Collision Detection

Steve Rotenberg
CSE291: Physics Simulation
UCSD
Spring 2019

Collision Detection

- *Collision detection* is the geometric process of determining whether two (or more) objects intersect
- It is closely related to *collision response*, which is the physics process of determining collision and contact forces
- However, it is still separate and can be designed as a self-contained module that purely does geometry analysis and no physics
- The higher level physics code can use the collision detection code to determine if collisions occur and obtain the geometric results of the collision test

Collision Results

- There are different types of results that we may be interested in from our collision tests
  - For example, in some cases, we simply need to know if two objects intersect, but we don’t need any other information
  - In other cases, we want to obtain some basic information about the collision point, such as the position and normal
  - In other cases, we may want a full 3D intersection geometry generated for the overlapping region

Collision Phases

- *Narrow phase*: geometric intersection testing of individual primitives
- *Mid phase*: optimization layer to handle objects made from many primitives (1000s of triangles, etc.)
- *Broad phase*: top-level optimization layer focused on reducing the number of pairs of objects tested for collisions (1000s of objects, etc.)

Collision Primitives

- There are various *primitives* we may want test for collisions, such as:
  - Point
  - Line
  - Triangle
  - Sphere
  - Ellipsoid
  - Cylinder, capsule
  - Axis-aligned box
  - Oriented box
  - k-DOP
  - Convex polyhedron
  - Nonconvex ‘watertight’ polyhedron
  - Nonconvex ‘polygon soup’
- Each of these tends to require dedicated code for the collision analysis, so supporting n different primitive types may require up to $n(n+1)/2$ primitive-primitive tests to be implemented
- Most systems work with a subset of this list and may have some limitations on what can be collided with what

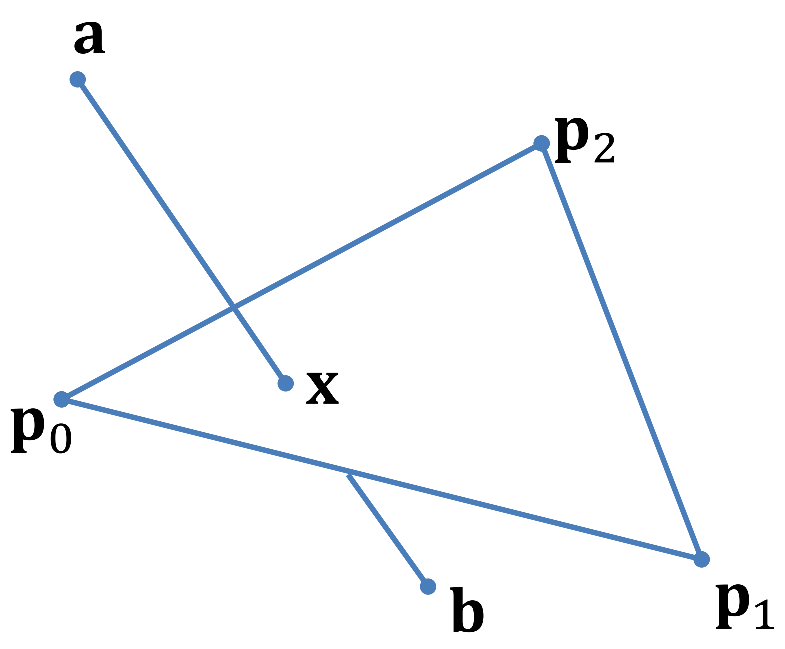
Discrete vs. Continuous

- Some collision testing algorithms are discrete meaning that they analyze a single discrete moment in time, where every object has a single configuration
- Other methods are continuous meaning they consider an initial and final configuration over a time step and determine if and when a collision happened at any point in between

Basic Intersection Tests

Line Segment vs. Triangle

- Consider the case of testing a line segment $\mathbf{ab}$ against triangle $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$:



Line Segment vs. Triangle

- First, compute signed distances of $\mathbf{a}$ and $\mathbf{b}$ to the plane:

$$d_a = (\mathbf{a} - \mathbf{p}_0) \cdot \mathbf{n}$$
$$d_b = (\mathbf{b} - \mathbf{p}_0) \cdot \mathbf{n}$$

- Where $\mathbf{n}$ is the unit length normal of the triangle
- Reject if both are above or both are below plane
- Otherwise, find intersection point $\mathbf{x}$:

$$\mathbf{x} = \frac{d_a\mathbf{b} - d_b\mathbf{a}}{d_a - d_b}$$

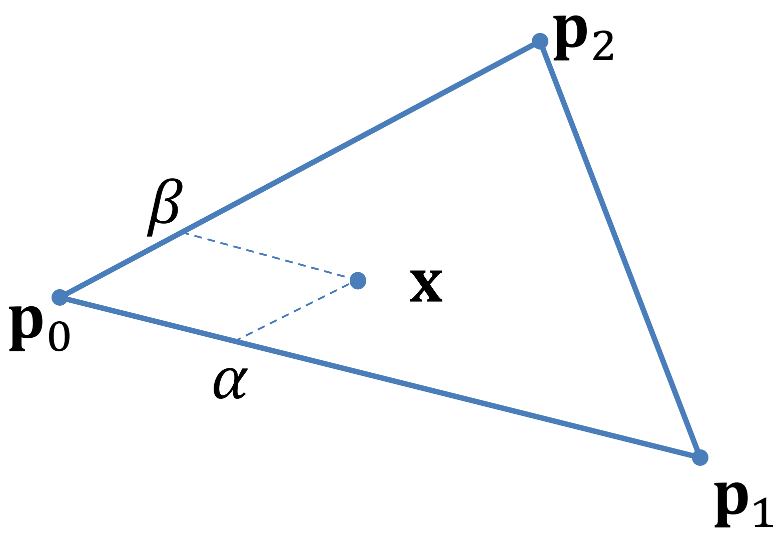
Point Inside Triangle

- To determine if the point $\mathbf{x}$ is inside the triangle, we compute the barycentric coordinates α and β :

$$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$$
$$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$$
$$\mathbf{w} = \mathbf{x} - \mathbf{p}_0$$

$$\alpha = \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{v}) - (\mathbf{v}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})}$$

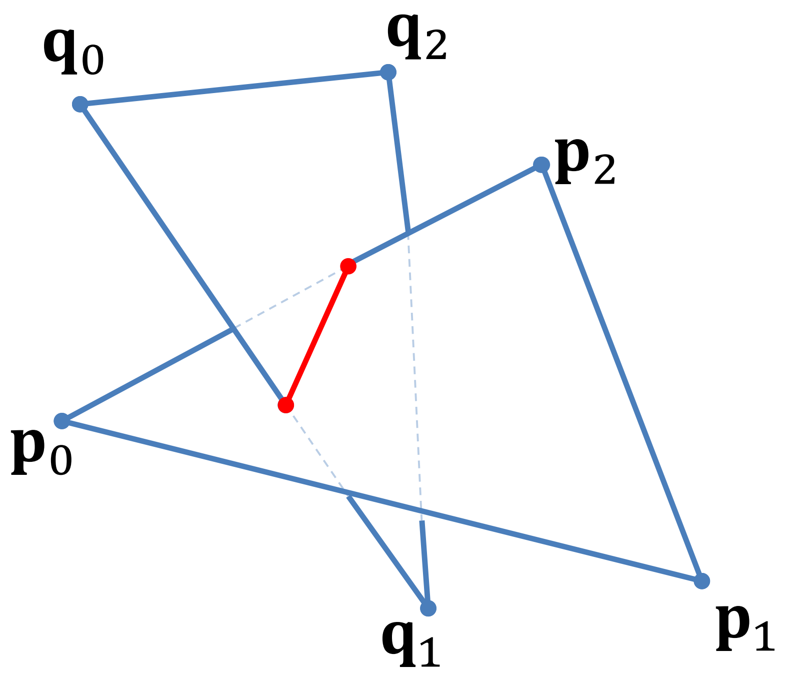
$$\beta = \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u}) - (\mathbf{u}\cdot\mathbf{u})(\mathbf{w}\cdot\mathbf{v})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})}$$



- The point is inside the triangle if $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta \leq 1$

Triangle vs. Triangle

- Consider testing triangle $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$ against triangle $\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$:



Triangle vs. Triangle

- To test two triangles, we start by computing the signed distance of the verts of each one to the plane of the other
- If they are all on one side, there is no intersection
- Otherwise, we intersect the two edges of one with the plane of the other in the same fashion as we did for a single segment
- We then clip the line segment connecting the two points to the triangle in the 2D plane
- If any part of the line segment remains, it represents the intersection of the two triangles

Sphere vs. Sphere

- Two spheres will intersect if the distance between their centers is less than the sum of their radii

$$|\mathbf{c}_1 - \mathbf{c}_2| \leq r_1 + r_2$$

- It’s quicker to avoid the square root in the distance computation by squaring both sides of this equation (which is safe, as both sides are guaranteed to be positive)

$$|\mathbf{c}_1 - \mathbf{c}_2|^2 \leq (r_1 + r_2)^2$$

Sphere vs. Line Segment

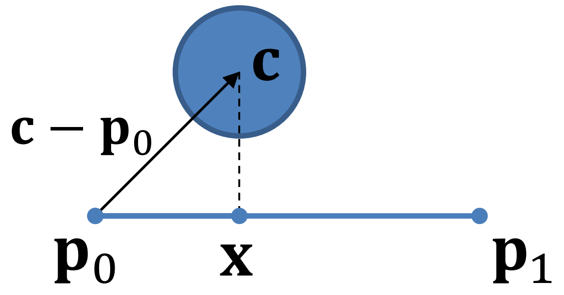
- To test a sphere with center $\mathbf{c}$ and radius r against a line segment $\mathbf{p}_0\mathbf{p}_1$, we find the parametric distance of the point along the line segment that is closest to the sphere center

$$t = (\mathbf{c} - \mathbf{p}_0) \cdot \frac{\mathbf{p}_1 - \mathbf{p}_0}{|\mathbf{p}_1 - \mathbf{p}_0|^2}$$

- If $t \leq 0$, the sphere hits $\mathbf{p}_0$ if $|\mathbf{c} - \mathbf{p}_0|^2 \leq r^2$
- Else if $t \geq 1$, the sphere hits $\mathbf{p}_1$ if $|\mathbf{c} - \mathbf{p}_1|^2 \leq r^2$
- Else if t is between 0 and 1, then the closest point between the sphere and the line is:

$$\mathbf{x} = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0)$$

- The sphere hits $\mathbf{x}$ if $|\mathbf{c} - \mathbf{x}|^2 \leq r^2$
- Otherwise, there is no intersection



Sphere vs. Triangle

- Testing sphere vs. triangle is equivalent to testing the distance of a point to a triangle:
  1. We first find the distance from the center of the sphere to the plane of the triangle. If this is greater than the radius or less than the negative radius, they don’t intersect
  2. We then use the distance to find the closest point on the plane to the center of the sphere. If this point is within the triangle, the sphere hits the plane of the triangle
  3. We then test if the sphere hits any of the edges of the triangle
  4. Finally, we test if the sphere intersects any of the vertices
- If a collision is identified in step 2, 3, or 4, each case has a straightforward way to find the contact point and normal
- This method generalizes to sphere vs. mesh, where we can save time by not testing duplicate vertices or edges

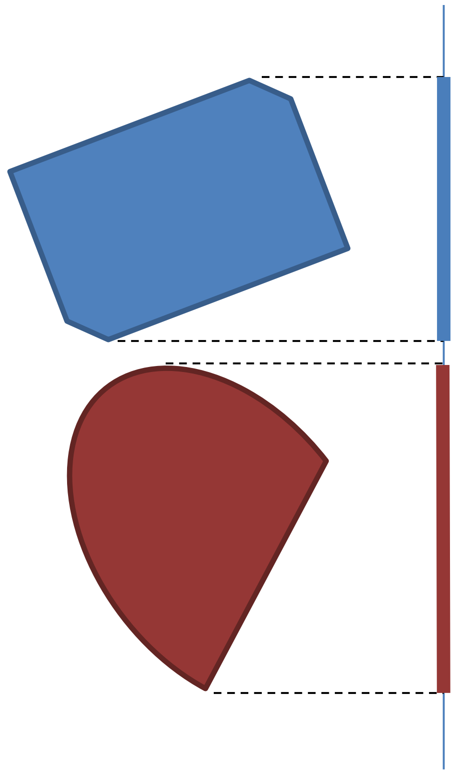
Capsules

- *Capsules* are cylinders with rounded caps
- They can also be thought of as a line segment with a radius, which leads to some fast algorithms
- They are mainly useful as bounding volumes for optimization
- Capsule vs. sphere reduces to testing the distance from a line segment to a point
- Capsule vs. capsule reduces to testing the distance between two line segments

Separating Axis Theorem

Separating Axis Theorem

- If we have two convex solids that are not intersecting, we should be able to find a separating plane such that each object is on a different side and neither intersects the plane
- Another way of looking at this is to project all of the points of both objects onto a line normal to the plane and observe that they form two separate non-overlapping intervals
- This is known as the *separating axis theorem* (SAT)



Separating Axis Theorem

- If we can find an axis such that the two objects project onto non-overlapping intervals, then we know the objects don’t intersect
- If we limit ourselves to convex polyhedra, we can identify a finite number of possible axes to test, but it may still be a large number
- If we further limit ourselves to oriented boxes, we can reduce the number of possible axes to 15

Axis Testing

- Let's say we have two convex polyhedra and we want to project them onto an axis $\mathbf{a}$ to test for overlap
- For each vertex $\mathbf{x}_i$ of an object, we find its projection onto the axis as $\mathbf{x}_i \cdot \mathbf{a}$
- We project all of the verts of each object and keep track of the min & max value for each
- Then, we simply check if the min/max ranges overlap
- If they don't overlap, then we know for sure that the two objects don't intersect
- If they do overlap, we simply know that this particular axis doesn't separate them, but it doesn't say whether or not a different axis will

Testing All Axes

- Given two non-intersecting convex polyhedra, we should be able to find a finite number of possible separating planes to test
- We can use the planes of the faces of each polyhedra as a start
- However, we can still find non-intersecting cases where none of the face planes separate them (consider two boxes nearly touching edge to edge)
- We must also test the planes formed by any edge of one object with any edge of the other object
- If f_n is the number of faces in object n and e_n is the number of edges, then the maximum number of possible axes we would have to test would be:

$$e_1 e_2 + f_1 + f_2$$

- However, we can reduce this by removing parallel planes and edges

Box Testing

- For the special case of boxes (with arbitrary rotation), this reduces down to only 3 unique planes per object and only 3 unique edges, leaving 3*3+3+3=15 total axes to test
- If each box has a 3x3 orientation matrix $\mathbf{M}_n$ whose column vectors are $\mathbf{M}_n = [\mathbf{a}_n \quad \mathbf{b}_n \quad \mathbf{c}_n]$, then we would want to test all of the following axes:

$$\begin{array}{lll}
\mathbf{a}_1 & \mathbf{b}_1 & \mathbf{c}_1 \\
\mathbf{a}_2 & \mathbf{b}_2 & \mathbf{c}_2 \\
\mathbf{a}_1 \times \mathbf{a}_2 & \mathbf{a}_1 \times \mathbf{b}_2 & \mathbf{a}_1 \times \mathbf{c}_2 \\
\mathbf{b}_1 \times \mathbf{a}_2 & \mathbf{b}_1 \times \mathbf{b}_2 & \mathbf{b}_1 \times \mathbf{c}_2 \\
\mathbf{c}_1 \times \mathbf{a}_2 & \mathbf{c}_1 \times \mathbf{b}_2 & \mathbf{c}_1 \times \mathbf{c}_2
\end{array}$$

- As soon as we find an axis that doesn’t overlap, we are done
- If all of them overlap, then the objects must intersect

Box Test Performance

- To test two boxes for overlap, we need to loop through up to 15 axes
- For each axis, we need to project 8 corners of 2 boxes, requiring 16 dot products total, as well as an equal number of min & max operations, which are typically done in a single clock cycle
- Finding the corner positions in world space requires very little effort due to symmetries in the box, and this only needs to happen once per object (i.e., can be re-used to test against other objects later)
- Overall, we would expect a single box-box test to run extremely fast on a modern CPU/GPU and this is also a very good case for SIMD vector operations

Recovering Contact Information

- If we determine that the two objects do intersect, we typically would want to know some additional information about the contact site
- The minimal information would be a position and normal
- More detailed information might include the contact manifold (intersection of the two object's boundaries), or the full contact polyhedron (intersection of the two object's volumes)

Recovering Contact Information

- If the objects intersect, then all of the axes will have some amount of overlap
- The axis with the least overlap represents the axis that allows the two objects to be separated with the least movement
- In other words, it is reasonable to base the collision point and normal off of the axis with the least overlap
- We can use the axis itself as the normal
- If the axis is based on a plane, then we can find the collision point as the point that went deepest into the plane
- If the axis is based on the cross product of edges, we find the nearest point between the two edges and use that as the position
- This process is very simple and fast and works about 99% of the time. There are occasional cases where the shallowest overlap actually doesn’t correspond to an actual collision, and we must go with the second smallest overlap. This case can be identified if we first use the shallowest one but then determine that the resulting point is not inside both boxes

Convex Collision Detection

Convex vs. Non-convex

- Collision detection with well modeled manifold non-convex geometry is difficult
- Collision detection with poorly modeled non-manifold ‘polygon soups’ is very tricky and poorly defined in many cases
- Most systems prefer to use convex objects for collision detection, as it is much easier to come up with consistent, reliable algorithms

GJK Algorithm

- The *Gilbert-Johnson-Keerthi* (GJK) algorithm is a popular method for computing the distance between two arbitrary convex objects
- One nice feature is that it can be adapted to handle intersections between different object classes (such as spheres vs. convex polyhedral) with minimal effort
- It is also an incremental algorithm that can be initialized when two objects get close enough, and then incrementally updated very quickly as they remain close
- It’s worth being familiar with it, but it is a bit complex to go over in this lecture. Google it!

Convex Decomposition

- One can take a non-convex model and automatically split it up into multiple convex models
- This is called a *convex decomposition*
- It is also popular to do an *approximate convex decomposition* (ACD), where the original model is split up into meshes that are within some tolerance of being convex
- This allows for fast, reliable collision detection of general solid objects

Convex Decomposition

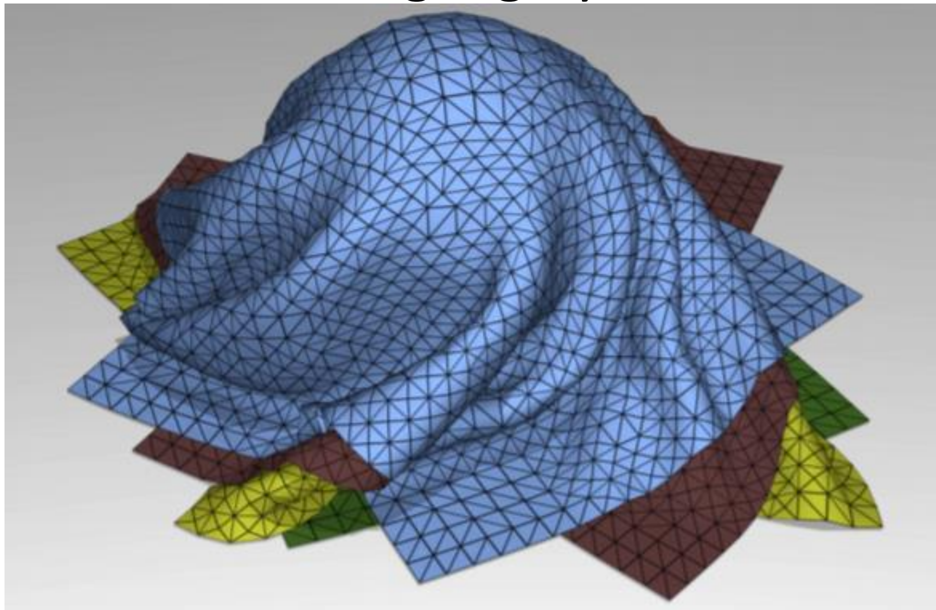
- “Approximate Convex Decomposition of Polyhedra”, Lien, Amato, 2007

- “A Simple and Efficient Approach for 3D Mesh Approximate Convex Decomposition”, Mamou, Ghorbel, 2010

Continuous Collision Detection

Continuous Collision Detection

- A variety of *continuous collision detection* methods have been devised over the years
- These methods use starting and ending configurations for each object to test if any collisions happened in the time between
- Related approaches are also useful for handling highly deformable objects like cloth



Continuous Collision Detection

- "Fast Continuous Collision Detection Between Rigid Bodies", Redon, Kheddar, Coquillart, 2002
- This paper introduced a popular method that has been extended in various ways
- The basic approach assumes that each object moves along a straight path while rotating at a constant angular velocity. This implies that any point on the objects will follow a helical path (sometimes called *screw motion*)
- Algebraic equations are set up to test the motion trajectories of points and edges of the objects and these are evaluated using *interval arithmetic* to compute bounded results

Interval Arithmetic

- An interval is defined as $[a, b] = \{x \in \mathbb{R} | a \leq x \leq b\}$, where $a = -\infty$ and $b = \infty$ are allowed

- Basic arithmetic operations are:

$$[x_1, x_2] + [y_1, y_2] = [x_1 + y_1, x_2 + y_2]$$
$$[x_1, x_2] - [y_1, y_2] = [x_1 - y_2, x_2 - y_1]$$
$$[x_1, x_2] \cdot [y_1, y_2] =$$

$$[\min(x_1 y_1, x_1 y_2, x_2 y_1, x_2 y_2)\,, \max(x_1 y_1, x_1 y_2, x_2 y_1, x_2 y_2)]$$

- As well as more advanced operations such as division

CCD References

- Some good CCD papers:
- “Fast and Exact Continuous Collision Detection with Bernstein Sign Classification”, Tang, Tong, Wang, Manocha, 2014
- “Efficient Geometrically Exact Continuous Collision Detection”, Brochu, Edwards, Bridson, 2012
- “Continuous Collision Detection for Articulated Models using Taylor Models and Temporal Culling”, Zhang, Redon, Lee, Kim, 2007
- “Interactive Continuous Collision Detection for Non-Convex Polyhedra”, Zhang, Lee, Kim, 2006

Collision Optimization

Collision Phases

- So far, we’ve mainly looked at narrow phase collision testing of primitive to primitive
- We definitely want fast narrow phase algorithms, but even the fastest narrow phase algorithms will be slow when lots of primitives are involved and any primitive may potentially intersect with any other
- To optimize cases with complex geometry, deformable geometry, and/or many moving objects, we need to use some spatial data structures

Mid-Phase Testing

- Mid-phase testing refers to the intersection testing of two complex objects, each made from potentially thousands of primitives
- It would also apply to dealing with deformable but contiguous objects, like cloth
- If we have two objects, each with 1000 triangles, we need to potentially test 1000000 triangle-triangle pairs
- Mid-phase optimization aims to drastically reduce this to as few as possible

Bounding Volume Hierarchies

- *Bounding volume hierarchies* (BVH) are tree-like data structures that group sub-trees into simple bounding volumes like spheres or boxes
- The volumes can overlap, so that a point may be contained in several leaf volumes
- Common examples include:
  - Sphere tree
  - AABB tree (axis-aligned bounding box)
  - OBB tree (oriented bounding box)
  - k-DOP tree (discreet oriented polytope)

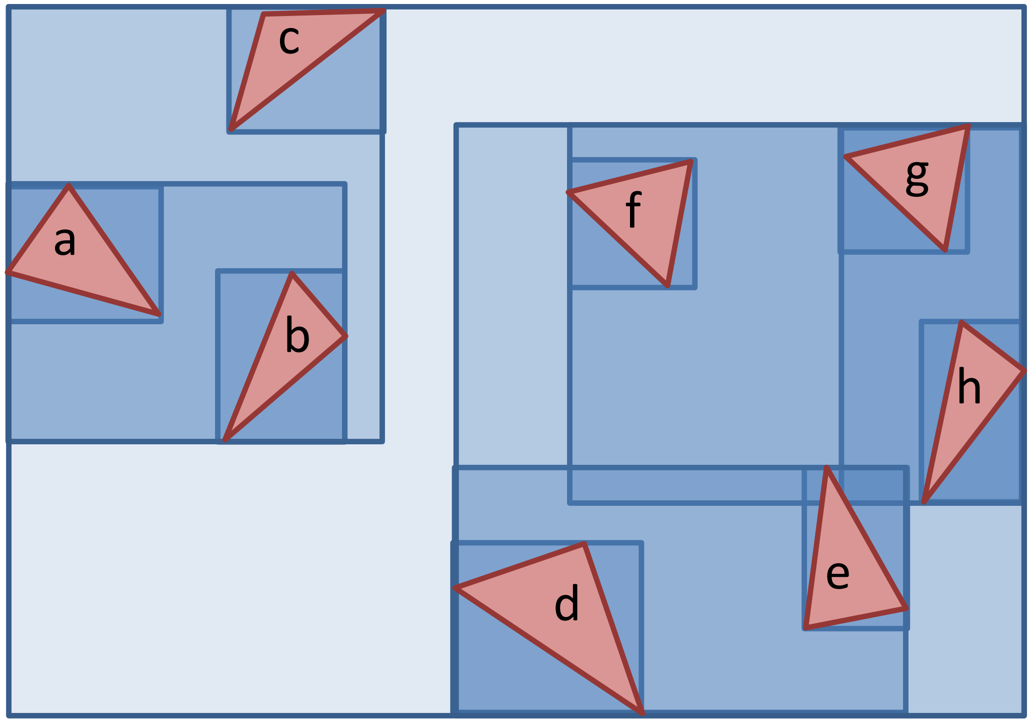
Sphere Trees

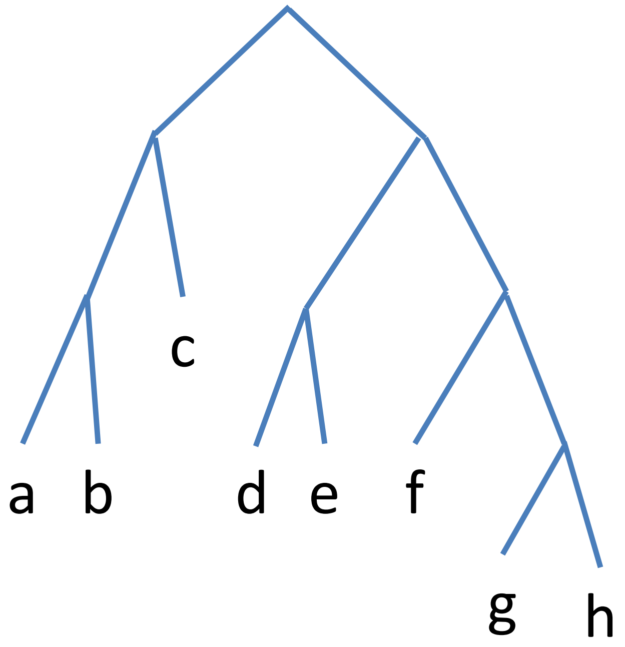
- Sphere-sphere tests are probably the fastest primitive collision test, making sphere trees a simple and tempting option for a BVH
- However, spheres are notoriously poor fitting for many objects, leading to lots of extra tests
- It is also relatively expensive to compute a tight fitting sphere around a set of points, so they are not very good for dynamic geometry

AABB Trees

- Axis-aligned bounding box (AABB) trees use a hierarchy of axis-aligned boxes making them simple and fast
- They can be generated fairly quickly, perform generally fast, and are easy to implement
- They can be updated very quickly, and so are a good choice for dynamic cases like deformable solids or scenes with many bodies
- Their main disadvantage is that they are not always a tight fit, and the tightness of the fit is also dependent on the orientation of the object

Axis Aligned Bounding Box Tree



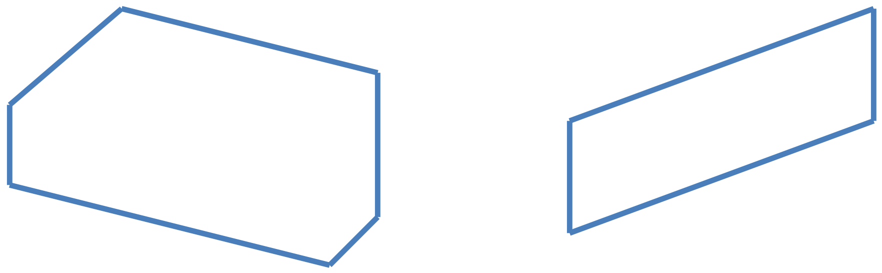


OBB Trees

- Oriented bounding box (OBB) trees use boxes that can be scaled and rotated in order to optimize how they fit around geometry
- Even though OBBs are more expensive to test than AABBs or spheres, they tend to out-perform them for rigid geometry, because they are able to fit tighter
- They are expensive to generate, and so can really only be used for rigid geometry

k-DOPs

- By the way, a k-DOP is a *discrete oriented polytope* bounded by k/2 pairs of parallel planes
- They are a very fast a potentially tightly fitting bounding volume that are preferred by some
- Their main disadvantage would be that they take more time to compute optimal bounding volumes than many other methods, so they are more applicable to static geometry



Spatial Partitions

- Spatial partitions divide up space into volumes such that a point within the top level volume will end up in exactly one leaf volume
- Hierarchical examples:
  - Octree
    - Top level volume is a cube
    - Each level splits into 8 equal cubes
  - KD-Tree (k-dimensional)
    - Top level volume is a box or infinite
    - Each level splits into 2 at arbitrary point along x, y, or z
  - BSP-Tree (binary separating plane)
    - Top level volume is a convex polyhedron or infinite
    - Each level splits into 2 along an arbitrary plane
- Non-hierarchical examples:
  - Uniform grid
  - Spatial hash table

Spatial Hash Table

- Spatial hash tables are like an infinite grid, where only occupied cells use memory
- They are excellent for scenes with a large number of similar sized objects
- It is best if no object is larger than 1 cell, so that no object can overlap more than 8 boxes
- They are ideal for situations where all objects are the same size, such as SPH fluid simulations and granular simulations
- In these cases, they should perform in linear O(*n*) time

Broad Phase Testing

- *Broad phase* testing refers to the optimizations and related data structures for dealing with a large number of distinct objects moving around and potentially colliding in a scene
- It is also referred to as *pair reduction*, as the goal is to reduce the number of potential collision pairs down from the worst case n^2 performance
- Several of the mid phase approaches are also valid here, such as AABB trees and spatial hash tables, which both can be updated quickly to accommodate moving objects
- In addition to these, the *sweep and prune* algorithm is also popular for broad phase testing

Sweep and Prune

- The basic *sweep and prune* algorithm chooses an axis (such as *x*-axis) and projects all objects to it to find their start and end interval along the axis
- Next, the min and max values for all objects are sorted into a list
- Then, we sweep down the list, maintaining an 'active list' that starts out empty
  - When we get to a new min value, we pair it with all objects on the active list and add these pairs to another list of potentially colliding pairs. We then add the new object to the active list
  - When we get to a new max value, we remove the associated object from the active list
- Once we finish with the axis, we can do the same thing for the y- and z-axes. Once we finish that, we find any pairs that overlap on all 3 axes and do further collision testing on them (mid or narrow phase)
- The basic algorithm is simple and fast and can be improved by using incremental sorting algorithms, as the sorted list changes very little from frame to frame
- There are many other enhancements to the algorithm, such as adaptations to parallel processors or GPUs