Data Structures & Algorithms for Geometry

- Agenda:
  - More bounding volumes
    - Spheres
    - Oriented bounding boxes (OBBs)
    - k-DOPs
  - Bounding volumes for visibility culling
    - BV-frustum intersection
  - First assignment

Bounding Spheres

- Very commonly used

Bounding Spheres

- Very commonly used
  - Used last year in VGP351 for collision detection.

Bounding Spheres

- Very commonly used
  - Used last year in VGP351 for collision detection.
- Simple compact representation

Bounding Spheres

- Very commonly used
  - Used last year in VGP351 for collision detection.
- Simple compact representation
  - Store the location of the center and the length of the radius...just 4 floats

Bounding Spheres

- Very commonly used
  - Used last year in VGP351 for collision detection.
- Simple compact representation
  - Store the location of the center and the length of the radius...just 4 floats
- Intersection test is very simple

Bounding Spheres

- Very commonly used
  - Used last year in VGP351 for collision detection.
- Simple compact representation
  - Store the location of the center and the length of the radius...just 4 floats
- Intersection test is very simple
  - If distance between centers < sum of radii, then intersect.

Bounding Spheres

- Very commonly used
  - Used last year in VGP351 for collision detection.
- Simple compact representation
  - Store the location of the center and the length of the radius...just 4 floats
- Intersection test is very simple
  - If distance between centers < sum of radii, then intersect.
- Update also trivial

Bounding Spheres

- Very commonly used
  - Used last year in VGP351 for collision detection.
- Simple compact representation
  - Store the location of the center and the length of the radius...just 4 floats
- Intersection test is very simple
  - If distance between centers < sum of radii, then intersect.
- Update also trivial
  - Transform center with object's transform.

Sphere Creation

- Non-trivial exercise.
  - Good thing the update procedure is so trivial!
- A variety of algorithms exist
  - Brute-force minimum sphere is $O(n^5)$.

Sphere Creation

- Non-trivial exercise.
  - Good thing the update procedure is so trivial!
- A variety of algorithms exist
  - Brute-force minimum sphere is $O(n^5)$.
  - Statistical methods can be used to produce a good approximation in $O(n)$.

Sphere Creation

- Non-trivial exercise.
  - Good thing the update procedure is so trivial!
- A variety of algorithms exist
  - Brute-force minimum sphere is $O(n^5)$.
  - Statistical methods can be used to produce a good approximation in O(n).
  - A recursive method can produce minimum sphere in O(n), but a robust implementation is complex.

Sphere Creation

- Non-trivial exercise.
  - Good thing the update procedure is so trivial!
- A variety of algorithms exist
  - Brute-force minimum sphere is $O(n^5)$.
  - Statistical methods can be used to produce a good approximation in O(n).
  - A recursive method can produce minimum sphere in O(n), but a robust implementation is complex.
  - An iterative approach can get within 5% of minimum in O(n), but has a higher constant factor.

Brute-force

- A plane is defined by 3 non-colinear.
- A sphere is defined by 3 points on a plane and one additional point not on the plane.
  - In other words, a tetrahedron...4-sided die for the D&D geeks. ;)
- Consider the sphere defined by all combinations of 4 non-coplanar points, keep the smallest that contains all the points.

Ritter's Algorithm

- Given an initial guess that is too small, can find bounding sphere within 10% of minimum.
- Easy to understand and easy to implement.
  - I implemented a version in 68000 assembly years ago.

Ritter's Algorithm (cont.)

```
void bounding_sphere(Sphere &sphere, vector *p, unsigned num)
{
    float r_squared = sphere.radius * sphere.radius;

    for (unsigned i = 0; i < num; i++) {
        const vector d = p[i] - sphere.center;
        const float dist_squared = d.dot3(d);

        if (dist_squared > r_squared) {
            const float dist = sqrt(dist_squared);
            const float r = (sphere.radius + dist) / 2.0f;
            const float k = (r - sphere.radius) / dist;

            sphere.radius = r;
            sphere.center += d * k;
            r_squared = r * r;
        }
    }
}
```

Ritter's Algorithm (cont.)

- What's the big assumption in this algorithm?

Ritter's Algorithm (cont.)

- What's the big assumption in this algorithm?
  - That we have a *good* way to come up with an initial sphere.
  - The better our initial estimate is, the better the final result.

Statistical Estimation

- Definitions:
  - Mean – sum of all elements divided by number of elements (aka average). Describes the central "location" of a random distribution.

$$u = \frac{1}{n} \Sigma_{i=1}^{n} x_i$$

  - Variance – sum of the squared difference between actual values and expected values. Describes how spread out a distribution is.

$$\sigma^2 = \frac{1}{n} \Sigma_{i=1}^{n} (x_i - u)^2 = \frac{1}{n} \left(\Sigma_{i=1}^{n} x_i^2 \right) - u^2$$

  - Standard deviation – square root of the variance.

Extending to Multiple Dimensions

- Mean is calculated the same way, but is a vector instead of a scalar.
- Covariance becomes a matrix:

$$C_{ij} = \frac{1}{n} \Sigma_{k=1}^{n} \left(P_{k,i} - u_i \right) \left(P_{k,j} - u_j \right)$$

$$C_{ij} = \frac{1}{n} \left(\Sigma_{k=1}^{n} P_{k,i} P_{k,j} \right) - u_i u_j$$

  - Here *i* and *j* are elements of the source vectors.

Principal Components Analysis

- Covariance by itself does nothing for us.
  - A statistical technique called *principal components analysis* (PCA) can help us.
- We first calculate the *eigenvectors* and *eigenvalues* of the covariance matrix.
  - Eigenvector - vector that is either left unaffected or simply multiplied by a scale factor after the transformation (from Wikipedia).
  - Eigenvalue – Scale factor of a non-zero eigenvector.

Eh?

- The eigenvector with the largest eigenvalue is the axis along which the original data has the largest variance.
- Similarly the eigenvector with the smallest eigenvalue is the axis along which the original data has the smallest variance.

Ah!

- The eigenvector with the largest eigenvalue is the axis along which the original data has the largest variance.
- Similarly the eigenvector with the smallest eigenvalue is the axis along which the original data has the smallest variance.
- If we know the axis with the largest variance, we can find the two widest spread points along that axis to get our initial sphere estimate!

Welzl's Algorithm

- If we have a bounding sphere, *S*, for set of points, *P*, and we add a point, *U*, that “extends” the sphere, we **know** that *U* is on the boundary of the new sphere.

Welzl's Algorithm

- If we have a bounding sphere, *S*, for set of points, *P*, and we add a point, *U*, that “extends” the sphere, we **know** that *U* is on the boundary of the new sphere.
  - We can track the points on the boundary of the current sphere in a “support set.”

Welzl's Algorithm (cont.)

- On each iteration, remove a point, *U*, from the set, and invoke the algorithm on the remaining set.
- If *U* is inside the returned sphere, return that sphere now.
- If *U* is outside the sphere, add it to the support set and *re*-invoke the algorithm with the remaining set.

Welzl's Algorithm (cont.)

- At the tail of the recurrsion (when the point set is empty) return the sphere created from the at most 4 points in the support set.

Welzl's Algorithm (cont.)

- This algorithm is a bit complicated to think about, but that's not the only problem.
  - There are two recursions, and the first one can easily cause a stack overflow.
  - That can be worked around, but complicates things futher.
- Inspite of all that, it still runs in *expected* O(n) time and yields a minimum bounding sphere.

Ritter's Algorithm Revisited

- Remember that Ritter's algorithm needs a good initial guess?

Ritter's Algorithm Revisited

- Remember that Ritter's algorithm needs a good initial guess?
- Use the output of one iteration to seed the next!
  - Take the result and shrink it a bit.
  - Add the points in random order.
  - Lather, rinse, repeat.

Break

You've earned it!

Oriented Bounding Boxes (OBBs)

- OBBs have some interesting subtleties.

Oriented Bounding Boxes (OBBs)

- OBBs have some interesting subtleties.
- Update is trivial.
  - Perform one matrix multiply, and transform one point.

Oriented Bounding Boxes (OBBs)

- OBBs have some interesting subtleties.
- Update is trivial.
  - Perform one matrix multiply, and transform one point.
- Intersection test more complex than spheres or AABBs
  - Can use a similar overlap test, but it is more complex and requires more computation.

Oriented Bounding Boxes (OBBs)

- OBBs have some interesting subtleties.
- Update is trivial.
  - Perform one matrix multiply, and transform one point.
- Intersection test more complex than spheres or AABBs
  - Can use a similar overlap test, but it is more complex and requires more computation.
- Creation of an *optimal* OBB is challenging.

OBB Representation

- How would you represent an OBB?

OBB Representation

- How would you represent an OBB?
- Storing 8 points seems like an obvious choice, but has some drawbacks.
  - Requires a lot of storage: 8 points $\times$ 3 floats $\times$ 4 bytes = 96 bytes per OBB.
  - Leads to suboptimal overlap test.

OBB Representation

- Best method is an extension of the best AABB representation:
  - Store the center, per-axis radii, *and* a transformation (rotations only) matrix.
- To update, simply transform the center and append the object's transformation to the OBBs base transform.

OBB Intersection

- Surprisingly complicated.
  - Can't just test box extent overlaps like AABBs.
  - Can't just test corners of box A to see if they are in box B.
- Have to use the *Separating Axis Test*.
  - We'll cover this in more detail when we get to chapter 5.

Separating Axis Test

- Find an axis in space that we can project the BVs and have them *not* overlap.
  - Simplified version for AABBs: project onto the principal axes.
  - For OBBs, there are 15 axes that must be tested.
    - Full mathematical proof is beyond our scope.
    - Table 4.1 in the textbook lists them.
- Note: the test is made efficient by transforming one OBB to the other OBBs coordinate system.

Separating Axis Test

- Find an axis in space that we can project the BVs and have them *not* overlap.
  - Simplified version for AABBs: project onto the principal axes.
  - For OBBs, there are 15 axes that must be tested.
    - Full mathematical proof is beyond our scope.
    - Table 4.1 in the textbook lists them.
- Note: the test is made efficient by transforming one OBB to the other OBBs coordinate system.
  - Use the inverse of the OBBs base transform.

OBB Creation

- Any thoughts?

OBB Creation

- Any thoughts?
  - Could probably start using a bounding sphere to estimate longest axis.

OBB Creation

- Any thoughts?
  - Could probably start using a bounding sphere to estimate longest axis.
  - If we have the convex hull, we know that one of the sides of the hull **must** be coplanar with one side of the OBBs. Could probably get an $O(n^2 \log n)$ from that.

OBB Creation

- Any thoughts?
  - Could probably start using a bounding sphere to estimate longest axis.
  - If we have the convex hull, we know that one of the sides of the hull **must** be coplanar with one side of the OBBs. Could probably get an $O(n^2 \log n)$ from that.
  - Sphere calculation is a good idea...could we apply PCA to get an OBB?

PCA for OBB

- Once we have the eigenvectors and eigenvalues, we have the axes for the OBB.
  - After normalizing, these can be used as the base transform for the OBB.

PCA for OBB

- Once we have the eigenvectors and eigenvalues, we have the axes for the OBB.
  - After normalizing, these can be used as the base transform for the OBB.
- The bad news is that PCA based OBBs are not optimal.
  - Non-uniform distribution of object points can skew the calculation.
  - Using the convex hull helps but isn't a silver bullet.

Improving PCA-based OBBs

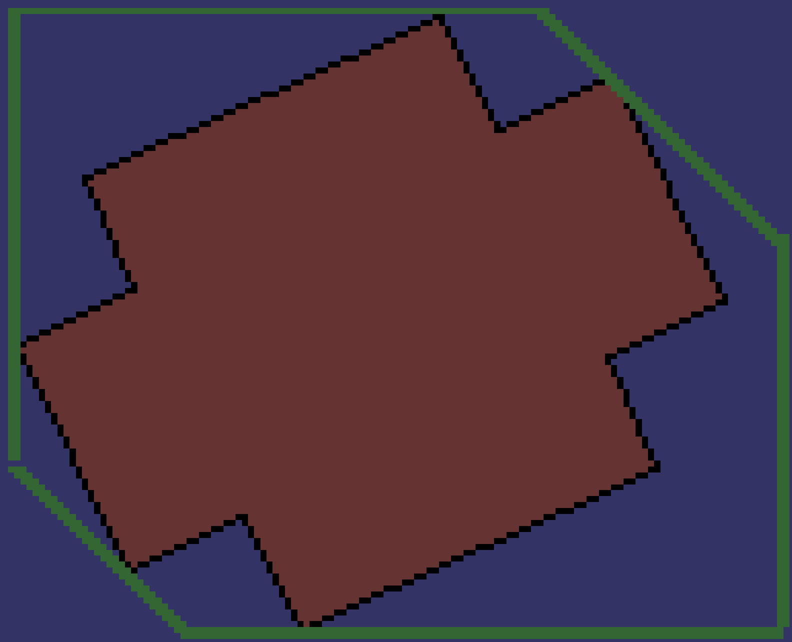
- Start by projecting all points onto the plane defined by the *minimum* eigenvector.
- Then find the minimum area rectangle enclosing the points.
  - This rectangle defines the other two edges of the OBB.
  - Compute in O(n log n) by computing the 2D convex hull and testing each rectangle that has a side colinear with a side of the hull.
- Repeat on the new OBB.

k-DOPs

- Select *n* axes.
  - The same axes are used for all objects.
  - Selected in advance and, typically, hard-coded.
- Find the minimum and maximum distances from each axis.
- Store these 2*n* values.
  - $2n = k$

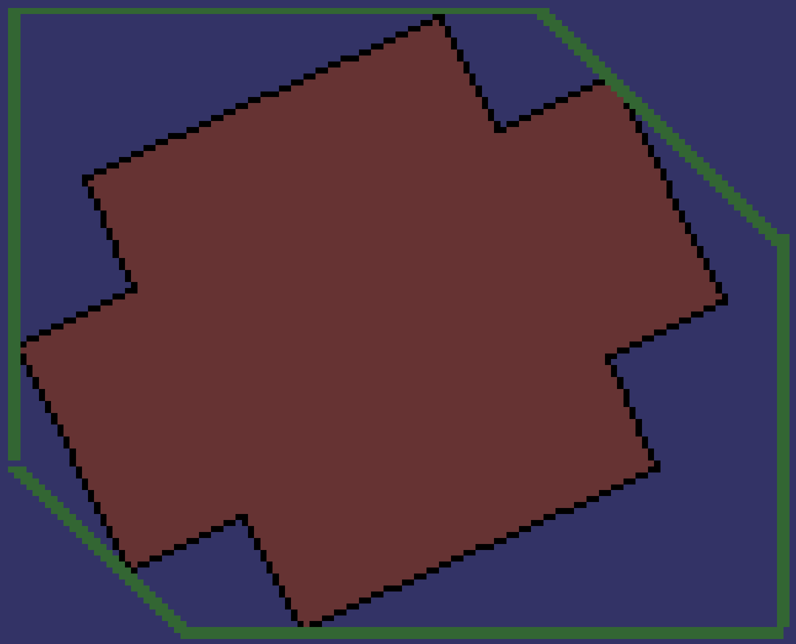
Example

- 2D 6-DOP
  - Note the improvement over an AABB



Example

- 2D 6-DOP
  - Note the improvement over an AABB
  - Notice that removing one axis would make a 4-DOP that is an AABB.



k-DOP Intersection Test

- Since AABBs are really k-DOPs, we can generalize the AABB intersection test.

```
bool kdop_intersect(kdop &a, kdop &b)
{
    for (unsigned i = 0; i < a.k / 2; i++) {
        if (a.min[i] > b.max[i]
            || a.max[i] < b.min[i])
            return false;
    }

    return true
}
```

k-DOP Update

- Again, think of k-DOPs as a generalization of AABBs, and apply the same techniques.

BV Intersections with Frustums

- Of fundamental importance: determine which side of a plane, *P*, a point, *p*, is on.
  - We call the side of the plane with the normal the “positive” side and the other side the “negative” side.
  - The formal name for a side is *half-space.*
- Plug *p* into the plane equation of *P*.

$$(n_p \cdot p) + d_P$$

  - If the result is negative, the point is in the negative half-space.

Point in Frustum Test

- A frustum is defined by 6 planes.
  - Assume the normals point *out*.
- A point is inside the frustum if it is in the negative half-space of every plane.

Sphere in Frustum Test

- “Grow” the frustum by the radius of the sphere.
  - Move each plane in the direction of it's normal by the radius of the sphere.

Sphere in Frustum Test

- "Grow" the frustum by the radius of the sphere.
  - Move each plane in the direction of it's normal by the radius of the sphere.

$$(n_i \cdot p) + (d_i + r_{sphere})$$

Sphere in Frustum Test

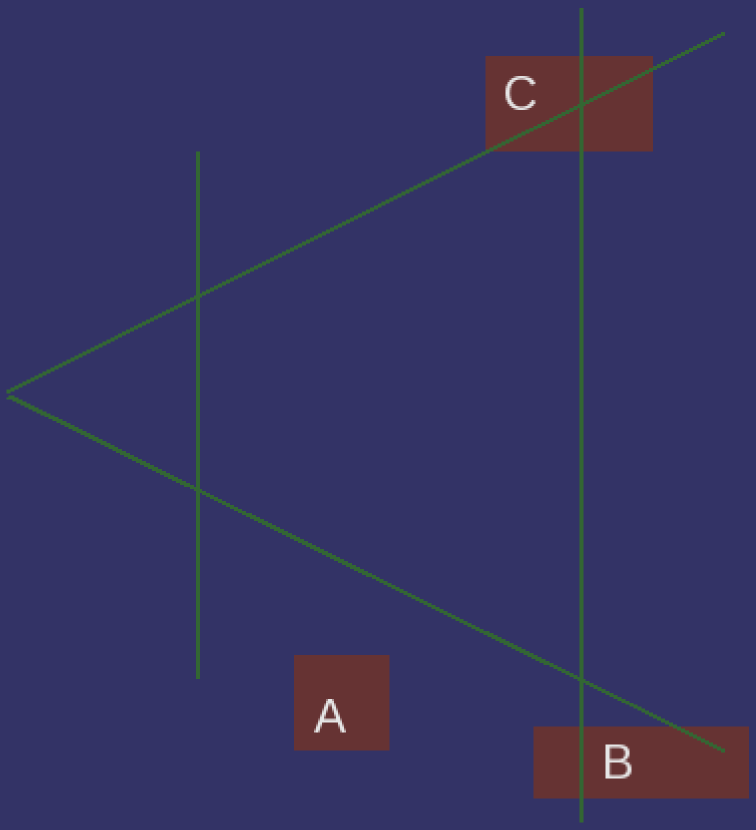
- "Grow" the frustum by the radius of the sphere.
  - Move each plane in the direction of it's normal by the radius of the sphere.

$$(n_i \cdot p) + (d_i + r_{sphere})$$

  - Treat the sphere as a point (i.e., shrink the sphere by its radius), and test the point against the new frustum.

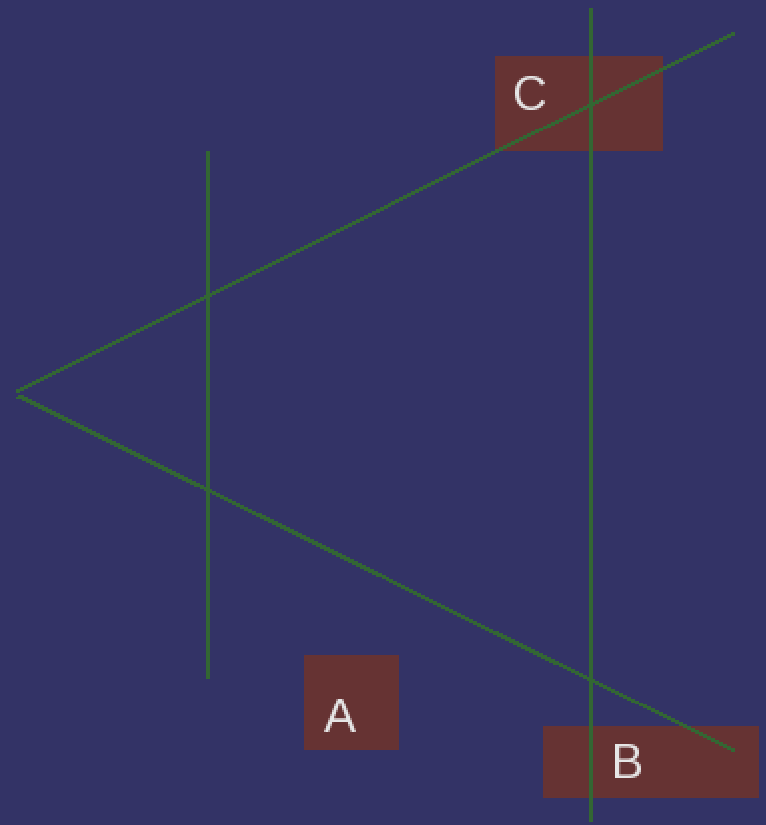
Box in Frustum Test

- Test each corner of the box. If all corners are outside the frustum, then box is outside.



Box in Frustum Test

- ~~Test each corner of the box. If all corners are outside the frustum, then box is outside.~~ Wrong!
- If all corners are on positive side of any one plane, then the box is outside.



Better Box / Frustum Test

- Lots of extra tests.
  - We don't need to test all 8 points.

Better Box / Frustum Test

- Lots of extra tests.
  - We don't need to test all 8 points.
- Pick the points that should be “most positive” and “most negative” for each plane.
  - Call these the *p-vertex* and the *n-vertex*.
- Just test those points.
  - If both are on the same side of the plane, then **all** of the points must be on that same side.

Finding n-vertex and p-vertex

- Assume the frustum is in the box's coordinate space.
- Look at the signs of the components of the plane's normal.
- Use the signs to determine which corner the normal points toward.
  - Example: If the normal signs are { +, +, - }, then the p-vertex is { box.radius.x, box.radius.y, -box.radius.z }.

Pseudo Code

```
int frustum_aabb(Plane *planes, Aabb &aabb)
{
    bool intersect = false;
    for (unsigned i = 0; i < 6; i++) {
        vector vn =
          get_negative_far_point(planes[i], aabb);
        if (vn.dot3(planes[i].n) + planes[i].d > 0)
            return OUTSIDE;

        vector vp =
          get_positive_far_point(planes[i], aabb);
        if (vp.dot3(planes[i].n) + planes[i].d > 0)
            intersect = true;
    }

    return (intersect) ? INTERSECTING : INSIDE;
}
```

References

http://www.ce.chalmers.se/~uffe/vfc_bbox.pdf

http://www.ce.chalmers.se/~uffe/vfc.pdf

Break

Next week...

- Convex hulls (this time for sure!)
- Bounding volume hierarchies
  - Building
  - Traversing
  - Merging
- Assignment #1 due.
- Assignment #2 assigned.

Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM or the Art Institute of Portland.
- OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.
- Khronos and OpenGL ES are trademarks of the Khronos Group.
- Other company, product, and service names may be trademarks or service marks of others.