GAME2016
Mathematical Foundation of Game Design and Animation

Lecture 12

Collision Detection

Dr. Paolo Mengoni
pmengoni@hkbu.edu.hk
Senior Lecturer @HKBU Department of Interactive Media

Agenda

- What is Collision Detection
  - Useful definitions: lines and rays
- Bounding Spheres and Circles
- Bounding Boxes
- Collision Testing
- Final Considerations

What is Collision Detection

What is Collision Detection?

- Given two geometric objects, determine if they overlap.

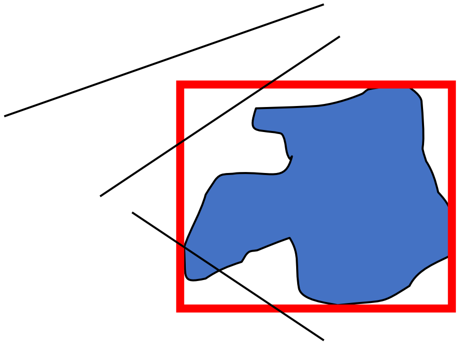
- Typically, at least one of the objects is a set of triangles.
  - Rays/lines
  - Planes
  - Polygons
  - Frustums
  - Spheres
  - Curved surfaces

When to use it

- Often in simulations.
  - Objects move – find when they hit something else
- Other examples.
  - Ray tracing speedup.
  - Culling objects/classifying objects in regions.
- Usually, needs to be fast.
  - Applied to lots of objects, often in real-time applications.

Bounding Volumes

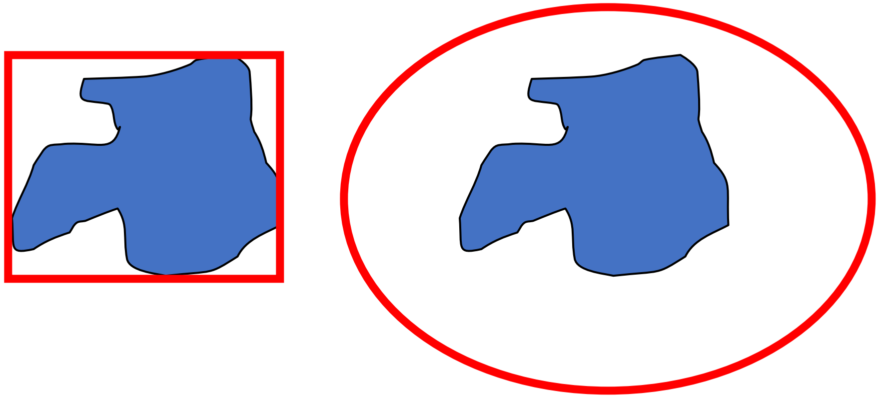
- Key idea:
  - Surround the object with a (simpler) bounding object (the bounding volume).
  - If something does not collide with the bounding volume, it does not collide with the object inside.
  - Often, to intersect two objects, first intersect their bounding volumes



- Choosing a Bounding Volume can be difficult.
  - Lots of choices, each with tradeoffs.

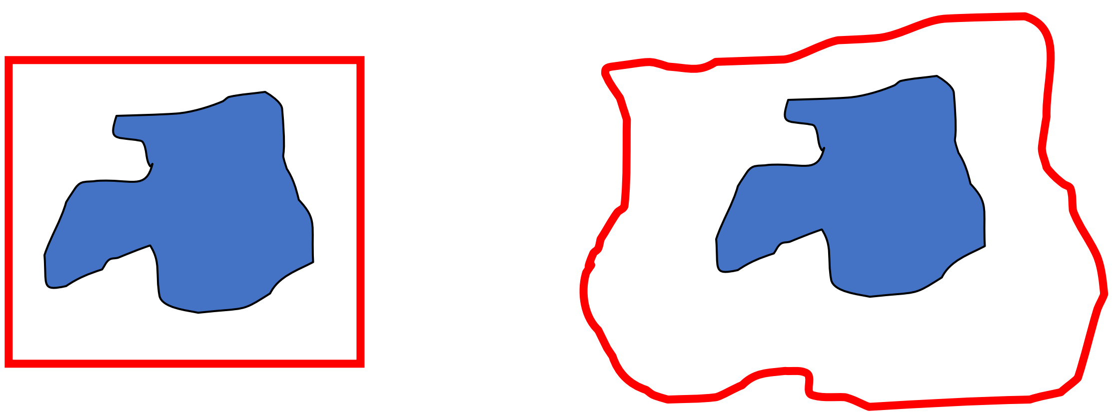
Choosing a Bounding Volume

- Lots of choices, each with tradeoffs
- Tighter fitting is better
  - More likely to eliminate “false” intersections



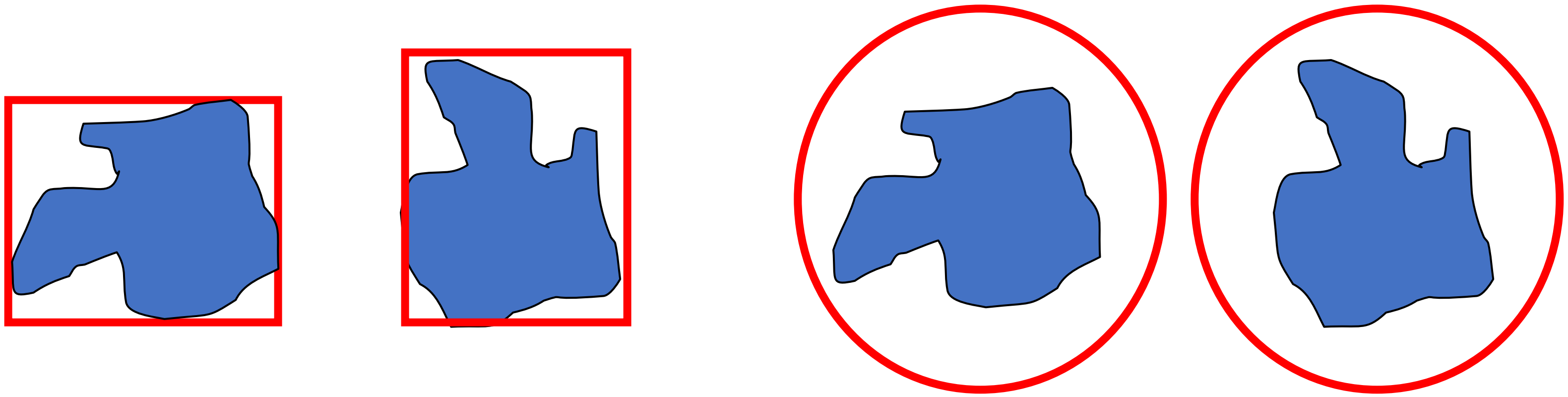
Choosing a Bounding Volume

- Lots of choices, each with tradeoffs
- Tighter fitting is better
- Simpler shape is better
  - Makes it faster to compute with



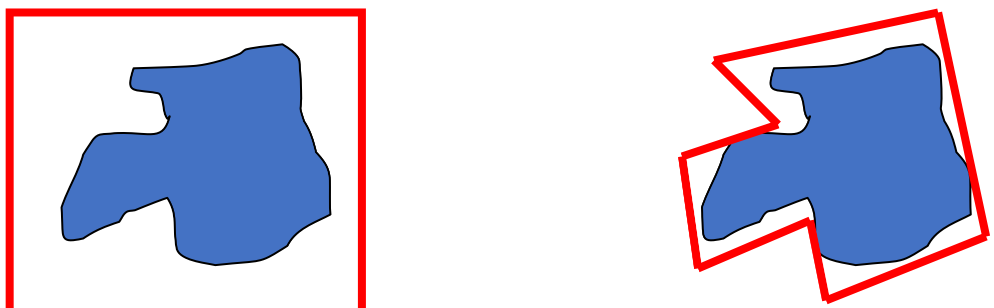
Choosing a Bounding Volume

- Lots of choices, each with tradeoffs
- Tighter fitting is better
- Simpler shape is better
- Rotation Invariant is better
  - Easier to update as object moves



Choosing a Bounding Volume

- Lots of choices, each with tradeoffs
- Tighter fitting is better
- Simpler shape is better
- Rotation Invariant is better
- Convex is usually better
  - Gives simpler shape, easier computation



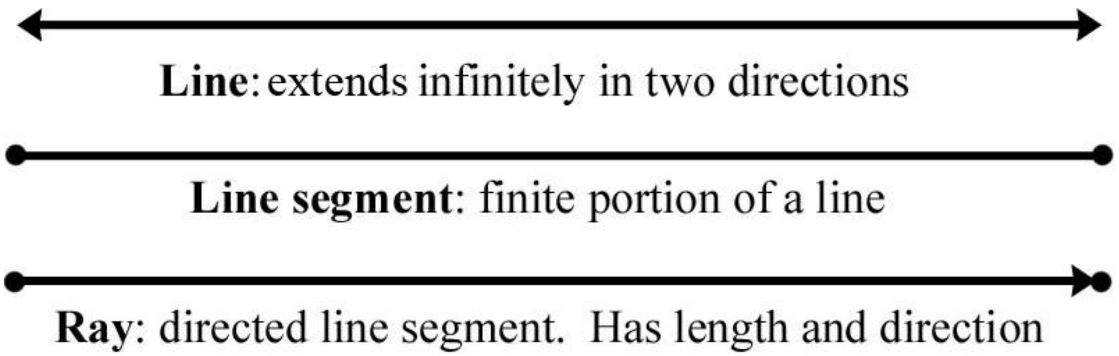
Useful definitions: lines and rays

Lines and Rays

- Classical definitions:
  - A *line* extends infinitely in two directions.
  - A *line segment* is a finite portion of a line that has two endpoints.
  - A *ray* is half of a line that has an origin and extends infinitely in one direction.
- Computer graphics definition:
  - A *ray* is a directed line segment.
    - A mix of *line segment* and *ray* in the classical definition

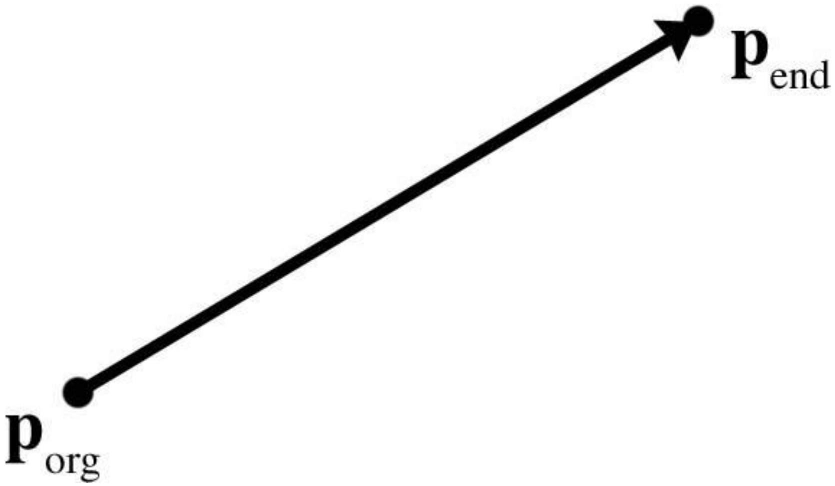
The Importance of Being Ray

- A ray will have an origin and an endpoint.
- A ray defines a position, a finite length, and (unless it has zero length) a direction.
- A ray also defines a line and a line segment.
- Rays are important in computational geometry and computer graphics.



Two Points Representation of Rays

- Give the two points that are the ray *origin* and the ray *endpoint*: $\mathbf{p}_{\text{org}}$ and $\mathbf{p}_{\text{end}}$.



Parametric Representation of Rays

Three equations in *t*:

$$x(t) = x_0 + t\,\Delta x$$
$$y(t) = y_0 + t\,\Delta y$$
$$z(t) = z_0 + t\,\Delta z$$

The parameter *t* is restricted to $0 \le t \le 1$.

Vector Notation

Alternatively, use vector notation:

$$\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{d}$$

Where:

$$\mathbf{p}(t) = [\ x(t)\ \ y(t)\ \ z(t)\ ]$$

$$\mathbf{p}_0 = [\ x_0\ \ y_0\ \ z_0\ ]$$

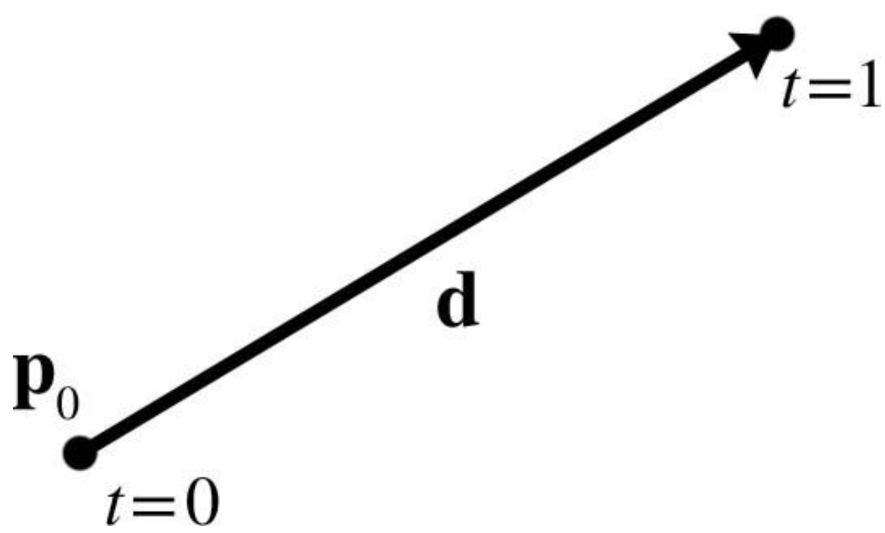
$$\mathbf{d} = [\ \Delta x\ \ \Delta y\ \ \Delta z\ ]$$

Vector Notation

$\mathbf{p}(0) = \mathbf{p}_0$ is the origin point.

$\mathbf{p}(1) = \mathbf{p}_0 + \mathbf{d}$ is the end point.

$\mathbf{d}$ is the ray's length and direction.



Variant

- Let **d** be a unit vector.
- Vary t in the range $[0, \ell]$, where ℓ is the length of the ray.
- $\mathbf{p}(0) = \mathbf{p}_0$ is the origin point.
- $\mathbf{p}(\ell) = \mathbf{p}_0 + \ell\mathbf{d}$ is the end point.
- **d** is the ray's direction.

Lines in 2D

Implicit representation of a line:

$$ax + by = d$$

Some people prefer the longer:

$$ax + by + d = 0$$

Vector notation: let $\mathbf{n} = [\, a \;\; b \,]$, $\mathbf{p} = [\, x \;\; y \,]$ and use dot product:

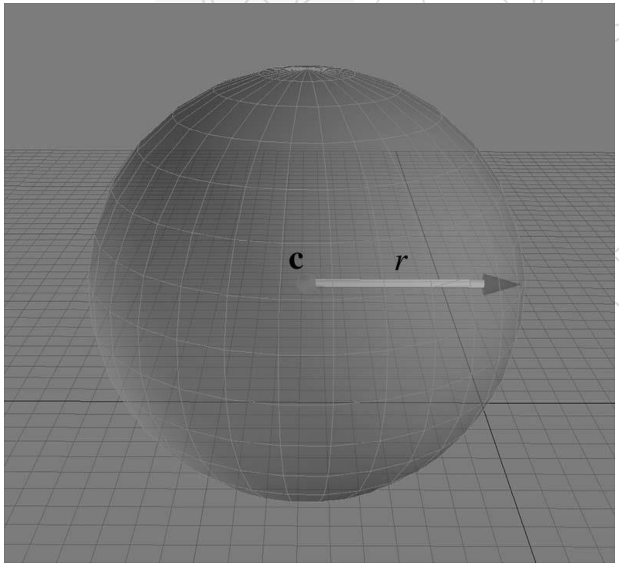
$$\mathbf{p} \cdot \mathbf{n} = d$$

Some special cases for this representation exist

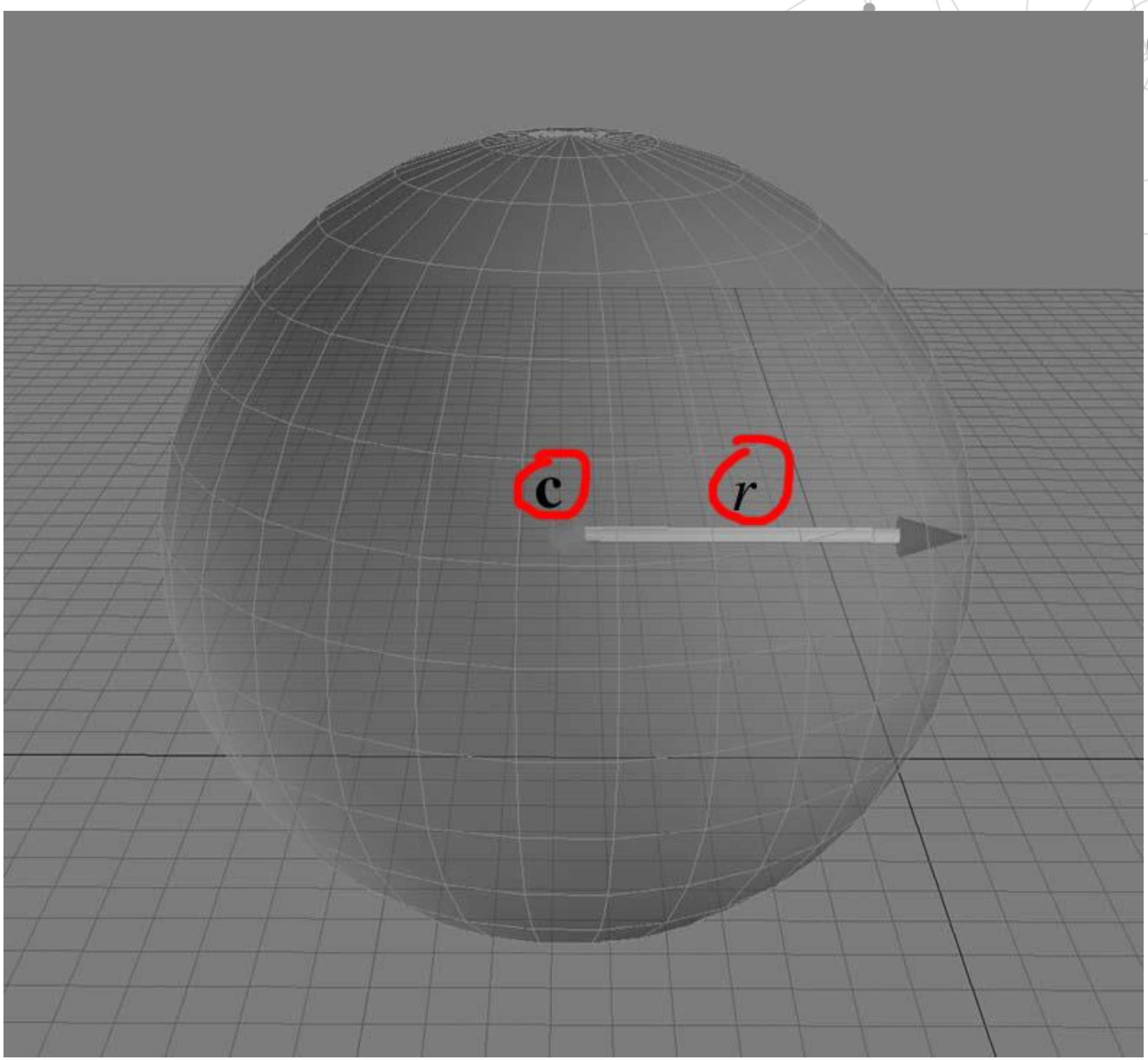
Bounding Spheres and Circles

Circles and Spheres

- A sphere is the set of all points that are a given distance from a given point.



- The distance from the center of the sphere to a point is known as the *radius* of the sphere.
- The straightforward representation of a sphere is its center *c* and radius *r*.
- A circle is a 2D sphere, of course. Or a sphere is a 3D circle, depending on your perspective.


c
r

Spheres in Collision Detection

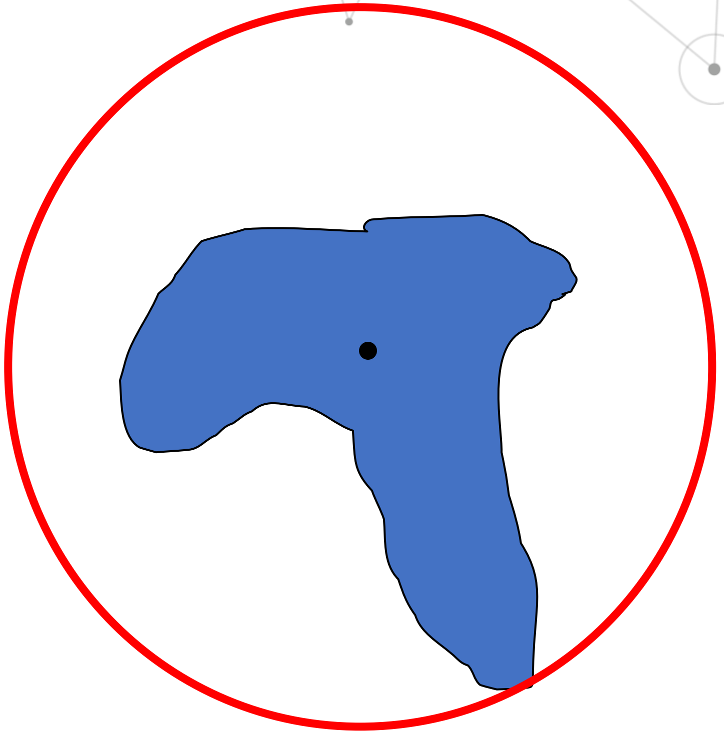
- A *bounding sphere* is often used in collision detection for fast rejection because the equations for intersection with a sphere are simple.

- Rotating a sphere does not change its shape.

- A bounding sphere can be used for an object regardless of the orientation of the object.

Common Bounding Volumes: Sphere

- Rotationally invariant
  - Usually
- Usually fast to compute
- Store: center point and radius
  - Center point: object's center of mass
  - Radius: distance of farthest point on object from center of mass.
- Often not very tight fit



Implicit Representation

The implicit form of a sphere with center **c** and radius *r* is the set of points **p** such that:

$$\|\mathbf{p} - \mathbf{c}\| = r.$$

For collision detection, **p** is inside the sphere if:

$$\|\mathbf{p} - \mathbf{c}\| \leq r.$$

Expanding this, if **p** = [*x y z*]:

$$(x - c_x)^2 + (y - c_y)^2 = r^2 \quad \text{(2D circle)}$$

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2 \quad \text{(3D sphere)}$$

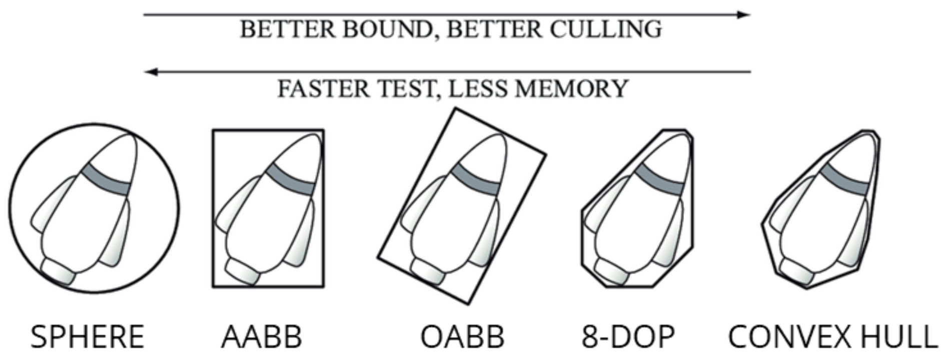
Bounding Boxes

Types of Bounding Box

- Like spheres, bounding boxes are also used in collision detection.

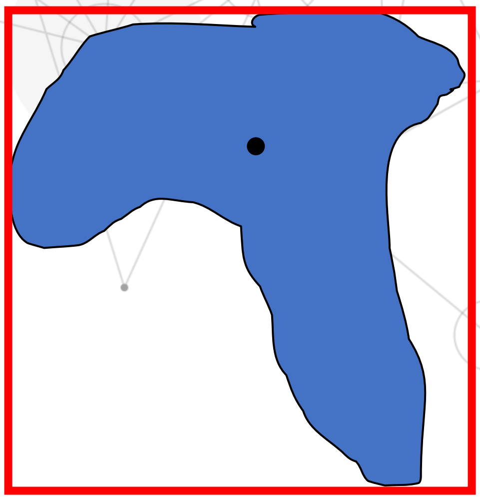
- AABB: *axially aligned bounding box*.
  - sides aligned with world axes
- OBB: *object aligned bounding box*.
  - sides aligned with object axes
- K-DOP: k-discrete oriented polytopes
- Convex Hull

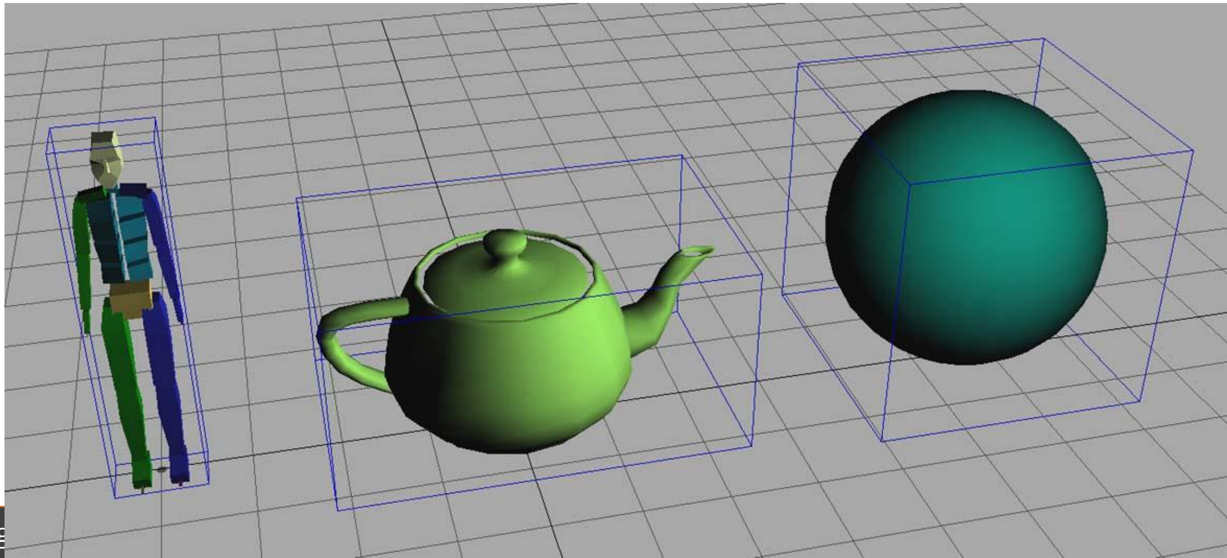
- Axially aligned bounding boxes are simpler to create and use.



Axis Aligned Bounding Box (AABB)

- Very fast to compute
- Store: max and min along x,y,z axes.
  - Look at all points and record max, min
- Moderately tight fit
- Must update after rotation, unless a loose box that encompasses the bounding sphere



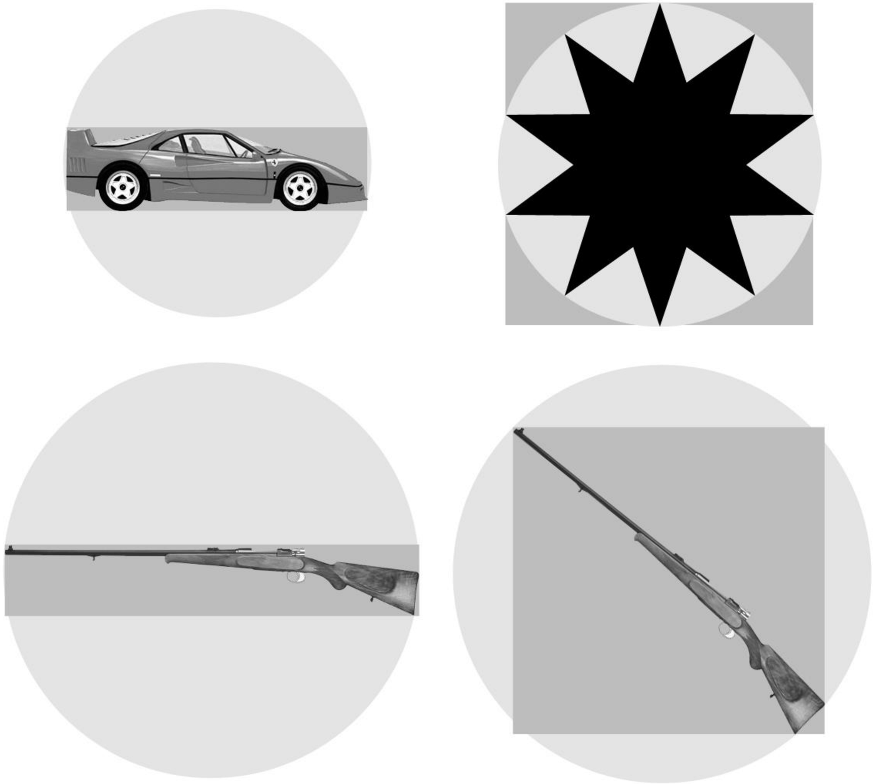


AABBs vs Spheres

- Computing the optimal AABB for a set of points is easy and takes linear time.
- Computing the optimal bounding sphere is a much more difficult problem.
- For many objects that arise in practice, AABBs usually provide a “tighter” bounding volume, and thus better trivial rejection.

Which is Best?

- Of course, for some objects, the bounding sphere is better.
- In the worst case, AABB volume will be just under twice the sphere volume.
- However, when a sphere is bad, it can be *really* bad.

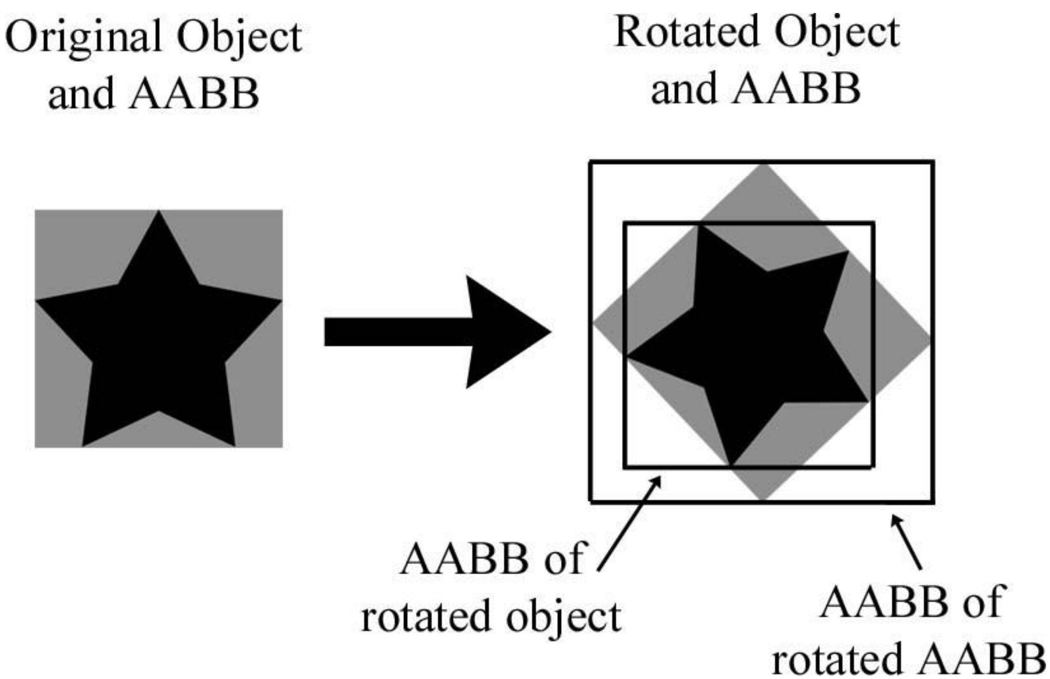


Transforming an AABB

- When you transform an object, its AABB changes.
- Can recompute a new AABB from the transformed object. This is slow.
- Faster to transform the AABB itself.
- But the transformed AABB may not be an AABB.
- So, transform the AABB, and compute a new AABB from the transformed box.
- There are some small but significant optimizations for computing the new AABB.

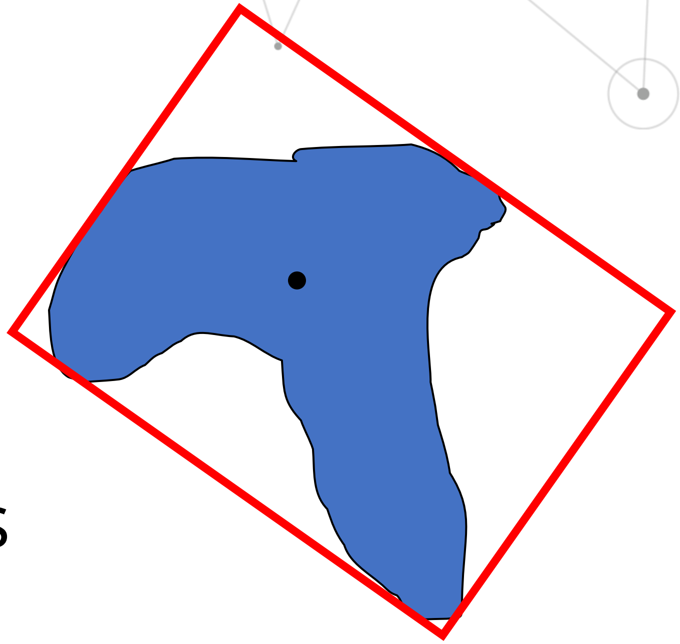
Downside to Transforming the AABB

- Transforming an AABB may give you a larger AABB than recomputing the AABB from the object.



Object Aligned Bounding Box (OBB)

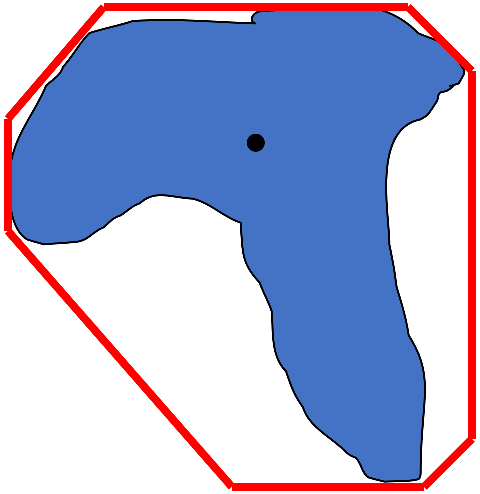
- Store rectangular parallelepiped oriented to best fit the object
- Store:
  - Center
  - Orthonormal set of axes
  - Extent along each axis
- Tight fit, but takes work to get good initial fit
- OABB rotates with object, therefore only rotation of axes is needed for update
- Computation is slower than for AABBs, but not as bad as it might seem



k-DOPS

- k-discrete oriented polytopes

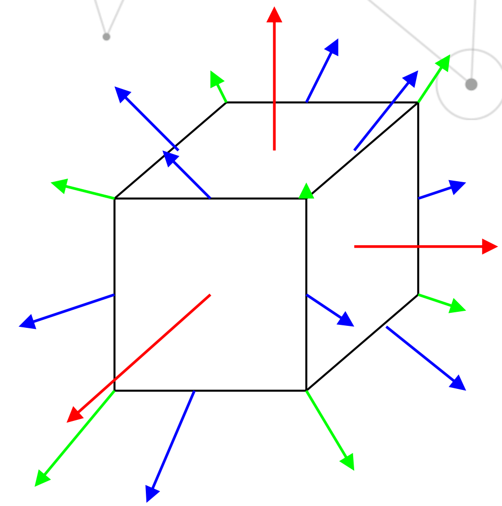
- Same idea as AABBs, but use more axes.
- Store: max and min along fixed set of axes.
  - Need to project points onto other axes.
- Tighter fit than AABB, but also a bit more work.



Choosing axes for k-dops

- Common axes: consider axes coming out from center of a cube.

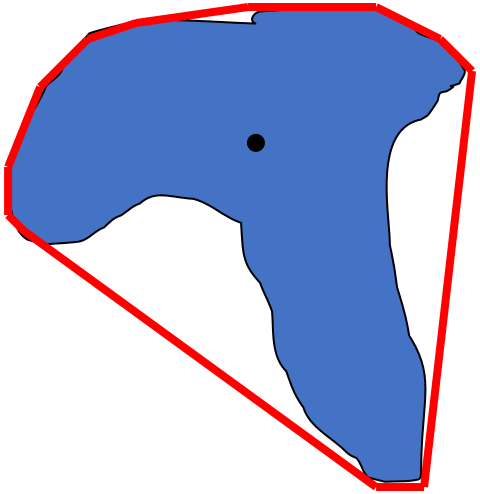
- Through faces: 6-dop
  - same as AABB
- Faces and vertices: 14-dop
- Faces and edge centers: 18-dop
- Faces, vertices, and edge centers; 26-dop



- More than that is not really helpful
  - Empirical results show 14 or 18-dop performs best.

Convex Hull

- Very tight fit (tightest convex bounding volume)
- Slow to compute
- Store: set of polygons forming convex hull
- Can rotate CH along with object.
- Can be efficient for some applications



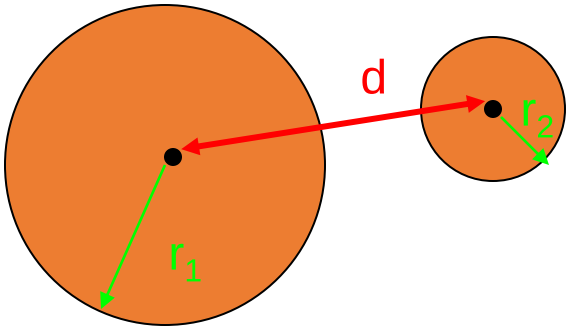
Collision Testing

Testing for Collision

- Will depend on type of objects and bounding volumes.
- Specialized algorithms for each:
  - Sphere/sphere
  - AABB/AABB
  - OABB/OABB
  - Ray/sphere (already introduced)

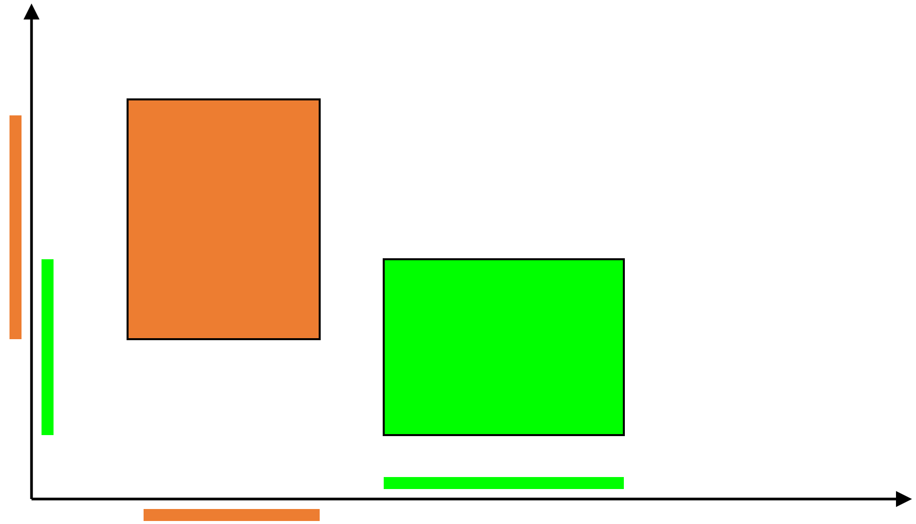
Sphere-Sphere

- Find distance between centers of spheres
- Compare to sum of sphere radii
  - If distance is less, they collide
- For efficiency, check squared distance vs. square of sum of radii



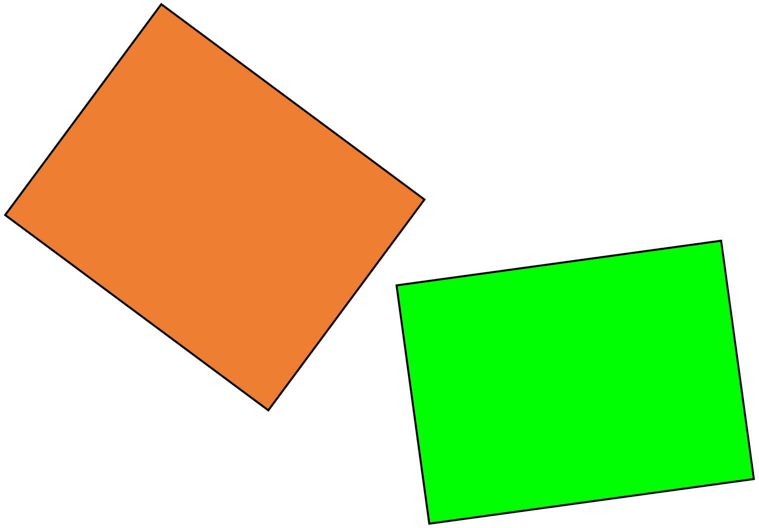
AABB-AABB

- Project AABBs onto axes
  - i.e. look at extents
- If overlapping on *all* axes, the boxes overlap.
- Same idea for k-dops.



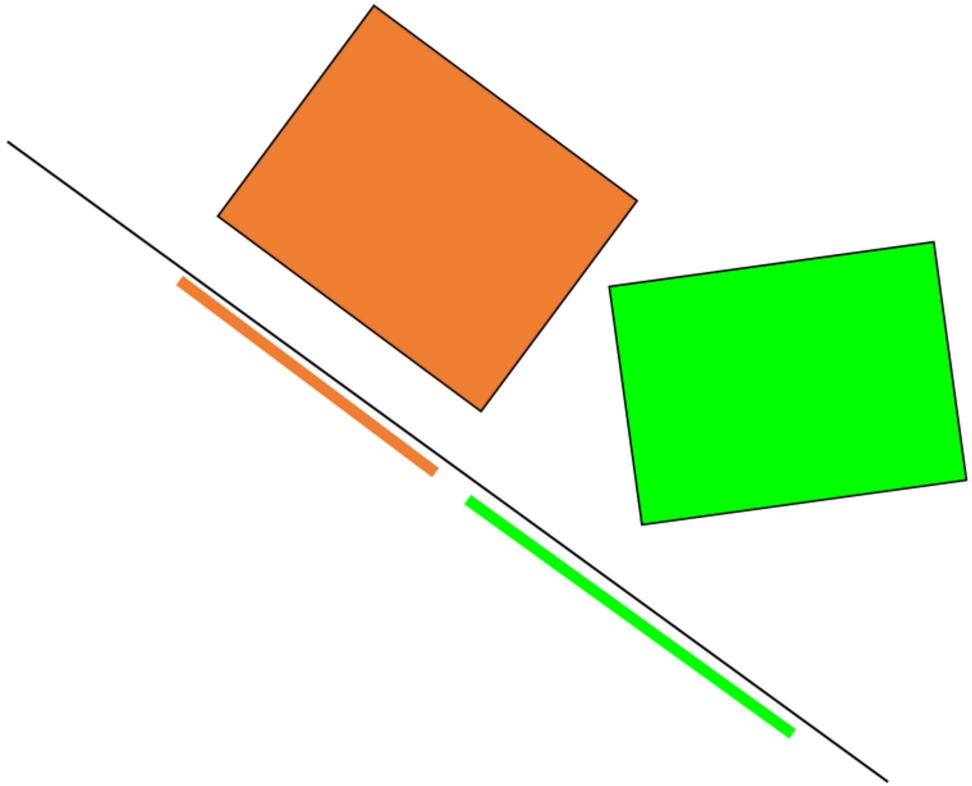
OBB - OBB

- How do we determine if two oriented bounding boxes overlap?



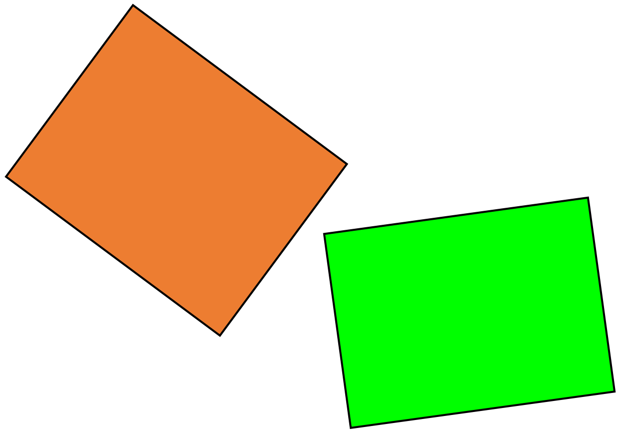
Separating Axis Theorem

- Two convex shapes do not overlap if and only if there exists an axis such that the projections of the two shapes do not overlap



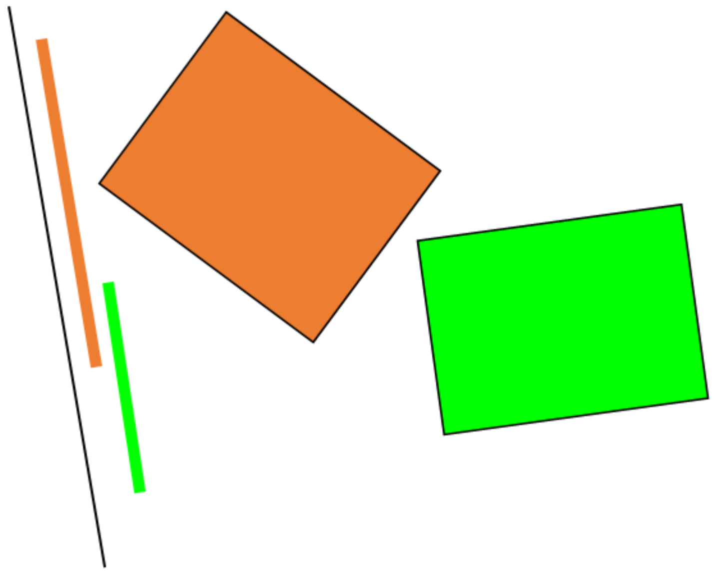
Enumerating Separating Axes

- 2D: check axis aligned with normal of each face
- 3D: check axis aligned with normals of each face and cross product of each pair of edges



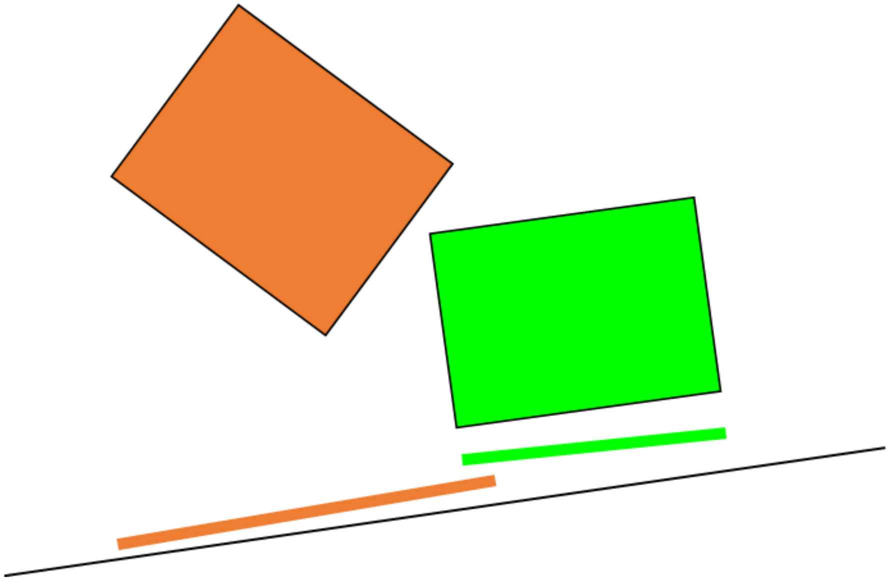
Enumerating Separating Axes

- 2D: check axis aligned with normal of each face
- 3D: check axis aligned with normals of each face and cross product of each pair of edges



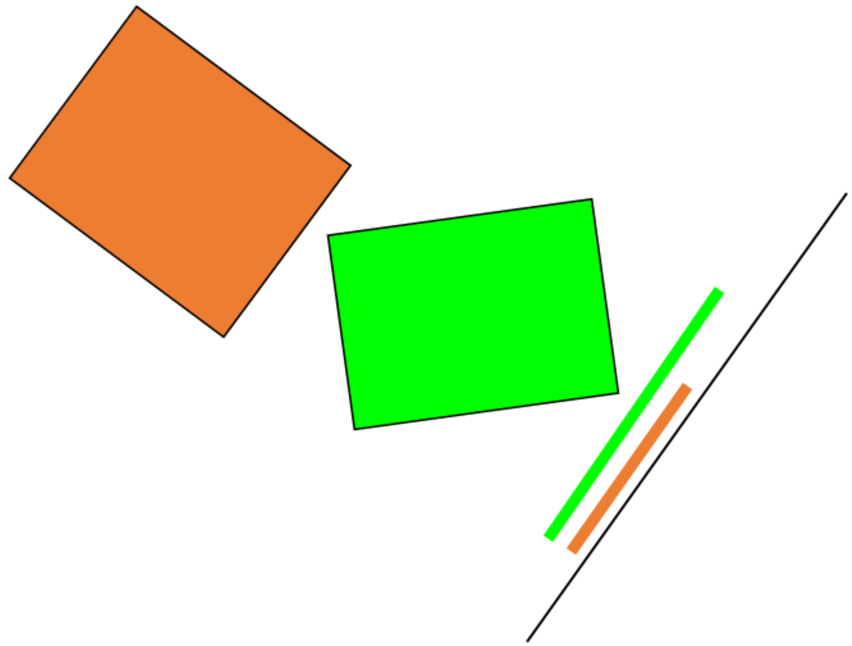
Enumerating Separating Axes

- 2D: check axis aligned with normal of each face
- 3D: check axis aligned with normals of each face and cross product of each pair of edges



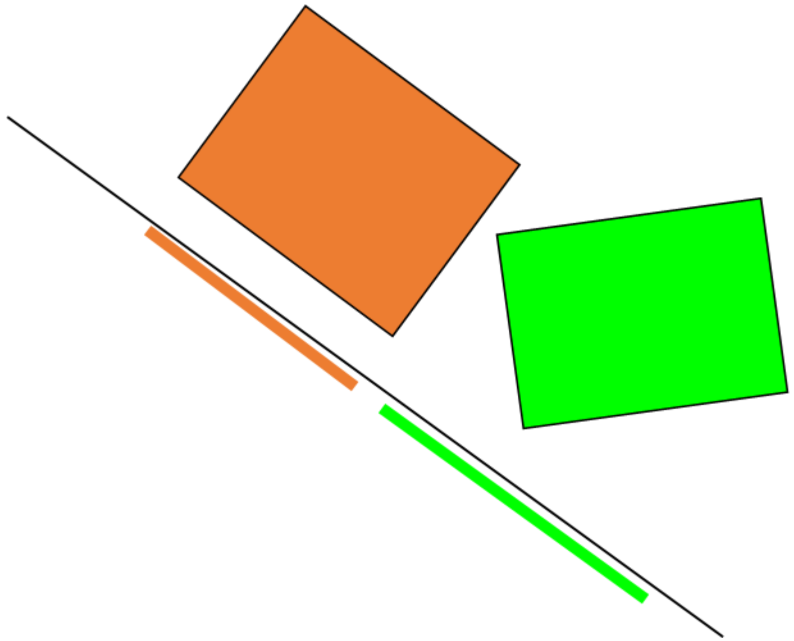
Enumerating Separating Axes

- 2D: check axis aligned with normal of each face
- 3D: check axis aligned with normals of each face and cross product of each pair of edges



Enumerating Separating Axes

- 2D: check axis aligned with normal of each face
- 3D: check axis aligned with normals of each face and cross product of each pair of edges



Bounding Volume Hierarchies

- What happens when the bounding volumes do intersect?

- We must test whether the actual objects underneath intersect.

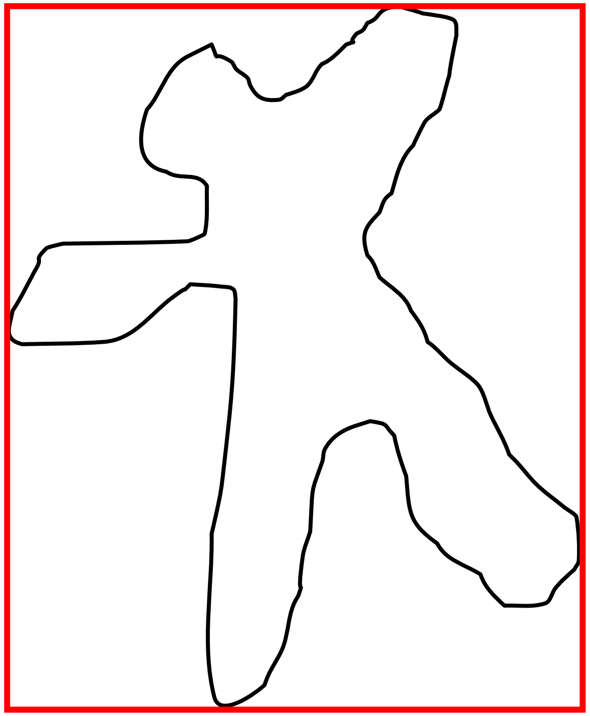
- For an object made from lots of polygons, this is complicated.

- We will use a bounding volume hierarchy

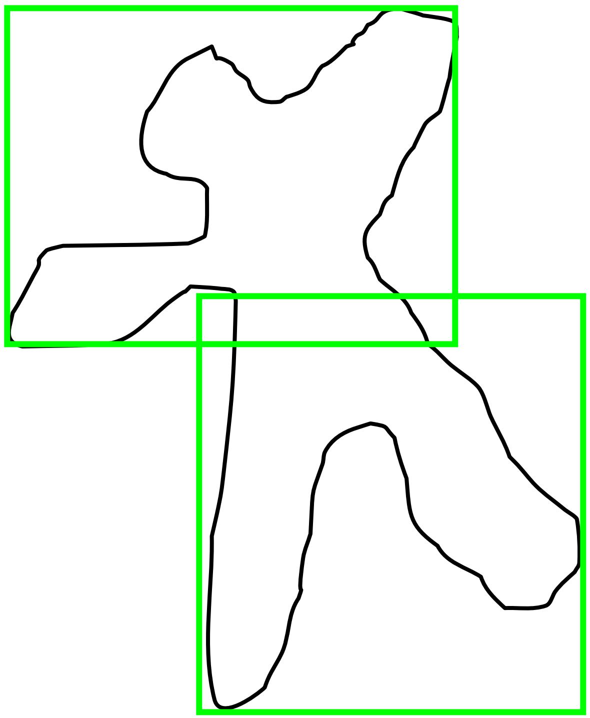
Bounding Volume Hierarchies

- Highest level of hierarchy – single BV around whole object.
- Next level – subdivide the object into two (or maybe more) parts.
  - Each part gets its own BV
- Continue recursively until only one triangle remains.

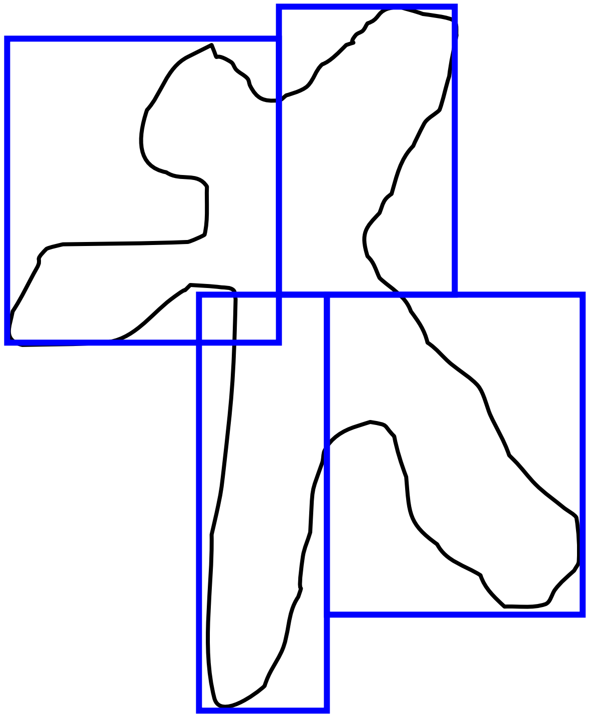
Bounding Volume Hierarchy Example



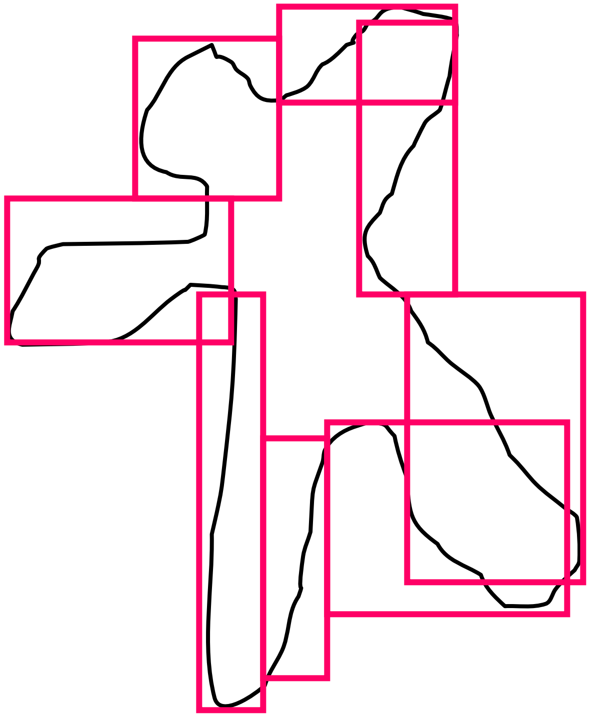
Bounding Volume Hierarchy Example



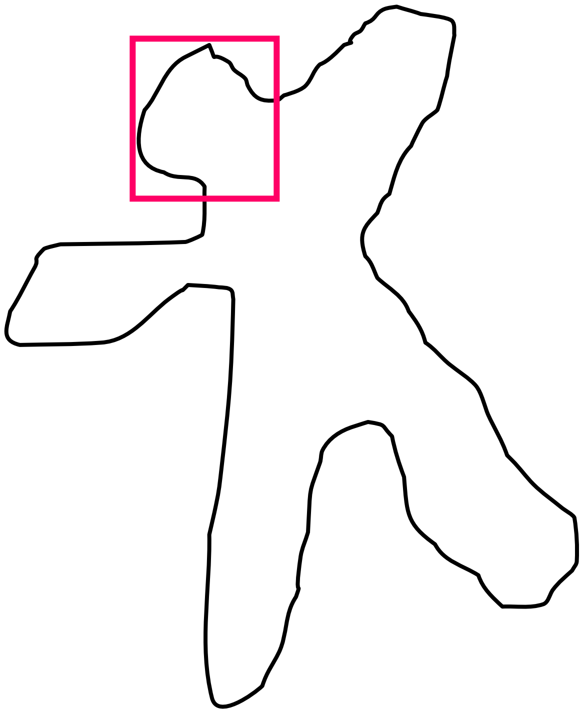
Bounding Volume Hierarchy Example



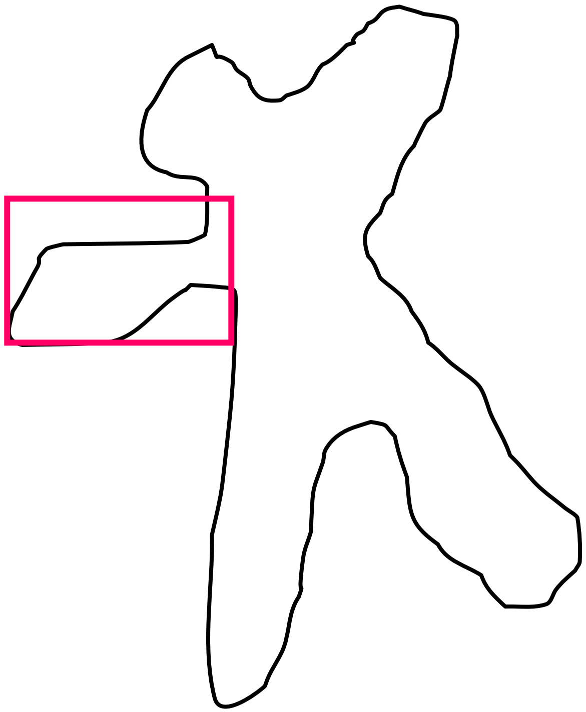
Bounding Volume Hierarchy Example



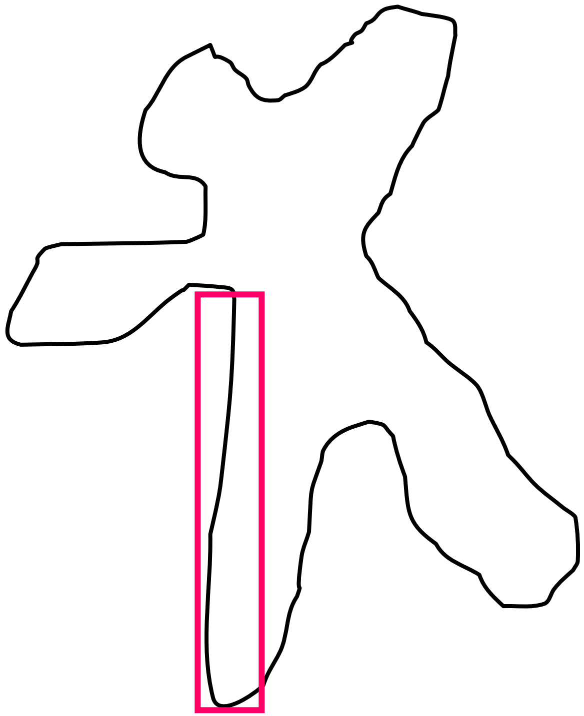
Bounding Volume Hierarchy Example



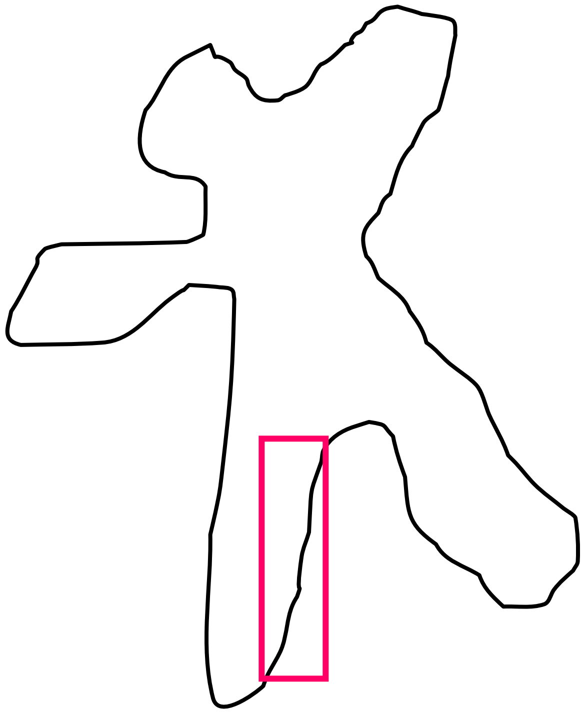
Bounding Volume Hierarchy Example



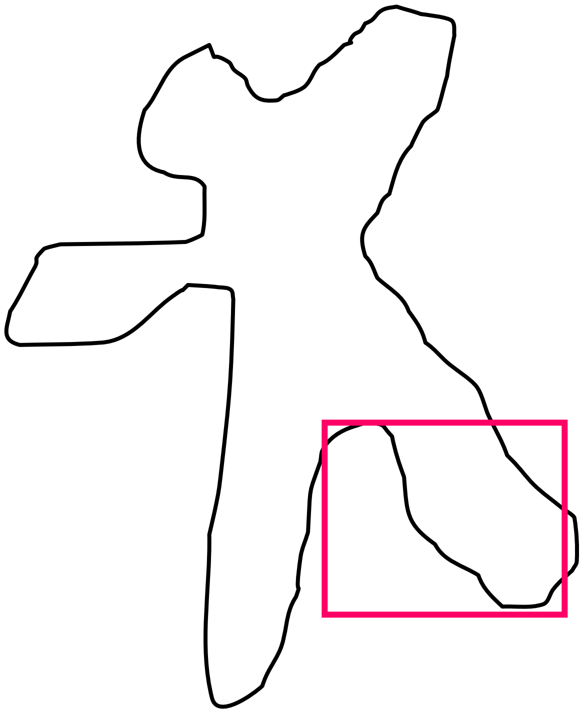
Bounding Volume Hierarchy Example



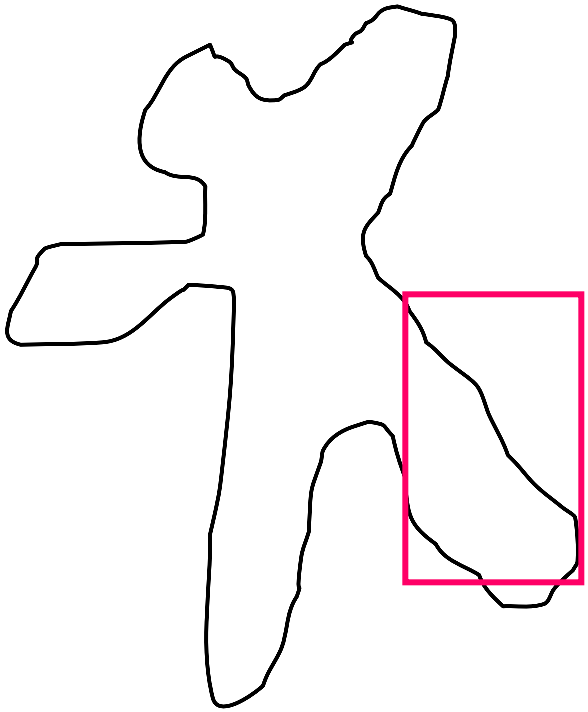
Bounding Volume Hierarchy Example



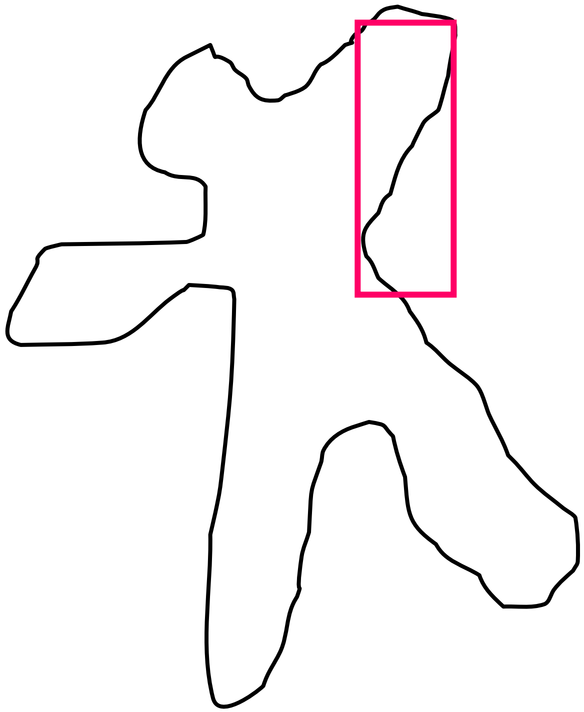
Bounding Volume Hierarchy Example



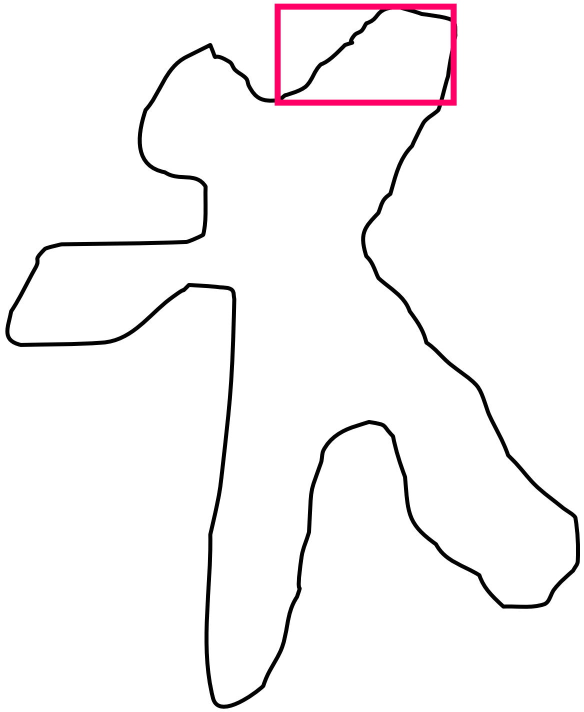
Bounding Volume Hierarchy Example



Bounding Volume Hierarchy Example



Bounding Volume Hierarchy Example



Intersecting Bounding Volume Hierarcies

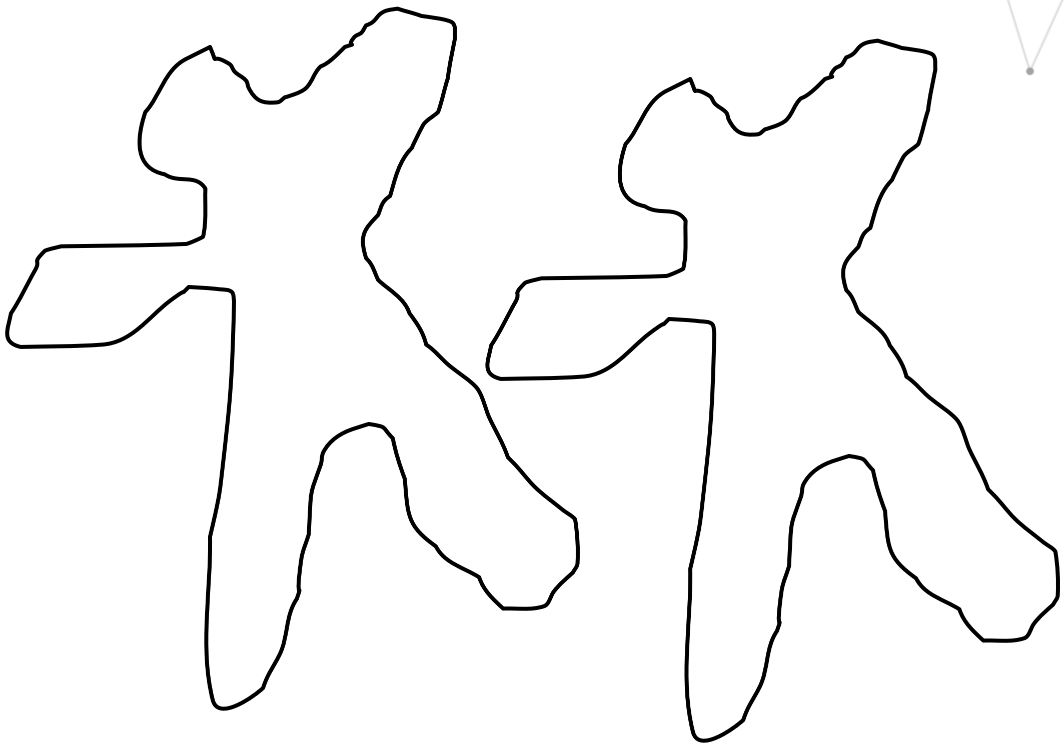
- For object-object collision detection

- Keep a queue of potentially intersecting BVs
  - Initialize with main BV for each object

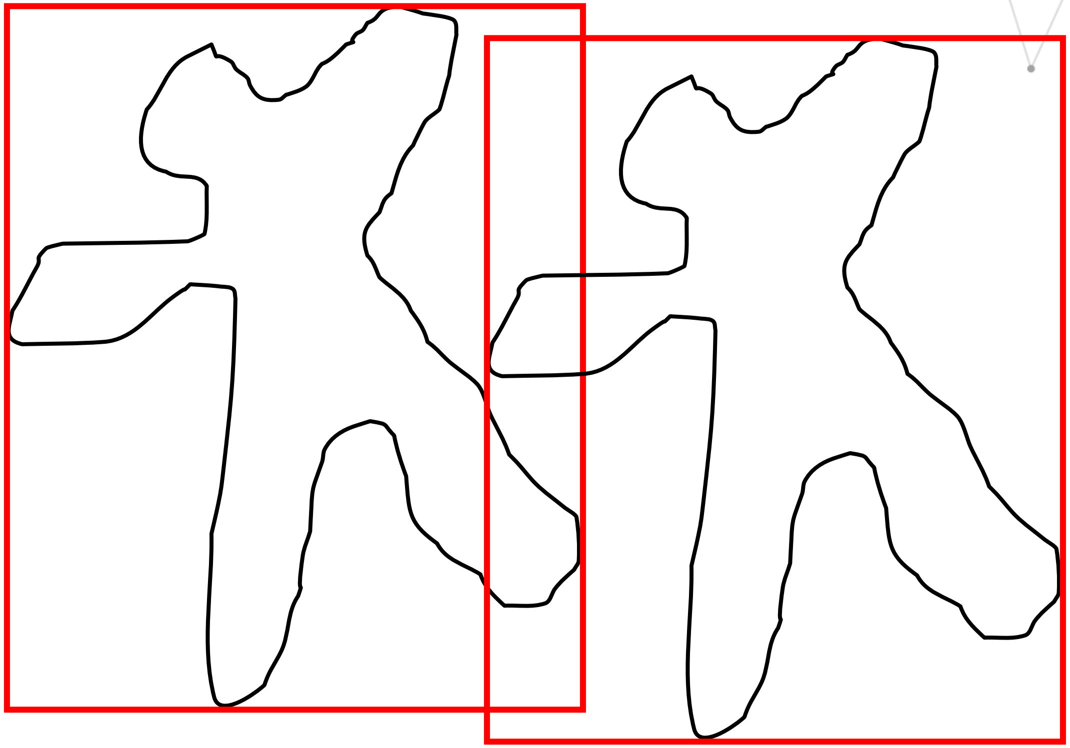
- Repeatedly pull next potential pair off queue and test for intersection.
  - If that pair intersects, put pairs of children into queue.
  - If no child for both BVs, test triangles inside

- Stop when we either run out of pairs (thus no intersection) or we find an intersecting pair of triangles

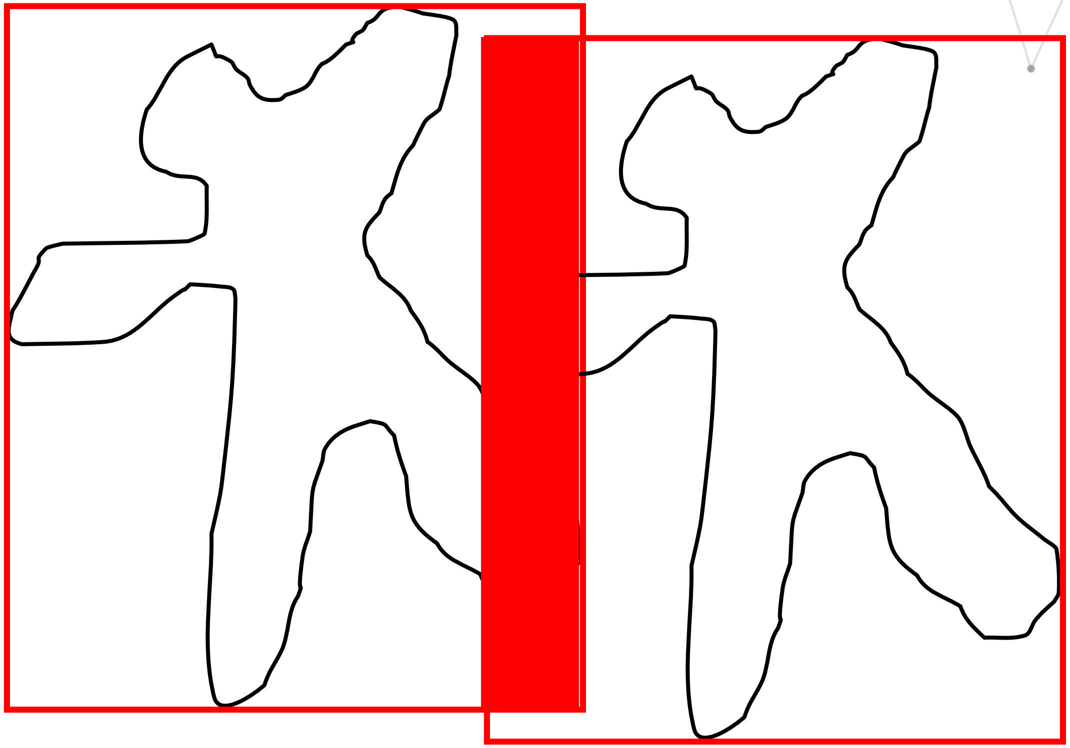
BVH Collision Test example



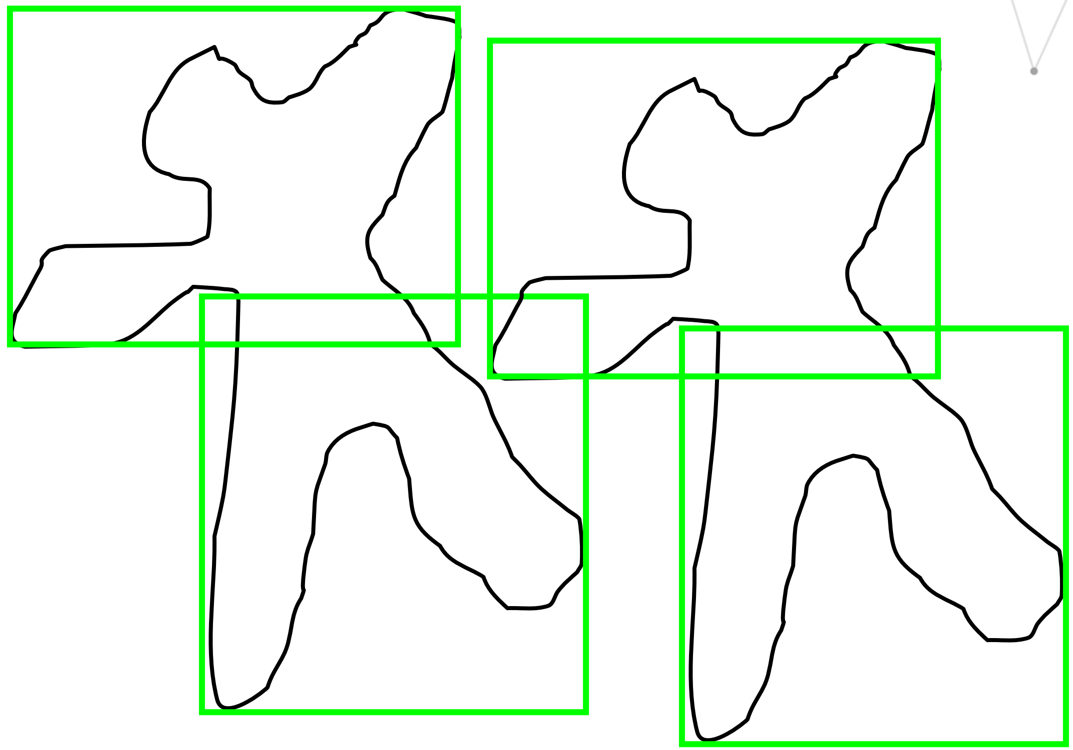
BVH Collision Test example



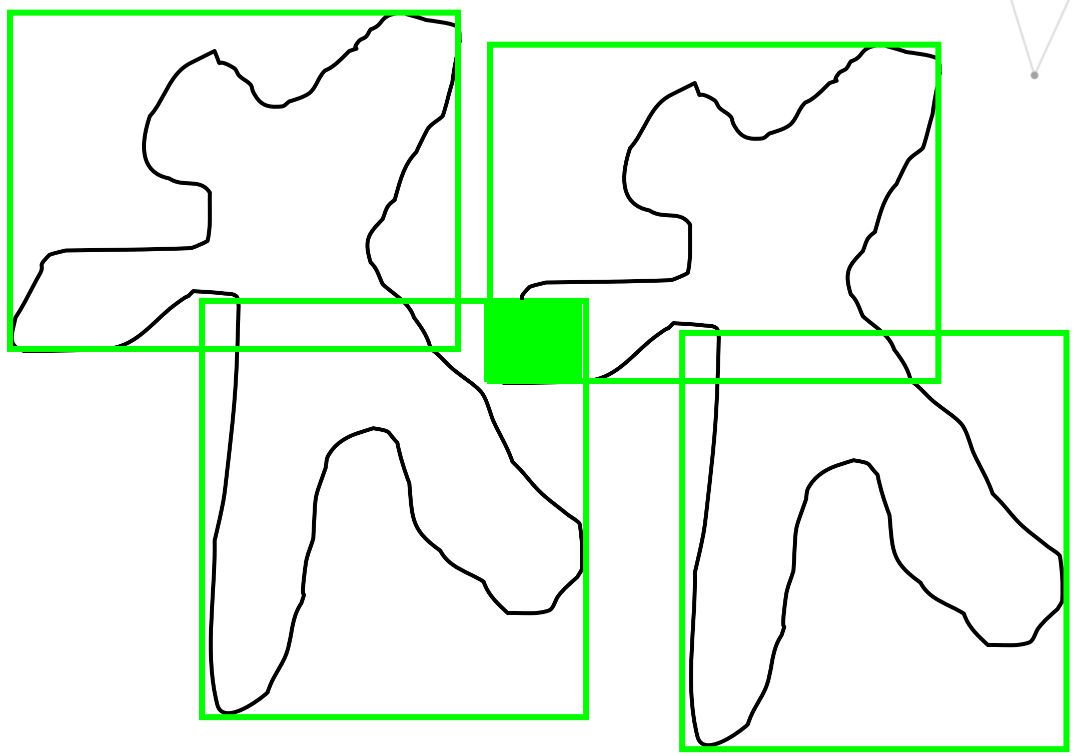
BVH Collision Test example



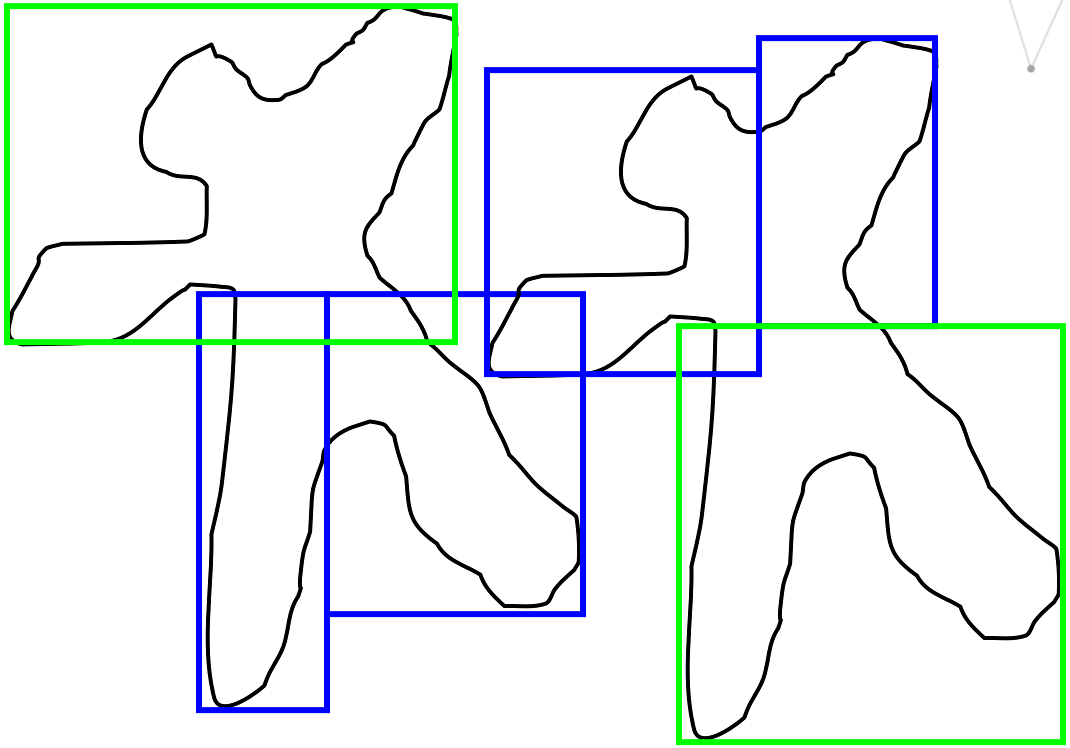
BVH Collision Test example



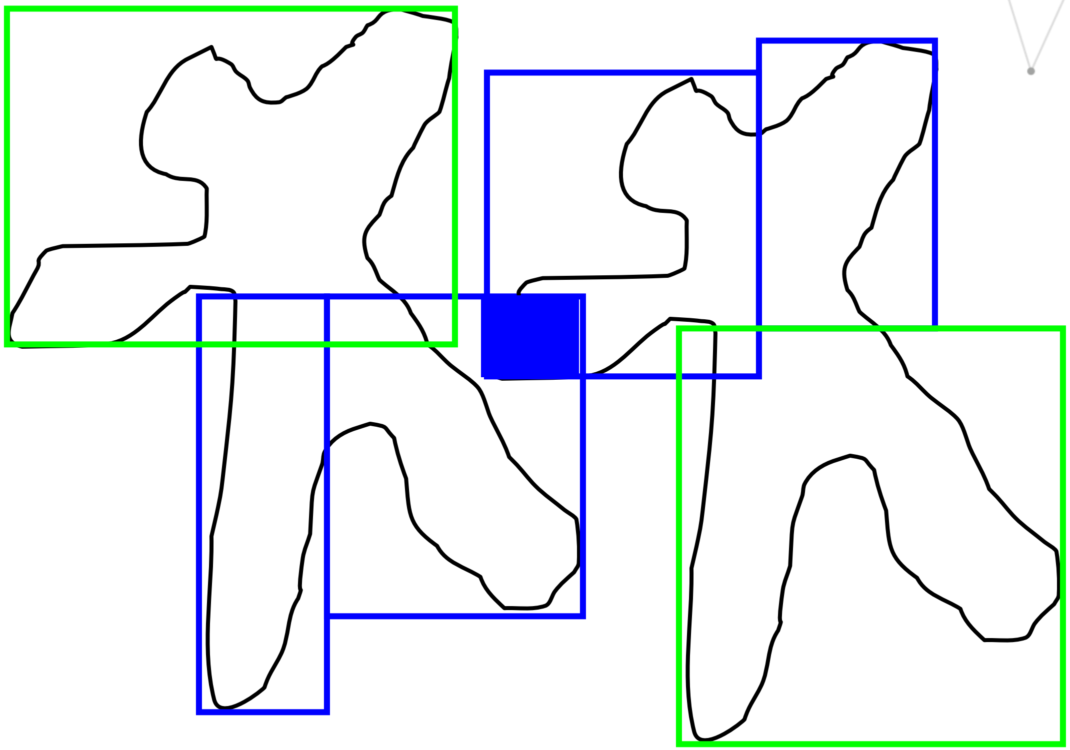
BVH Collision Test example



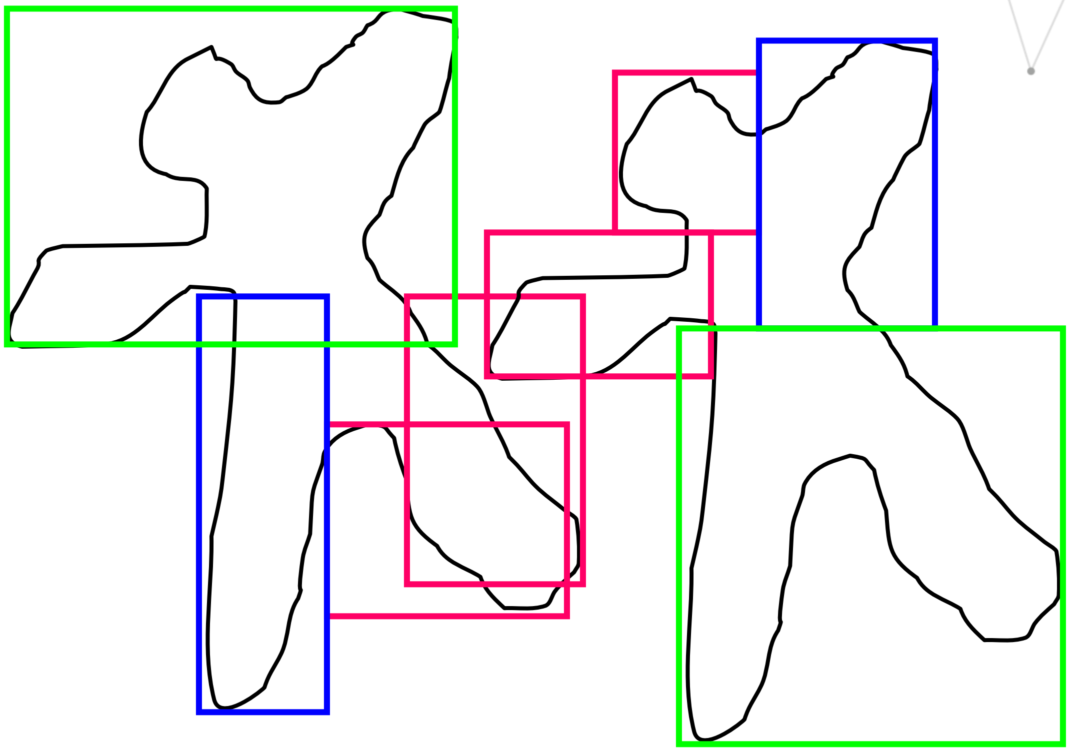
BVH Collision Test example



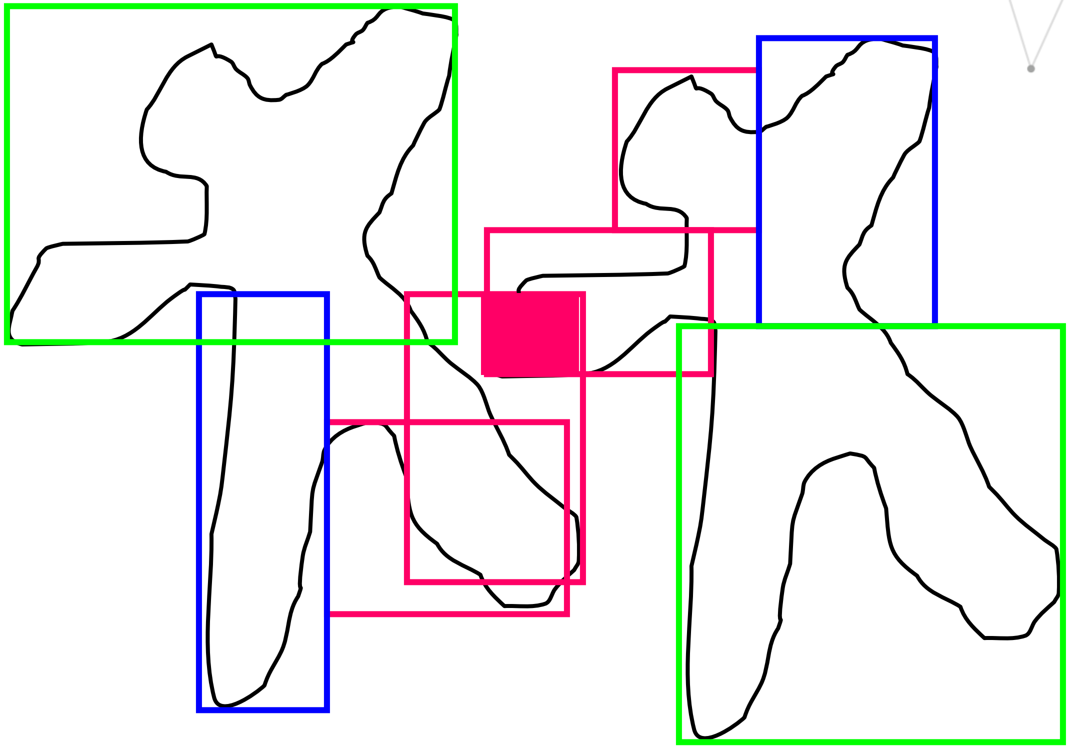
BVH Collision Test example



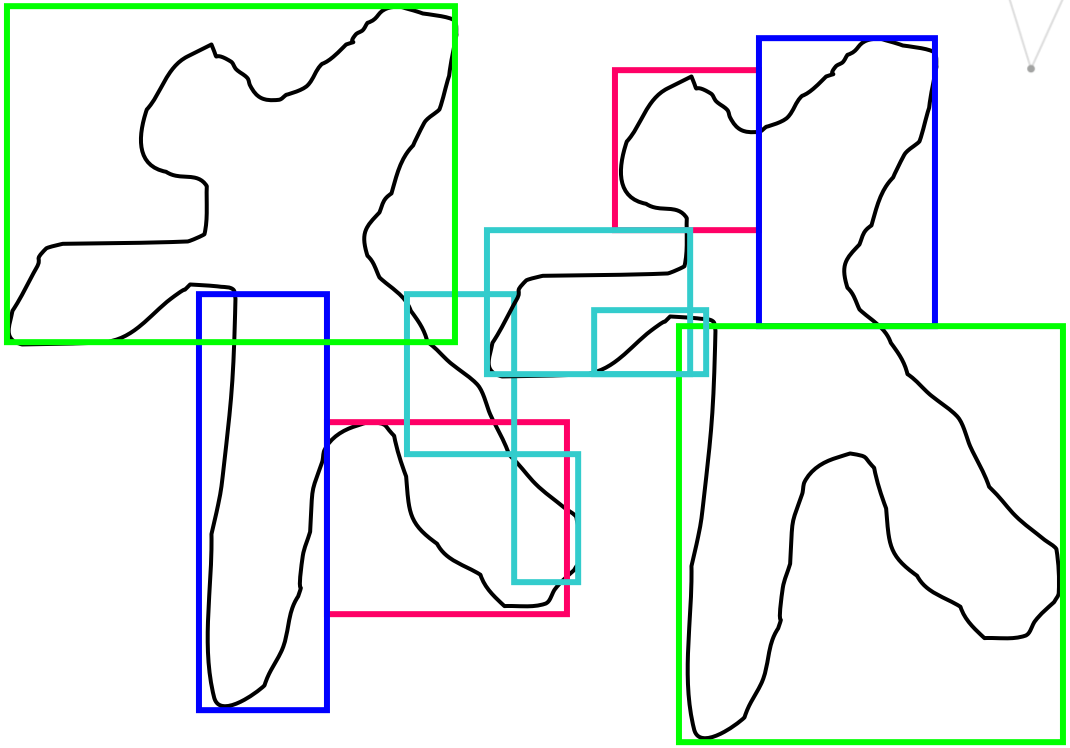
BVH Collision Test example



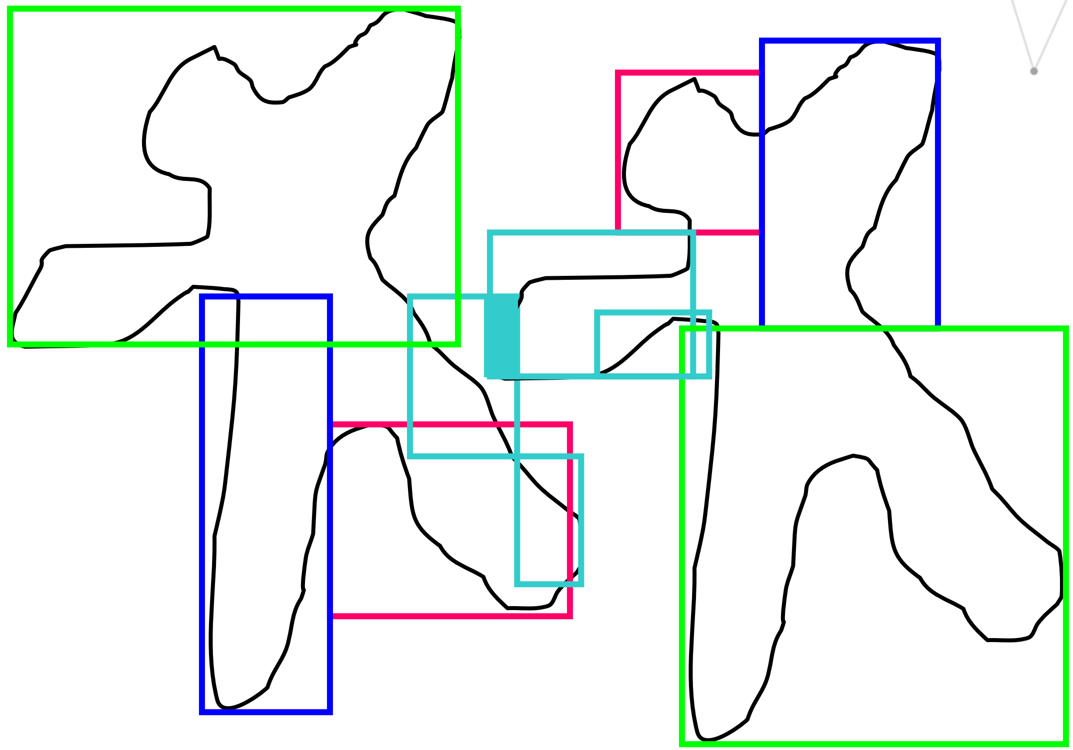
BVH Collision Test example



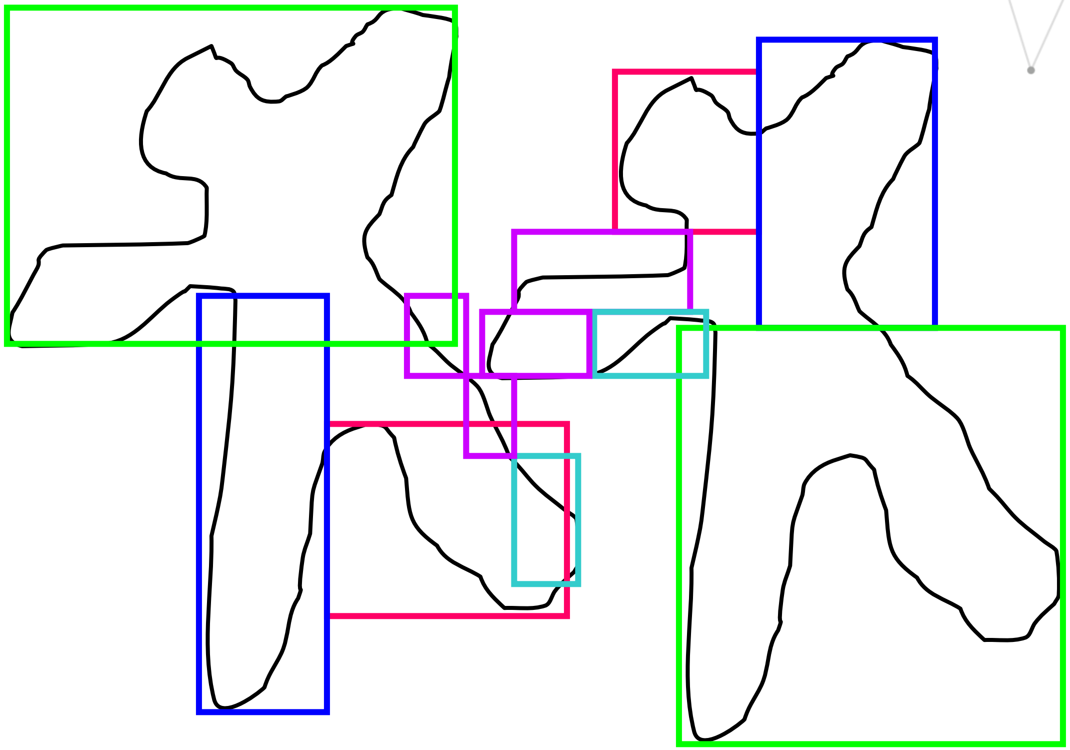
BVH Collision Test example



BVH Collision Test example



BVH Collision Test example



No Collision!

Broad Phase vs. Narrow Phase

- What we have talked about so far is the “narrow phase” of collision detection.
  - Testing whether two particular objects collide

- The “broad phase” assumes we have a number of objects, and we want to find out all pairs that collide.

- Testing every pair is inefficient

Broad Phase Collision Detection

- Form an AABB for each object
- Pick an axis
  - Sort objects along that axis
  - Find overlapping pairs along that axis
  - For overlapping pairs, check along other axes.
- Limits the number of object/object tests
- Overlapping pairs then sent to narrow phase

Final Considerations

World Physics

- Collision Detection in a physically-based simulation

- Must account for object motion.
  - Obeys basic physical laws – integration of differential equations.

- Collision detection: yes/no.
  - Collision **determination**: *where* do they intersect.
  - Collision **response**: how do we adjust the motion of objects in response to collision.

- Collision determination/response are more difficult, but are key for physically based simulation.

Some Other Issues

- Constructing an optimal BV hierarchy
- Convergence of BV hierarchy (i.e. how fast do the BVs approach the actual object).
  - OABBs usually better for this task.
- Optimizing individual tests
- Handling stacking and rest contacts