Game Architecture

Collision Detection

Making things interact.

Today’s Agenda

- What is collision detection?
- Bounds and Shapes
- Separating Axis Theorem
- GJK
- Case Reduction

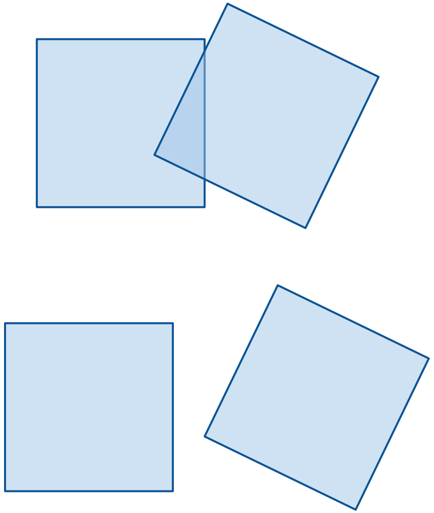
Has something hit something else?

What is collision detection?

- Geometric Intersection
- Pairwise reduction
- Collision resolution

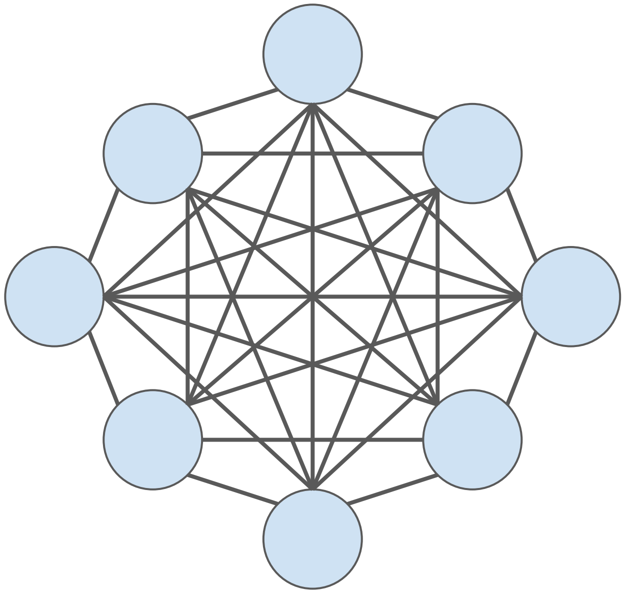
What is collision detection?

- **Geometric Intersection**
- Pairwise reduction
- Collision resolution



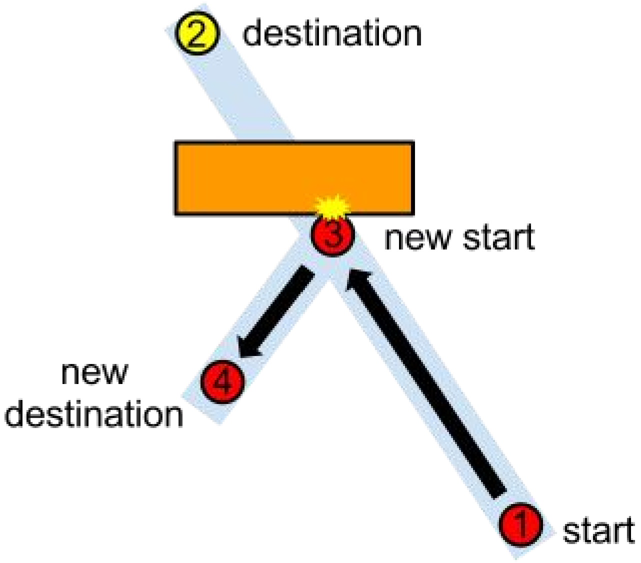
What is collision detection?

- Geometric Intersection
- **Pairwise reduction**
- Collision resolution



What is collision detection?

- Geometric Intersection
- Pairwise reduction
- **Collision resolution**



Today we'll cover...

- **Geometric Intersection**
- **Pairwise reduction**
- Collision resolution

Middleware

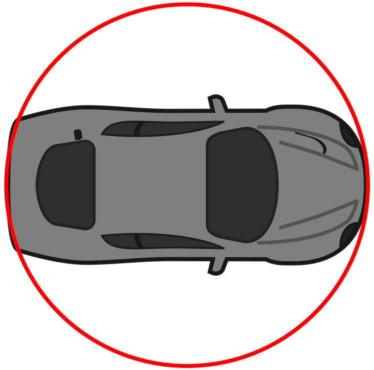
- There exists, of course, middleware that solves this for you.
  - PhysX
  - Havok
  - Bullet
- It’s likely you’ll work with one of these.

Collision detection is really just the implementation of the physical behavior of solids.

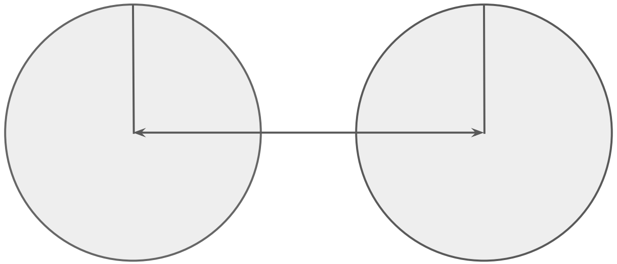
Geometric Intersection

- Intersection testing is costly
  - Triangle-to-triangle test several times costlier than circle-to-circle
  - Thousands of triangles means thousands of times the cost
- Approximate bounds with simpler shapes
- There are trade-offs
  - Accuracy
  - Complexity

Shapes: Sphere

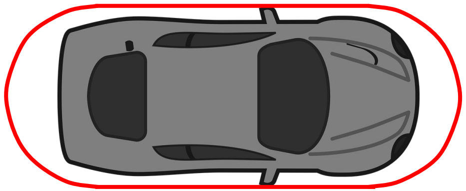


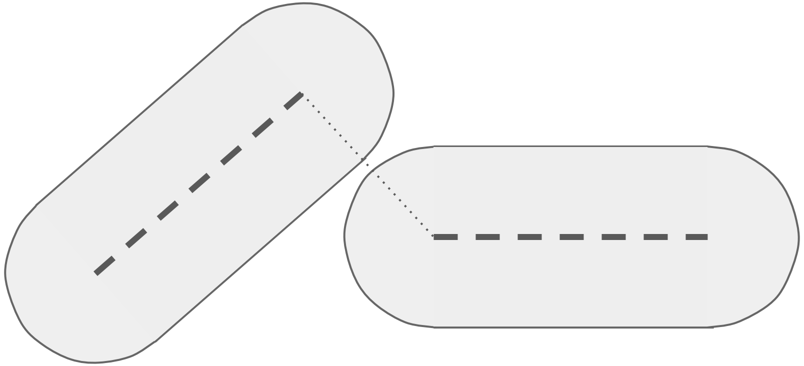
$$|C_1 - C_2| < r_1 + r_2$$



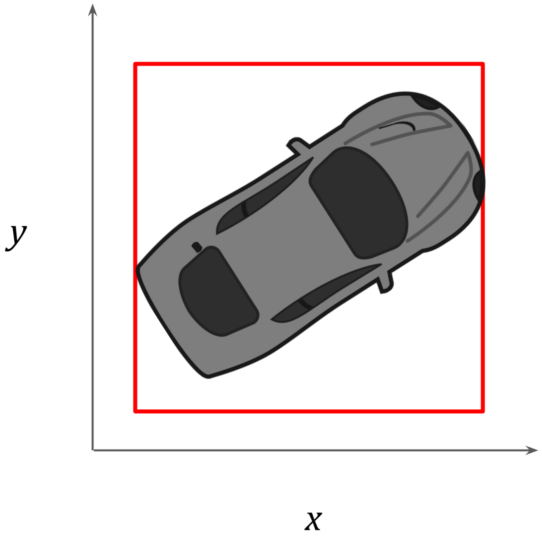
Shapes: Capsule

$$dist(L_1, L_2) < r_1 + r_2$$





Shapes: AABB



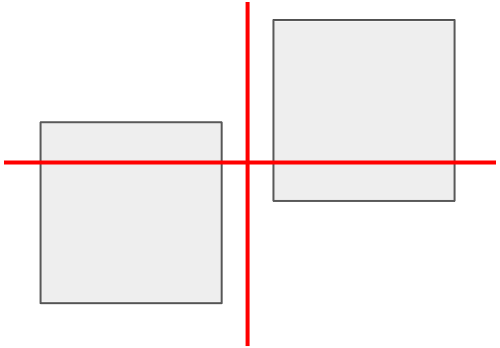
$$C_1.x - C_2.x < w_1 + w_2$$

and

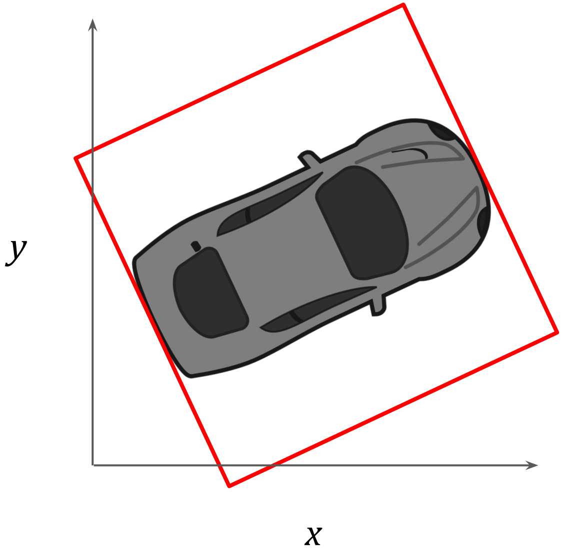
$$C_1.y - C_2.y < h_1 + h_2$$

and

$$C_1.z - C_2.z < l_1 + l_2$$



Shapes: OOBB



???

Separating Axis Theorem

For two convex shapes, if an axis exists on which the projections of both shapes do not overlap, the objects are not intersecting.

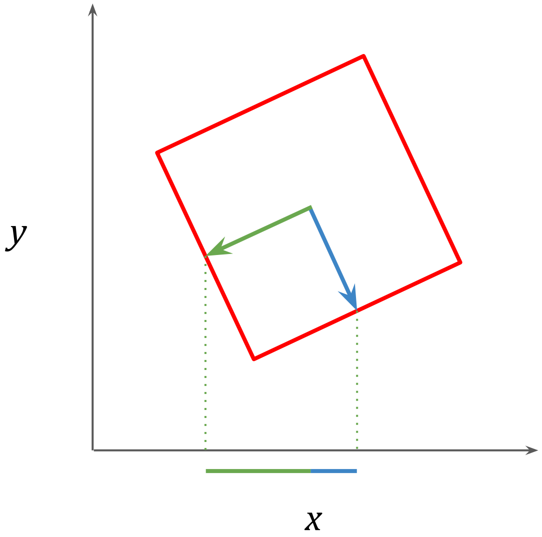
Two objects **do not** collide

if

there exists a **plane** that **separates** them.

Demo!

Projection



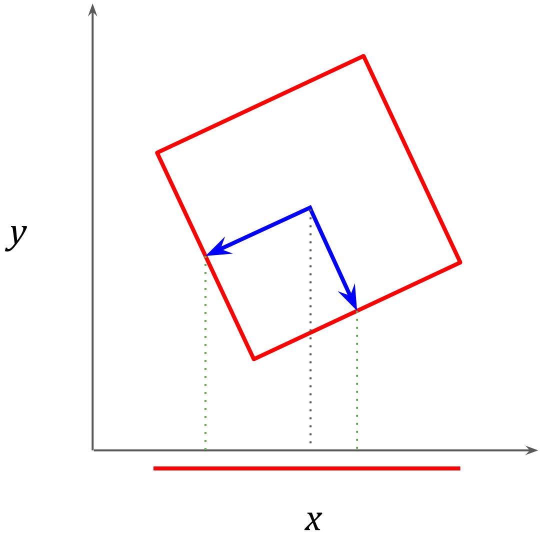
- Sum the projection of the half vectors to get the 'radius.'
- Project a onto b:

$$\left(\frac{a \cdot b}{b \cdot b}\right)b$$

- When b is a unit vector:

$$(a \cdot b)b$$

Projection



- Double the projection and center it on the box's position along the axis.
  - In practice, the 'center point' is the center of the box dotted with the axis.
- Check for overlap of projections along that axis.
- Repeat for each axis.
- Reject on first non-overlap.

GJK

- Named after E.G. Gilbert, D.W. Johnson, and S.S. Keerthi
  - University of Michigan
  - Hence “GJK”
- Collision detection algorithm based on **Minkowski difference**.

- Brace yourselves...

Minkowski Sum

- Wait… weren't we talking about the Minkowski *difference*?
- The **Minkowski sum** is the result of taking every point within one shape, and adding it to every point in another.

$$[(\mathbf{A}_i + \mathbf{B}_j)]$$

- So, the difference is simply the subtraction of **B** from **A**.

$$[(\mathbf{A}_i - \mathbf{B}_j)]$$

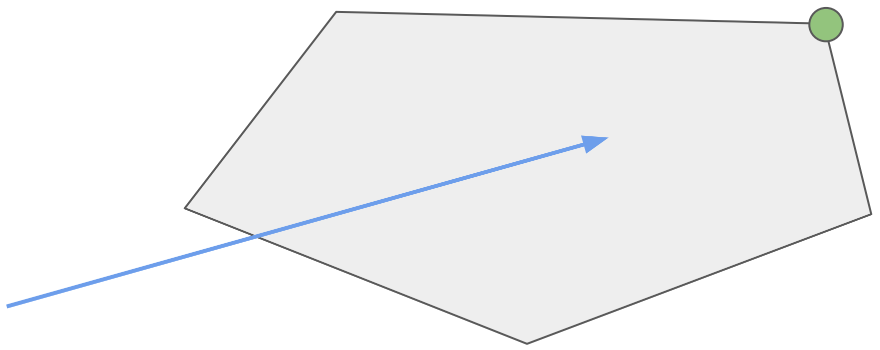
- Demo!

At a High Level

- The Minkowski difference (**M**) of two convex shapes is itself convex, and...
- The Minkowski difference of two **intersecting** convex shapes contains the origin.
  - This is mostly intuitive - if the shapes are intersection, they share some of the same points.
  - Those points subtracted from themselves equals the origin.
- The GJK algorithm seeks to locate the origin within the Minkowski difference of two convex shapes.
  - It does so by constructing a **simplex** (in 3D, a tetrahedron) within the difference which contains the origin.

Supporting Vertex

- At each step, the algorithm searches for a vertex to add to the simplex.
  - This vertex must lie in the direction of the origin, since we want to encompass it.
- A **supporting vertex** is a vertex on the convex hull of **M** which lies closest to the origin in the direction of the search vector.



Support Function

- We can define a **support mapping function** which gives a supporting vertex of shape **A** for search vector **v**:

$$S_{\mathbf{A}}(\mathbf{v})$$

- The support mapping function of a point in the Minkowski difference of shapes **A** and **B** follows:

$$S_{\mathbf{M}}(\mathbf{v}) = S_{\mathbf{A}}(\mathbf{v}) - S_{\mathbf{B}}(-\mathbf{v})$$

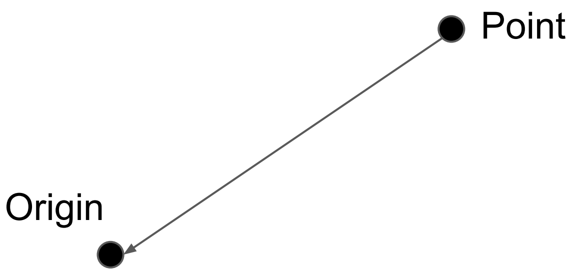
- Wait… why is **-v** used for **B**'s support function?
    - The difference operation essentially negates each point in **B**, so we need to search in the opposite direction.

Simplex Selection

- At each step of the algorithm, we examine our current simplex and determine which **feature** is closest to the origin.
  - A feature may be a point, edge, or face.
- Thus, there are several unique cases to consider:
  - Point
  - Line
  - Triangle
  - Tetrahedron

Simplex Selection

- **Point**
- Line
- Triangle
- Tetrahedron

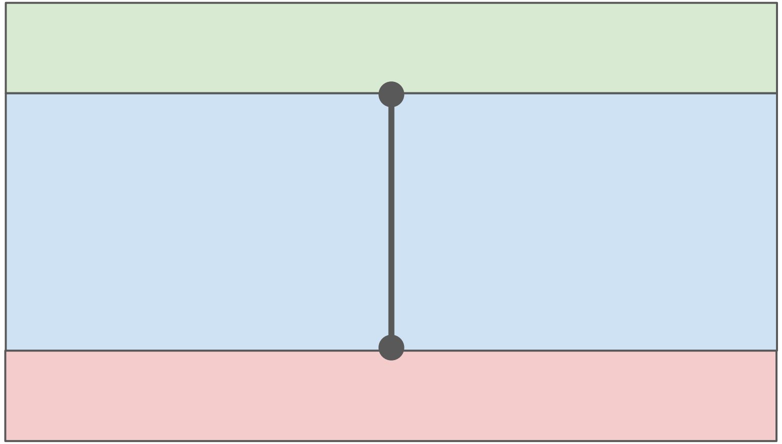


Easiest case.

- The search direction is the direction from the point to the origin.
- Done!

Simplex Selection

- Point
- **Line**
- Triangle
- Tetrahedron



The origin may be closest to:

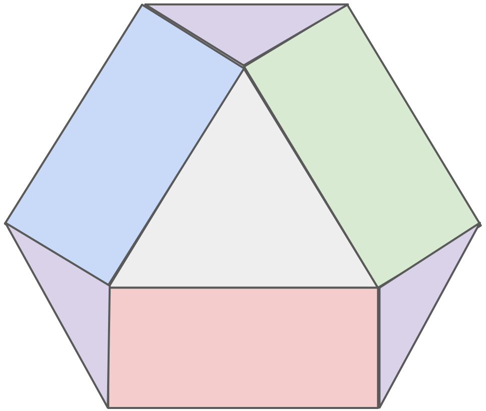
1) An endpoint.
2) The edge itself.

We handle these cases, respectively:

1) Discard the other point, refer to single point case.
2) Search direction perpendicular to line towards origin. Requires a few cross products.

Simplex Selection

- Point
- Line
- **Triangle**
- Tetrahedron



This time, there are three features:

1) Point
2) Edge
3) Face

1 and 2 we handle the same as before. For the face…

- Determine whether the origin is above or below the triangle, and use the corresponding face normal.

Simplex Selection

- Point
- Line
- Triangle
- **Tetrahedron**

Pretend there's a tetrahedron here.

At this point, we either contain the origin or not.

- If we do, we're done.
- If not…
  - Determine which face, edge, or point the origin is closest to.
  - Discard extraneous points to create new simplex.
  - Yea, there are a lot of cases.

Wrapping up GJK

- The last question is “how do we know when there’s no intersection?”
  - Recall that our goal is to encompass the origin.
  - If we want progress, we need our best supporting vertex to get us closer to the origin.
- If at any point, the best supporting vertex is closer to our original simplex than the origin, there is no intersection.

- This is one of those things that could have an entire lecture dedicated to it.
- Resources for more information available at the end of these slides.
- But first… another demo!

Case Reduction

- Many objects in the world
  - N^2 collision tests for N objects in the world.
  - N = 100 means 10,000 collision tests to perform per-frame!
  - Doesn’t scale.
- Reduce cases and reject early.
  - The less tests we have to perform, the better.
  - The earlier we can reject a collision, the better.

Case Reduction

- Phases
- Space partitioning
- Filtering

Case Reduction

- **Phases**
- Space partitioning
- Filtering

Often a collision detection system will operate in multiple phases of different granularities:

- Broad phase
  - Simplistic, large bounding shapes for fast rejection.
- Mid phase
  - Further refined bounding shapes.
- Narrow phase
  - Close convex hull intersection, such as GJK.

This is a technique used by Havok.

Case Reduction

- Phases
- **Space partitioning**
- Filtering

In lieu of a broad phase, a system may instead opt for a space partitioning system to bucket objects into different subspaces.

- Divide world space into partitions.
  - Octrees or BSP
- Bucket objects into partitions.
- Perform tests within partitions.

Case Reduction

- Phases
- Space partitioning
- **Filtering**

Some objects are actually allowed to intersect each other! Or, we know objects only need to intersect with particular things.

- Assign collidable types
  - Player
  - Static
  - Projectile
- Only allow collisions for specific types
- Example:
  - player = 0x4
  - filter = 0xFB
  - Reject if (filter & player == 0) (it is)

Worth Mentioning...

- Discrete tests can fail for fast-moving objects.
  - An object will pass entirely through another object over just one timestep.
- This can be solved with **continuous collision detection**.
- An object's path is swept through space and the resulting convex hull is used for collision tests.
  - You can see this in Ericson's GJK presentation.
- Alternatively, use more math to calculate **time of impact**.
  - Beyond the scope of this lecture.

On to Resolution… Next Time

- Need more information about collision.
    - Contact point.
    - Contact velocity.
    - Contact normal.
    - What object is “at fault?”
- This information helps us calculate the resolution step.
    - What direction will the objects move in?
    - How fast will they now move?
    - What angular impulse is applied to them?

Want to learn more?

- Tutorial on 2D separating axis theorem
  - http://www.metanetsoftware.com/technique/tutorialA.html
- A comprehensive reference table for geometric intersections
  - http://www.realtimerendering.com/intersections.html
- An excellent explanation of GJK
  - http://vec3.ca/gjk/
  - Pay less attention to the implementation...
- Christer Ericson's GJK lecture
  - http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt

Homework 5

- Let’s take a look at the homework 5 depot.

End of Lecture

- Homework 5 is due next Monday, 3/6.
- Next time we’ll cover dynamics and collision resolution.

Appendix

- The appendix is meant to cover other common geometric intersection and utility tests and their derivations.

Point-Line Distance

- Given a point, P, and two points on the line, L_0 and L_1:

$$\frac{|(\mathbf{p}-\mathbf{l}_0)\times(\mathbf{p}-\mathbf{l}_1)|}{|\mathbf{l}_1-\mathbf{l}_0|}$$

- This works because the cross product's magnitude depends on the angle between the two vectors.
    - A point on the line will yield a cross product of zero magnitude for the numerator.

Point-Plane Distance

- Given a point **p** and a plane defined by point $\mathbf{p}_0$ and normal **n**:

$$|((\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n})\mathbf{n}|$$

- Take a vector from the point on the plane to the point in space, and project it onto the plane's normal. The magnitude of the project vector is the distance.
- This can be simplified to:

$$(\mathbf{n} \cdot \mathbf{p}) - (\mathbf{n} \cdot \mathbf{p}_0)$$

Line-Line Distance

- Given points $\mathbf{p}_0$ and $\mathbf{p}_1$ on respective lines with unit direction vectors $\mathbf{u}_0$ and $\mathbf{u}_1$:

$$\frac{|(\mathbf{p}_1 - \mathbf{p}_0) \cdot (\mathbf{u}_1 \times \mathbf{u}_0)|}{|\mathbf{u}_1 \times \mathbf{u}_0|}$$

- If the distance is 0, the lines are intersecting.

Ray-Plane Intersection

- Given a ray defined by point $\mathbf{p}_r$ and direction $\mathbf{r}$, and a plane defined by point $\mathbf{p}_0$ and normal $\mathbf{n}$, we can find the point of intersection:

$$\mathbf{p}_r + [\frac{(\mathbf{n} \cdot \mathbf{p}_r) - (\mathbf{n} \cdot \mathbf{p}_0)}{-\mathbf{n} \cdot \mathbf{r}}]\mathbf{r}$$

- Basically use point-to-plane distance and dot product's cosine equation to derive the equation.
- If the scalar we scale $\mathbf{r}$ by is negative, the ray does not intersect the plane.

Ray-Triangle Intersection

- Use ray-plane intersection to find the intersection point with the plane the triangle lies on.
- Then, calculate the barycentric coordinates of the point in the triangle.
- If the point lies in the triangle, there is an intersection between the ray and the triangle.