Crashing 101

How to detect and respond to collisions

Will you have a list of who's presenting when?

Yes, I'll make that list once I have an idea of who's in what groups.

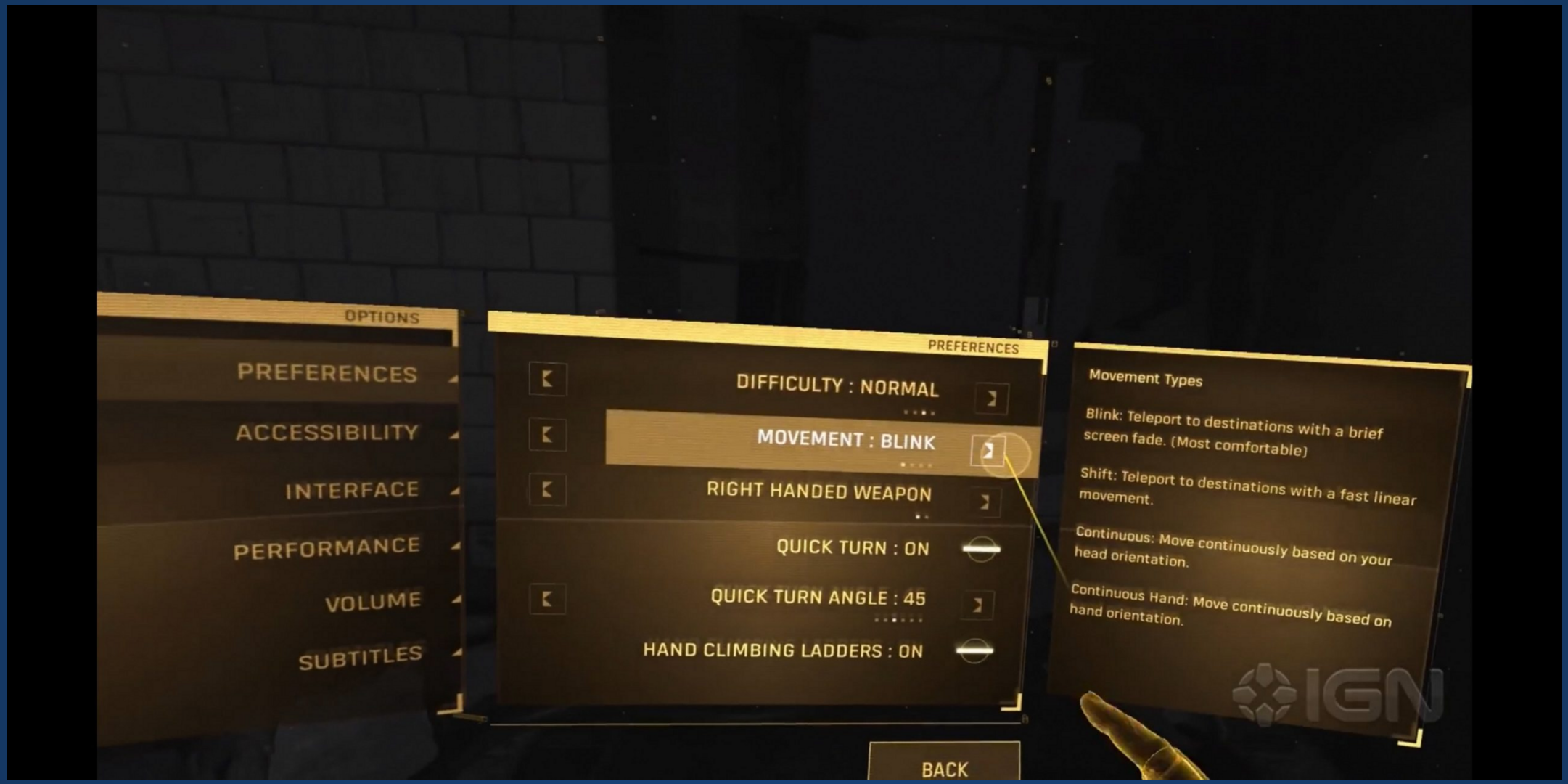
Do we need to find our groups by the project proposal?

Yes.

How do UIs work in 3D?

In most 3D games/settings, UIs are still 2D things drawn on top of the 3D world.

How do UIs work in 3D?



In VR (3D worlds), UIs are very much still experimental, and we don't know what form they'll take yet.

In Processing

Can either use a UI toolkit or use 2D drawing calls in a 3D environment.

```
1  RectangularButton b;
2  CircularButton c;
3
4  pushMatrix();
5  translate(200, 0, 0);
6  rotateY(angle / 2);
7  shape(box);
8  popMatrix();
9
10 popMatrix();
11 hint(DISABLE_DEPTH_TEST);
12
13 b.draw();
14 c.draw();
15
16 hint(ENABLE_DEPTH_TEST);
```

https://github.com/KrabCode/LazyGui

Are there jobs solely devoted to debugging?

Not really, no. Although depending on a developer's role within a team, they may be assigned to fix bugs more often than writing new functionality (or vice versa).

What else can a debugger do?

Debuggers are primitive scripting interfaces that interact with your code at runtime. You can do all sorts of stuff with them.

“ Stop on line 27, but only when `var` is 15 and `timer` is odd. Then print out the largest element of `list7` and tell me if it's larger or smaller than `florp`.

But these are advanced tricks.

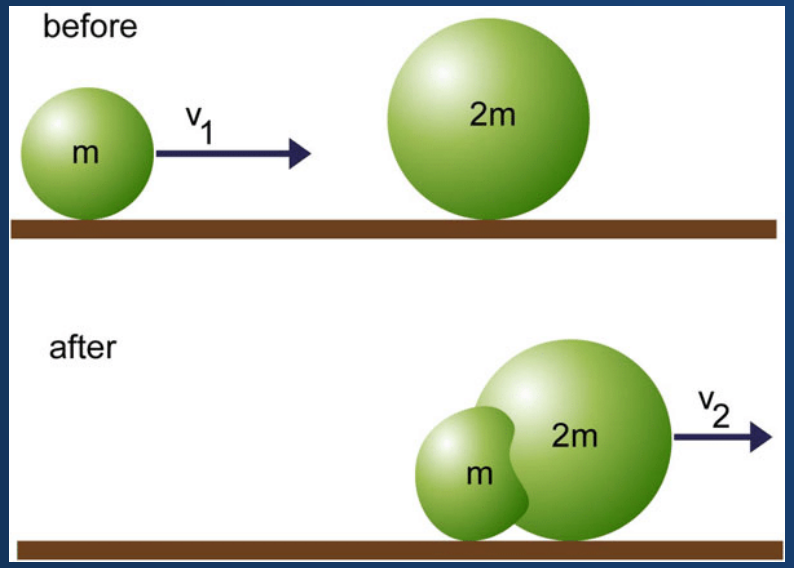
Button Reviews

Two General Styles:

- Button class that can either encapsulate a square or circular button.
- A base button class, and then either a circular or rectangular button that inherits from it.

What functionality is common and what is different between the two button types?

Simple Collisions

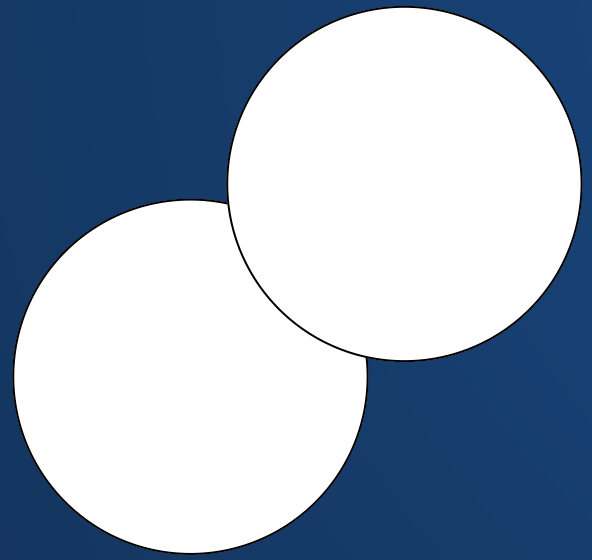


Disclaimer: Unless explicitly stated otherwise, we're assuming that the computer has perfectly-accurate math.

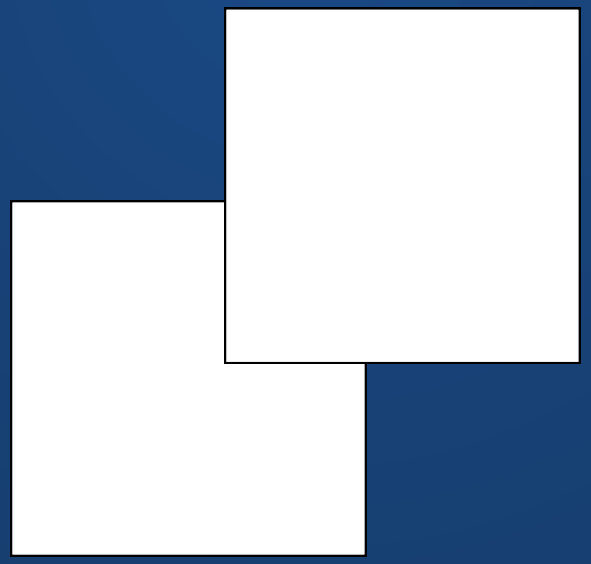
Floating-point accuracy introduces some nasty wrinkles into implementations.

There are three basic shapes we need to know how to handle.

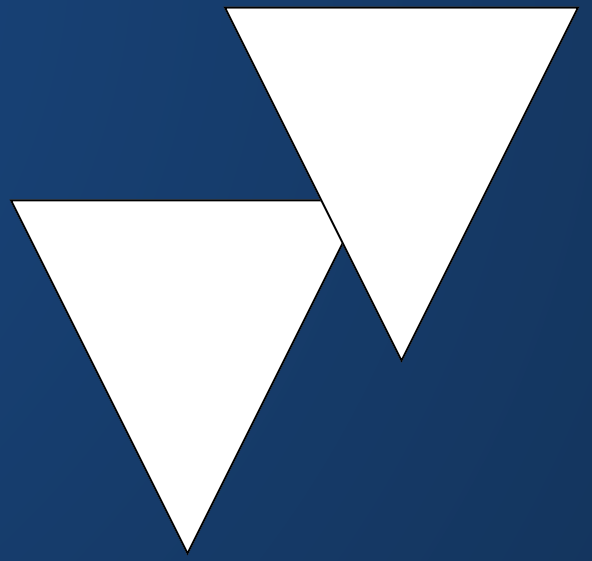
Sphere-Sphere



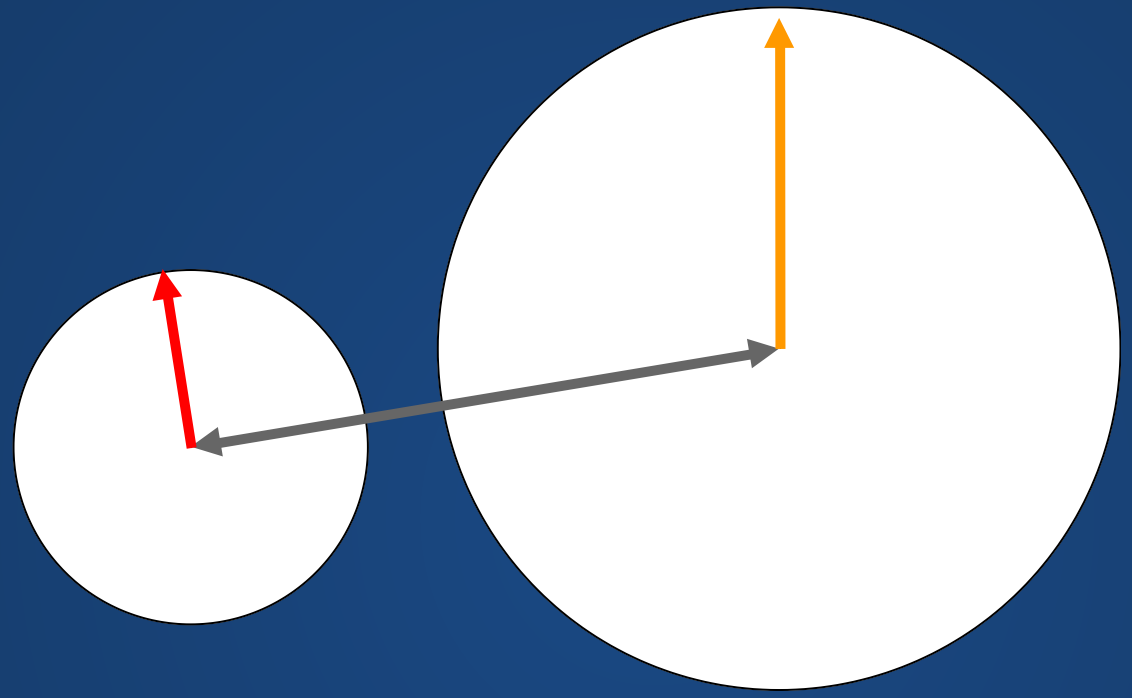
Box-Box



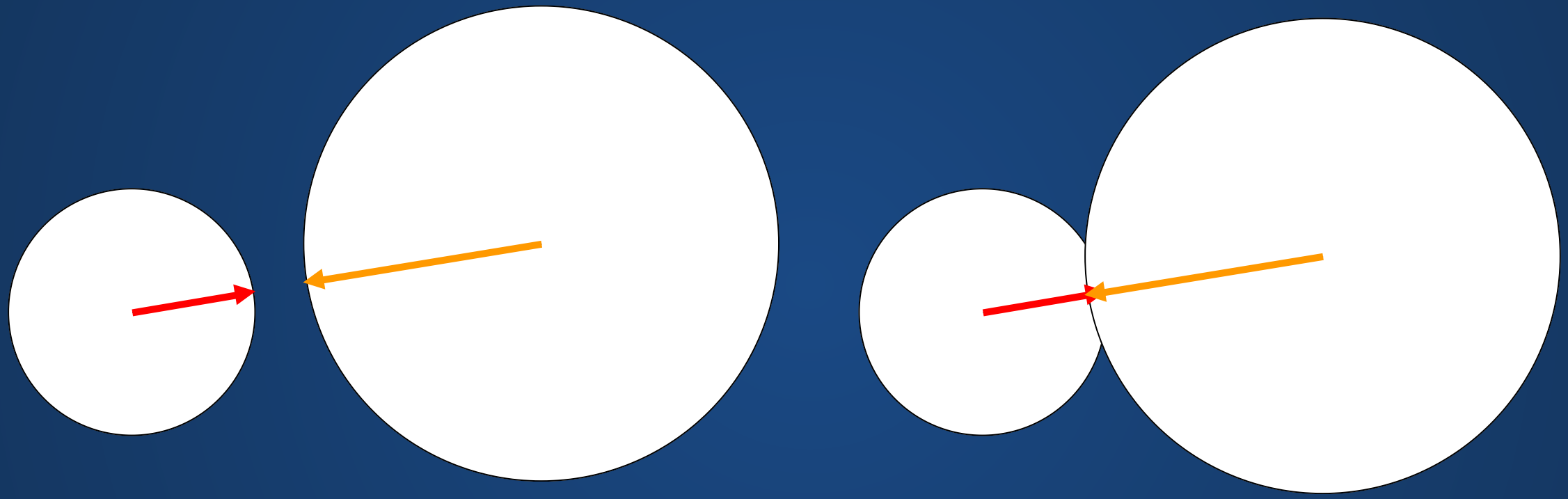
Triangle-Triangle



How do we tell if two spheres are colliding?

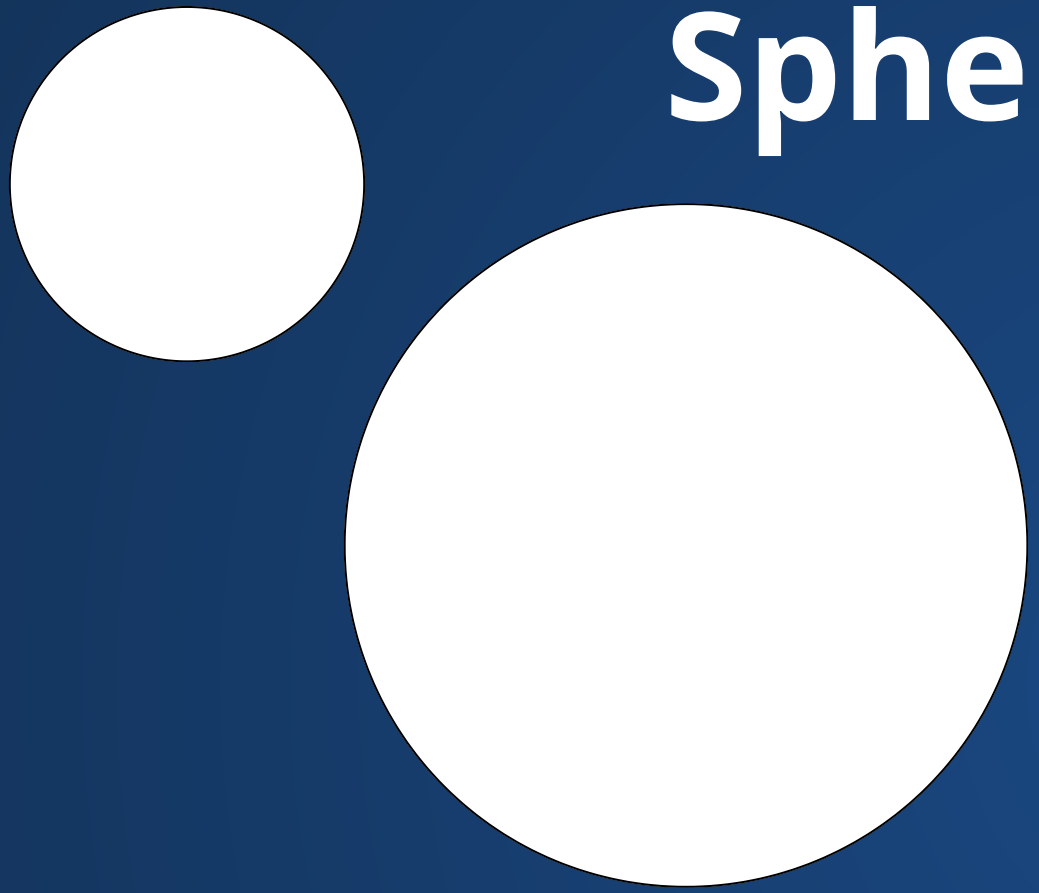


How do we tell if two spheres are colliding?



The shortest path between the two centers is a line. If the length of this line is less than the sum of the radii, they're colliding.

Sphere Collision



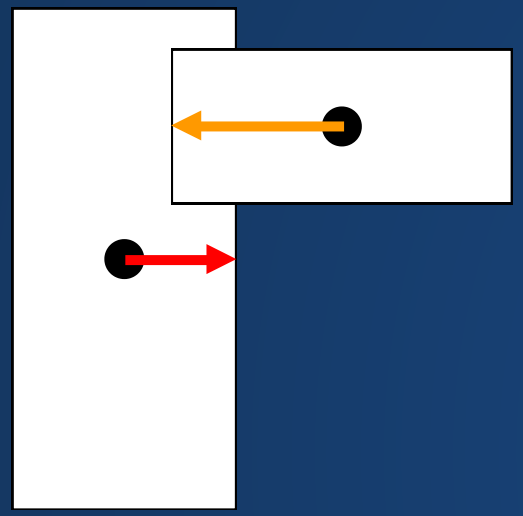
Sphere 1 located at (x_1, y_1, z_1) with a radius of r_1.

Sphere 2 located at (x_2, y_2, z_2) with a radius of r_2.

Spheres are colliding if

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} - (r_1 + r_2) < 0$$

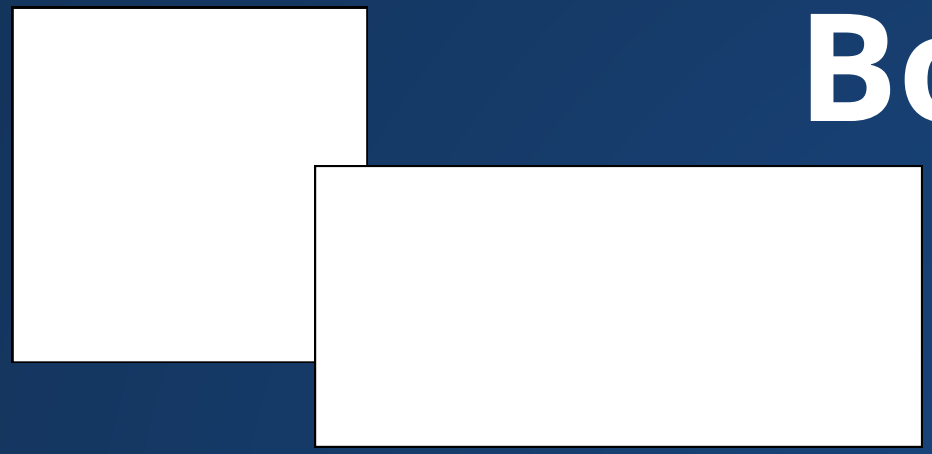
Box Collision



How can we tell if two boxes collide?

Similar idea to spheres, but we check along each axis (two in 2D, three in 3D).

Box collision



Box 1 centered at (x_1, y_1, z_1) with axial sizes of h_1, w_1, d_1.

Box 2 centered at (x_2, y_2, z_2) with axial sizes of h_2, w_2, d_2.

Boxes are **not** colliding if **any** of the following are true

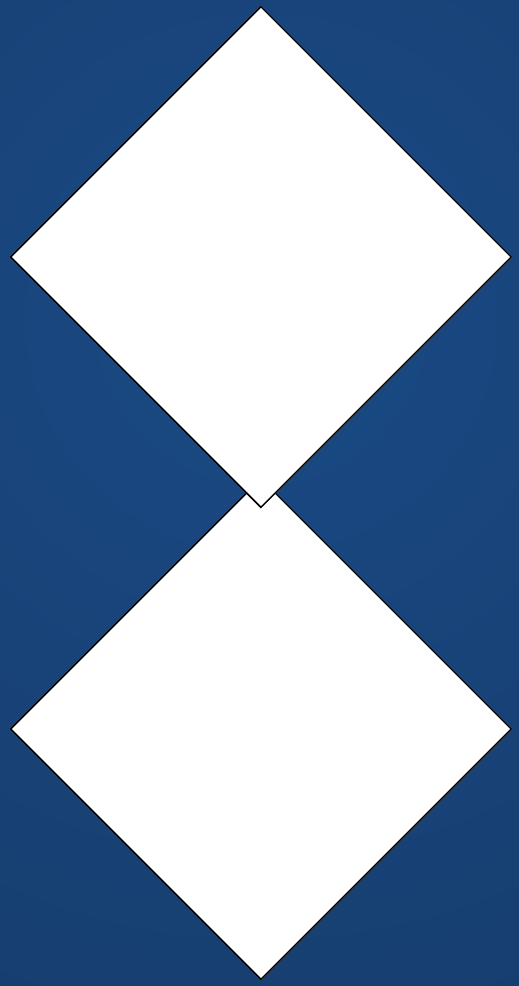
$$|x_1 - x_2| > \frac{w_1+w_2}{2}$$

$$|y_1 - y_2| > \frac{h_1+h_2}{2}$$

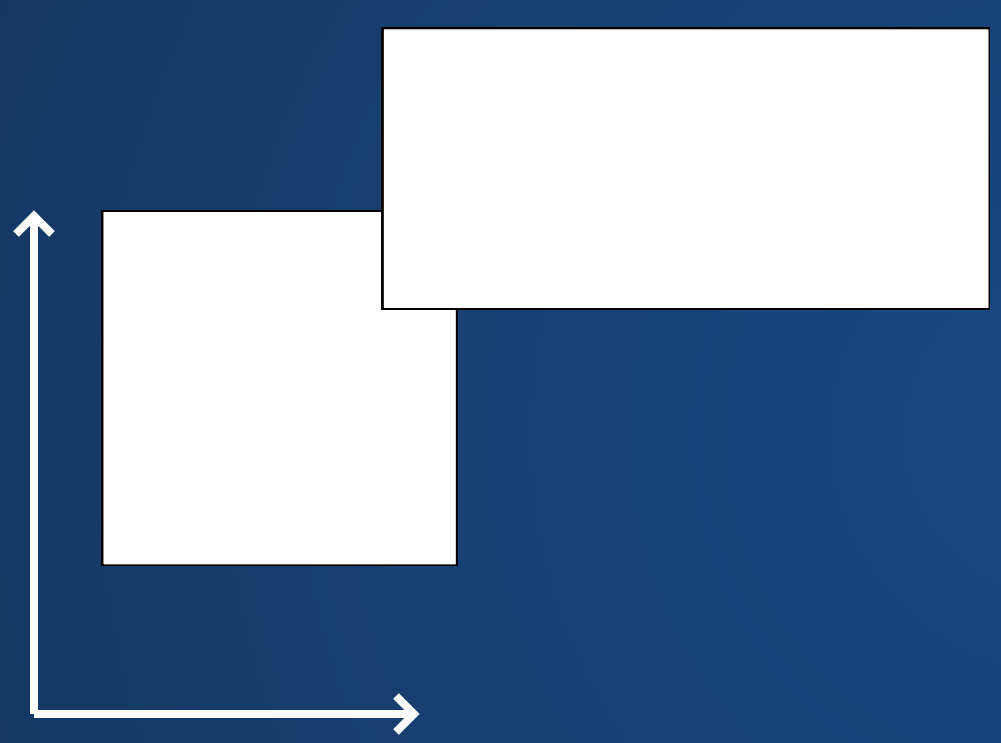
$$|z_1 - z_2| > \frac{d_1+d_2}{2}$$

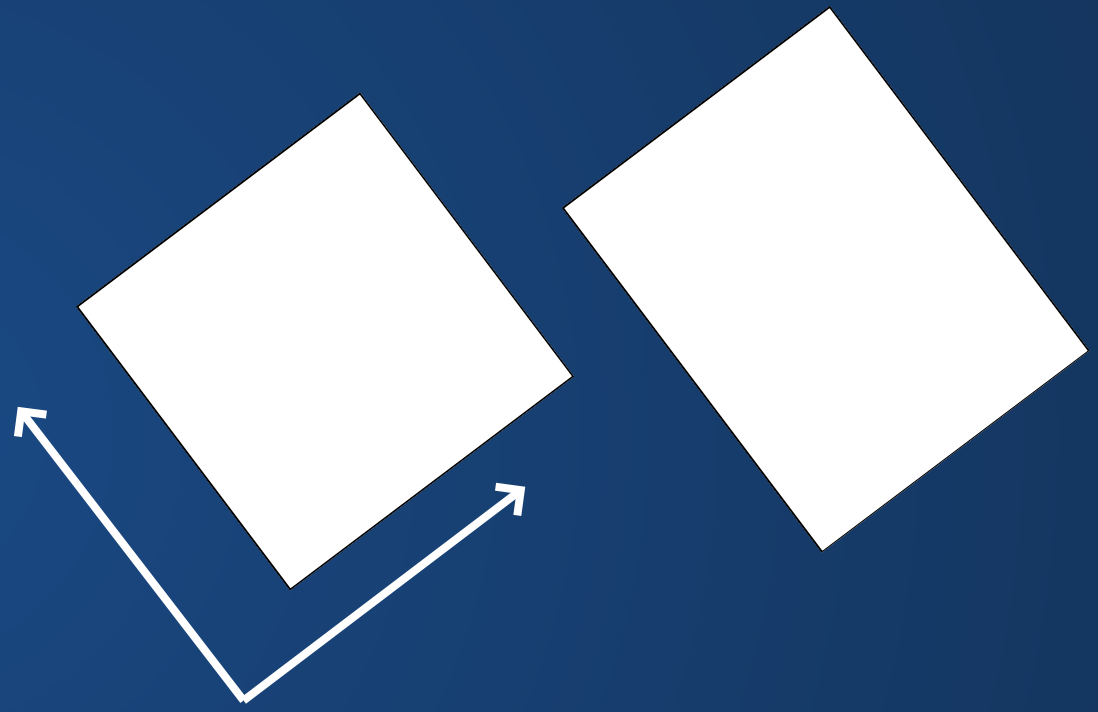
So boxes **are** colliding if all of the above are false.

Except this doesn't work :(



Axis-Aligned Box collision

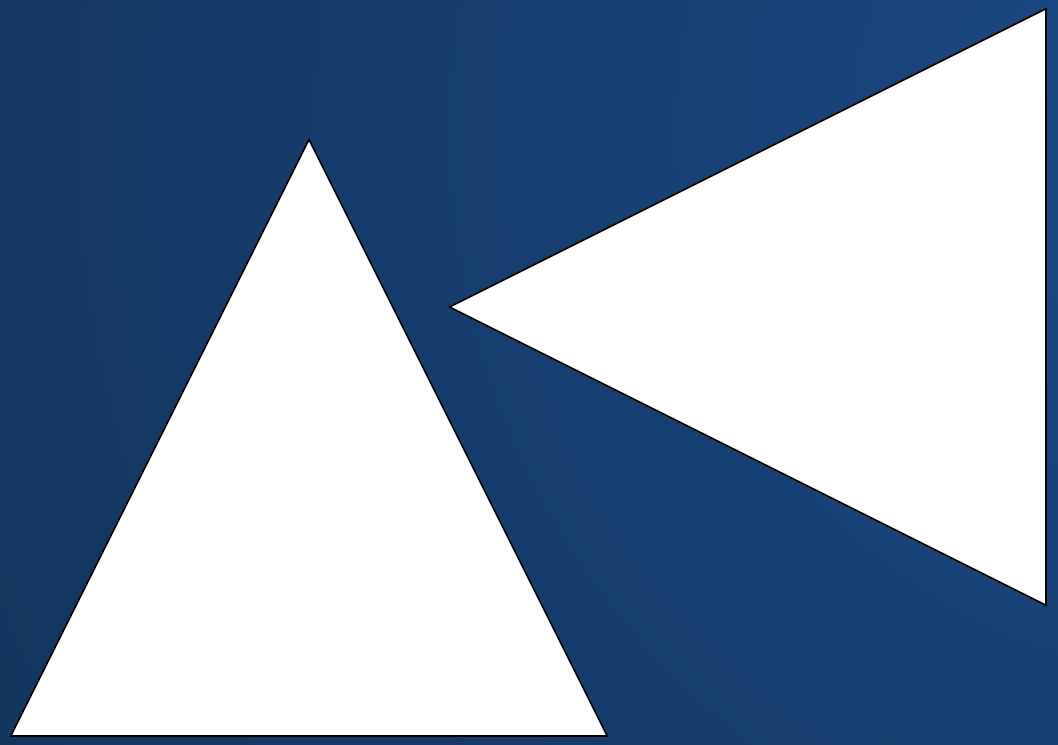




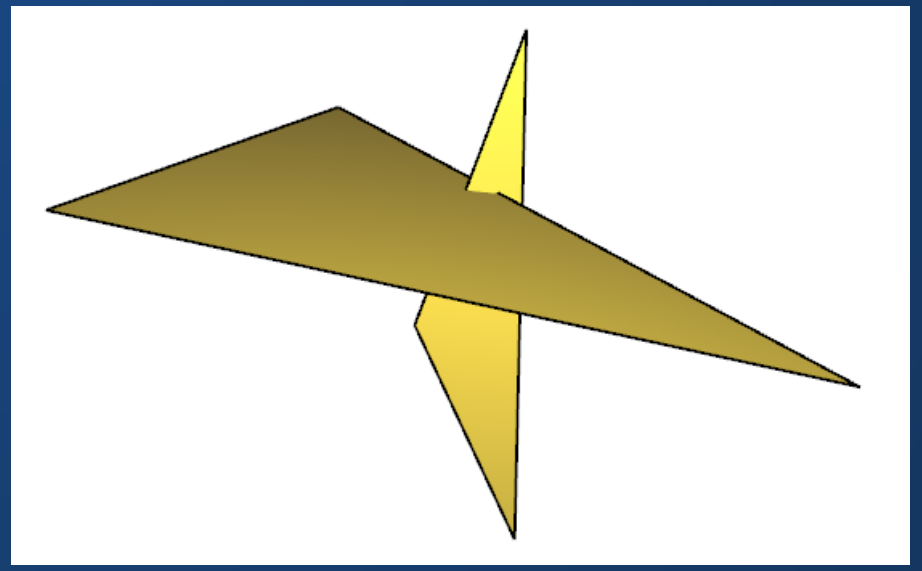
We require that the boxes are *axis-aligned*---that is, their unique directions line up with that of the coordinate system.

Triangle-Triangle Collision

This is a topic that's highly, highly studied. Many papers on the topic, not all of them correct.



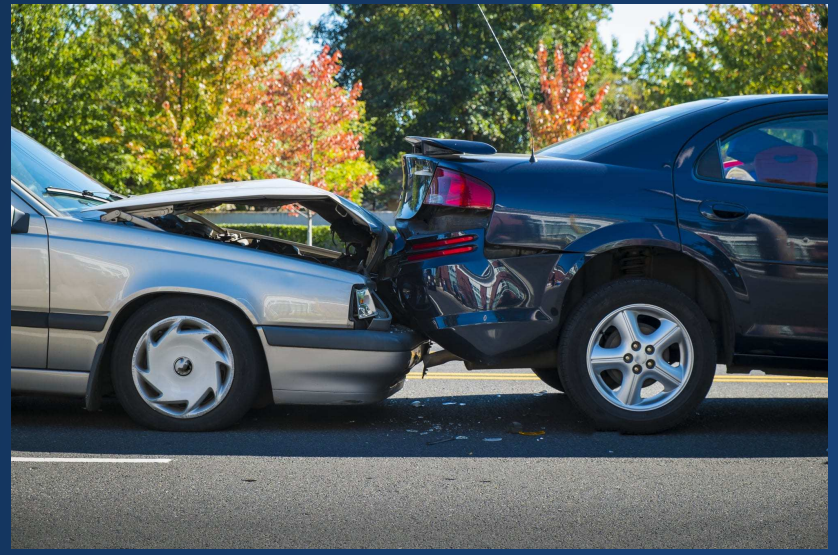
Also a little funky: we're trying to check if 2D shapes collide in 3D, but they don't have to be in the same plane!



Triangle-Triangle Collision

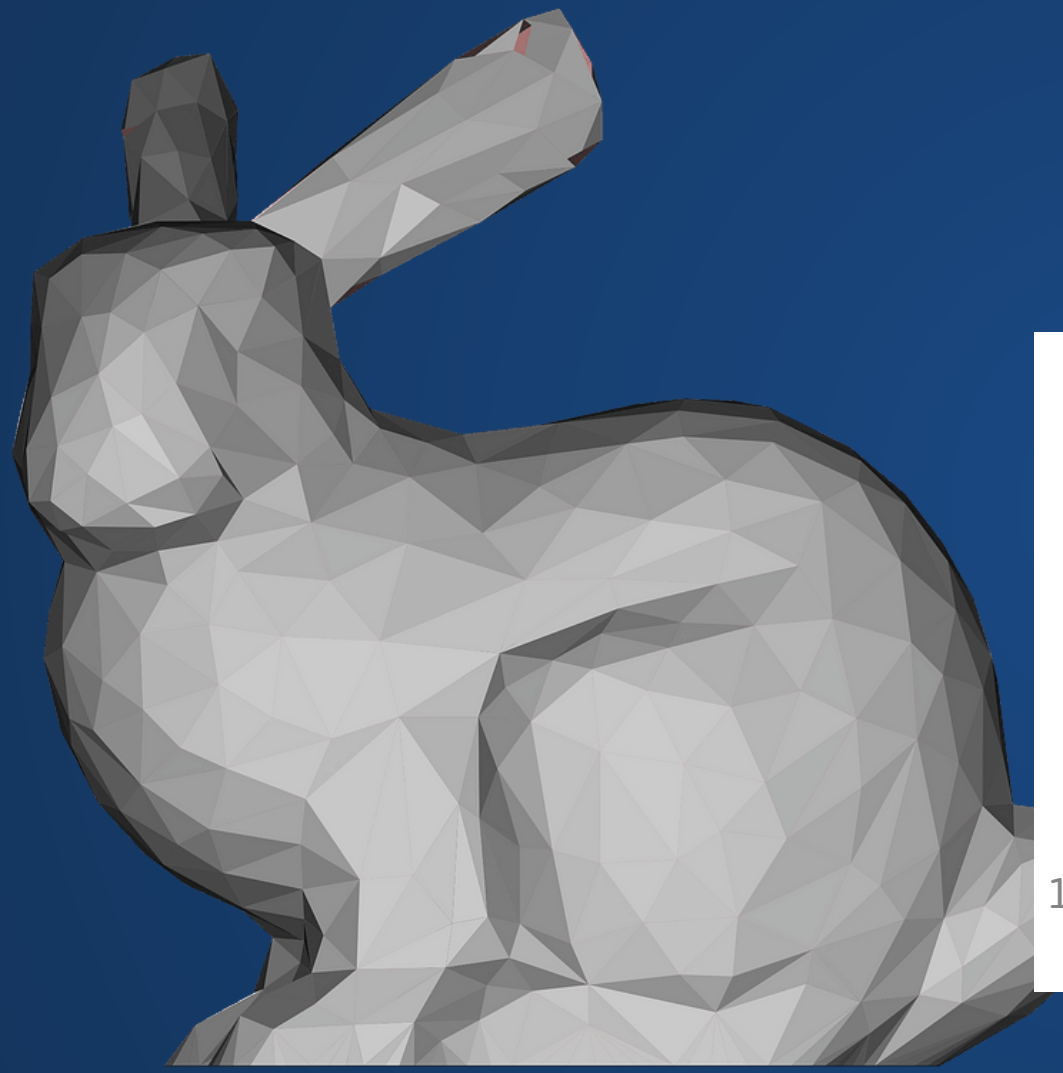
1. Take one triangle and extend it to an infinite plane.
2. For each edge of the other triangle:
    1. See if the edge intersects the infinite plane of the first triangle.
    2. If so, see if that intersection occurs inside or outside the first triangle.
    3. If the intersection occurs inside the first triangle, we have a collision.

This method is slow, but relatively foolproof. There are faster methods, but these often have issues in floating point math, or miss certain edge cases.



More Complex Shapes

Big Shape Made of Triangles



We can just intersect the triangles against each other!

```
1  boolean shapesIntersect(Shape s1, Shape s2){
2    for(int i = 0; i < s1.tri.length; i++){
3      for(int j = 0; j < s2.tri.length; j++){
4        if( s1.tri[i].intersects(s2.tri[j])){
5          return true;
6        }
7      }
8    }
9    return false;
10 }
```

Accelerating Collision Detection

Checking All Triangles is too slow!

The Stanford Dragon has 871,414 triangles.

How many checks do we need to see if two Stanford dragons collide with each other?

$$6 \times 871414 \times 871414 = 4\,556\,174\,156\,376$$

At 3 GHz, this is about 25 minutes.

Two-Phase Collisions

Suppose we have a scene with N objects. How many potential collisions are there?

Of those, how many do we expect to see actually occurring at any point?

Two-Phase Collisions

Observation: most of the time, only a few objects are colliding!

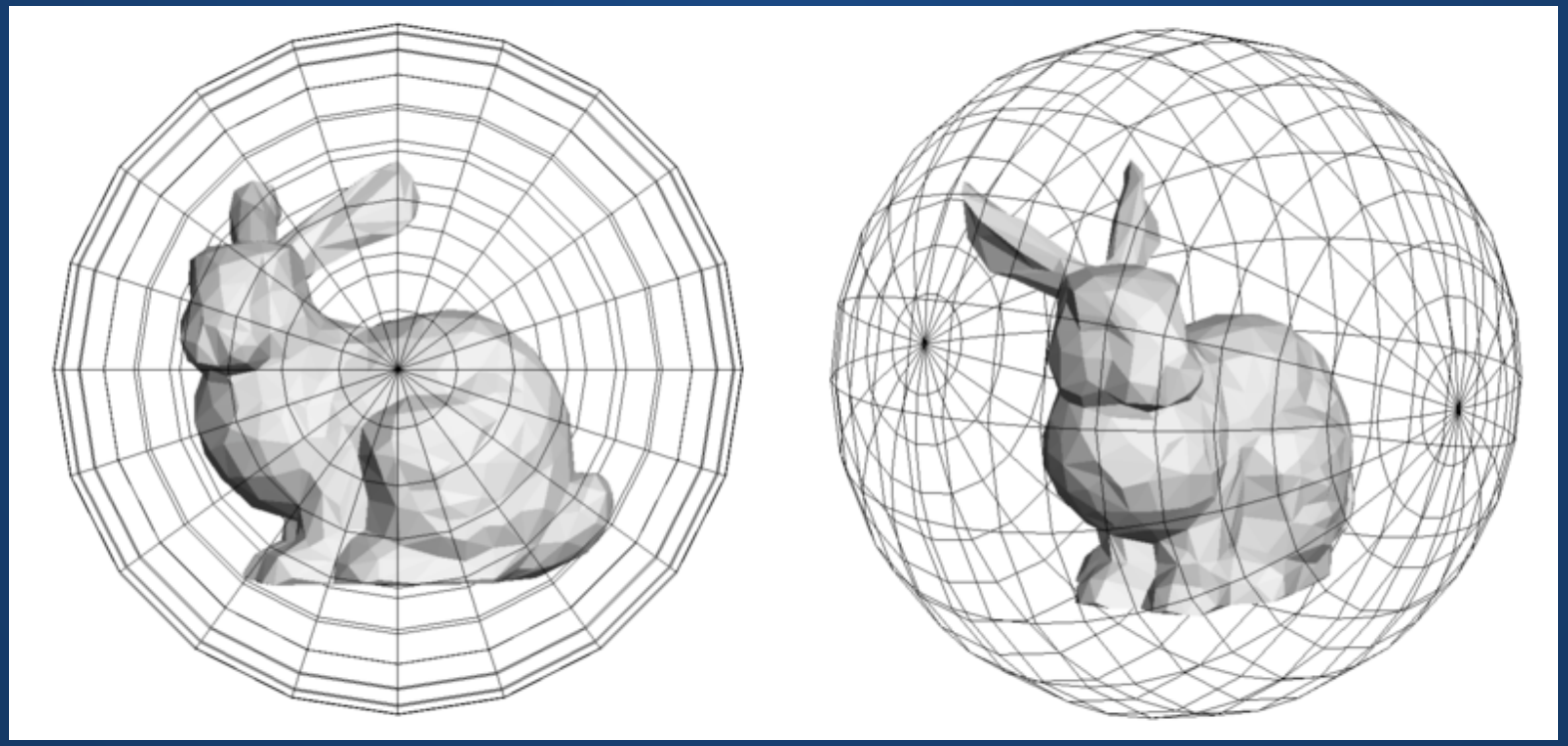
Separate our collision check into two phases:

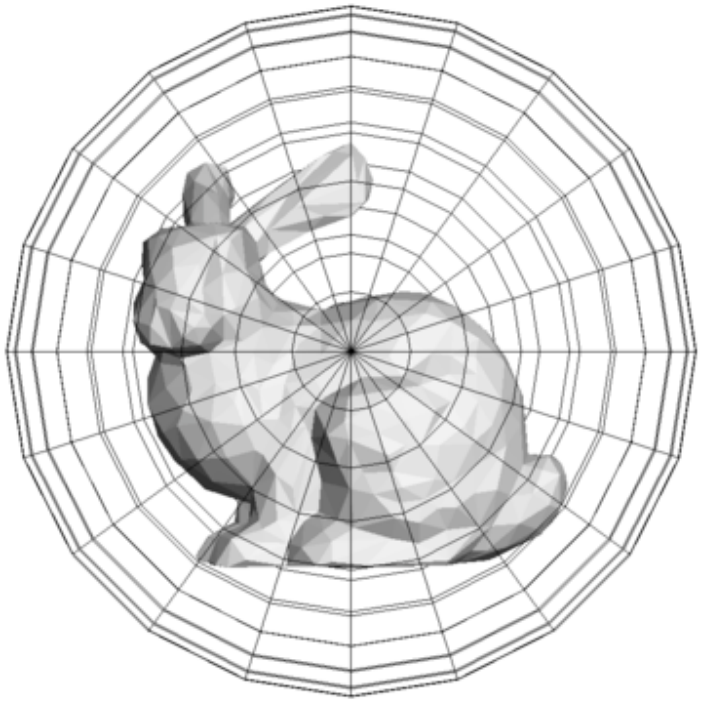
- *Broad phase*: run a quick, simple check to see if the objects might be colliding. We expect the answer to be "no", so let's make that the quick path.
- *Narrow phase*: if the shapes **might** be colliding, run a more in-depth scan to check (usually some variant of triangle-triangle).

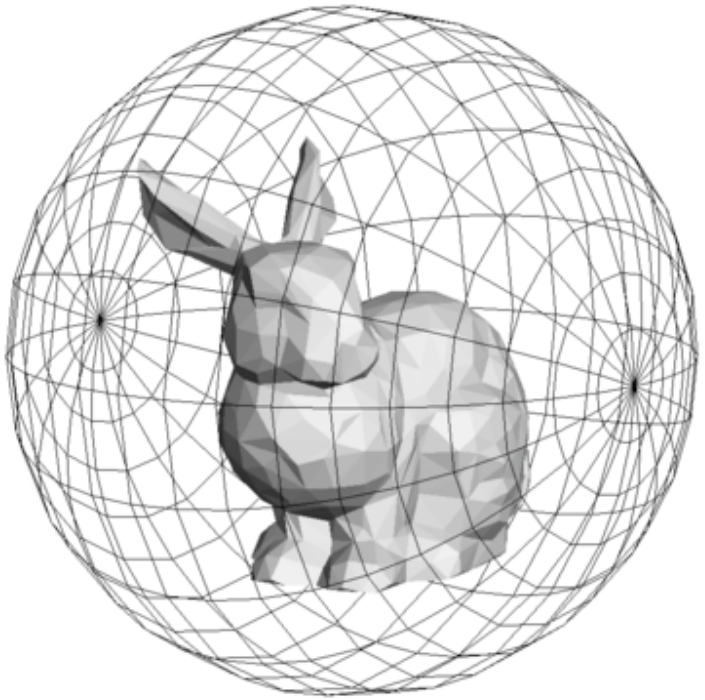
Broad-Phase Collision Checks

We need a check that satisfies the following:

- Responds to "Are the objects colliding" with either "no" or "maybe".
- Fast to compute.





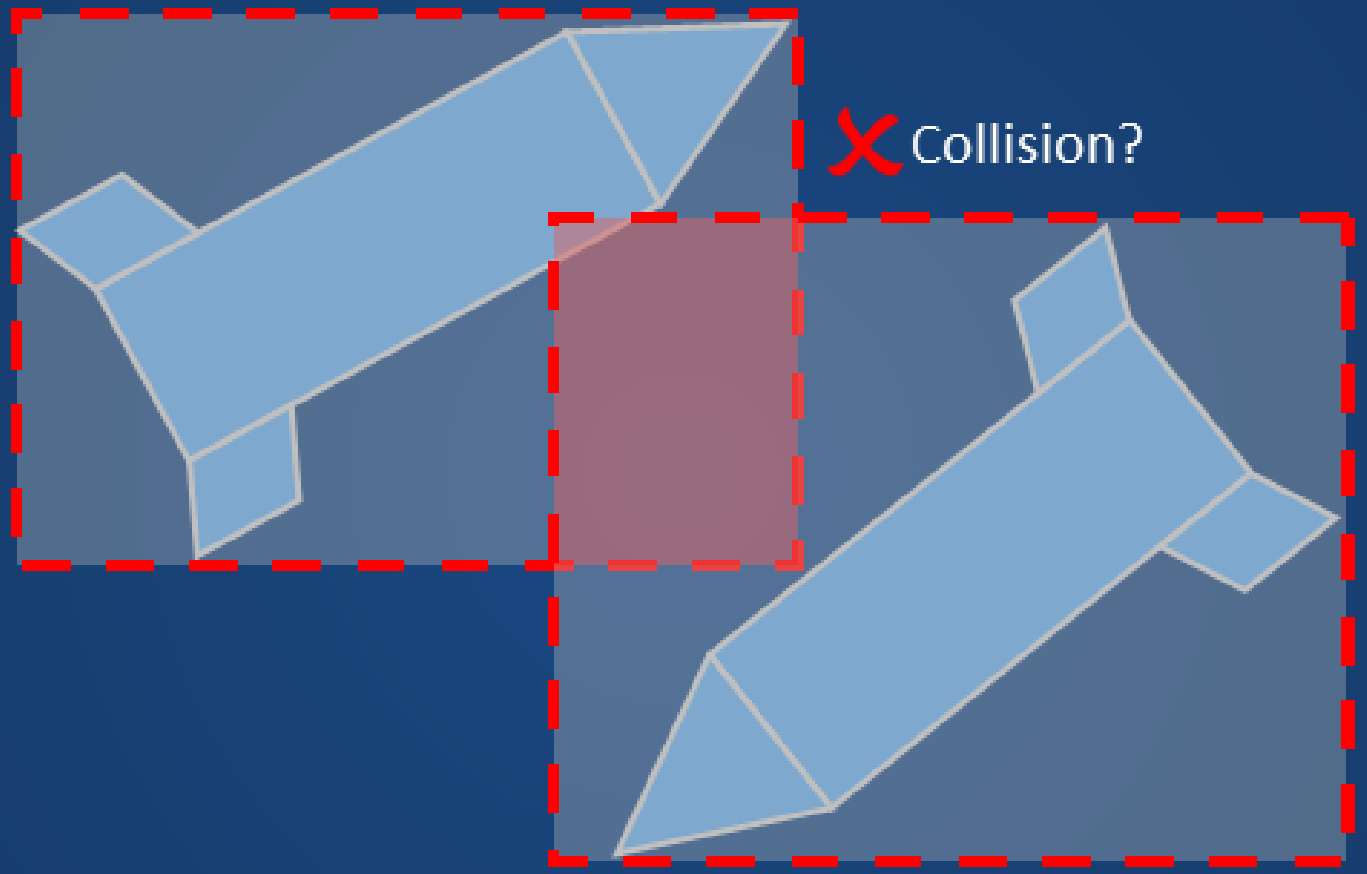


Sphere intersection is fast to compute.

If the spheres do not intersect, the objects **definitely** do not intersect.

If the spheres intersect, the objects **might** intersect. Proceed to narrow-phase.

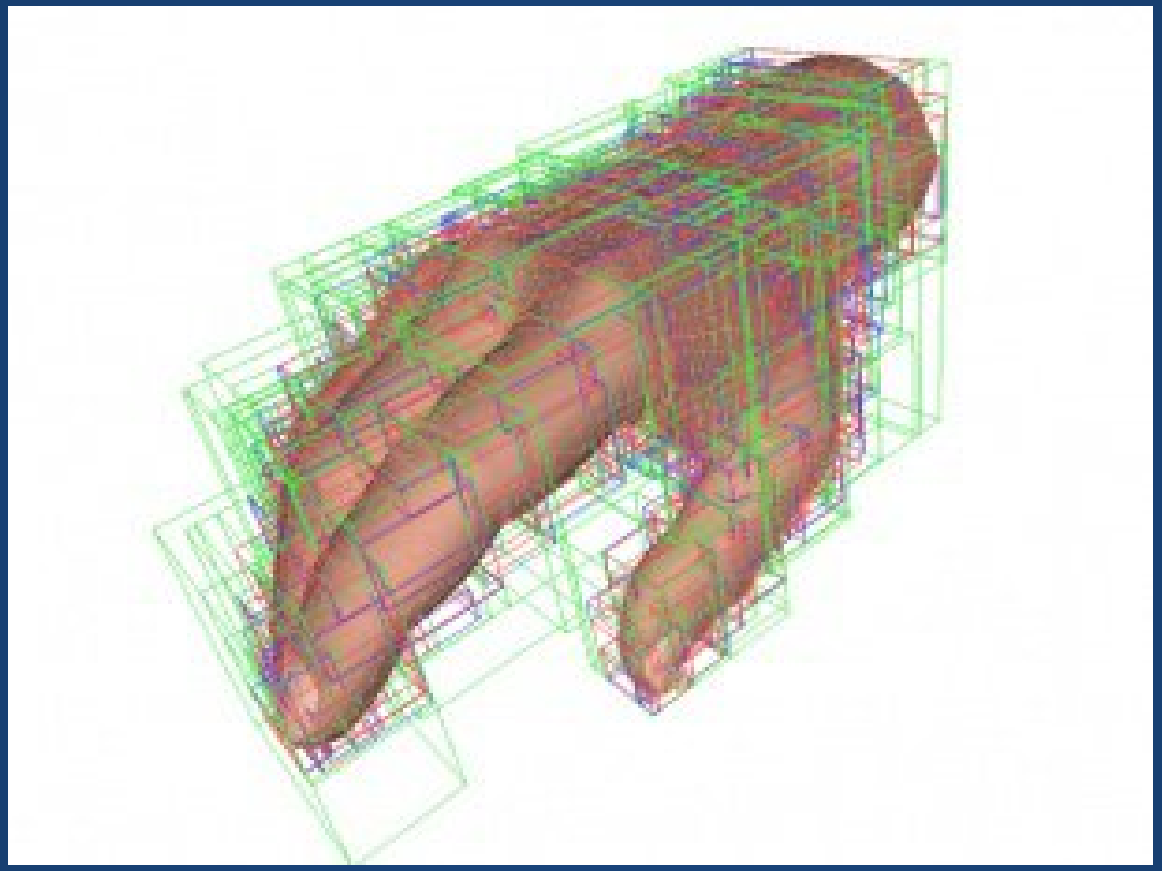
Axially-Aligned Bounding Box



More common for certain applications: AABBs.

1. Check intersection against outer (orange) box.
2. If intersection with orange box, check intersection against each of white boxes.
3. If intersection with any white box, proceed to narrow-phase detection.

Using Bounding Volumes in a broad-phase check is a good acceleration...but in some cases, it's still not fast enough.

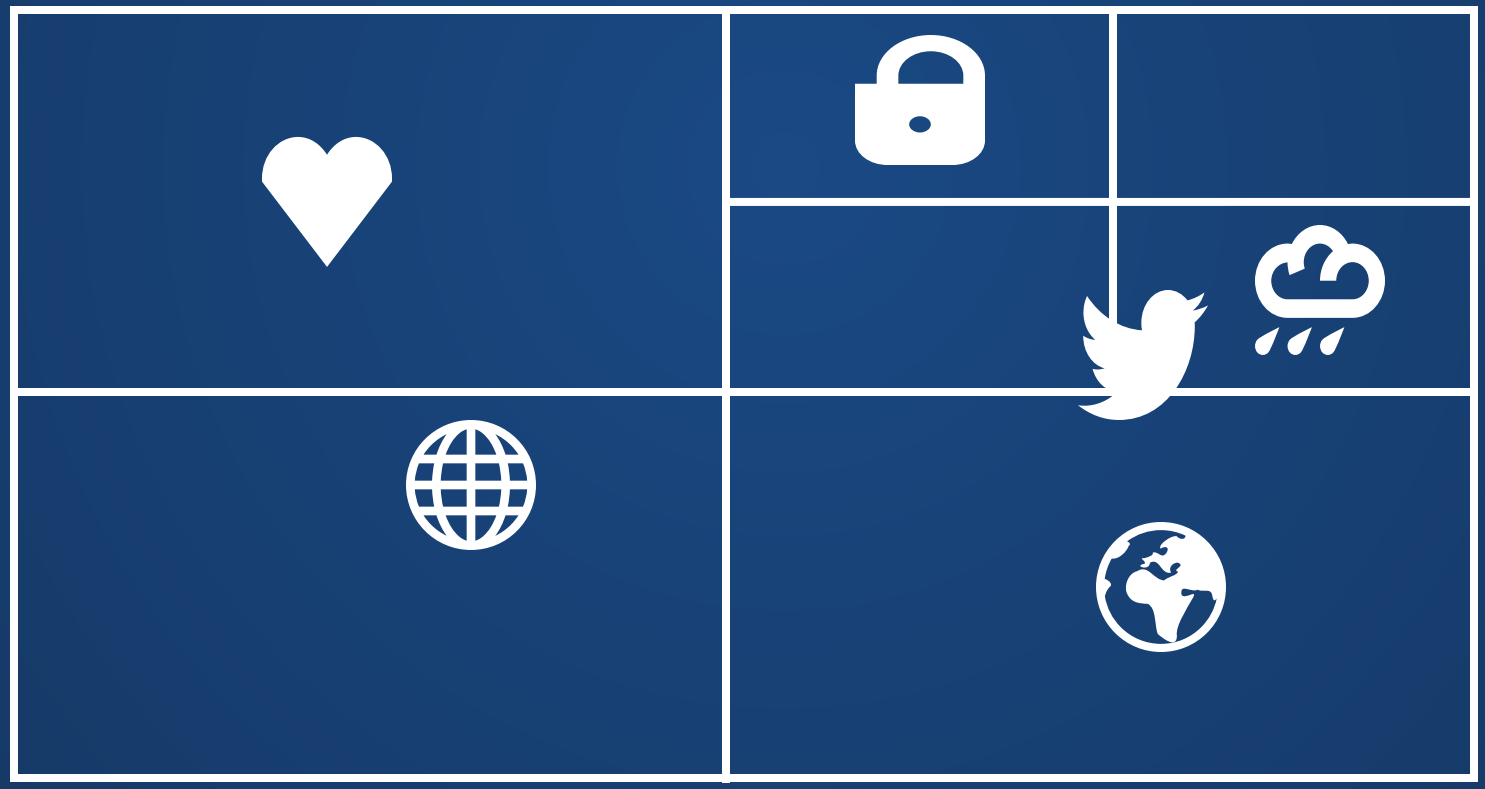


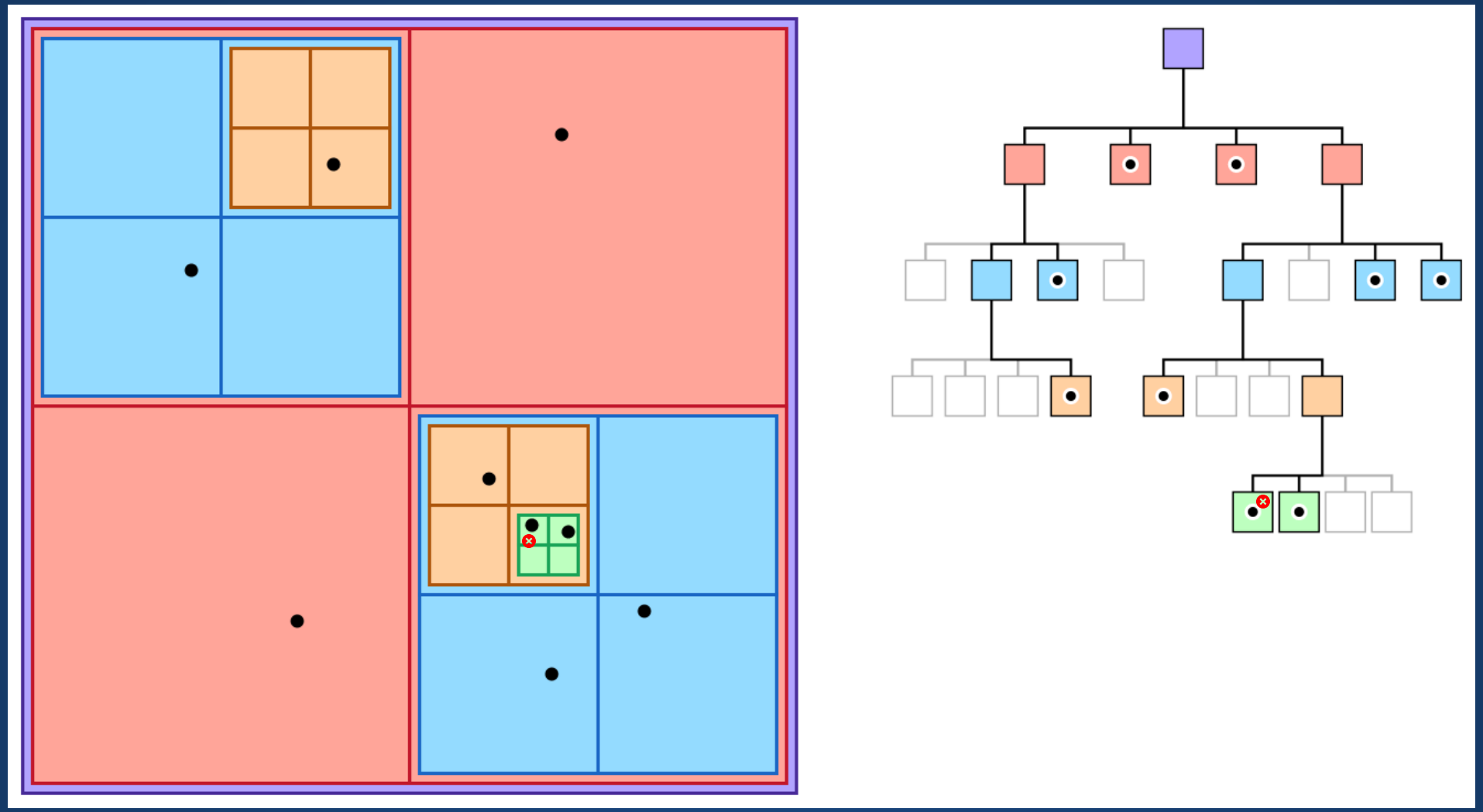
Instead of accelerating the collision checks, we need some way to avoid doing them altogether!

Spatial Data Structure

Similar idea to scene hierarchy, but tracks locations rather than transformation.

A way to organize shapes by their location.





Suppose we want to see if two of the circles here overlap. How would we check that?

The structure we have examined above is a *quadtree*. The 3D analogue structure is known as an *octtree*.

There are also other spatial data structures:

- Bounding Volume Hierarchy
- KD Tree
- Locality Sensitive Hash

Can combine these spatial data structures with bounding volumes to make some really fast checks!

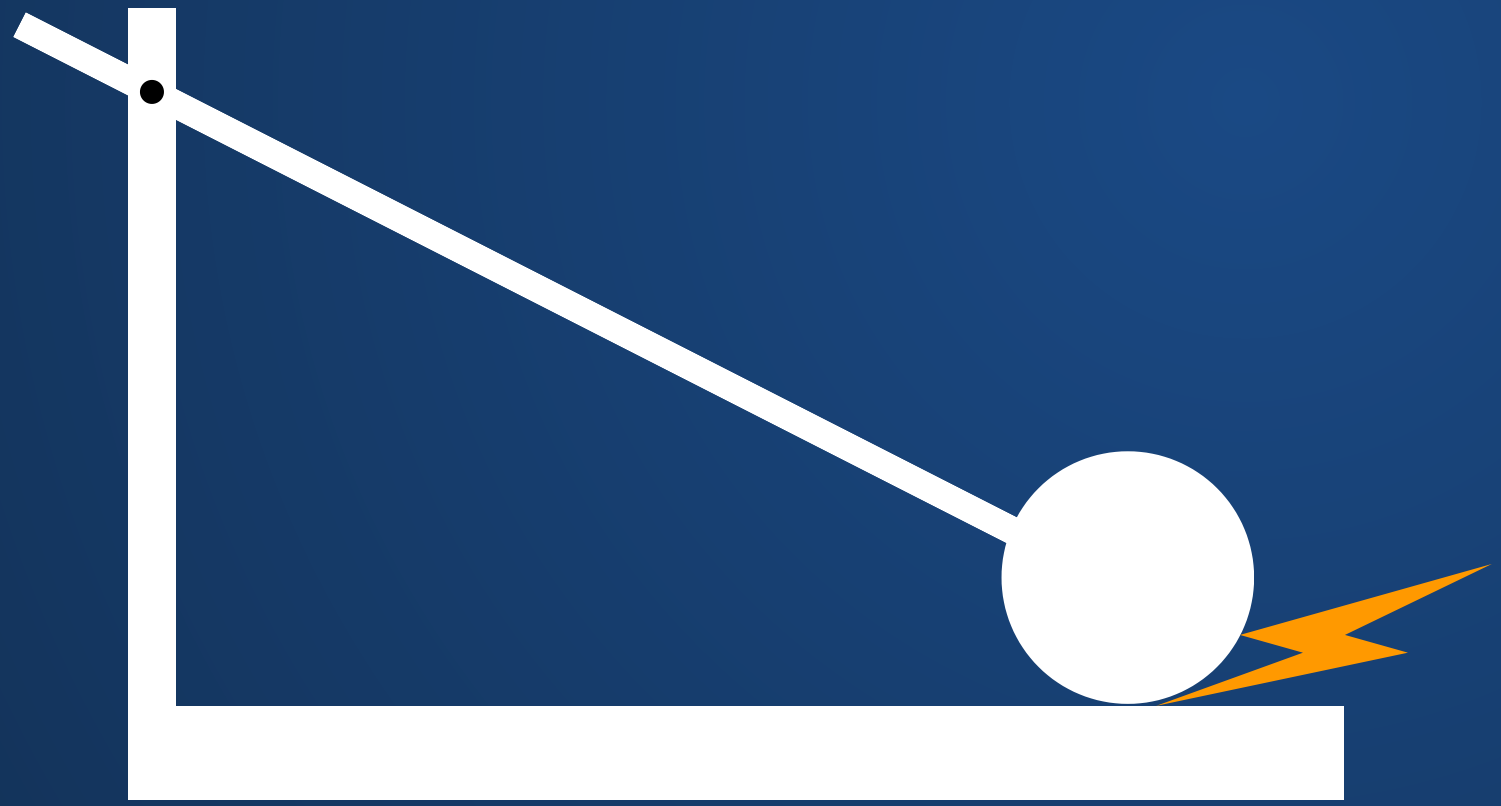
Collision Response

a.k.a. okay, they crashed. Now what?

we're going to deal with rigid bodies.

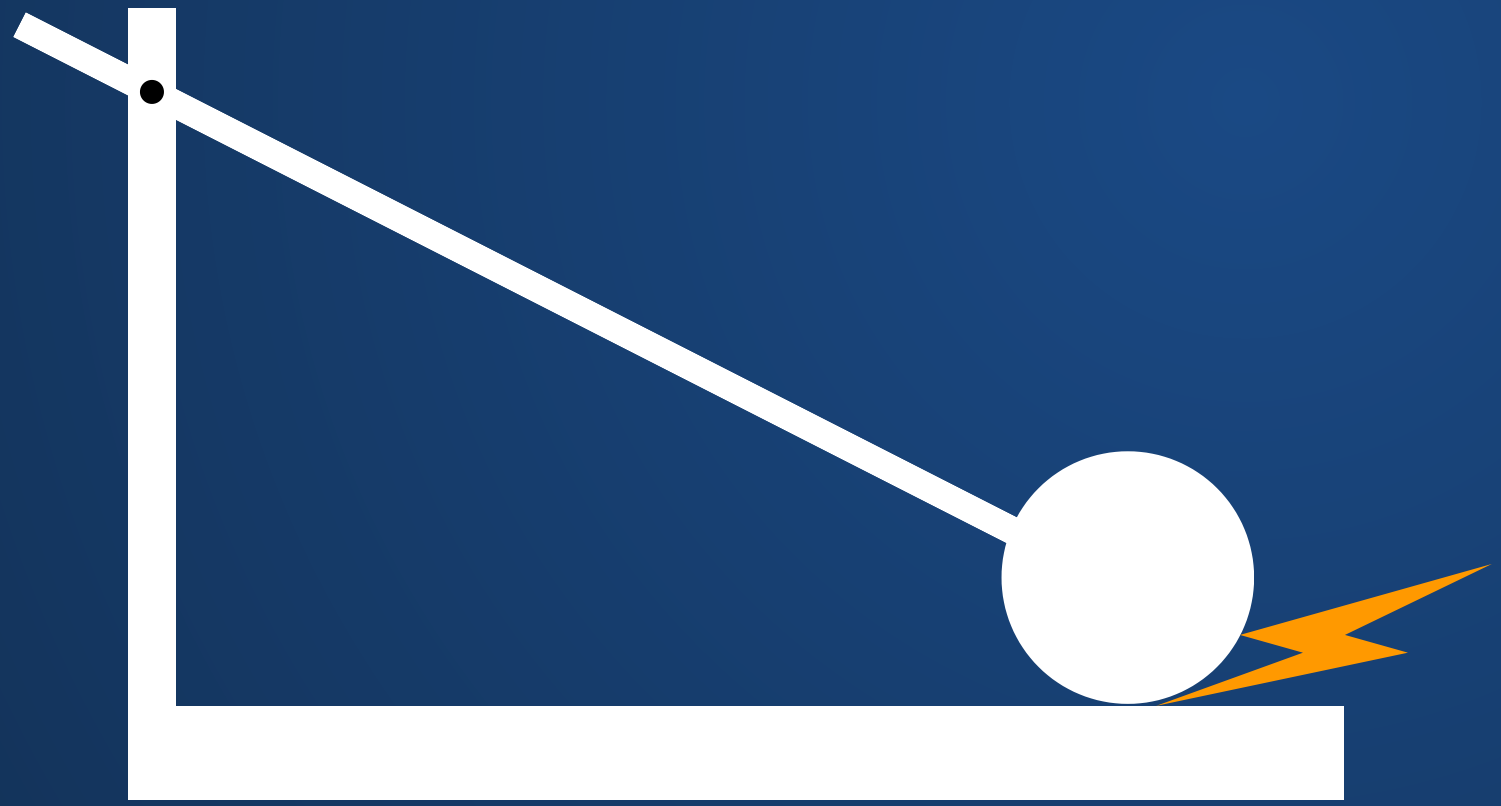
There are several strategies when a collision is detected

There are several strategies when a collision is detected

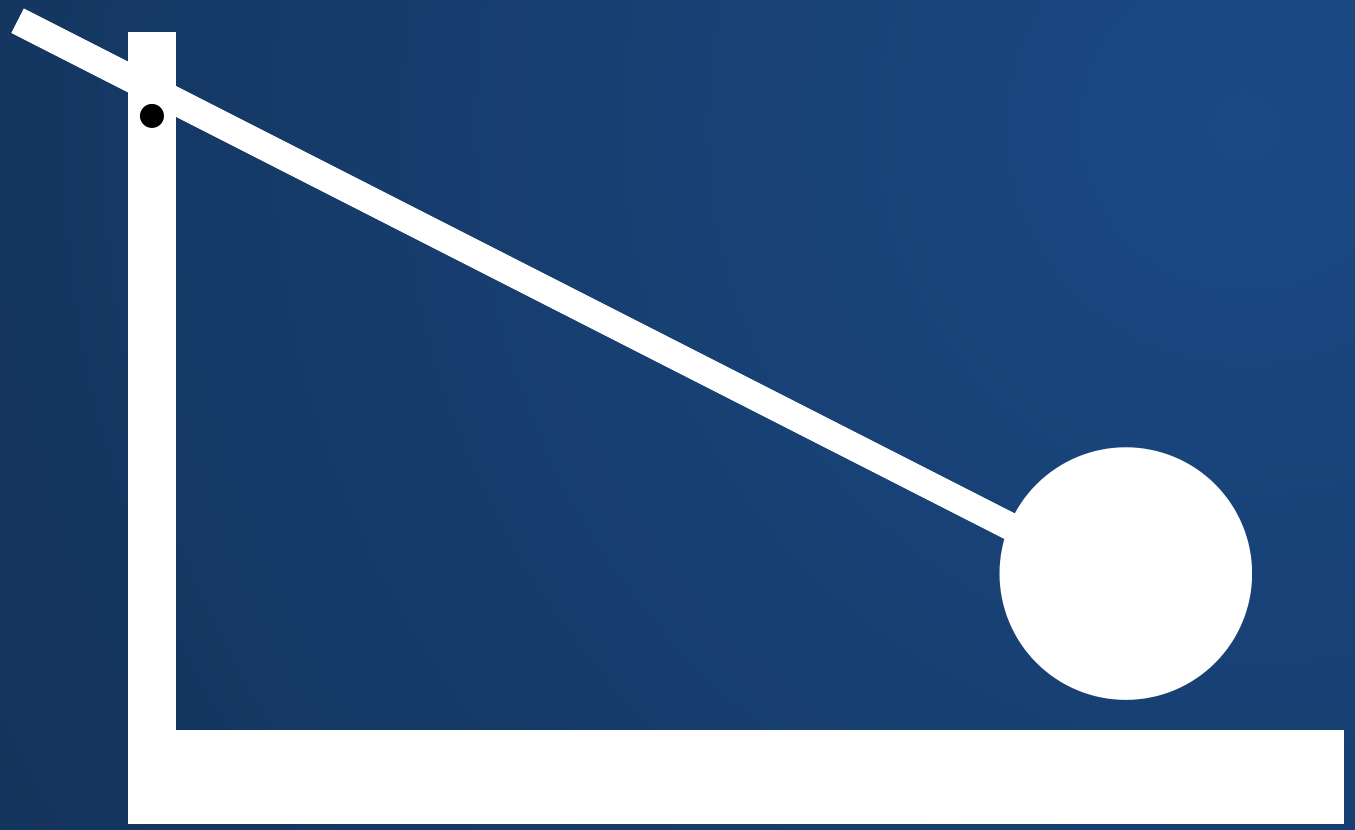


There are several strategies when a collision is detected

Option 1: Move the colliding shapes away from each other a small distance and restart the simulation.

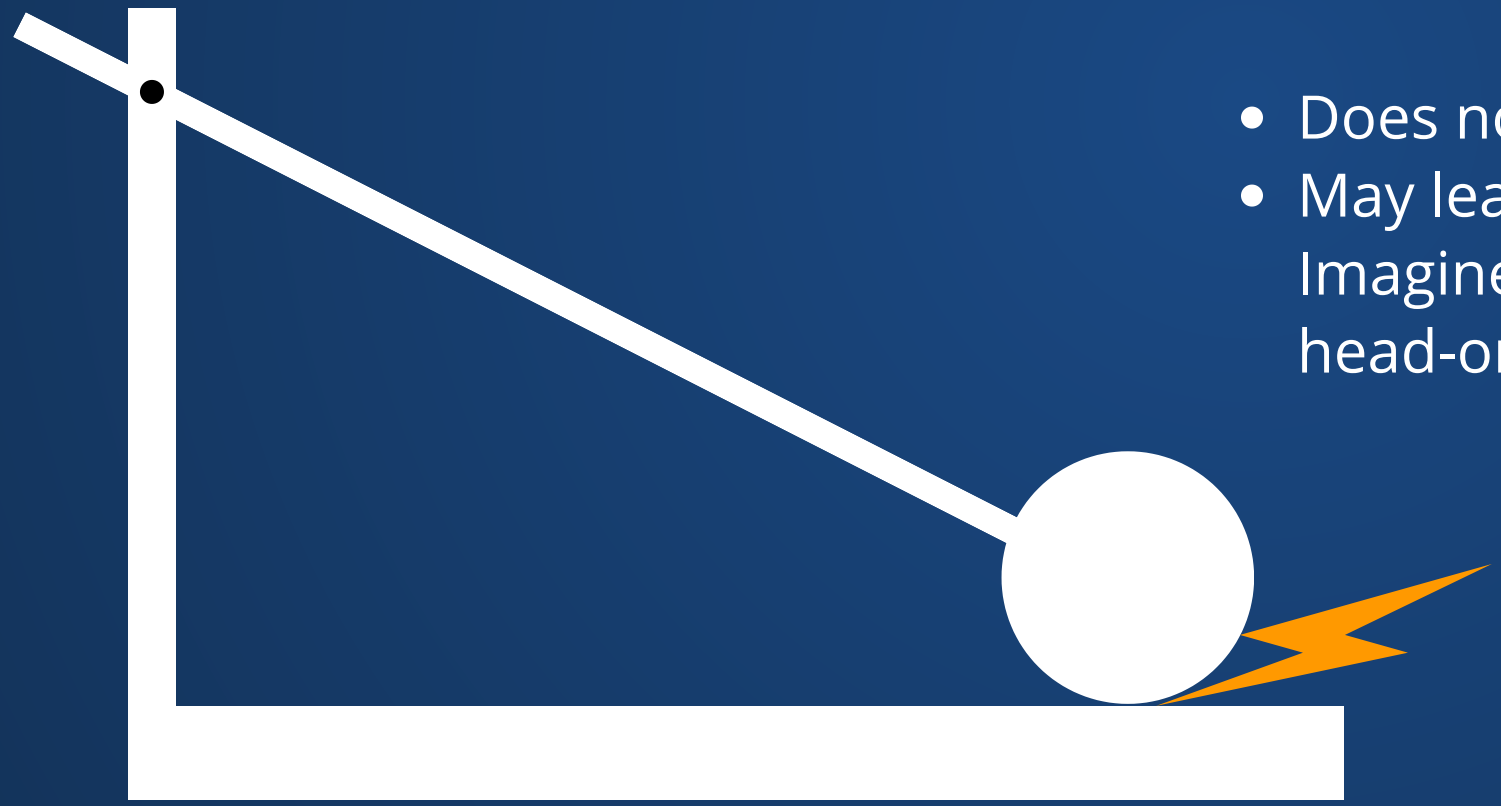


There are several strategies when a collision is detected



Option 1: Move the colliding shapes away from each other a small distance and restart the simulation.

There are several strategies when a collision is detected



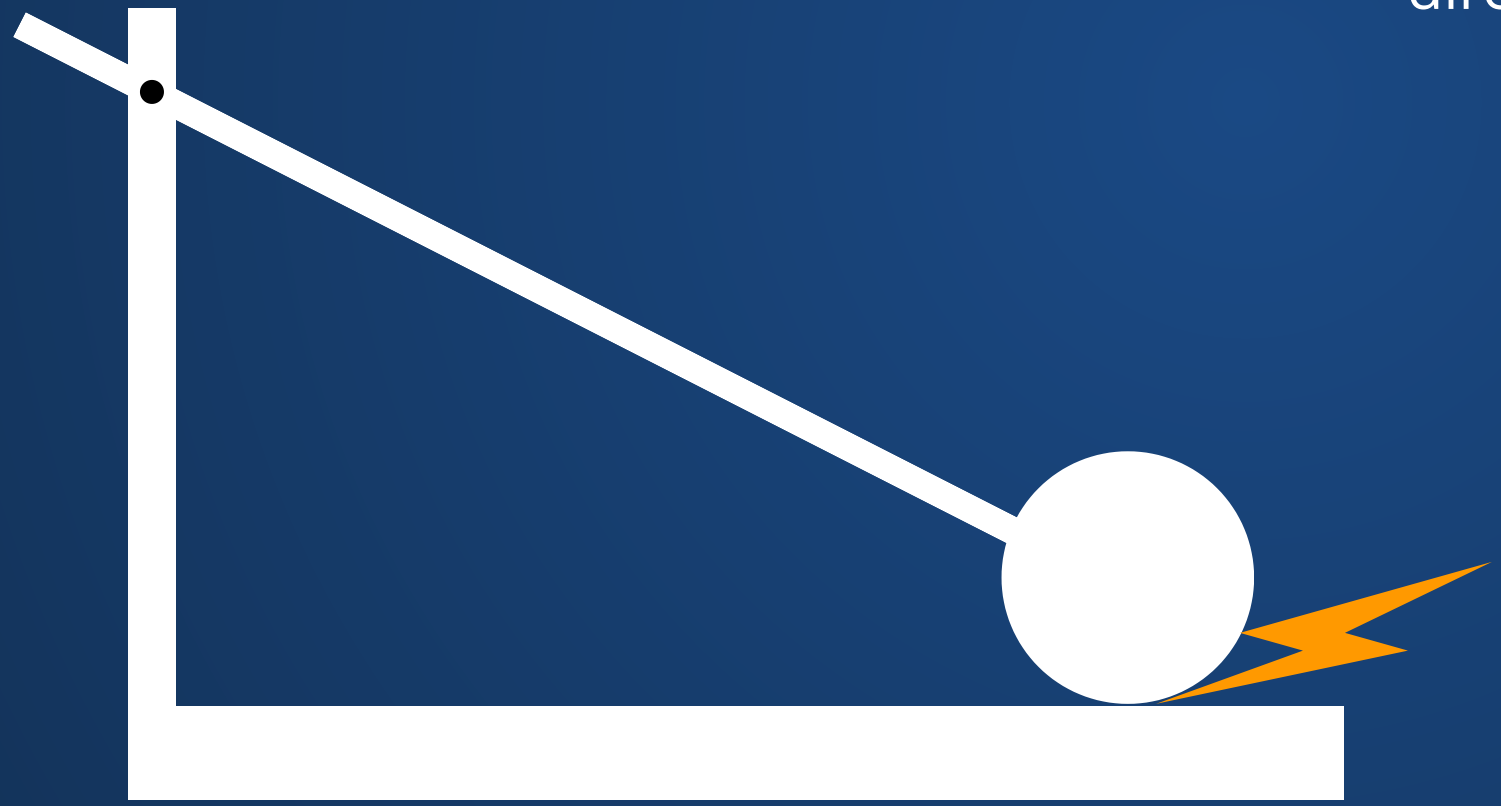
Yeah, this doesn't look great.

Other issues:

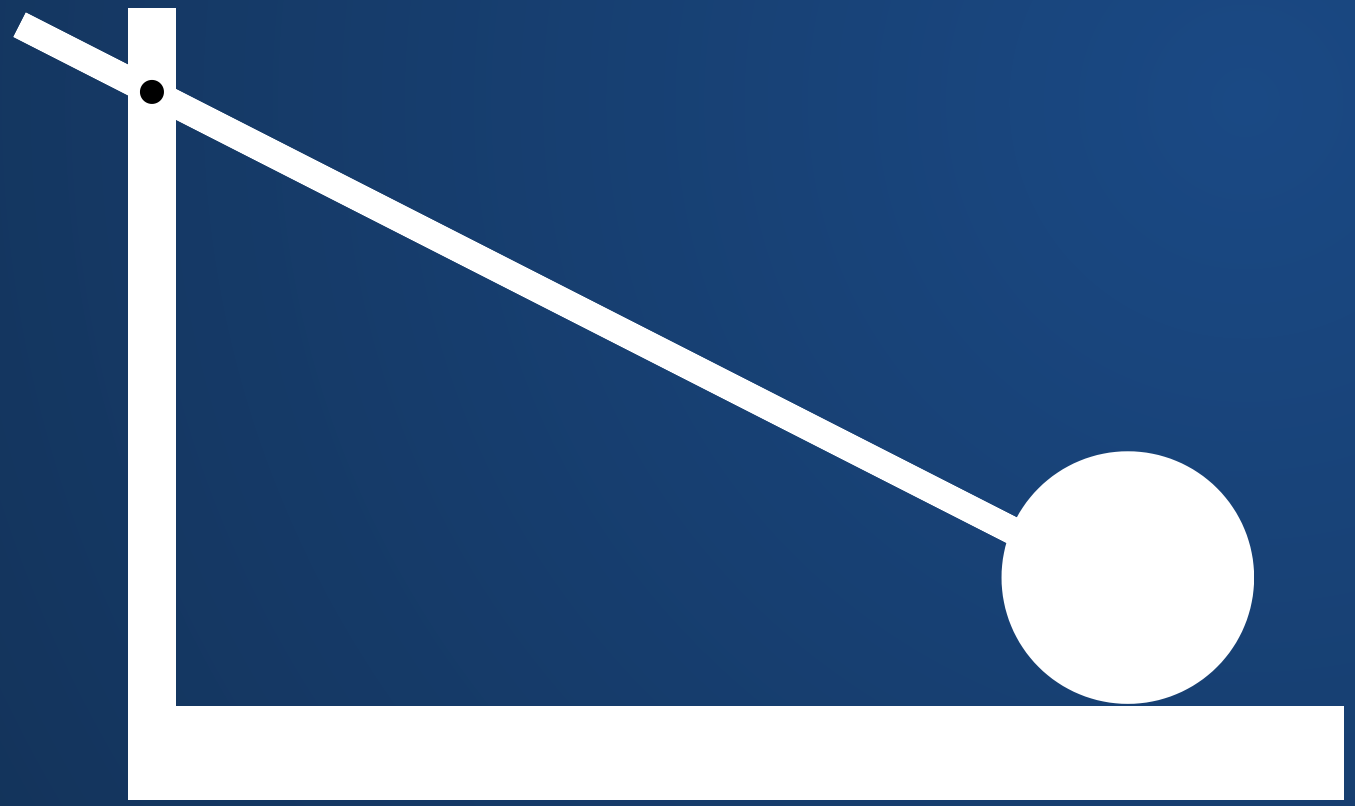
- Does not respect physics of simulation
- May lead to immediate re-collision. Imagine simulating two cars crashing head-on.

There are several strategies when a collision is detected

Option 2: Go for a completely inelastic collision: all velocity in the collision direction is immediately lost.

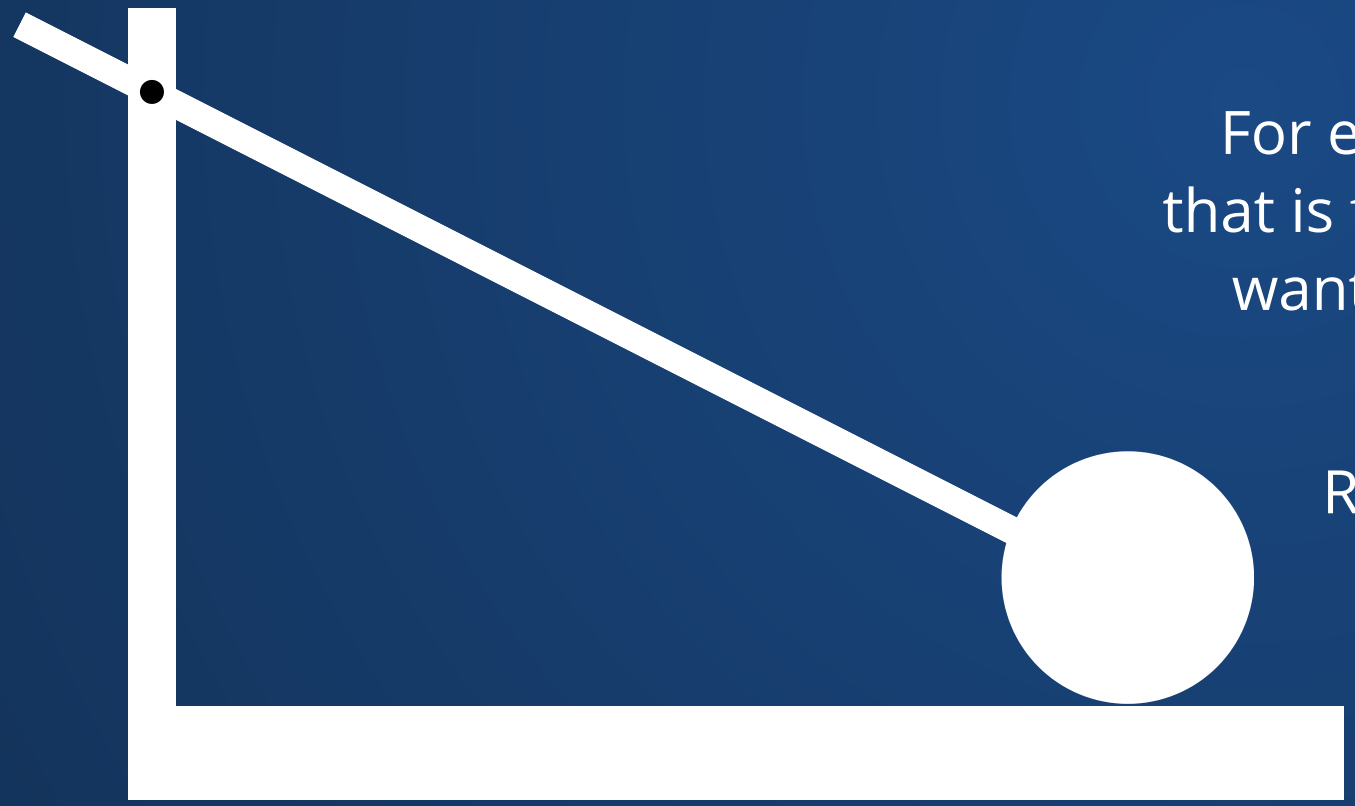


There are several strategies when a collision is detected



Option 2: Go for a completely inelastic collision: all velocity in the collision direction is immediately lost.

There are several strategies when a collision is detected



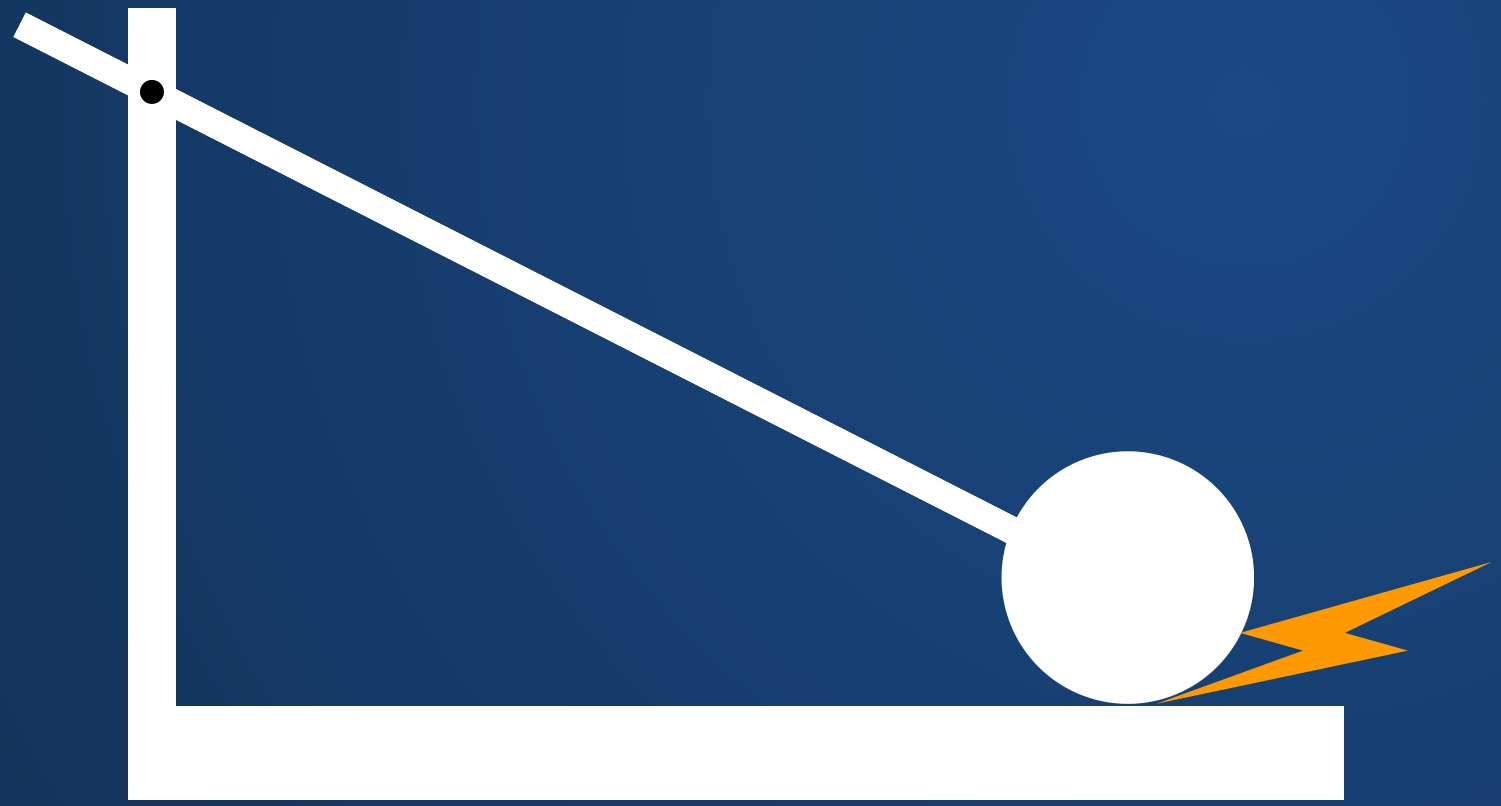
This is actually acceptable under certain situations!

For example, if we have a player character that is falling to the ground, we probably don't want to make them rebound into the sky.

Resolving all collisions this way leads to an unrealistic world.

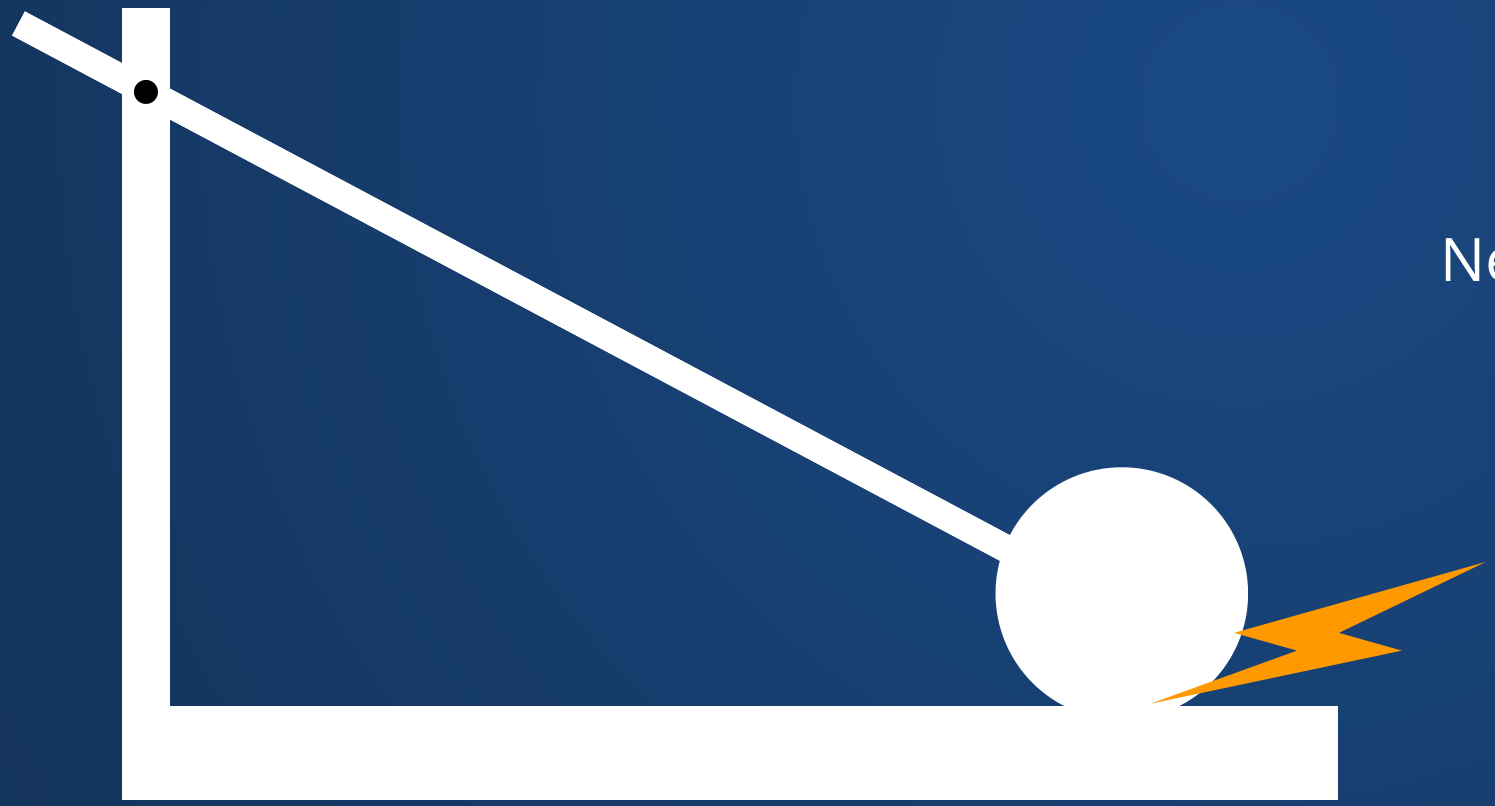
There are several strategies when a collision is detected

Option 3: Apply a force in the normal direction of the collision until the collision would no longer take place on the next timestep.



There are several strategies when a collision is detected

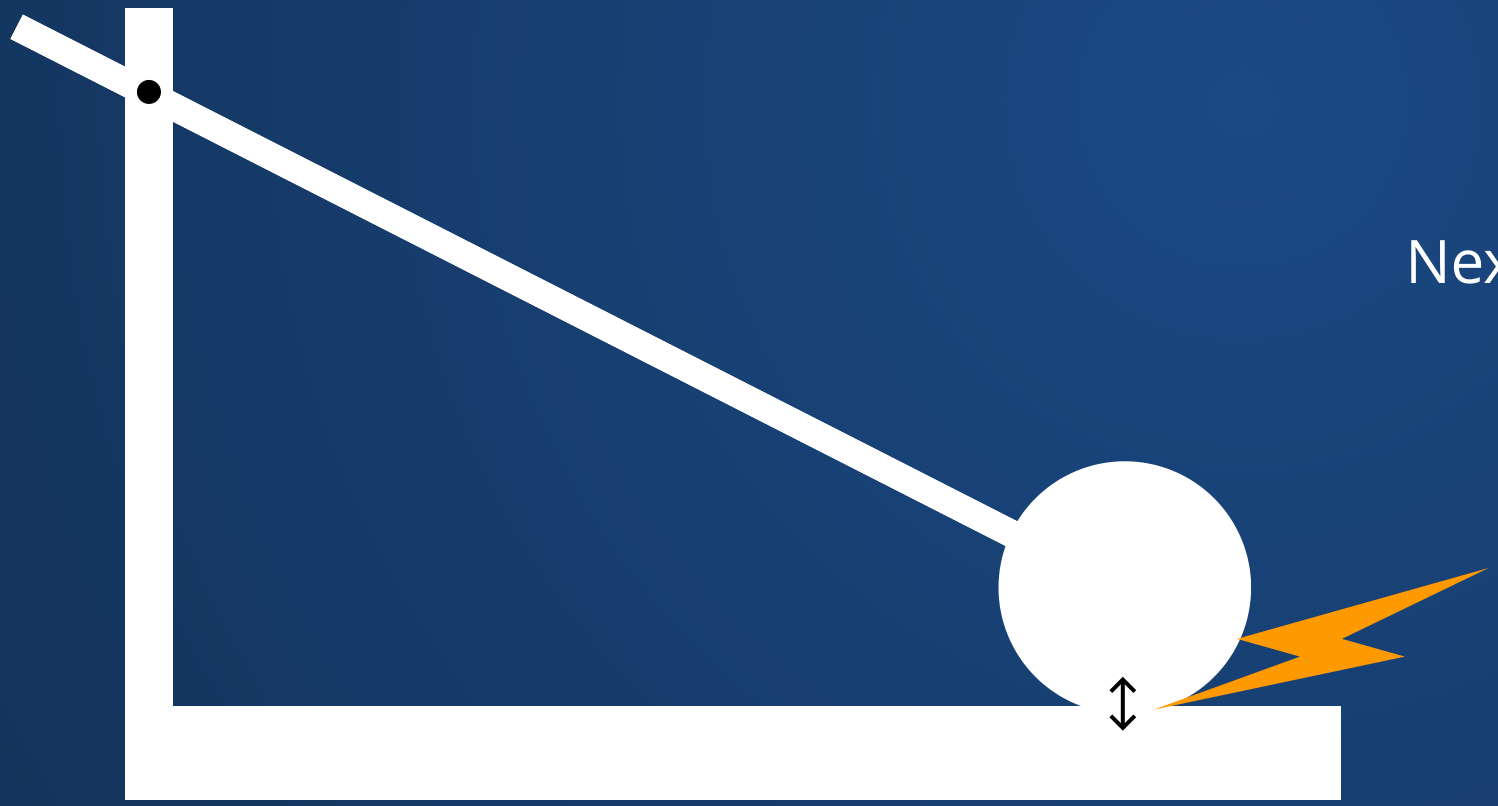
Option 3: Apply a force in the normal direction of the collision until the collision would no longer take place on the next timestep.



Next frame with no force

There are several strategies when a collision is detected

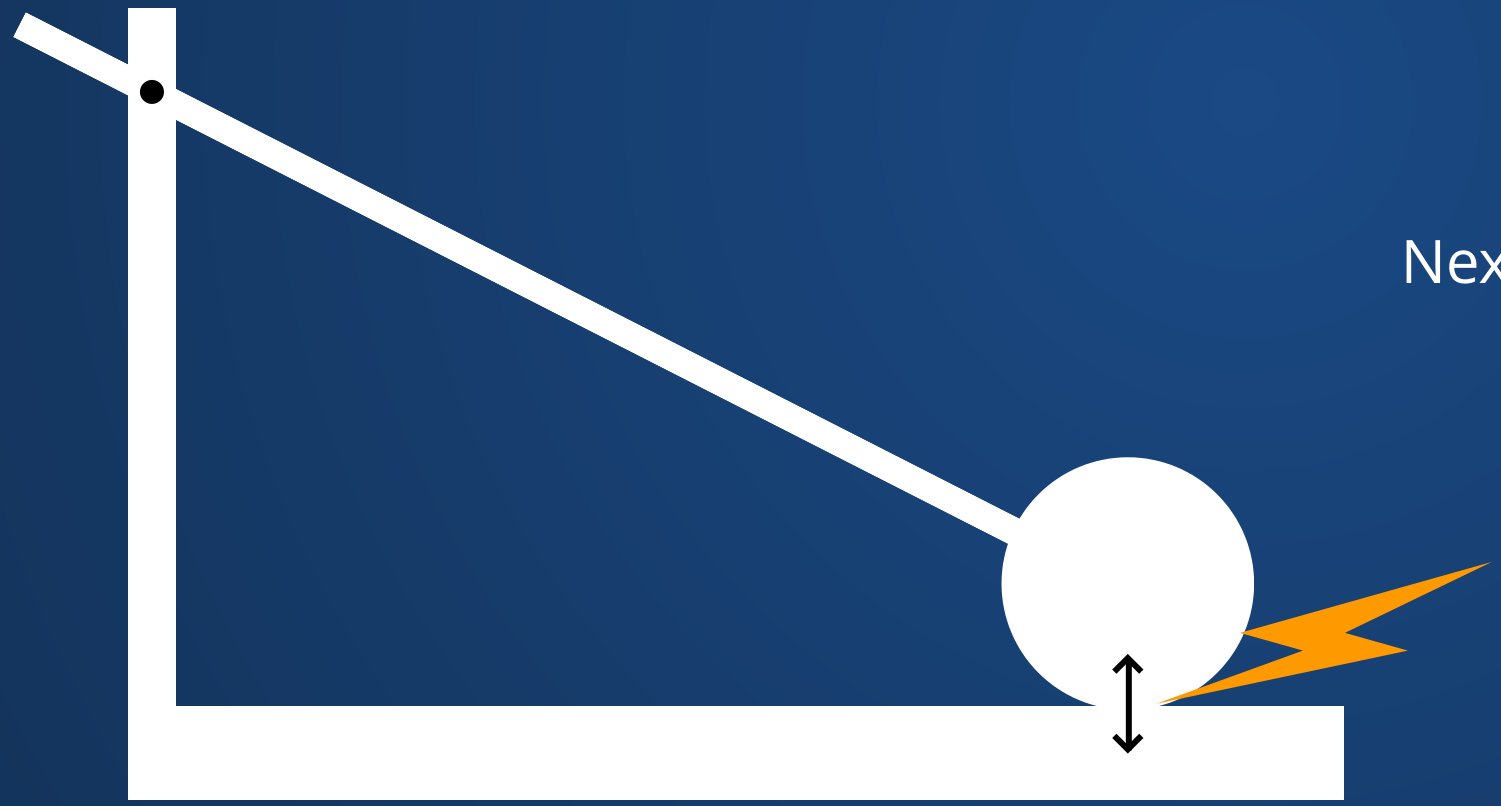
Option 3: Apply a force in the normal direction of the collision until the collision would no longer take place on the next timestep.



Next frame with small force

There are several strategies when a collision is detected

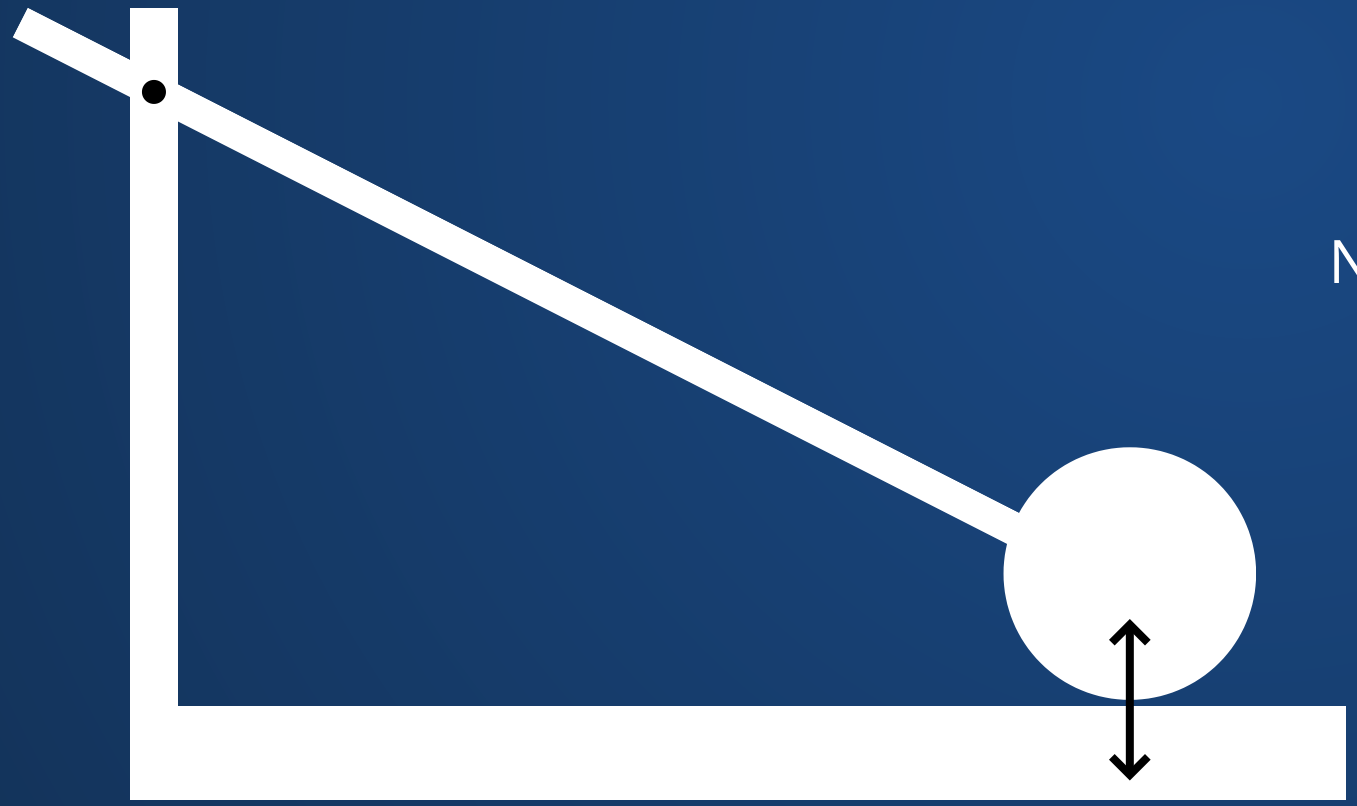
Option 3: Apply a force in the normal direction of the collision until the collision would no longer take place on the next timestep.



Next frame with larger force

There are several strategies when a collision is detected

Option 3: Apply a force in the normal direction of the collision until the collision would no longer take place on the next timestep.



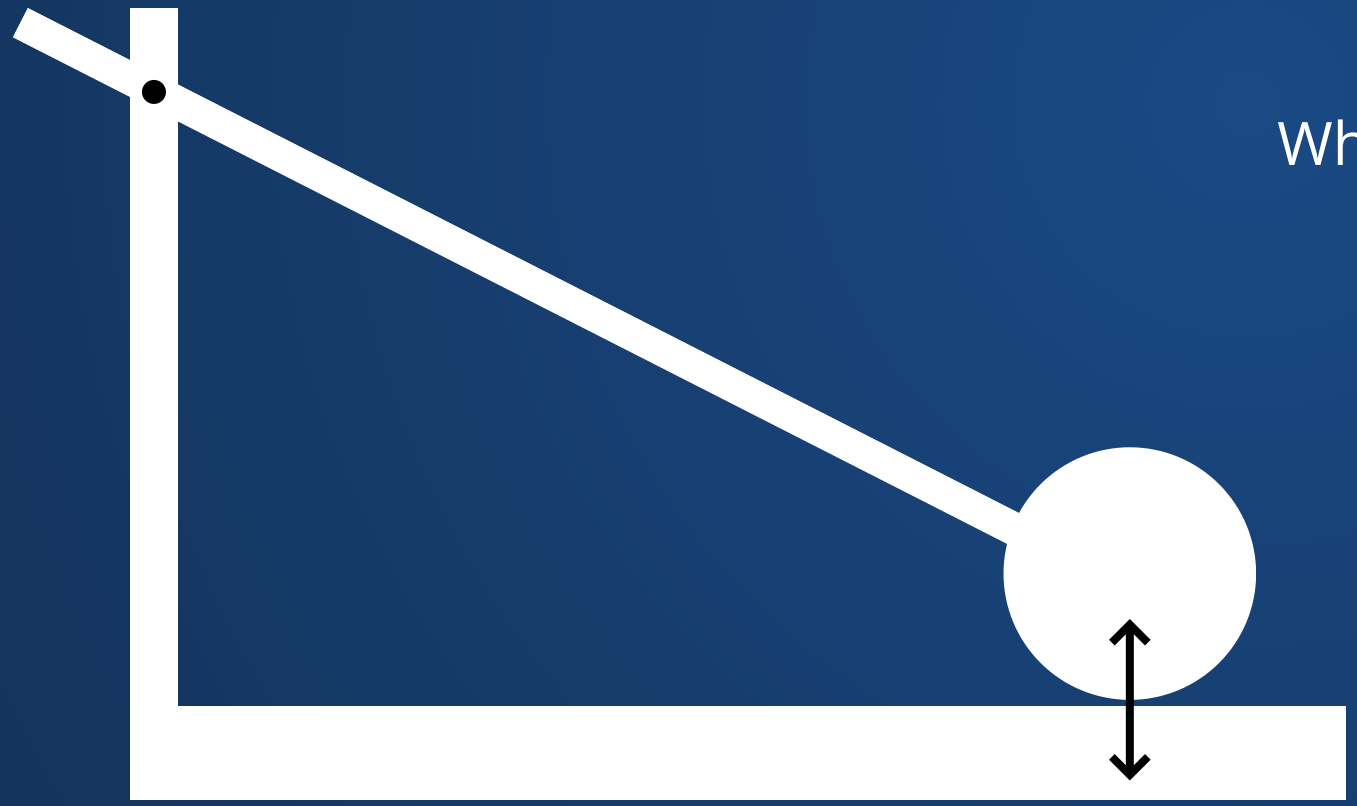
Next frame with even larger force

This force is sufficient to prevent the collision on this frame, so we apply this force to the ball and to the stand (Newton's Third Law)

There are several strategies when a collision is detected

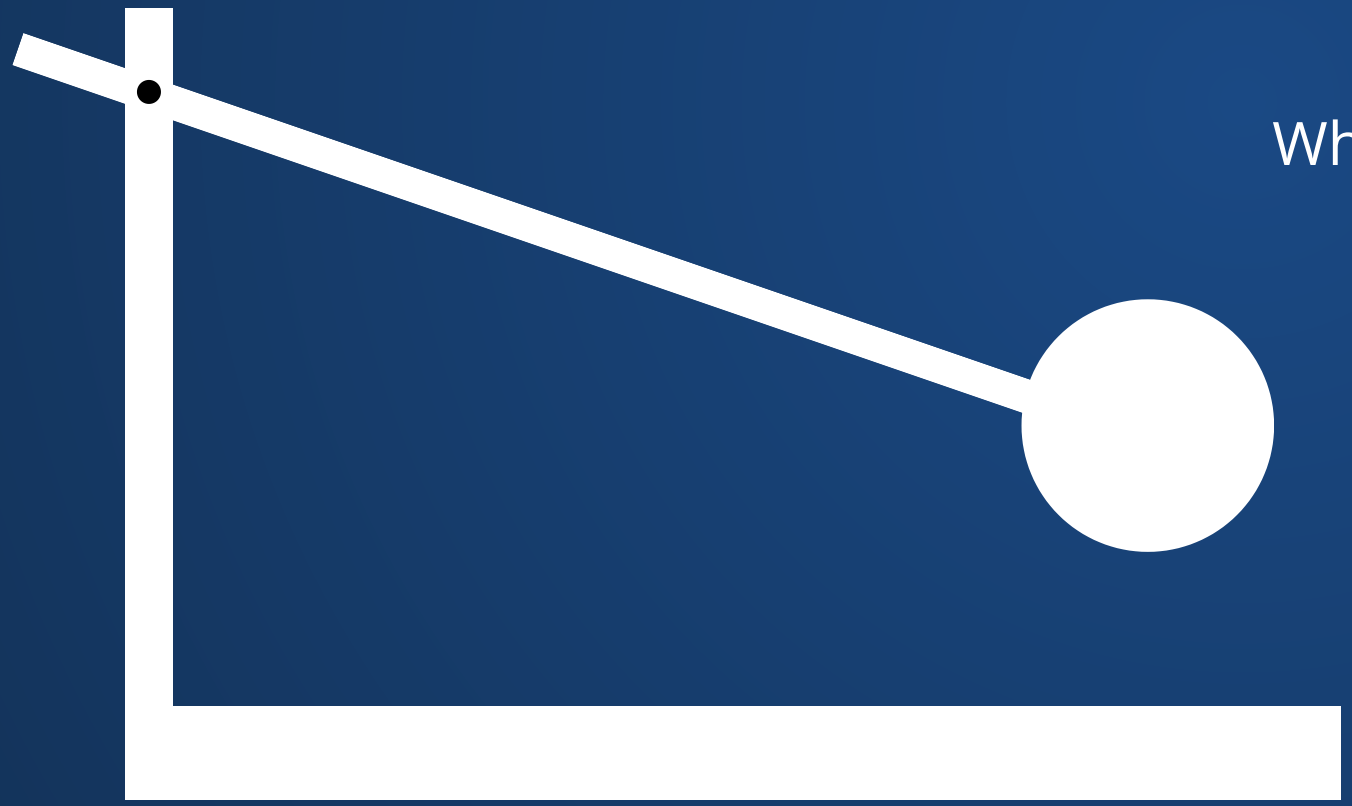
Option 3: Apply a force in the normal direction of the collision until the collision would no longer take place on the next timestep.

What effect does this force have on the next few timesteps of simulation?



There are several strategies when a collision is detected

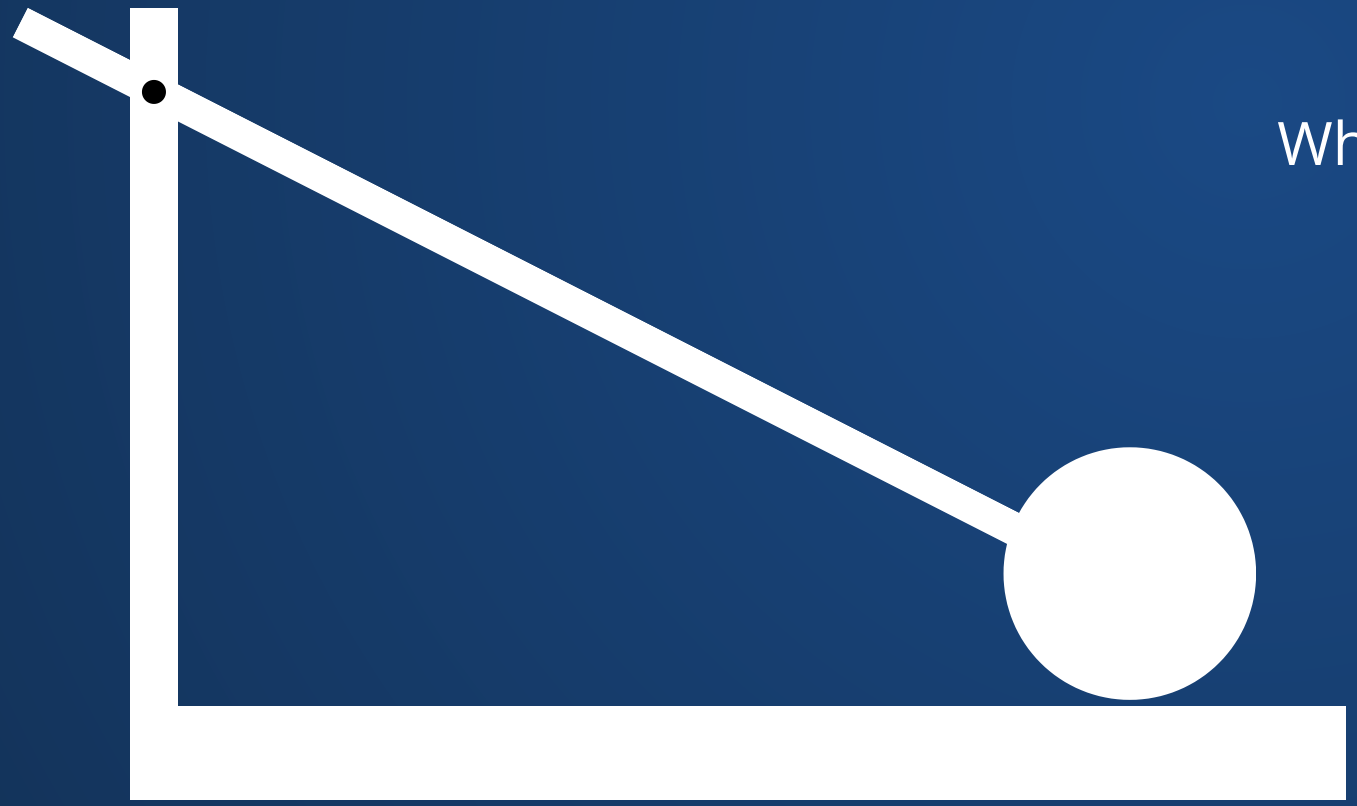
Option 3: Apply a force in the normal direction of the collision until the collision would no longer take place on the next timestep.



What effect does this force have on the next few timesteps of simulation?

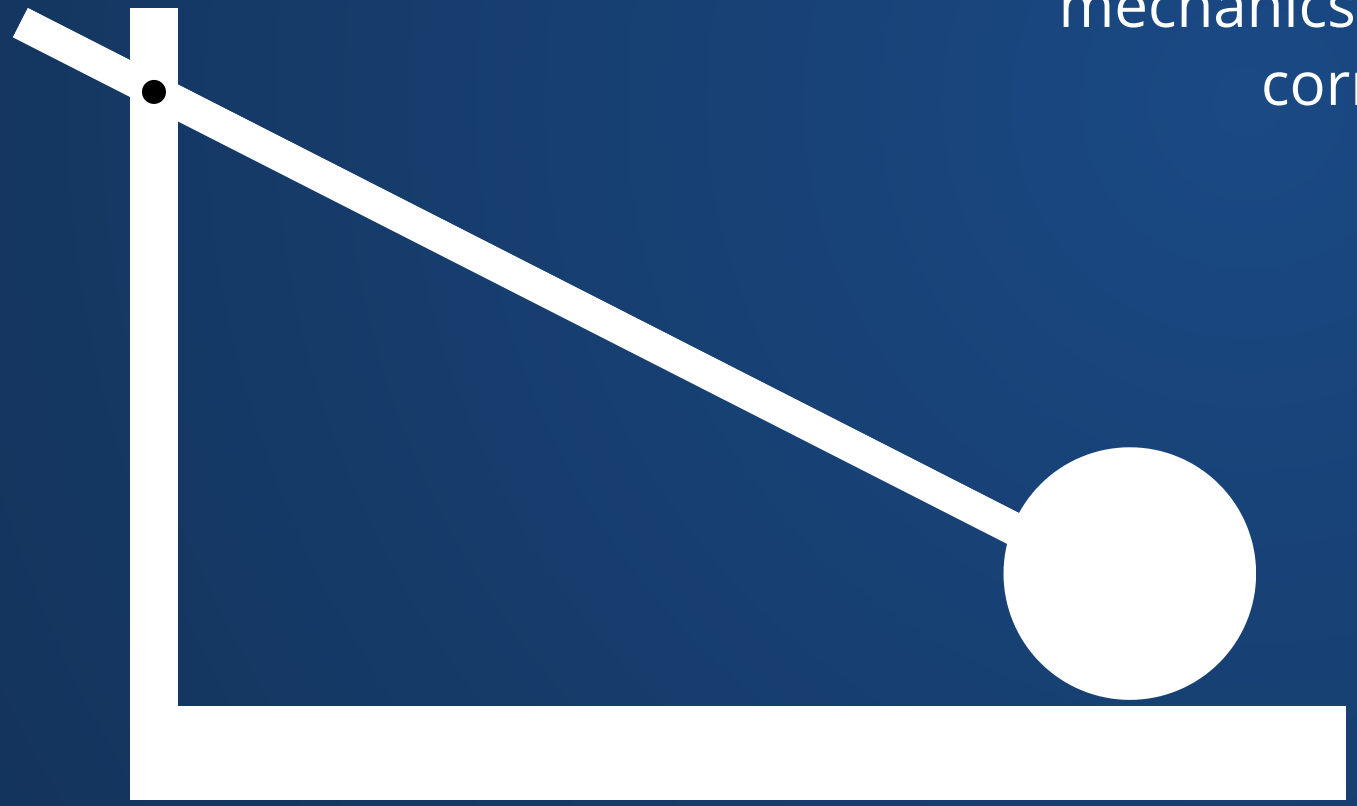
There are several strategies when a collision is detected

Option 3: Apply a force in the normal direction of the collision until the collision would no longer take place on the next timestep.



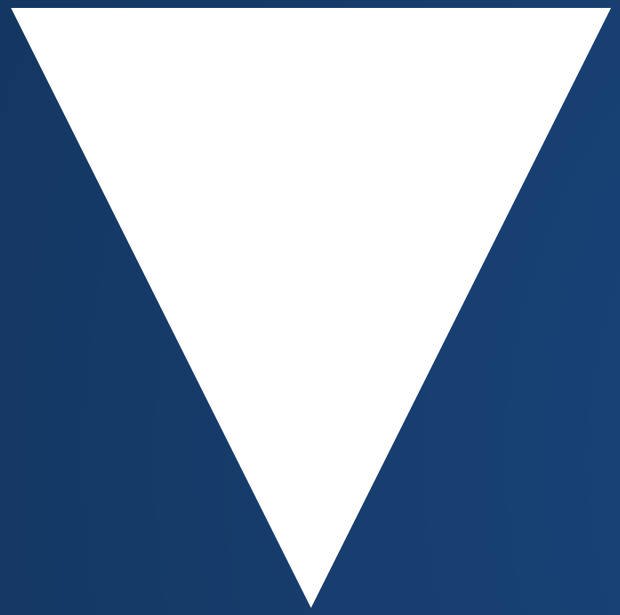
What effect does this force have on the next few timesteps of simulation?

There are several strategies when a collision is detected



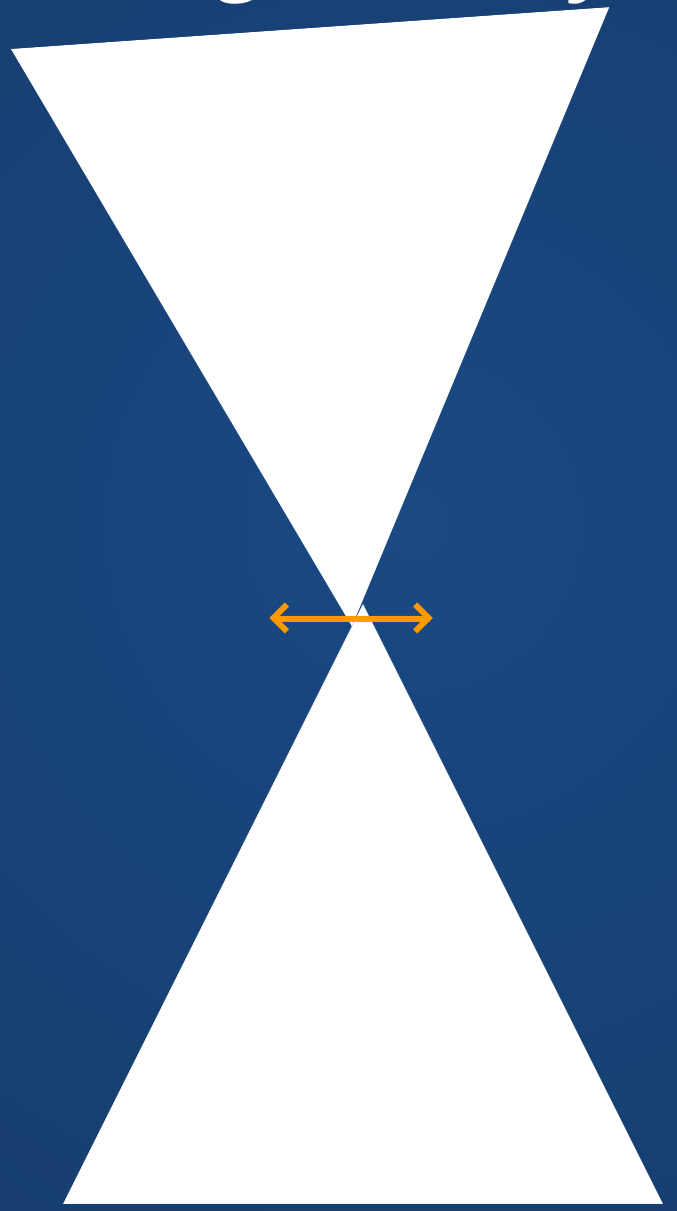
We don't actually have to try larger and larger forces until we find one that works--the language of *impulses* in classical mechanics allows us to compute the correct force to apply.

Impulse-based collision resolution usually has the most generality



Impulse-based collision resolution usually has the most generality

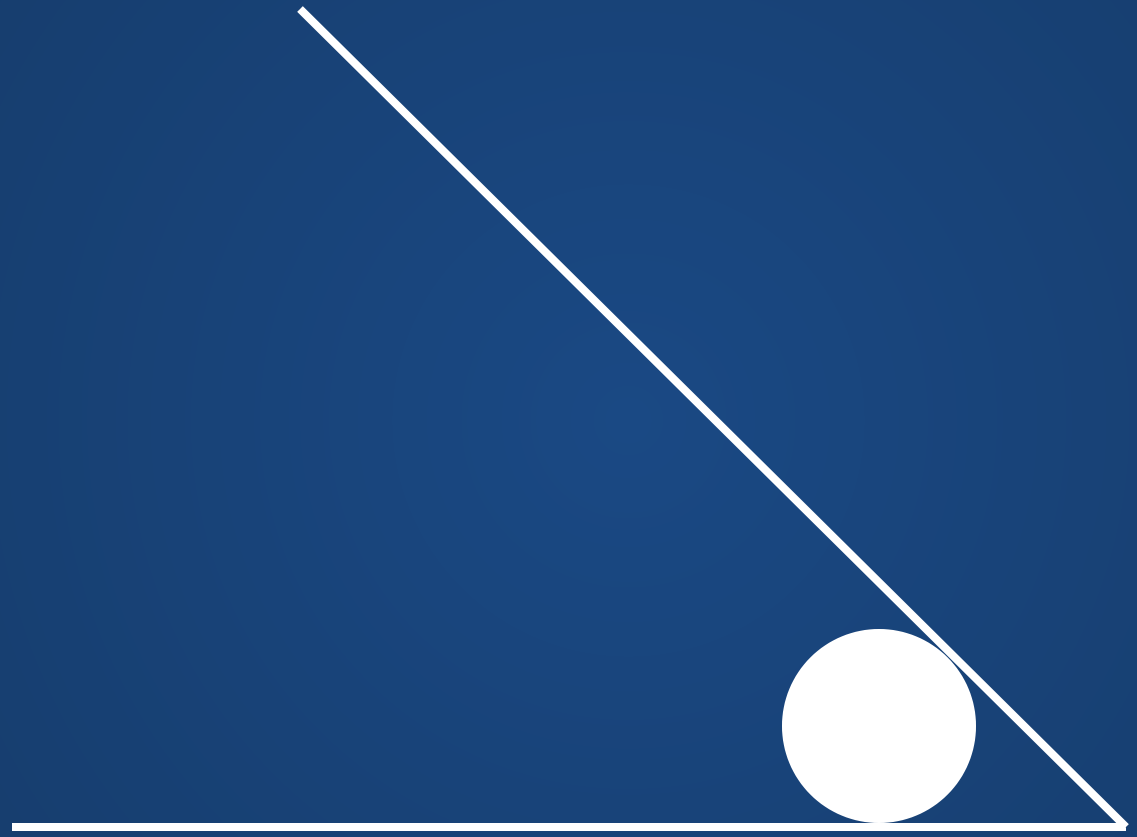
Impulse-based collision resolution usually has the most generality



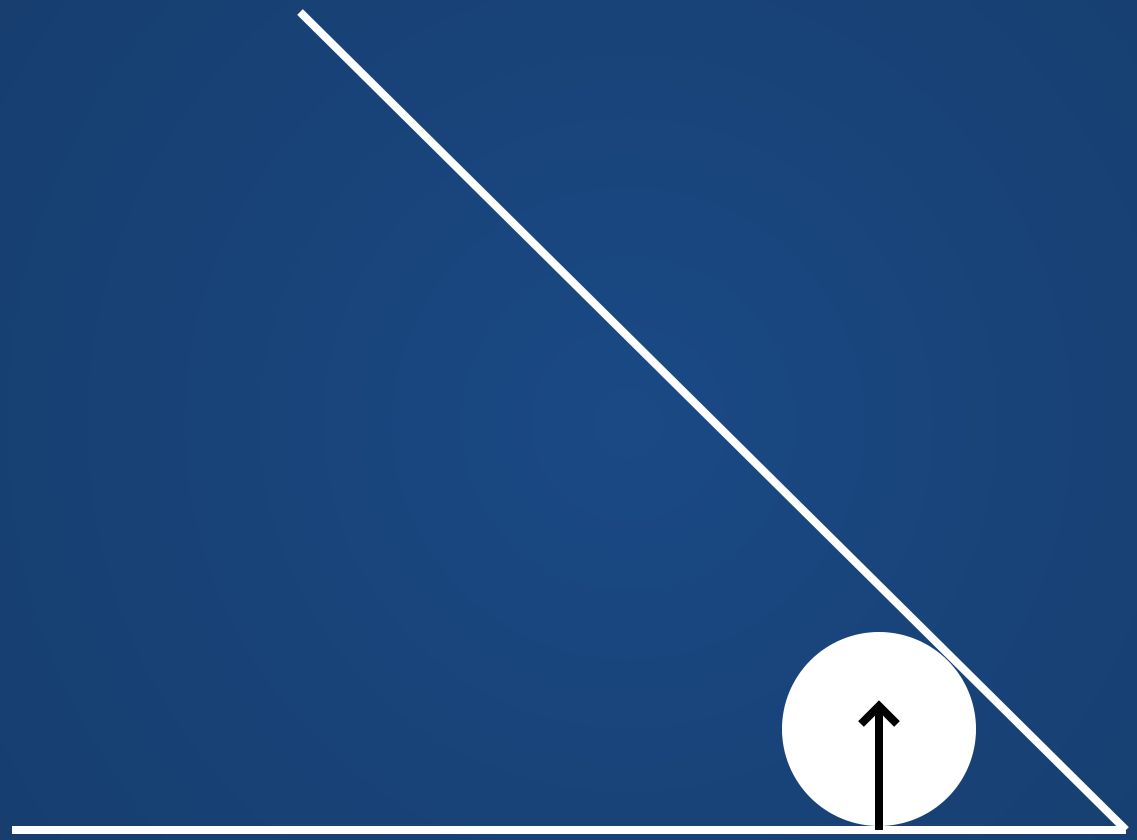
Impulse-based collision resolution usually has the most generality

But this has its own issues!

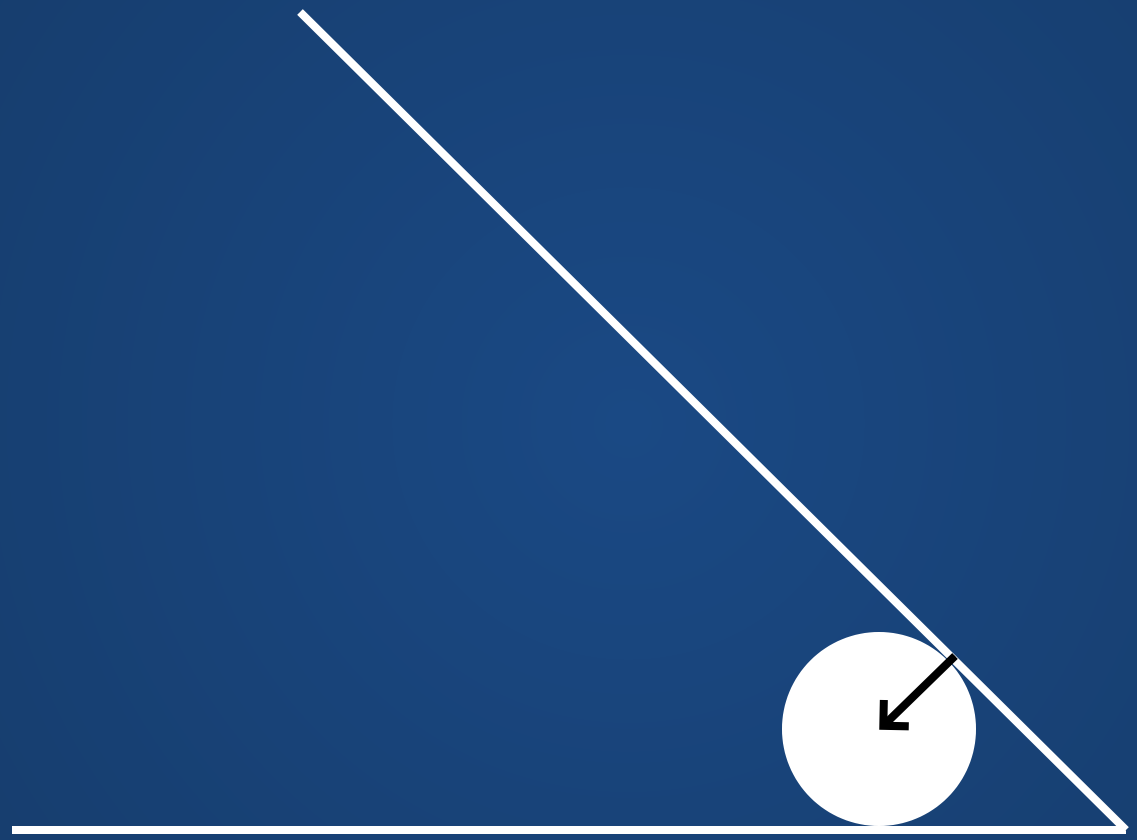
But this has its own issues!



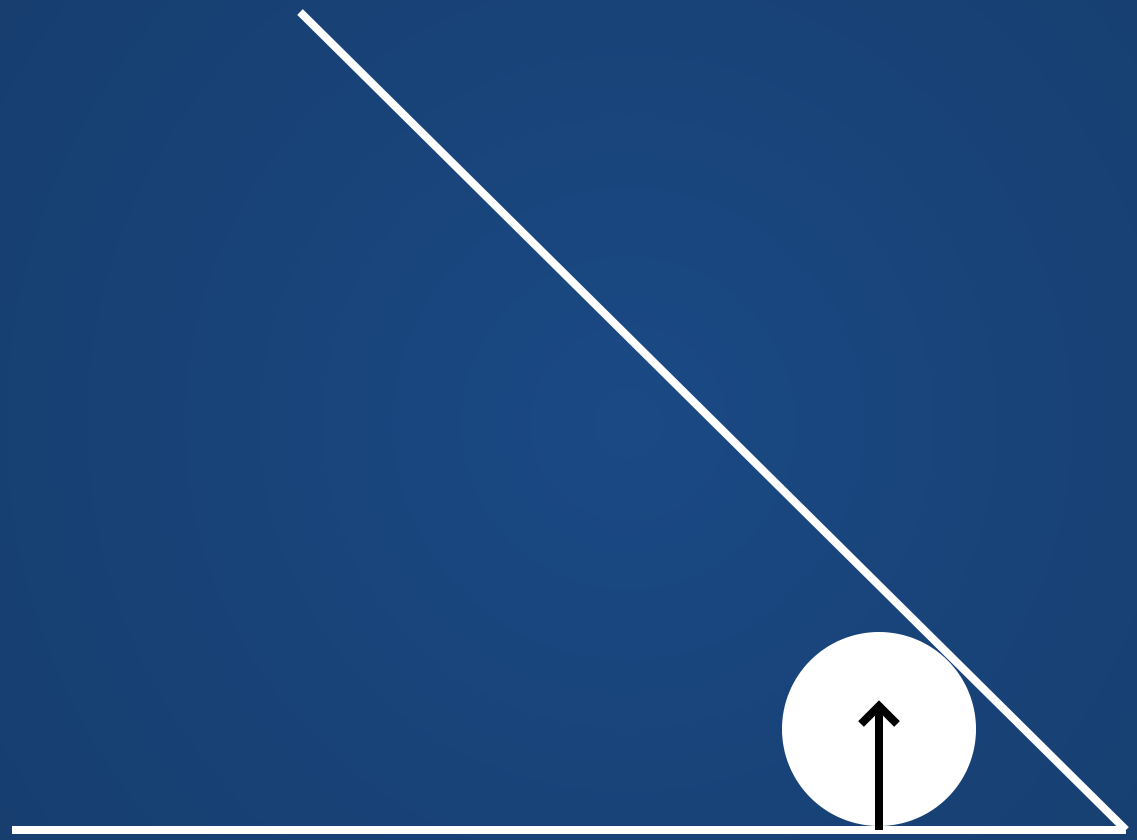
But this has its own issues!



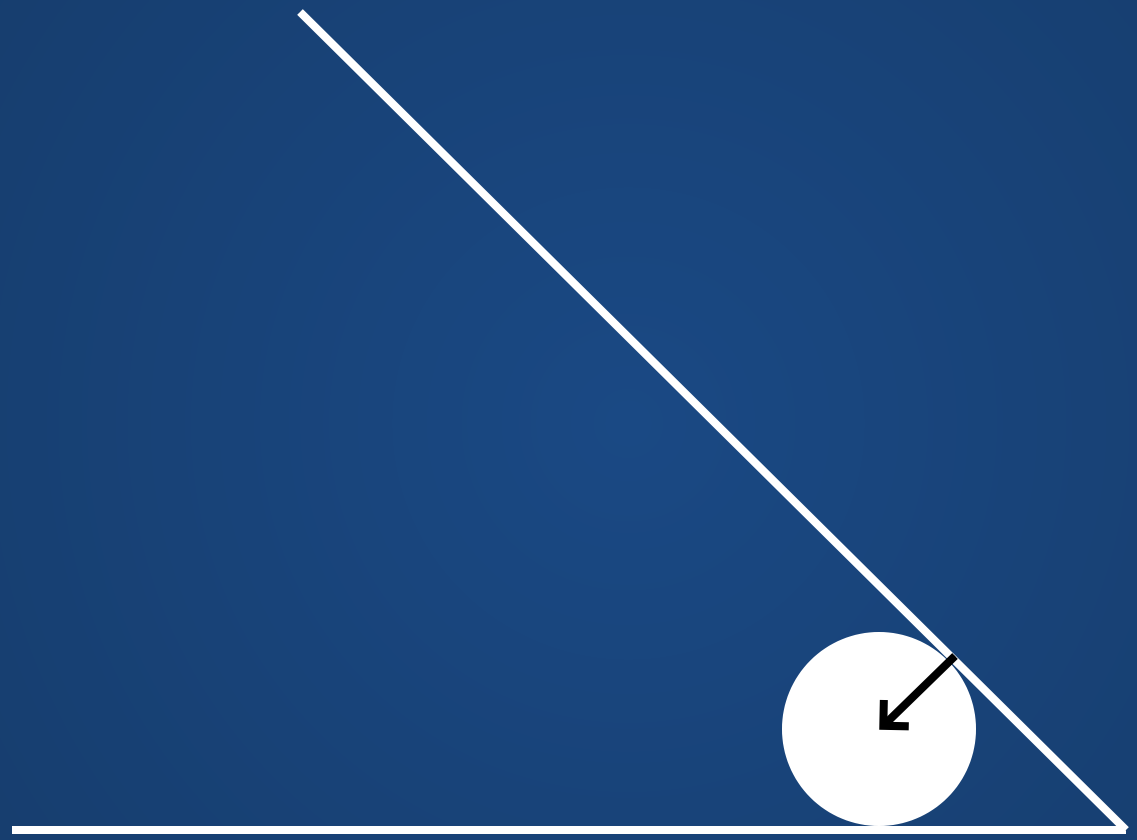
But this has its own issues!



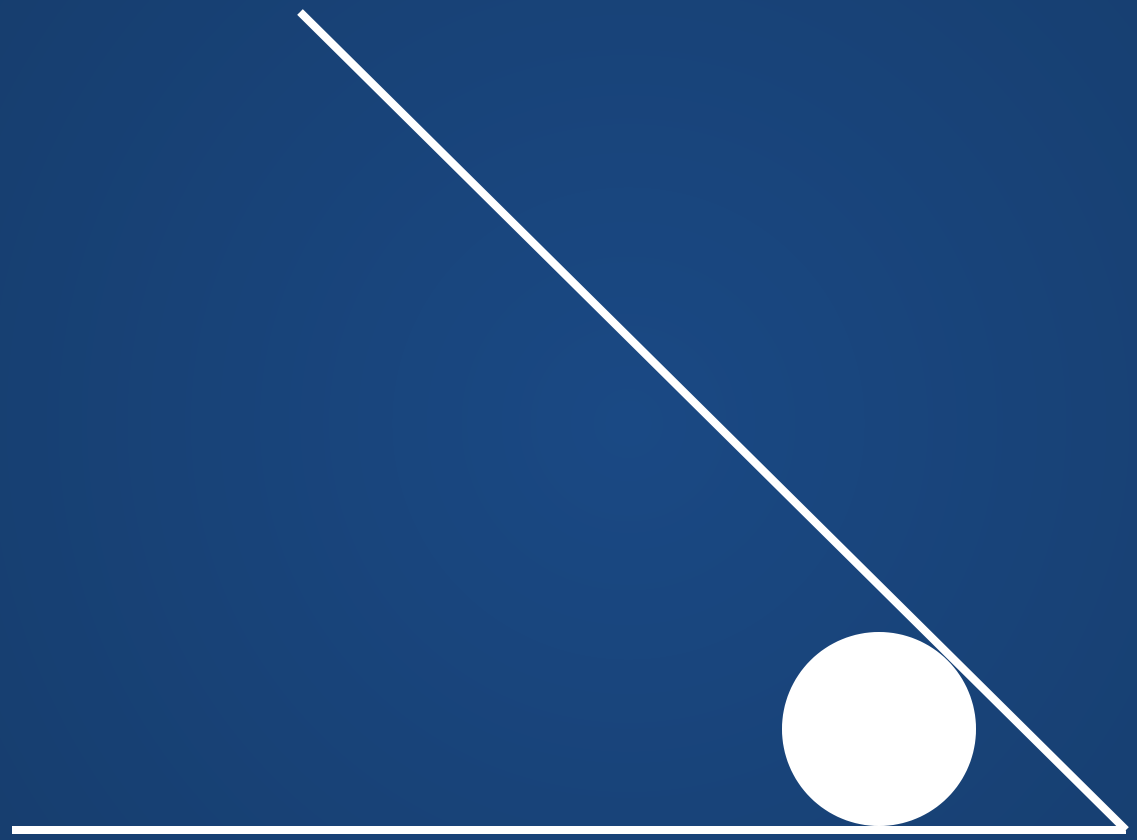
But this has its own issues!



But this has its own issues!



But this has its own issues!

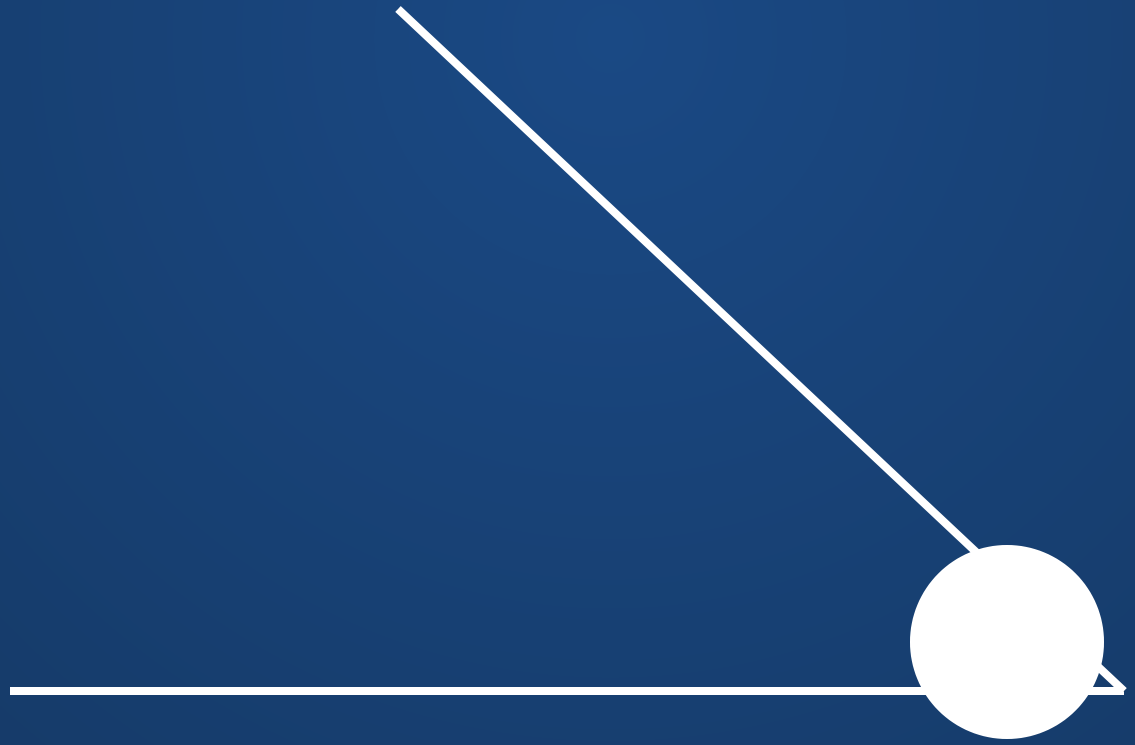


We get stuck in an infinite loop of applying impulses!

But this has its own issues!

State of the art: if we spend too long trying to resolve collisions, give up.

Allow the collision to go unresolved: will look ugly and break the system, but (probably?) better than a guaranteed lockup of the system.



Other Ideas

There are *tons* of other ideas and fields in collision detection. For example, we have not discussed:

- The Separating Axis Theorem (SAT) tests
- Continuous-Time Collision Detection (CTCD)
- Ray-Geometry intersection
  - Ray Tracing
  - Hitscan Detection
- Swept Volumes
- Signed Distance Fields

This field is very deep and rich. What we've seen today barely scratches the paint!

Hands-On: Collisions

1. Generate two shapes using either `beginShape()`/`endShape()` or by loading shapes from an SVG file.
2. Send the shapes toward each other using an explicit Euler-based simulation.
3. Implement a broad-phase collision detection using AABBs for your shapes.
4. When the shapes collide, apply a force to push them apart, using the axis-aligned vector of contact between the AABBs. That is, push only perpendicular to the face of the box.
5. OPTIONAL: Add a narrow-phase check + resolution. This will only be possible if you did **not** load shapes from an SVG.

Note: this assignment is worth twice the typical Hands-On activities, since it replaced two Hands-Ons from the sprites class.