Bounding Volumes & Culling

CS 385 - Class 12
3 March 2022

Bounding Volumes

Bounding Volumes

- A simple shape that entirely encloses an object
- Used in place of the (potentially) complex geometry of an object
- Usage scenarios:
  - extent computations
  - object collisions
  - culling
  - ray tracing

Bounding Sphere

- Enclose the entire object in a sphere
- Required parameters: (four values)
  - (x, y, z) coordinates of center of volume
  - radius (or diameter) of the sphere
- Advantages
  - simple representation / easy to generate
  - object orientation independent
  - easy intersection tests
- Disadvantages
  - update volume if object changes size or position
  - potentially inflated volume
    - increased false positives

Axis-Aligned Bounding Box (AABB)

- Enclose the entire object in box with sides aligned to coordinate axes
- Required parameters: (six values)
  - (x, y, z) coordinates of center of volume
  - (x, y, z) extents in coordinate axes directions
    - alternatively, opposite corners of the box
- Advantages
  - simple representation / simplest to generate
  - simplest intersection tests
- Disadvantages
  - update volume if object changes size, position, or orientation
  - potentially inflated volume

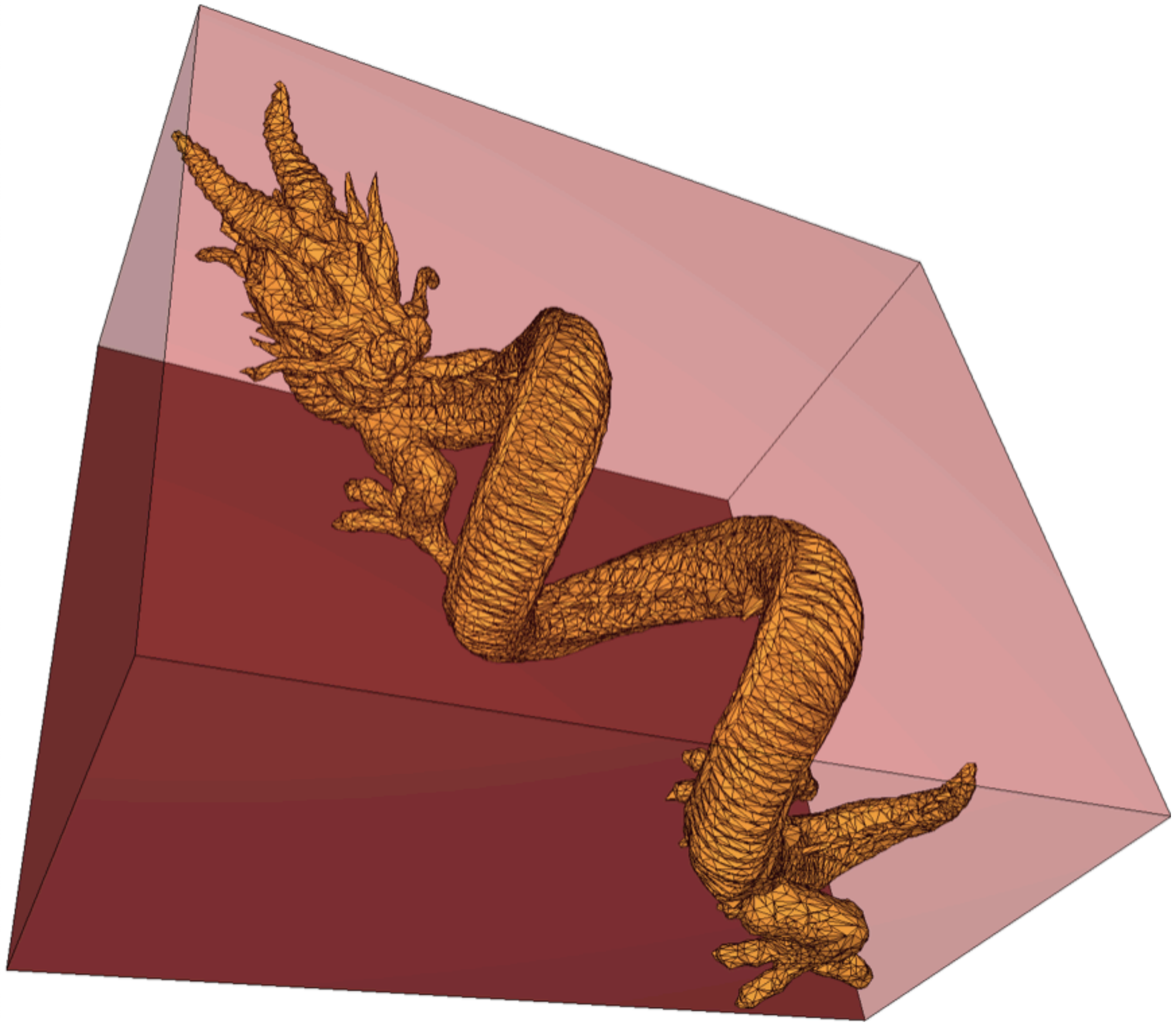


Image from CGAL documentation

Object-Oriented Bounding Box (OOBB)

- Enclose the entire object in box with sides aligned to object's axes
- Required parameters: (twelve values)
  - (x, y, z) coordinates of center of volume
  - (x, y, z) vectors in principal directions of object
- Advantages
  - minimizes enclosing volume
    - fewest false positives
- Disadvantages
  - more challenging to compute
  - more complex intersections
  - update volume if object changes size, position, or orientation

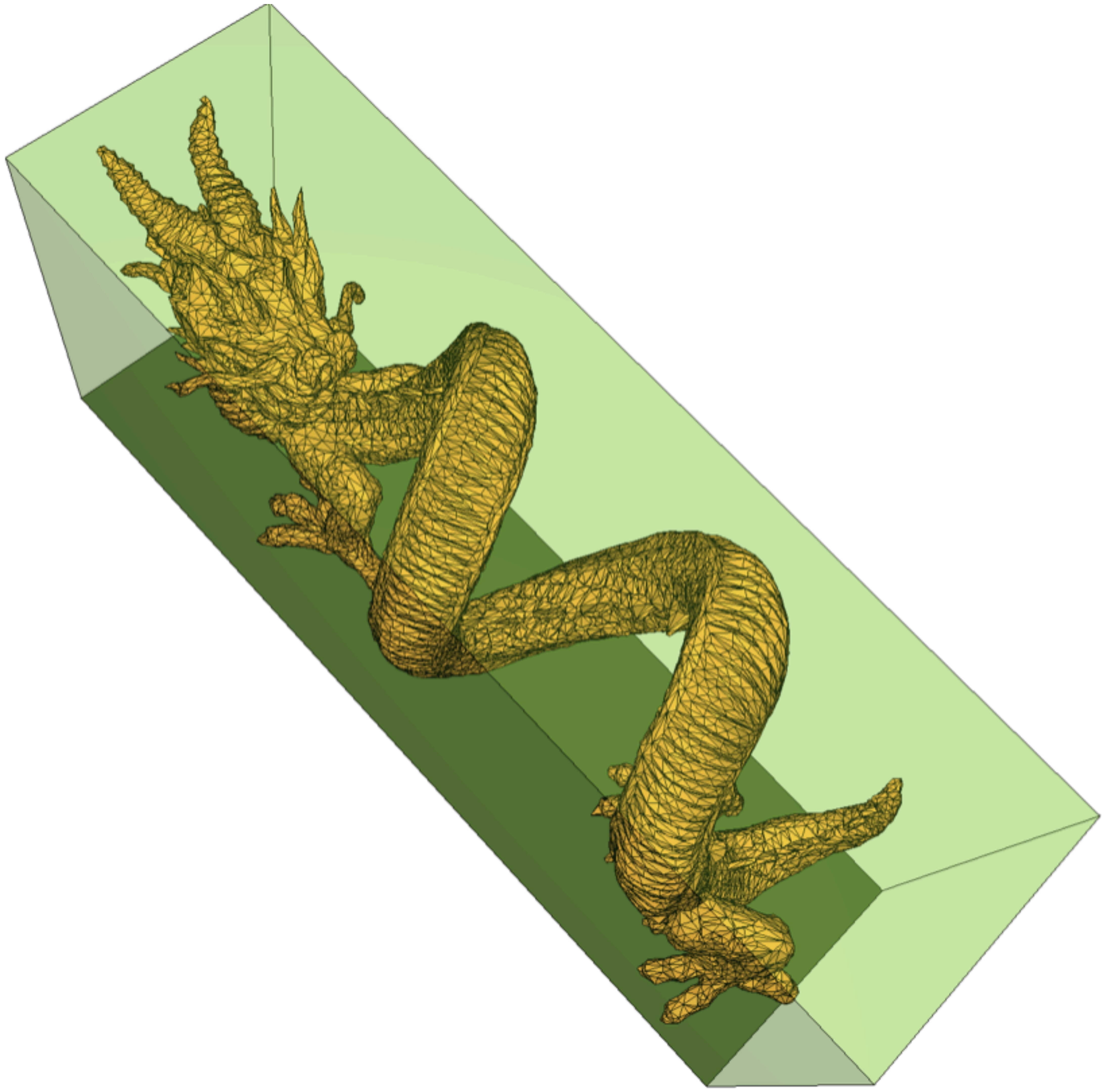


Image from CGAL documentation

AABB Generation

- Simply determine the minimum and maximum values for each dimension
- return as the coordinates of the corners of the volume

```
positions = [ [x, y, z], [x, y, z], ... ];

function GenerateAABB(p) {
  var min = [...p[0]];
  var max = [...p[0]];

  for (var i = 0; i < p.length; ++i) {
    for (var j = 0; p[i].length; ++j) {
      min[j] = Math.min(min[j], p[i][j]);
      max[j] = Math.max(max[j], p[i][j]);
    }
  }

  return { min: min, min: max };
}
```

Sphere Generation

- Determine the AABB of the object
- Derive the center and radius from AABB extents

```
positions = [ [x, y, z], [x, y, z], ... ];

function GenerateBoundingSphere(p) {
  var corners = GenerateAABB(p);
  var extents = [];
  var dist = 0.0;
  for (var i = 0; i < corners[i].length; ++i) {
    extents.push(corners.min[i]- corners.max[i]);
    dist += extents[i] * extents[i];
    extents[i] /= 2.0;
  }
  dist = Math.sqrt(dist);

  return { center: extents, radius: list };
}
```

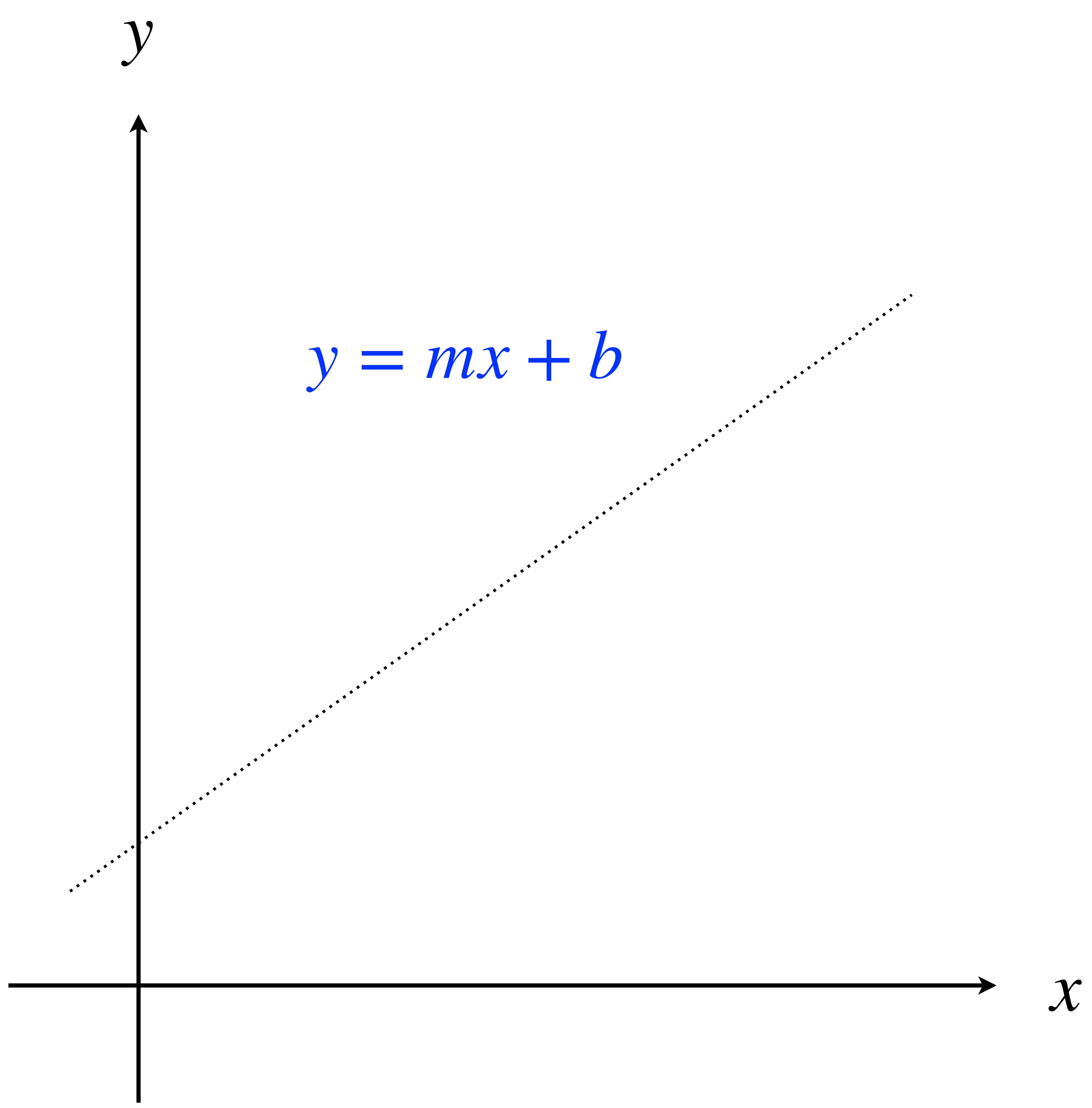
OOBB Generation

- Determine the principal axes of the object
- Compute the center

But First, Some Math

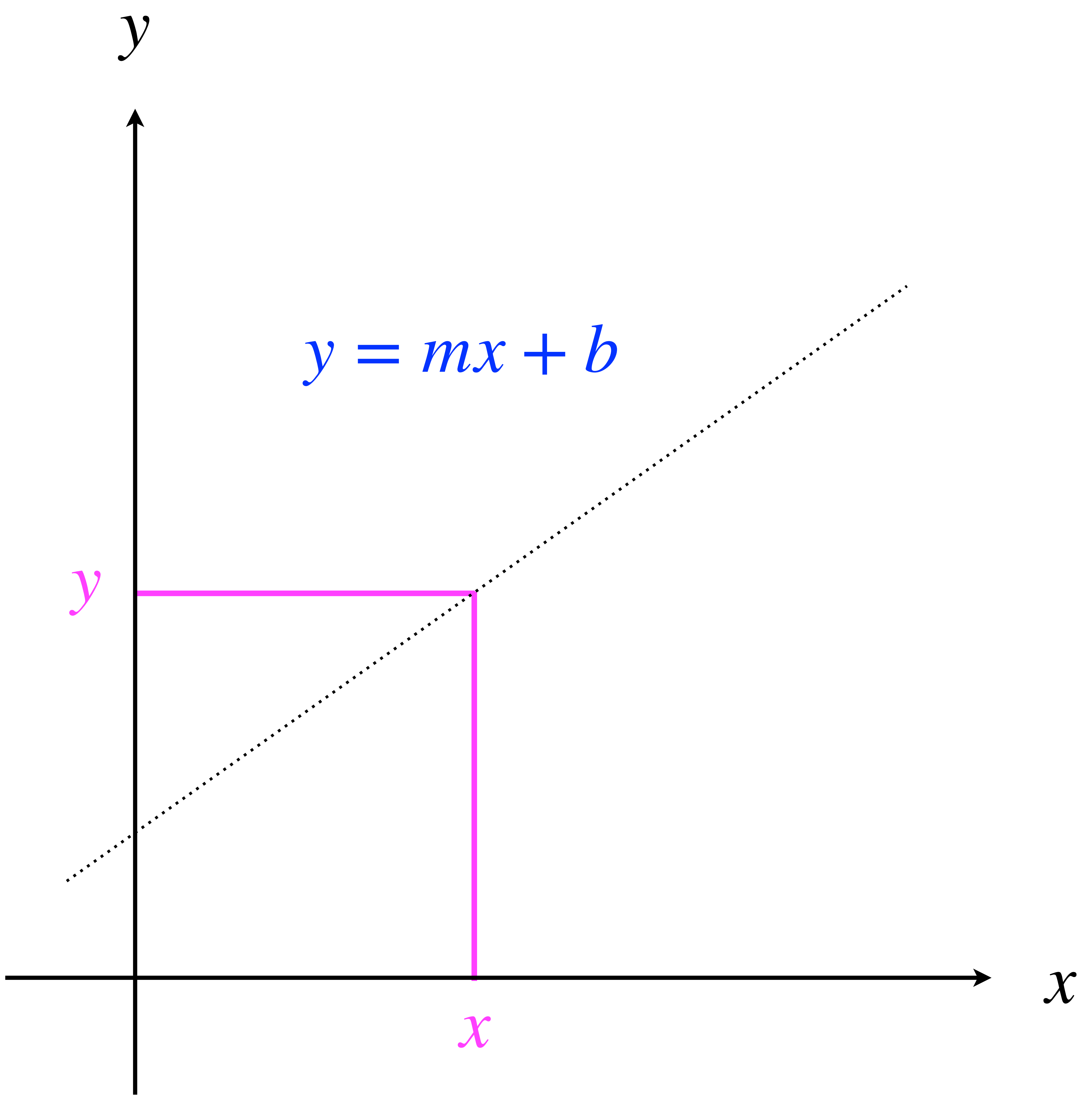
Remember Our Friend, the Line

- This form of a line is useful if you know x or y, and want to find the other value



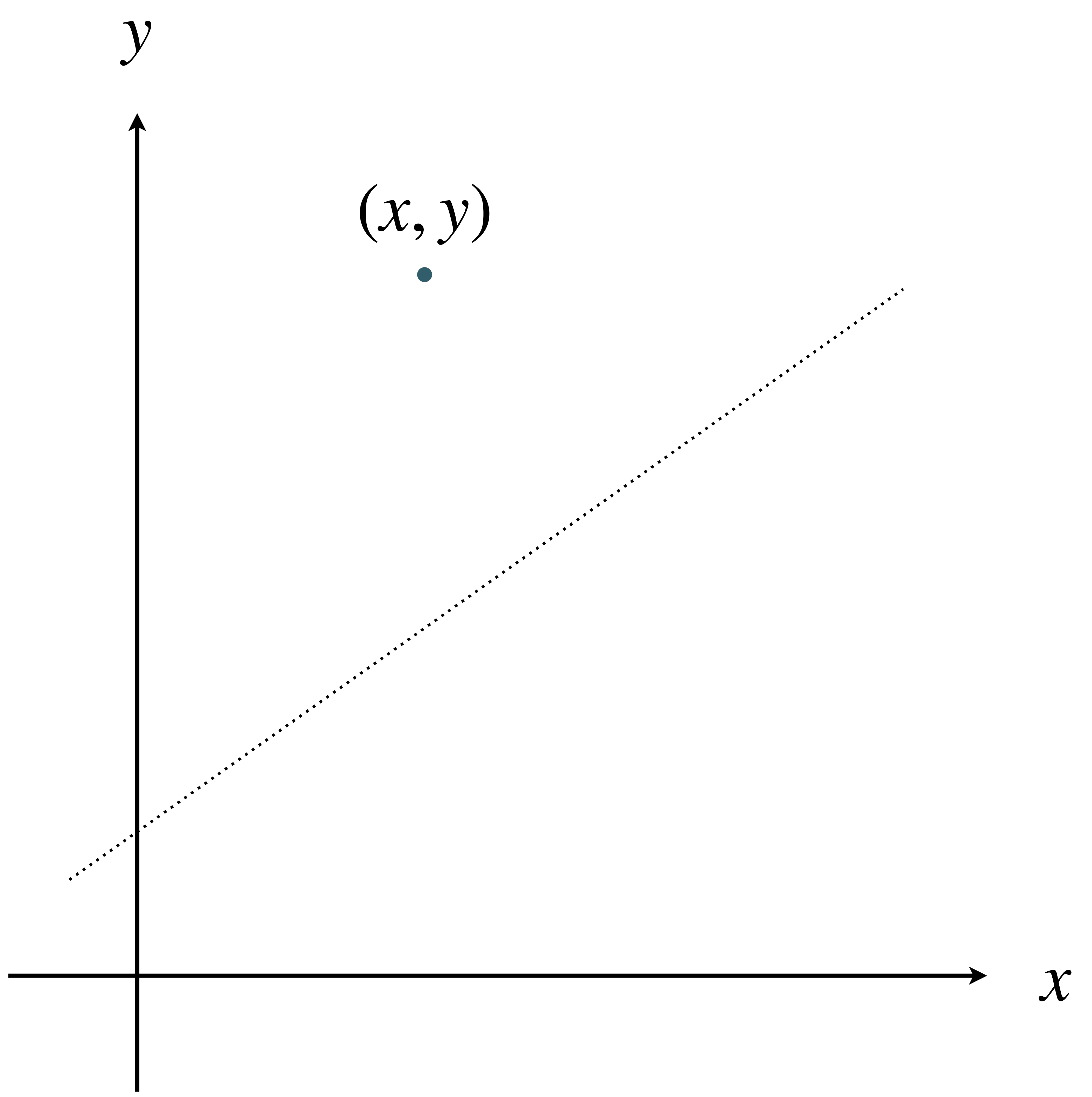
Remember Our Friend, the Line

- This form of a line is useful if you know x or y, and want to find the other value



A Different Problem

- Suppose we have a point (x, y) and want to know its relation to the line
- For example, suppose we want to know *which side* of the line (x, y) is
- What does that even mean?



A Little Algebra

$$mx + b = y$$

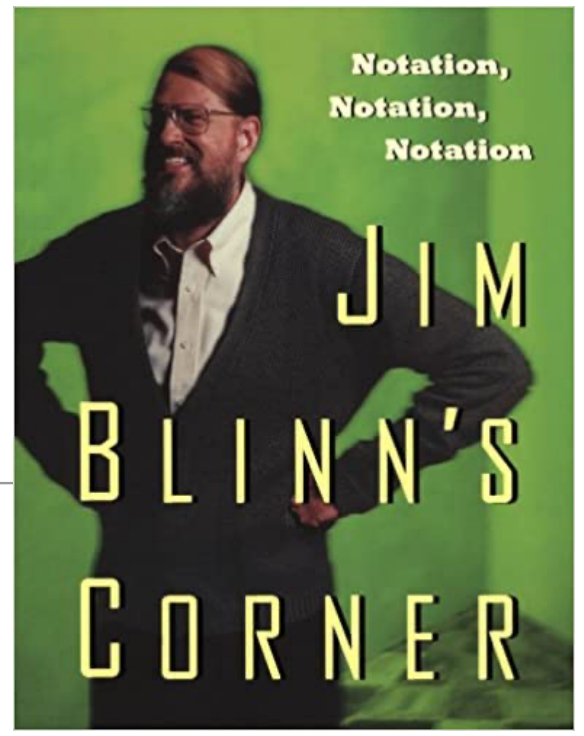
$$x + \frac{b}{m} = \frac{y}{m}$$

$$x - \frac{y}{m} + \frac{b}{m} = 0$$

$$(x, y) \cdot (1, -\frac{1}{m}) + D = 0 \quad (\text{define } D = \frac{b}{m})$$

And this helps how?

Notation, Notation, Notation

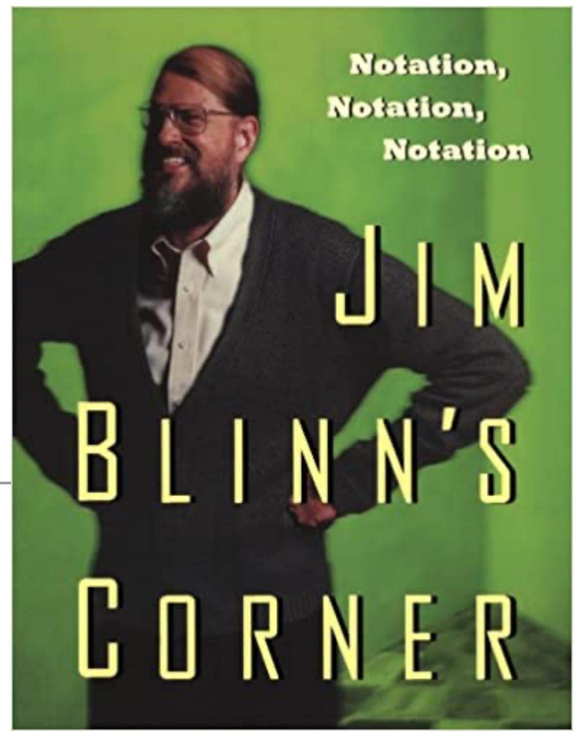


$$(x, y)$$

$$\Downarrow$$

$\vec{p}$ A point named p

Notation, Notation, Notation

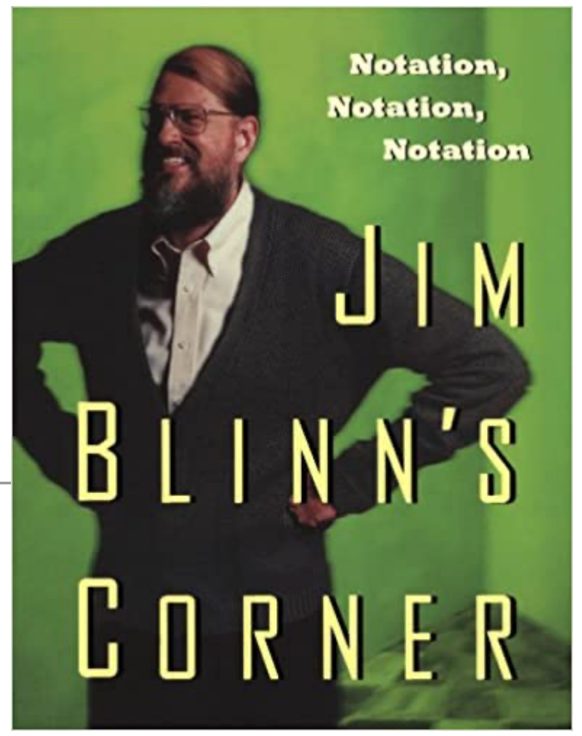


$$(x, y) \qquad \bullet$$

$$\Downarrow \qquad \Downarrow$$

$$\overrightarrow{p} \qquad \textit{Dot Product}$$

Notation, Notation, Notation



$$(x, y) \quad \bullet \quad \left(1, -\frac{1}{m}\right)$$

$$\Downarrow \qquad \Downarrow \qquad \Downarrow$$

$$\vec{p} \quad \bullet \quad \hat{n}$$

A *normalized* vector named n

Aside: Vector Length

- $||\overrightarrow{v}||$ is the notation for that length

$$\vec{v} = (x, y) \qquad \text{(2D)}$$

$$||\vec{v}|| = \sqrt{x^2 + y^2}$$

$$\vec{v} = (x, y, z) \qquad \text{(3D)}$$

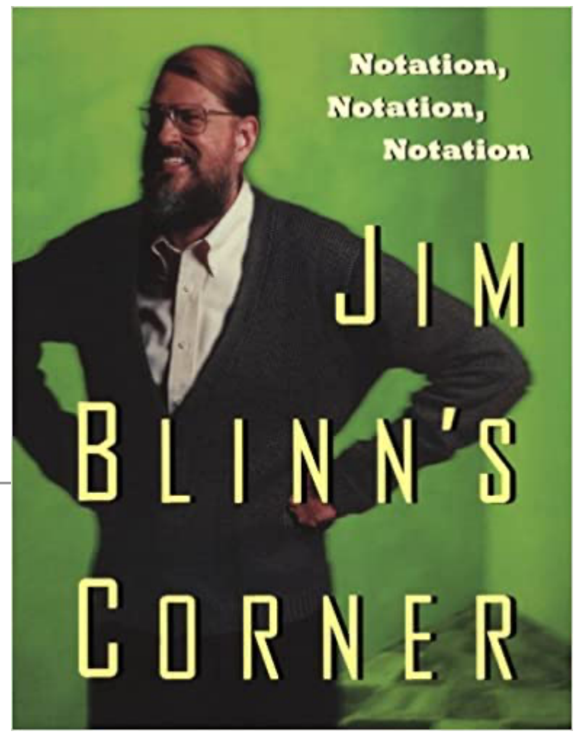
$$||\vec{v}|| = \sqrt{x^2 + y^2 + z^2}$$

Normal Vectors

- Just means the vector's *length* is equal to one
  - often called *unit vectors*
- Put a *hat* on our vector to indicate it's a unit vector: $\hat{v}$
- We'll use normal vectors a lot:
  - GLSL has a built-in `normalize()` function
  - MV.js defines the function as well

$$\hat{v} = \frac{\vec{v}}{||\vec{v}||}$$

Notation, Notation, Notation

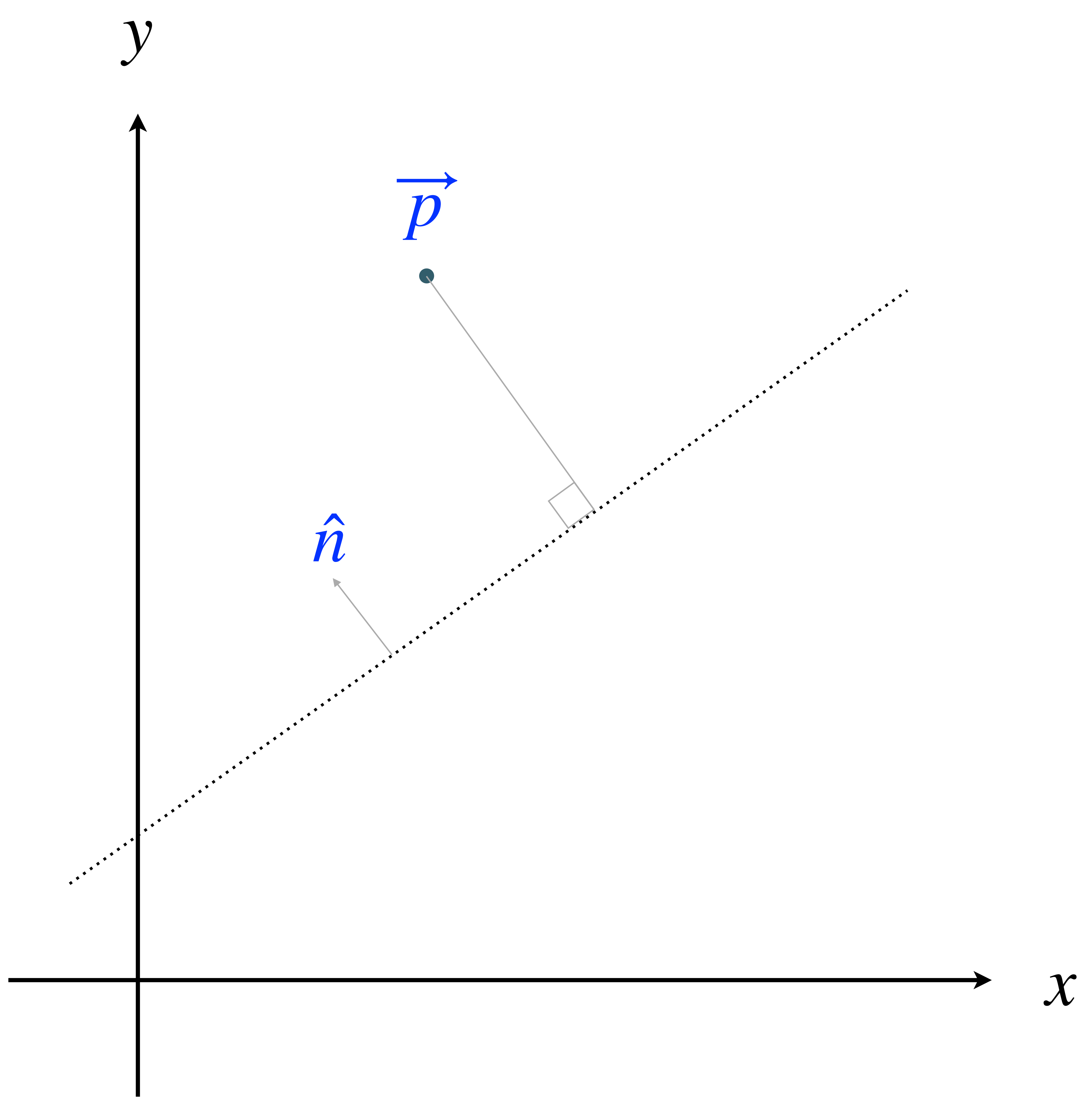


$$
\begin{array}{ccccccc}
(x, y) & \bullet & \left(1, -\dfrac{1}{m}\right) & + & D & = & 0 \\
\Downarrow & \Downarrow & \Downarrow & & \Downarrow & & \\
\vec{p} & \bullet & \hat{n} & + & D & = & 0
\end{array}
$$

yippee. So what?

A Different Line Equation

- As compared to $y = mx + b$, which tells you y for a specific x
- $\overrightarrow{p} \cdot \hat{n} + D$ tells you the distance $\overrightarrow{p}$ is from the line
  - zero indicates $\overrightarrow{p}$ lies on the line
  - any other value tells you two things:
    - the distance from the line
    - the sign of the value tells you which "side" of the line $\overrightarrow{p}$ is on
- $\overrightarrow{p} \cdot \hat{n} + D = 0$ is called the *implicit form* of a line

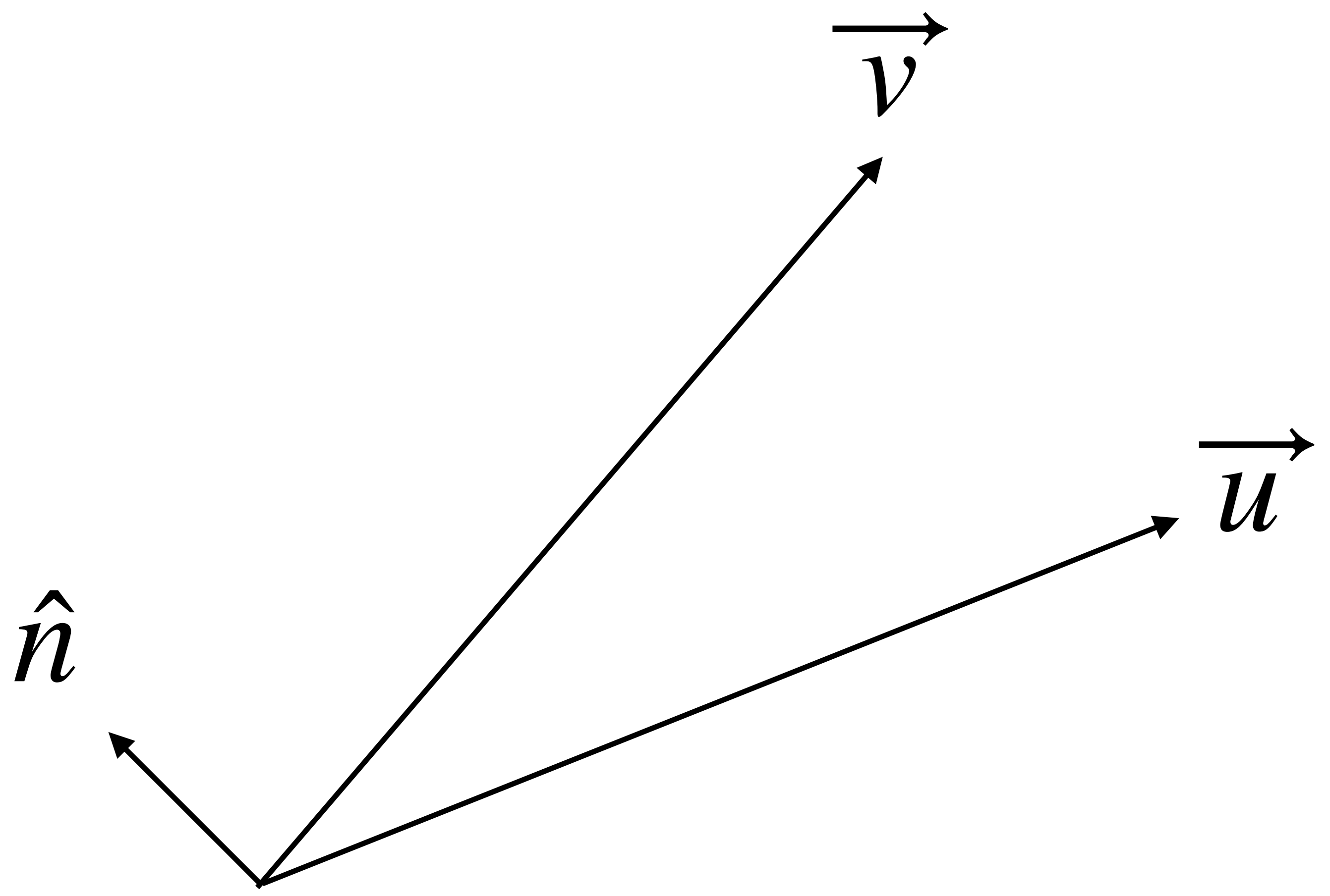


What's Really Cool About This!

- $\overrightarrow{p} \cdot \hat{n} + D = 0$ holds true for problems in any dimension!
- We'll use this technique most often in 3D
  - and in particular for our next topic, *culling*
    - we'll test points against our viewing frustum

Computing $\hat{n}$ and D

- $\hat{n}$ for 3D can often be computed simply (no, really)
  - If you know two vectors in the "plane", we can use the *cross product* to get the perpendicular vector
    - GLSL and **MV.js** have a `cross()` function
  - If you know a point $\overrightarrow{p}$ on the plane, then $D = \overrightarrow{p} \cdot \hat{n}$
    - If the plane passes through the origin, then $D = 0$
    - if the plane is perpendicular to coordinate axis, then D is $\overrightarrow{p}$'s coordinate in that dimension
  - See Appendix for these slides for computing these values for a line
    - since most of our work is in 3D, that's more relevant, but 2D is kinda fun



$$\hat{n} = \frac{\overrightarrow{u} \times \overrightarrow{v}}{||\overrightarrow{u} \times \overrightarrow{v}||}$$

Culling

Culling

- Drawing stuff you can't see is wasteful!
  - recall that geometry outside of the viewing frustum will be *clipped out* and not rasterized
    - for primitives entirely outside, no fragments are generated
  - However, your program is still
    - sending vertices to GPU through WebGL
    - executing the vertex shader for each of those vertices
    - having the rasterizer determine if any fragments should be generated