terms:

$$B = \begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} = J^T AJ = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

We can expand to get equations for to get b_{pq}:

$$b_{pq} = csa_{pp} - s^2 a_{pq} + c^2 a_{pq} - csa_{qq}$$

Solve for $b_{pq} = 0$

Instead of building this large matrix each time, we can further inspect the multiplications of $\boldsymbol{J^1 AJ}$. What we're after here is to compute the new matrix that will cause a_{pq} and a_{qp} which will go to 0.

With a little bit of work, we can expand this matrix multiplication to get all of the terms of B. However, we only care about the off-diagonal terms as we want to solve to make them equal to 0.
Note: I re-arranged some terms knowing that $a_{pq} = a_{qp}$

Jacobi Rotation

Expand and solve:

$$b_{pq} = 0 = csa_{pp} - s^2 a_{pq} + c^2 a_{pq} - csa_{qq}$$

To get:

$$\frac{a_{qq} - a_{pp}}{a_{pq}} = \frac{c^2 - s^2}{cs}$$

To make life easier divide both sides by 2

$$\frac{a_{qq} - a_{pp}}{2a_{pq}} = \frac{c^2 - s^2}{2cs}$$

Let $\beta = \frac{a_{qq} - a_{pp}}{2a_{pq}}$

Solve for s and c

With some simple re-arranging we can get all of the terms in a on one side and all cosine and sine terms on the other. Now since one side is just a bunch of constants (all the a terms) we can group them together: $\beta = \frac{a_{qq} - a_{pp}}{2a_{pq}}$. Now we have the equation $\beta = \frac{c^2 - s^2}{2cs}$ to solve.

Jacobi Rotation

Use the trig identities:

$$\cos(2\theta) = \cos^2(\theta) - \sin^2(\theta),$$
$$\sin(2\theta) = 2\sin(\theta)\cos(\theta),$$
$$\tan(2\theta) = \frac{2\tan(\theta)}{1-\tan^2(\theta)}.$$

Re-arrange into an equation in terms of tangent ($t = \tan(\theta)$):

$$\frac{1-t^2}{2t} = \beta \qquad \text{Or} \qquad t^2 + 2t\beta - 1 = 0$$

Now we have 1 equation with 2 unknowns (cosine and sine). We can use trig identities to re-arrange and get everything in terms of tangent (so we only have 1 unknown).

We had $\beta = \frac{c^2-s^2}{2cs}$, plugging the double angle formulas for cosine and sine in we now can get: $\beta = \frac{\cos(2\theta)}{\sin(2\theta)}$ which we can then relabel as $\beta = \frac{1}{\tan(2\theta)}$.

Now we can plug in the double angle formula for tangent to get: $\beta = \frac{1-\tan^2(\theta)}{2\tan(\theta)}$.

Now by labeling $t = \tan(\theta)$ we can re-arrange this equation to put everything on one side and we end up with a quadratic formula to solve in terms of t.

Jacobi Rotation

Special solution of Quadratic equation:

$$t = \frac{sign(\beta)}{|\beta| + \sqrt{\beta^2 + 1}}$$

Knowing: $\tan^2(\theta) + 1 = \sec^2(\theta)$

We can solve for $\sin(\theta)$ and $\cos(\theta)$

We can solve this specific version of the quadratic formula as shown above, where $sign(\beta)$ returns either -1 or 1 depending on the sign of β.
Finally, knowing a few more trig identities we can solve for cosine and sine.

Jacobi Rotation

Continue this process until:

1. The matrix is diagonal (or close enough)
2. We do enough iterations

So how long do we perform this for? Well the desired result is until our matrix becomes diagonal, or at least close enough. One good measure is to see if the sum of the squared off diagonal terms is below some epsilon value.

The other termination condition is just some max number of iterations. It could take too long to converge to a solution so in that case we just cap out at something like 50 iterations.

Jacobi Rotation

At the end we'll have: $A = \begin{bmatrix} e_x & 0 & 0 \\ 0 & e_y & 0 \\ 0 & 0 & e_z \end{bmatrix}$

Where e_i is the eigenvalue of eigenvector i.

To get the eigenvectors: $[\vec{v}_x \quad \vec{v}_y \quad \vec{v}_z] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * J_1 * J_2 * \cdots * J_n$

So after all that work we'll have the eigenvalues of our points which we can use to select the largest eigenvector, but how do we get the eigenvector? We simply transform the identity matrix by our series of rotation matrices and the columns of the resultant matrix are the eigenvectors.

PCA

1. Find the axis of max spread (Jacobi Rotation)
2. Find the two points furthest apart on this axis
3. The mid-point is the sphere's starting center
4. Iteratively expand until all points are contained

Now that we have the axis of most spread we can find the 2 points furthest away on this axis and choose their mid-point as the center of our sphere.
From here we can iteratively increase our sphere as before.

Sphere Computation

Method 4: Iterative refinement

- Take previous results
- Shrink sphere
- Try again

Another simple idea is to iterate on the results of a previous guess. When expanding a sphere to contain all points, the choice of the center and the order of the points greatly affect the result. We can take the previous sphere's center, shrink the radius by some amount (so some points are outside), and visit the points in a different order to produce a different sphere. At some point, this new sphere is likely to be smaller than our previous one. In this case we keep that sphere and continue iterating again. If we do this for a few iterations then we're likely to produce a close to ideal bounding sphere.

Sphere Computation

Method 5: Welzl's Algorithm (min bounding sphere)

```
Sphere Welzl(const std::vector<Vector3>& points, int numPoints, std::vector<Vector3>& sos, int numSos)
{
  if(numPoints == 0 || numSos == 4)
  {
    switch(numSos)
    {
    case 0: return Sphere();
    case 1: return Sphere(sos[0]);
    case 2: return Sphere(sos[0], sos[1]);
    case 3: return Sphere(sos[0], sos[1], sos[2]);
    case 4: return Sphere(sos[0], sos[1], sos[2], sos[3]);
    }
  }

  int index = numPoints - 1;
  Sphere smallestSphere = Welzl(points, numPoints - 1, sos, numSos);
  if(smallestSphere.ContainsPoint(points[index]))
    return smallestSphere;
  sos[numSos] = points[index];
  return Welzl(points, numPoints - 1, sos, numSos + 1);
}
```

There many are ways to compute a minimum bounding sphere from a collection of points. The simplest way to think of is a n^5 algorithm where you start with all combinations of 4 points (then 3 then 2) and keep the smallest sphere that contains all other points.

Fortunately, the problem of finding a minimum bounding sphere has been well researched and there is a better method (although still slow) known as Welzl's method. This method is a randomized algorithm that is expected to run in linear time. The idea is that if you have a bounding sphere and a point, if that point is inside the sphere then we're fine, however if the point is outside the sphere then the minimum sphere needs to be recomputed and that point is on the surface of the minimum bounding sphere.

Once you grasp the flow of the algorithm, the only parts left to cover are how to compute the minimum bounding sphere of all the different number of points. For 0, 1, and 2, points the result is trivial, however 3 and 4 are not so trivial. The simple idea is to find the point equidistance from each point but taking into account several edge cases. Details of this are not shown here, although I'll give a quick outline in class.

Obb

```
struct Obb
{
  Vector3 mPosition;
  Matrix3 mBasis;
  Vector3 mHalfExtents;
};
```

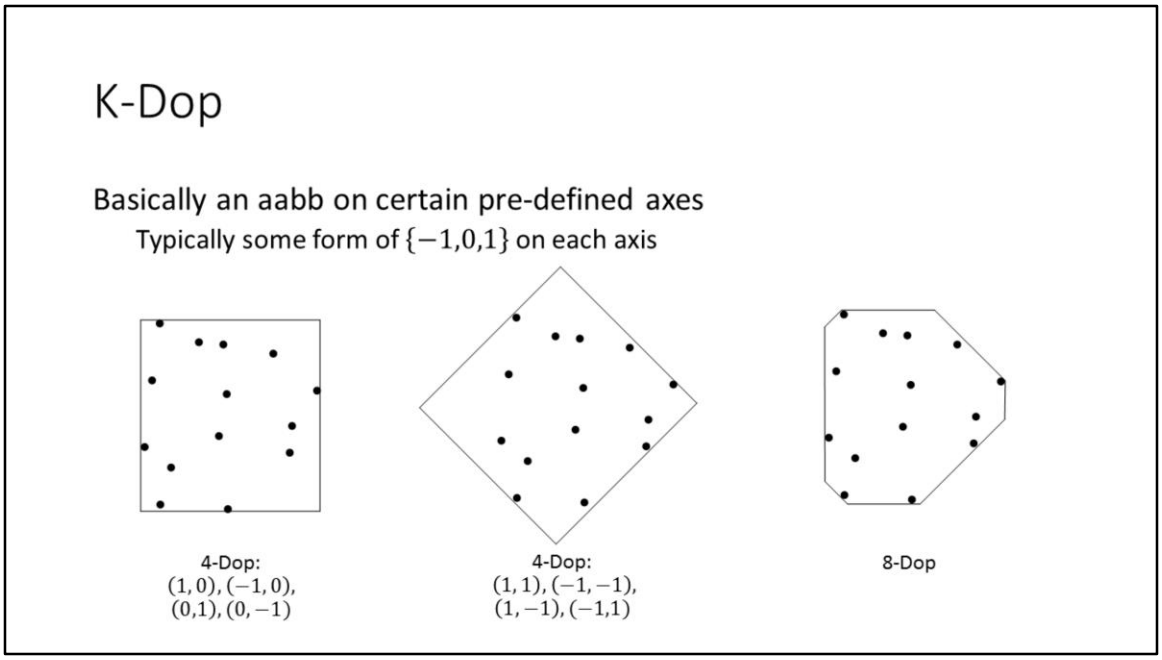
Not easy to compute – which axes?

Use PCA

Expensive intersection:

15 axis checks (more in SAT lecture)

Obbs (oriented bounding boxes) are a more tight fitting bounding volume than aabbs and spheres, however the computational cost of computing them is higher. On top of that, intersection tests are significantly more expensive (full SAT test required of 15 axes). Because of this I don't recommend using them and I won't go into any more detail on them.



One tighter fit bounding volume is what's known as a k-dop (Discrete-oriented polytope). K-Dops are very similar to Aabbs, except that we pick several other axes. For instance, a 6-Dop is equivalent to an aabb. Typically, the axes used are only of some form of $\{-1, 0, 1\}$. K-dops are fairly easy to compute, as they are just performing some dot products to find the furthest value in a given direction.

Intersections for k-dops are easy as it's just an extension of the aabb check, that is check all "axes" and see if the projection interval overlaps.
Updates are not as easy. The same methods to update an aabb can be used on a k-dop, albeit they are more expensive and complicated to implement.

Convex Hull

Best convex approximation possible

Expensive to compute
Expensive to intersect

At the extreme end of tight fit bounding volumes is the convex hull. A convex hull is the tightest fit convex shape that wraps your object. While they are tight fit, they are also fairly expensive to compute and test for intersection.

One common method of computing the convex hull is to use the quick-hull algorithm. At a high level, quick-hull sub-divides the points into two sides. On each side the point furthest in the direction of the normal is computed and then those points are split into the two new sides. This continues until all points are contained. 3d quick-hull follows the same algorithm but has a few more edge cases to deal with.

Another form of convex hull computation which is sometimes more beneficial is approximate convex hulls. The idea is to not take the tightest fit hull, but some approximation that will be good enough and require less detail.

To intersect convex hulls you either need to test all half-spaces or use some generic convex algorithm such as GJK (which will be covered later).

Questions?