UNIVERSITY OF MINNESOTA
Driven to Discover®

Essential Graphics Math

CSCI 4611: Programming Interactive Computer Graphics and Games

Evan Suma Rosenberg | CSCI 4611 | Fall 2022

Review: Vectors

$$\underset{\text{point}}{(2,0)} - \underset{\text{point}}{(5,0)} = \underset{\text{vector}}{(-3,0)}$$

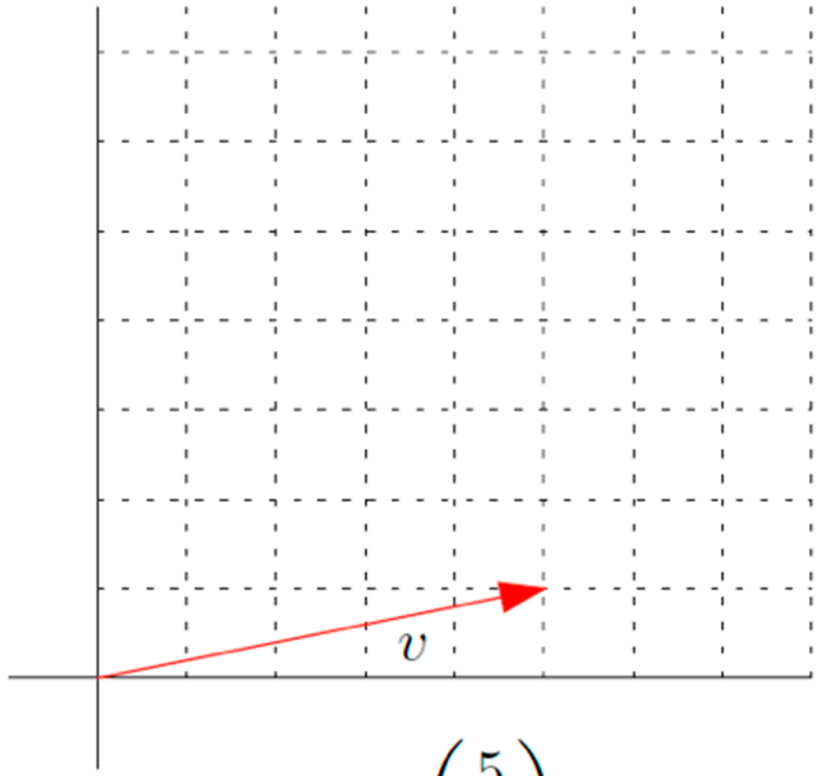
The difference between (2,0) and (5,0) is the direction and distance to travel to get to (2,0) from the starting point of (5,0).

Vector Length (Magnitude)

A vector has a **length**, denoted $\|\boldsymbol{v}\|$

$$\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_d^2}$$

```
const v = new gfx.Vector2(5, 1);
console.log(v.length());

>> 5.09901951359278 45
```



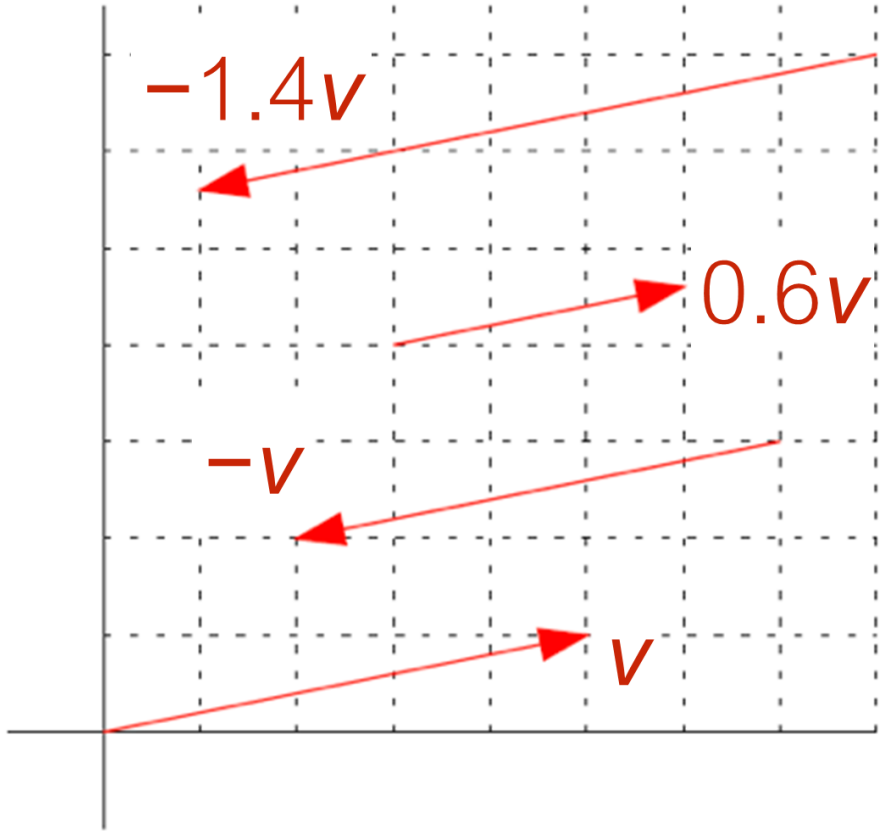
$$v = \begin{pmatrix} 5 \\ 1 \end{pmatrix}$$

Vector Normalization

A vector v is a **unit vector** if $||\boldsymbol{v}|| = 1$.

Normalizing creates a unit vector by dividing every component by the length.

```
const v = new gfx.Vector2(5, 1);
v.normalize();
```



Vector Addition

```
const v1 = new gfx.Vector2(1, 5);
const v2 = new gfx.Vector2(1, 4);

const sum = v1 + v2;

>> TS2365: Operator '+' cannot be applied to types...
```

Vector Addition

```
const v1 = new gfx.Vector2(1, 5);
const v2 = new gfx.Vector2(1, 4);

// This creates a new vector to hold the sum
const sum = gfx.Vector2.add(v1, v2);

// This changes the value of v1 to the sum
v1.add(v2);
```

Vector Subtraction

```
const v1 = new gfx.Vector2(1, 5);
const v2 = new gfx.Vector2(1, 4);

// This creates a new vector to hold the difference
const sum = gfx.Vector2.subtract(v1, v2);

// This changes the value of v1 to the difference
v1.subtract(v2);
```

Vector Addition/Subtraction vs. translate()

```
ship.rotation += Math.PI / 2;
ship.position.add(new Vector2(0, 1));
```

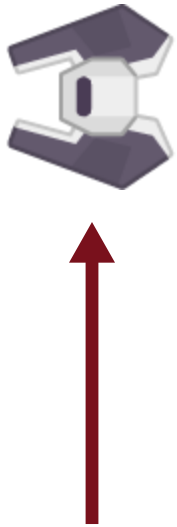
```
ship.rotation += Math.PI / 2;
ship.translate(new Vector2(0, 1));
```

Vector Addition/Subtraction vs. translate()

```
ship.rotation += Math.PI / 2;
ship.position.add(new Vector2(0, 1));
```

```
ship.rotation += Math.PI / 2;
ship.translate(new Vector2(0, 1));
```

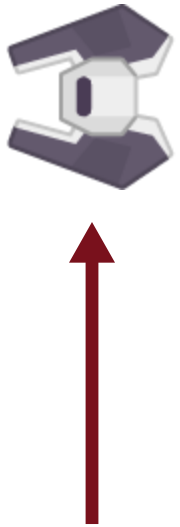
Vector Addition/Subtraction vs. translate()



```
ship.rotation += Math.PI / 2;
ship.position.add(new Vector2(0, 1));
```

```
ship.rotation += Math.PI / 2;
ship.translate(new Vector2(0, 1));
```

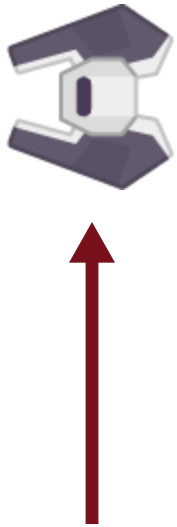
Vector Addition/Subtraction vs. translate()



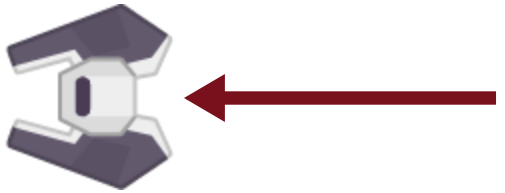
```
ship.rotation += Math.PI / 2;
ship.position.add(new Vector2(0, 1));
```

```
ship.rotation += Math.PI / 2;
ship.translate(new Vector2(0, 1));
```

Vector Addition/Subtraction vs. translate()



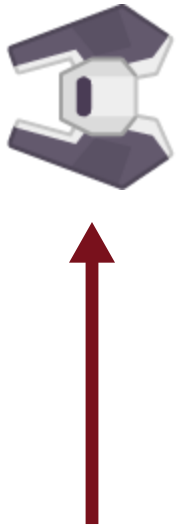
```
ship.rotation += Math.PI / 2;
ship.position.add(new Vector2(0, 1));
```



```
ship.rotation += Math.PI / 2;
ship.translate(new Vector2(0, 1));
```

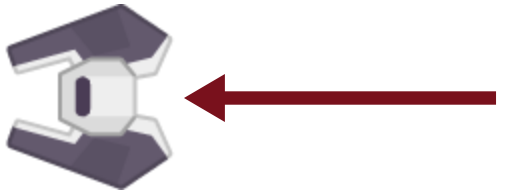
Vector Addition/Subtraction vs. translate()

Movement is **independent** of object's rotation!



```
ship.rotation += Math.PI / 2;
ship.position.add(new Vector2(0, 1));
```

Movement is **relative** to object's rotation!



```
ship.rotation += Math.PI / 2;
ship.translate(new Vector2(0, 1));
```

What does the translate() function do?

```
ship.translate(new Vector2(0,1));
```

What does the translate() function do?

```
ship.translate(new Vector2(0,1));
```

```
// The translate method moves the object in a direction relative to its current orientation
translate(translation: Vector2): void
{




}
```

What does the translate() function do?

```
ship.translate(new Vector2(0,1));
```

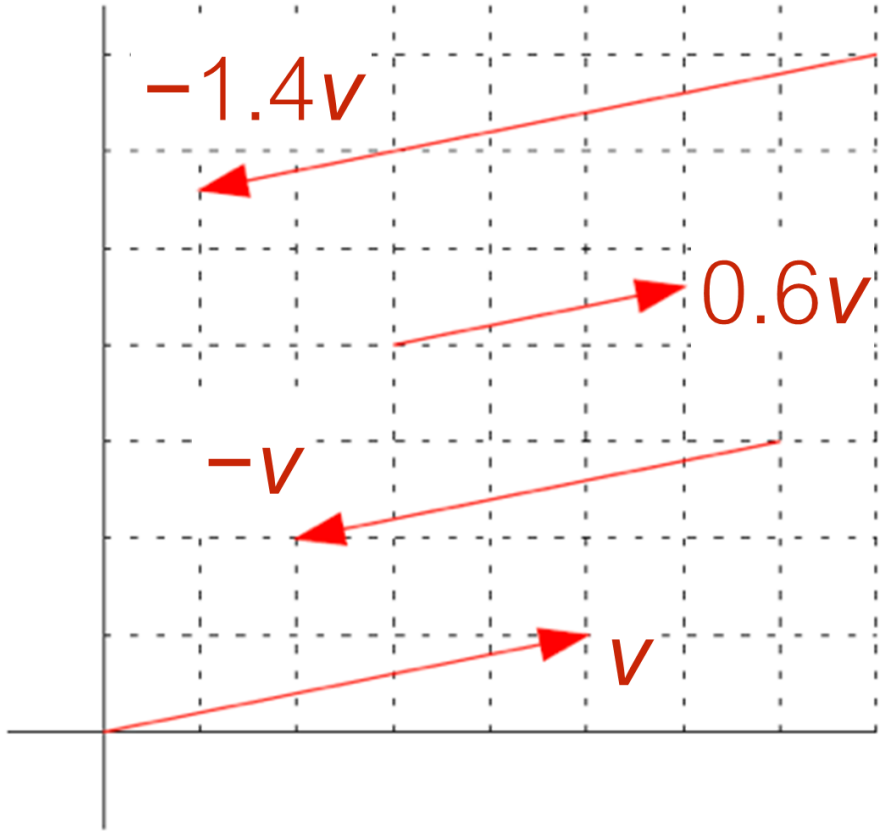
```
// The translate method moves the object in a direction relative to its current orientation
translate(translation: Vector2): void
{
    // Rotate the translation vector by the object's current rotation
    const localVector = Vector2.rotate(translation, this.rotation);

    // Add the rotated vector to the object's current position
    this.position.add(localVector);
}
```

Vector Scalar Multiplication

Multiplying a vector $\boldsymbol{v}$ by a scalar (real number) $\boldsymbol{c}$ gives a new vector,

$$\boldsymbol{cv} = (\boldsymbol{c}v_1, \boldsymbol{c}v_2, \ldots, \boldsymbol{c}v_d)$$

Note that $\boldsymbol{cv}$ has either the same or the opposite direction as $\boldsymbol{v}$.



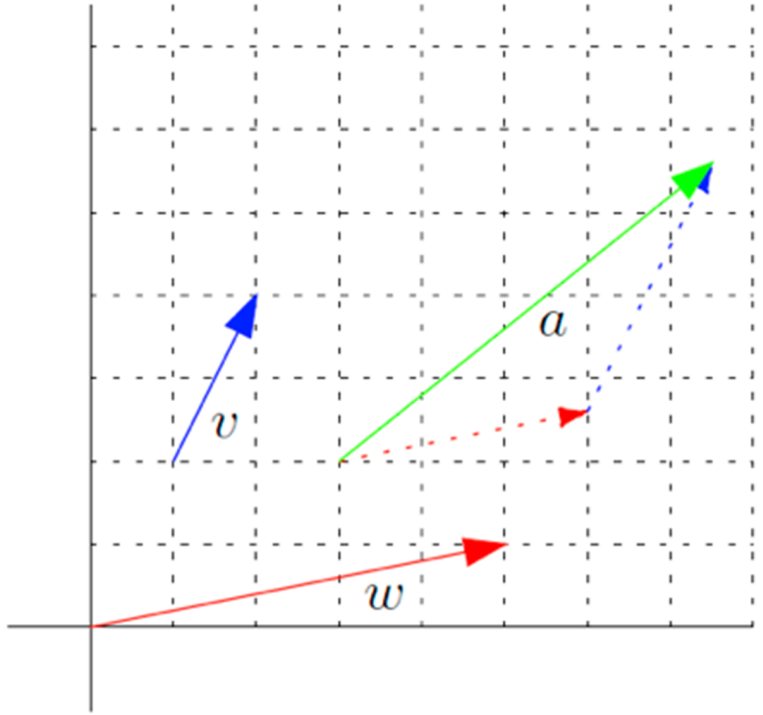
Vector Scalar Multiplication

```
const v = new gfx.Vector2(1, 5);

// This creates a new vector to hold the result
const vResult = gfx.Vector2.multiplyScalar(3);

// This changes the value of v to the result
v.multiplyScalar(3);
```

Linear Combinations

A **linear combination** of vectors is a sum of their scalar multiples.

$c_1\boldsymbol{v} + c_2\boldsymbol{w} + \ldots$



$a = 1.5v + 0.6w$

Coordinates

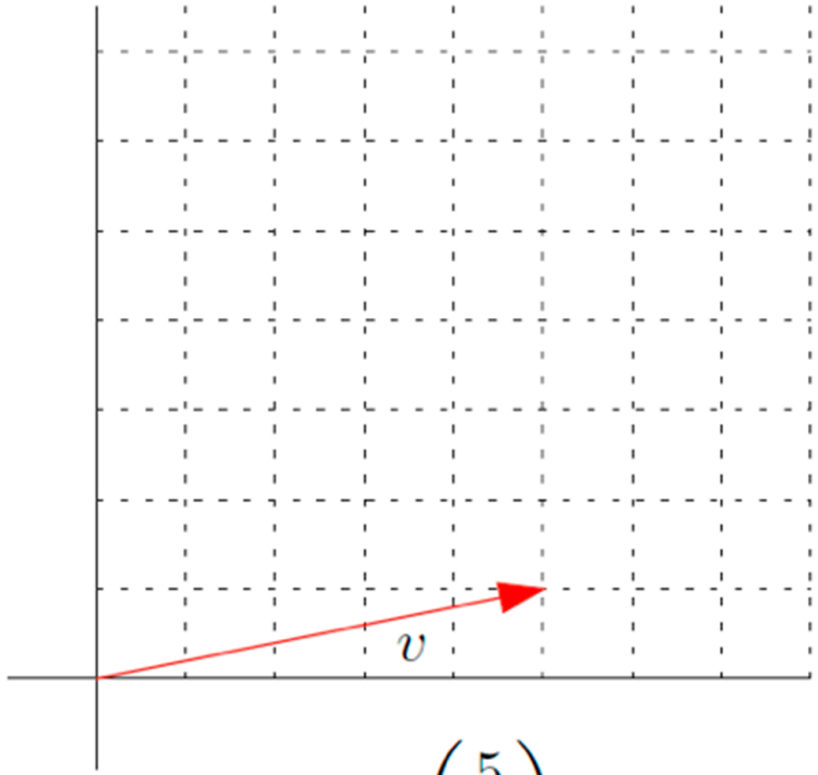
Recall that given a coordinate system, we can express a vector as a list of numbers.

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_d \end{pmatrix}$$

These numbers are the coefficients of a linear combination of the **basis vectors**.

$\hat{x} = (1, 0)$

$\hat{y} = (0, 1)$



$$v = \begin{pmatrix} 5 \\ 1 \end{pmatrix}$$

$v = 5\hat{x} + 1\hat{y}$

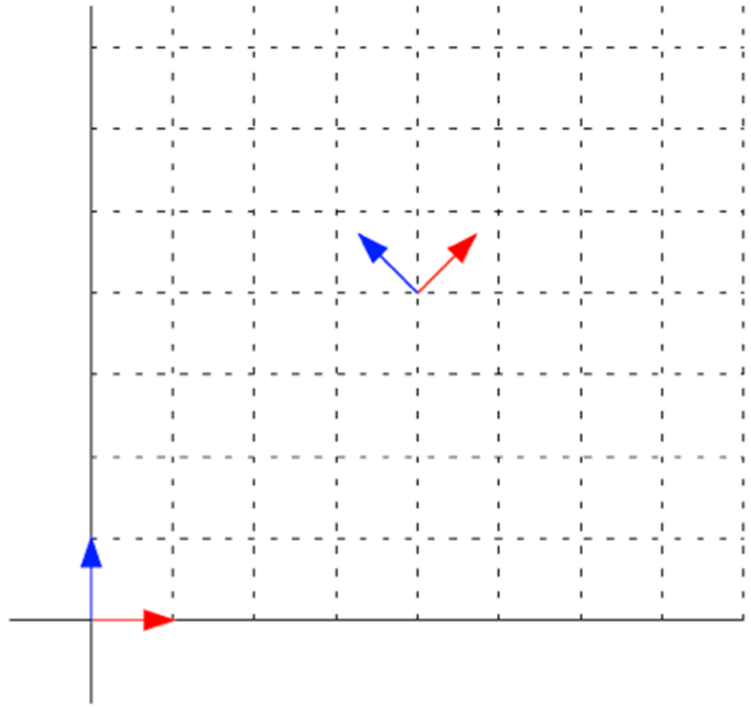
Orthogonal Bases

Two vectors form an **orthogonal basis** if:

1. They are both unit vectors, and

2. They are orthogonal, i.e. perpendicular.

The advantage of working with an orthogonal basis is that lengths of vectors, expressed in the bases, are easy to calculate.



Dot Products

For two vectors v and w, their **dot product** is a scalar, defined as:

$$v \cdot w = v_1 w_1 + v_2 w_2 + \cdots + v_d w_d$$

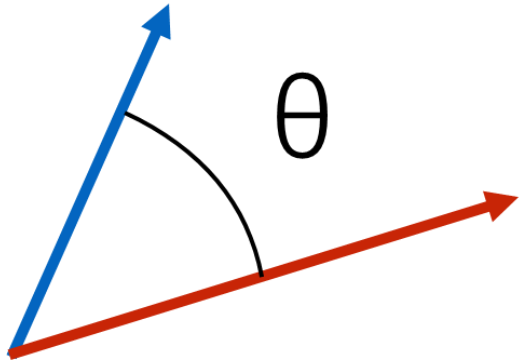
$$v \cdot w = \sum_{i=1}^{d} v_i w_i$$

Dot Products

Useful fact:

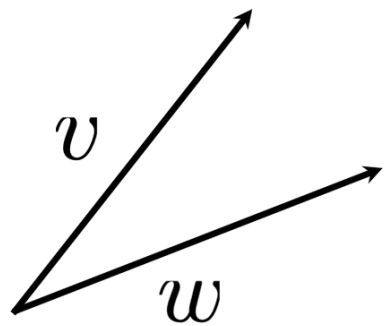
$$\boldsymbol{v} \cdot \boldsymbol{w} = ||\boldsymbol{v}||\ ||\boldsymbol{w}||\ \cos(\theta)$$

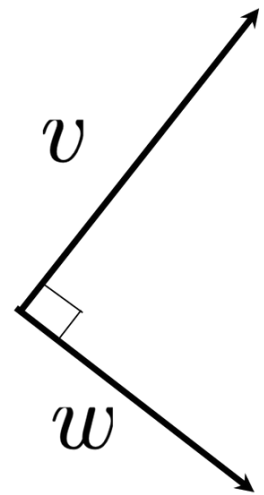
where θ is the angle between the vectors $\boldsymbol{v}$ and $\boldsymbol{w}$

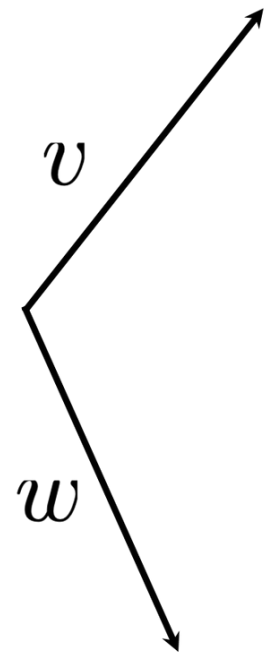


What do we know if $\boldsymbol{v} \cdot \boldsymbol{w} = 0$?

$$v \cdot w$$

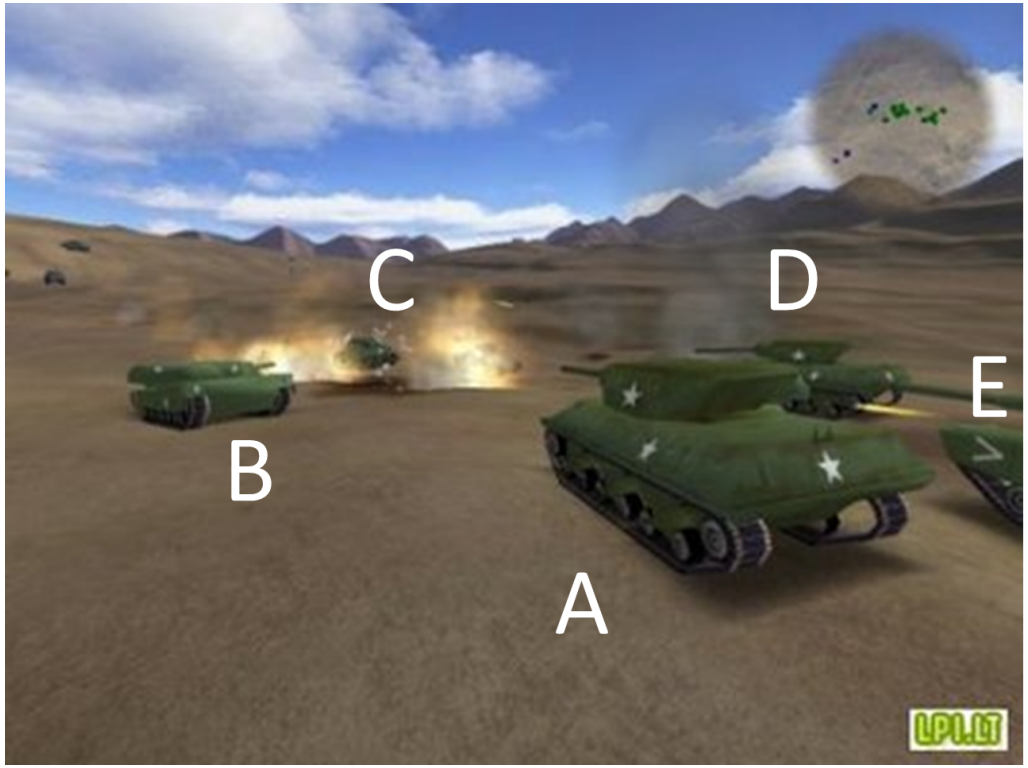


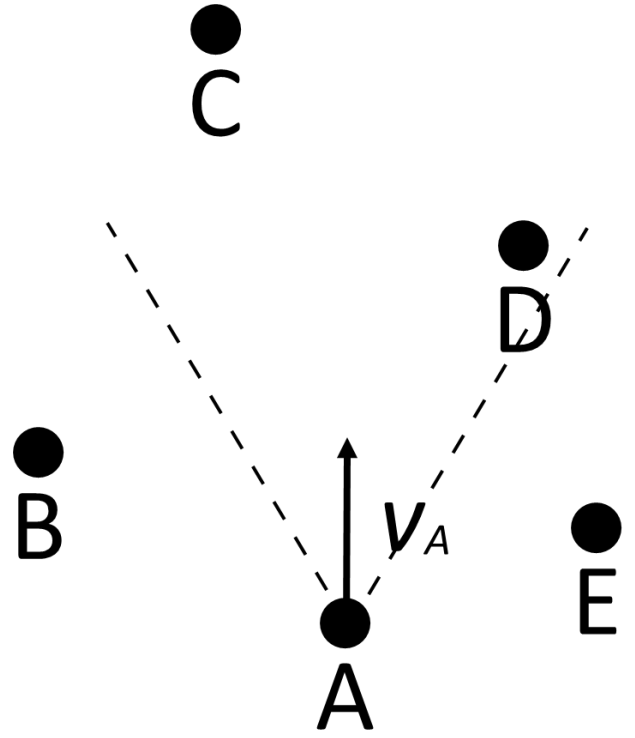




$v \cdot w > 0$ $\quad$ $v \cdot w = 0$ $\quad$ $v \cdot w < 0$

The operator inside tank **A** has a 60° field of view out the window.
(i.e. 30° on either side)





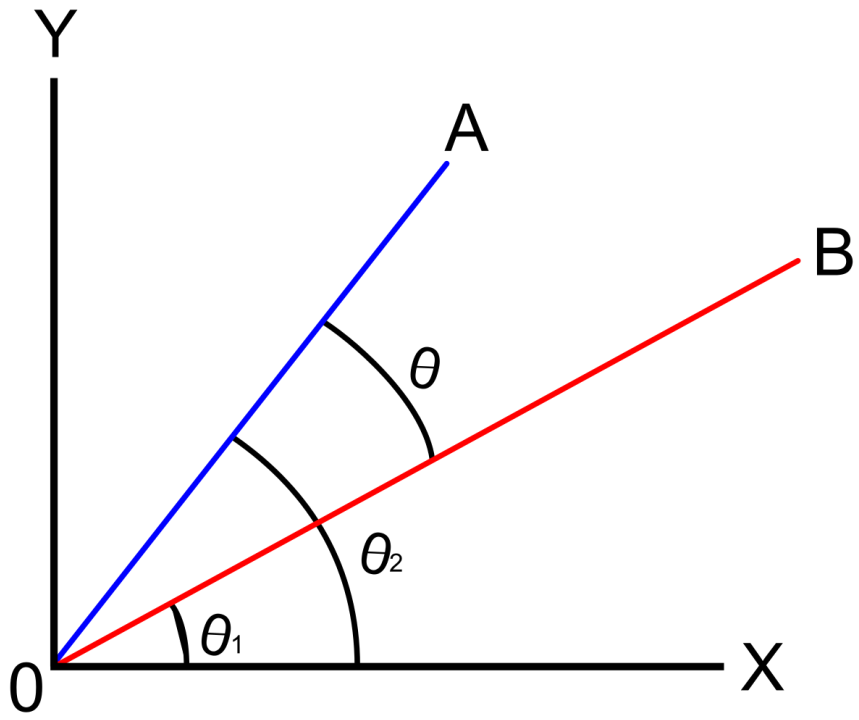
Can they see the other tanks located at points **B**, **C**, **D**, and **E**?

Angle Between Two Vectors

$\boldsymbol{a} \cdot \boldsymbol{b} = ||\boldsymbol{a}||\,||\boldsymbol{b}|| \cos\theta$

If both ***a*** and ***b*** are unit vectors, then $\boldsymbol{a} \cdot \mathbf{b}$ gives the cosine of the angle between them.

It is often useful to calculate the angle between two vectors. How would you do this?



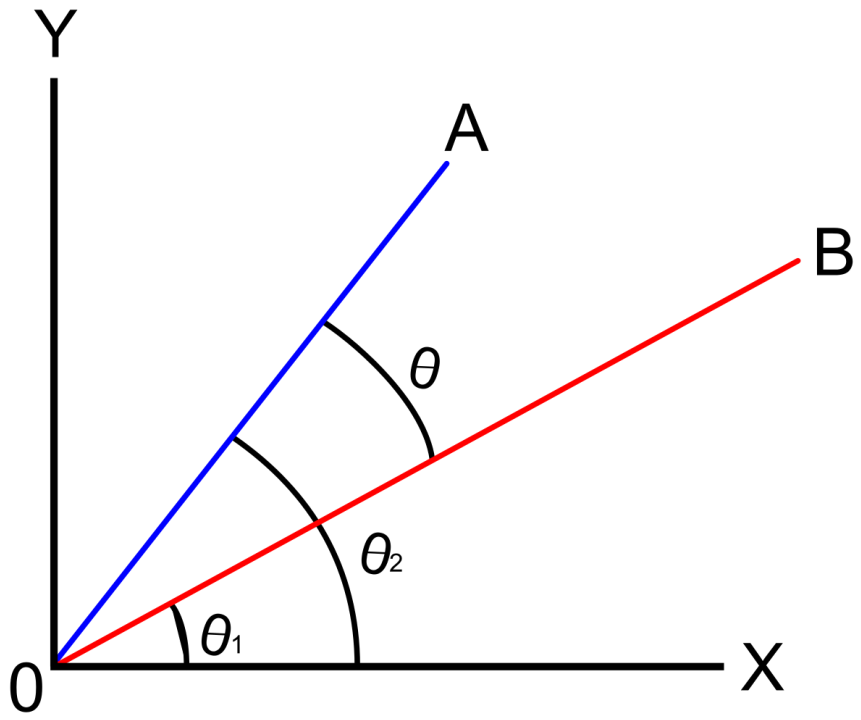
Angle Between Two Vectors

$\boldsymbol{a} \cdot \boldsymbol{b} = ||\boldsymbol{a}||\,||\boldsymbol{b}|| \cos\theta$

If both ***a*** and ***b*** are unit vectors, then $\boldsymbol{a} \cdot \mathbf{b}$ gives the cosine of the angle between them.

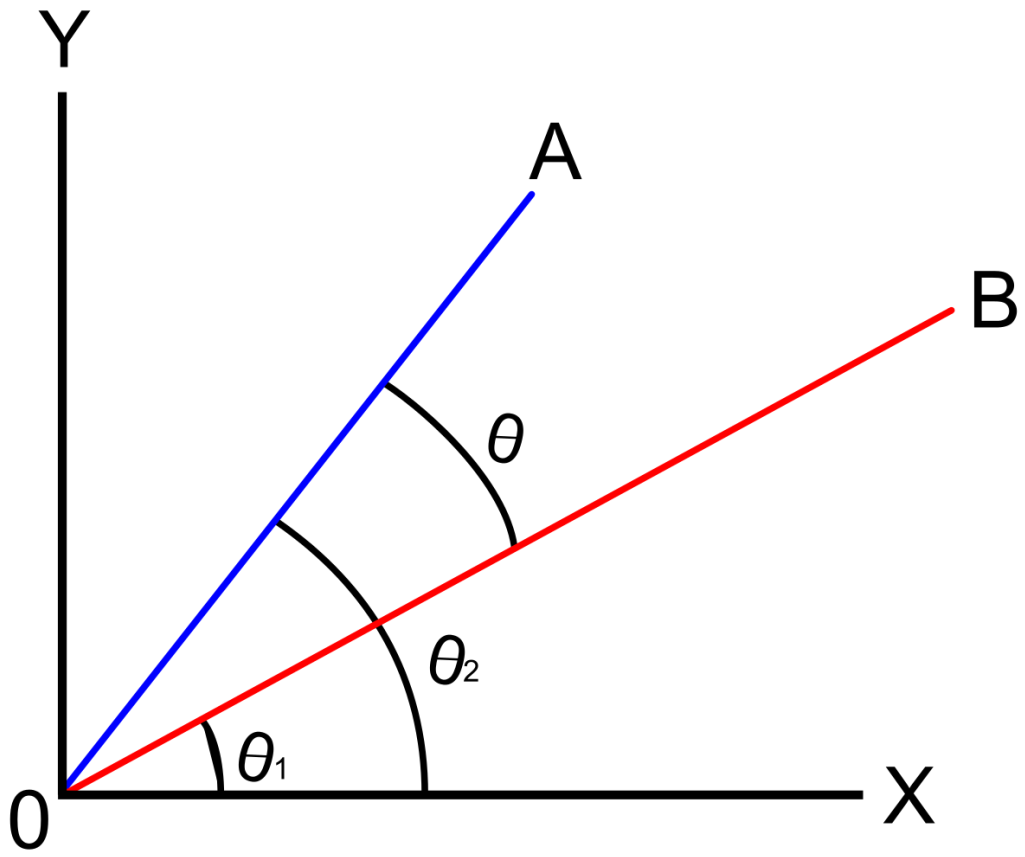
It is often useful to calculate the angle between two vectors. How would you do this?

$\theta = \text{acos}(||\boldsymbol{a}||\,||\boldsymbol{b}||)$



Angle Between Two Vectors

```
const A = new gfx.Vector2(3, 5);
const B = new gfx.Vector2(7, 3);

// Create a vector for the x axis
const X = new gfx.Vector2(1, 0);

// One way to compute the angle theta
const theta1 = X.angleBetween(A);
const theta2 = X.angleBetween(B);
const theta = theta1 - theta2;

// Another way to compute the angle theta
const theta = gfx.Vector2.angleBetween(A,
B);
```

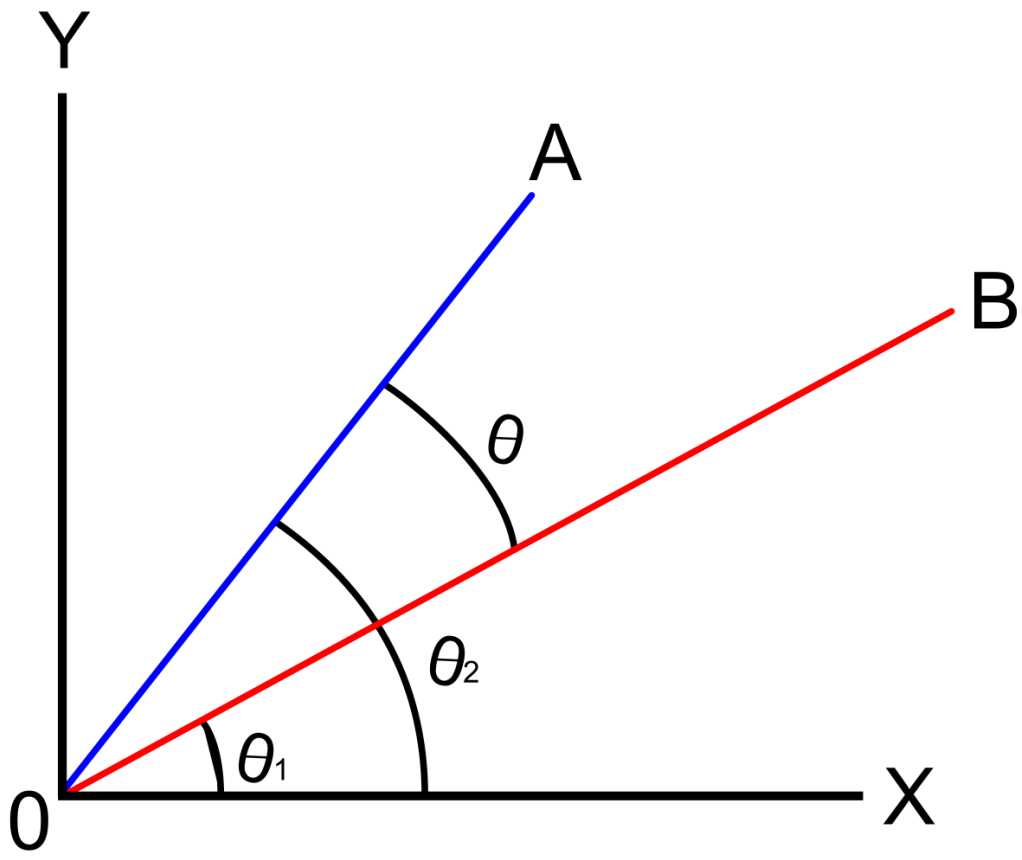


Signed Angle Between Two Vectors?

$\theta = \text{acos}(||\boldsymbol{a}|| \, ||\boldsymbol{b}||)$

This returns an angle between 0 and π.

```
const A = new gfx.Vector2(3, 5);
const B = new gfx.Vector2(7, 3);

console.log(gfx.Vector2.angleBetween(A, B));
>> 0.625485040239292

console.log(gfx.Vector2.angleBetween(B, A));
>> 0.625485040239292
```

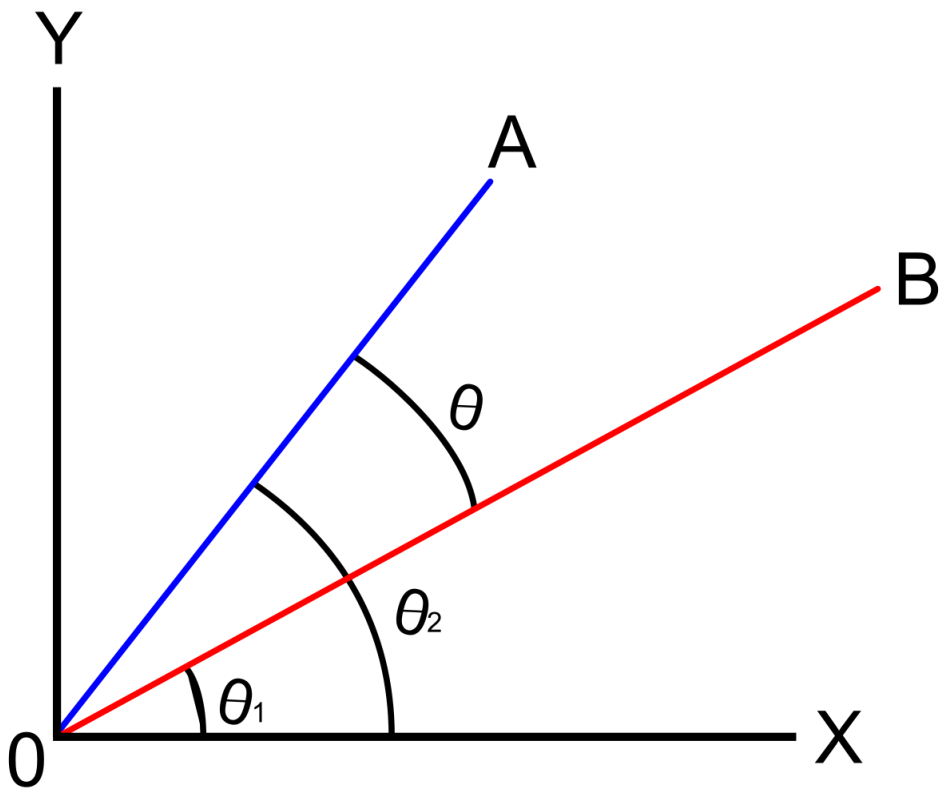


Signed Angle Between Two Vectors (2D)

$\theta = \text{atan2}(\, ||\boldsymbol{b}||.y, ||\boldsymbol{b}||.x\,) - \text{atan2}(\, ||\boldsymbol{a}||.y, ||\boldsymbol{a}||.x\,)$

This returns an angle between $-\pi$ and π.

```
const A = new gfx.Vector2(3, 5);
const B = new gfx.Vector2(7, 3);

console.log(gfx.Vector2.angleBetweenSigned(A,
B));
>> -0.6254850402392292

console.log(gfx.Vector2.angleBetweenSigned(B,
A));
>> 0.6254850402392292
```



What does the lookAt() function do?

```
ship.lookAt(this.mousePosition);
```

What does the lookAt() function do?

```
ship.lookAt(this.mousePosition);
```

```
// The lookAt method rotates the object to point towards a target position.
// The second optional parameter specifies the local axis that should point at the target.
// By default, this is the object's y-axis
lookAt(target: Vector2, lookVector = Vector2.UP): void
{









}
```

What does the lookAt() function do?

```
ship.lookAt(this.mousePosition);
```

```
// The lookAt method rotates the object to point towards a target position.
// The second optional parameter specifies the local axis that should point at the target.
// By default, this is the object's y-axis
lookAt(target: Vector2, lookVector = Vector2.UP): void
{
    // Compute the vector from the object's position to the target
    const targetVector = Vector2.subtract(target, this.position);









}
```

What does the lookAt() function do?

```
ship.lookAt(this.mousePosition);
```

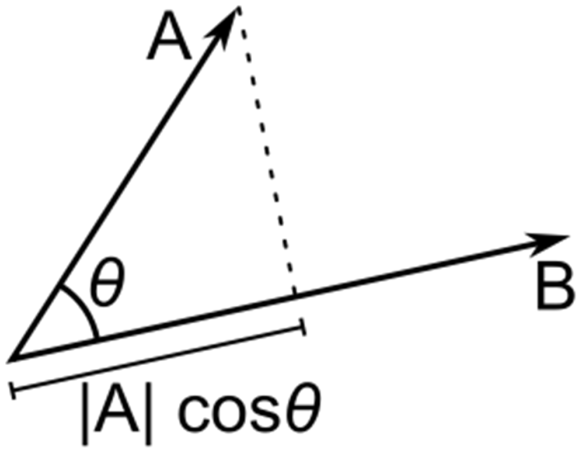
```
// The lookAt method rotates the object to point towards a target position.
// The second optional parameter specifies the local axis that should point at the target.
// By default, this is the object's y-axis
lookAt(target: Vector2, lookVector = Vector2.UP): void
{
    // Compute the vector from the object's position to the target
    const targetVector = Vector2.subtract(target, this.position);

    // Compute the signed angle between the look vector and target vector
    // and use it to set the object's current rotation
    if(targetVector.length() > 0)
        this.rotation = lookVector.angleBetweenSigned(targetVector);
}
```

Scalar Projection

$\boldsymbol{a} \cdot \boldsymbol{b} = ||\boldsymbol{a}||\,||\boldsymbol{b}|| \cos\theta$

If only ***b*** is a unit vector, then $\boldsymbol{a} \cdot \boldsymbol{b}$ gives the length of the projection of **a** in the direction of **b.**

This is known as the **scalar projection** of ***a*** onto ***b***.

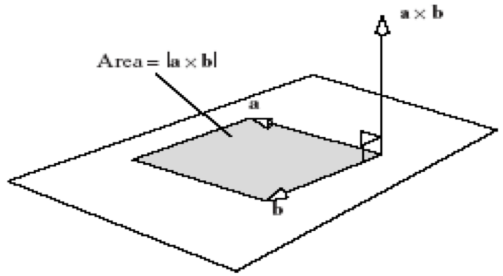


How much
does ***a*** point in
the direction of ***b***?

Cross Product (3D Only!)

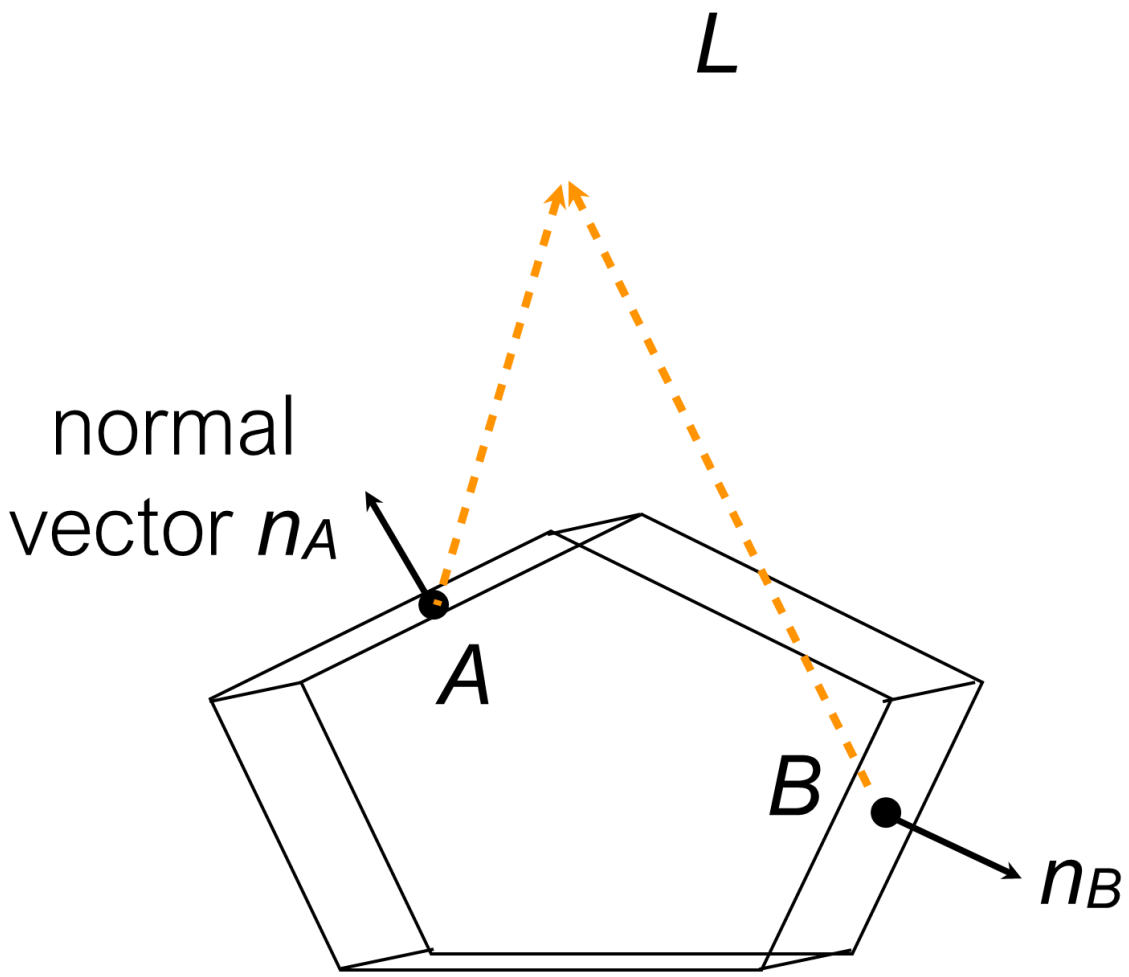
In 3D, the cross product of ***v*** and ***w*** is another vector, defined as:

$$v \times w = \begin{bmatrix} v_2 w_3 - v_3 w_2 \\ v_3 w_1 - v_1 w_3 \\ v_1 w_2 - v_2 w_1 \end{bmatrix}$$



length of ***v*** x ***w*** = area of parallelogram created by the two vectors

direction of ***v*** x ***w*** = orthogonal to the two vectors



Does the light ***L*** hit the inside or the outside of the object?

Review

Is the result a point, a vector, or a scalar?

(*P* and *Q* are points, *v* and *w* are vectors, all in 3D.)

$P + v = ?$

$v + w = ?$

$P + Q = ?$

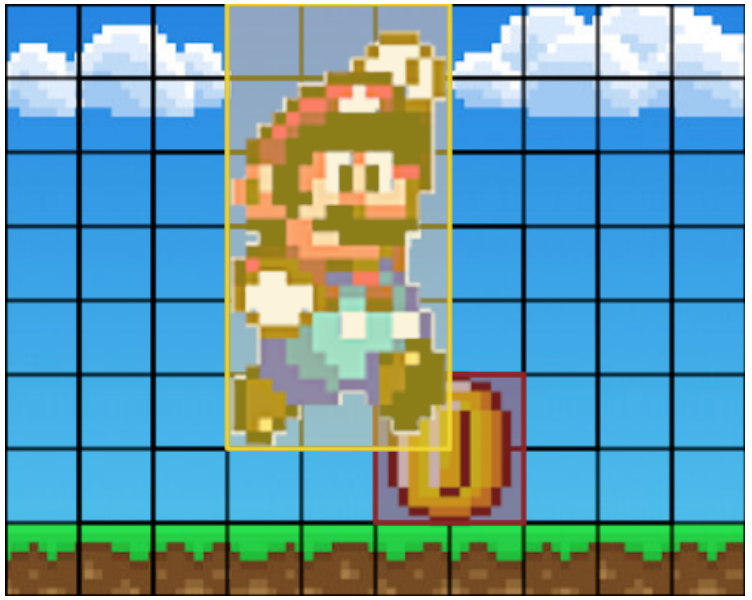
$v \cdot w = ?$

$v \times w = ?$

Intersection Tests

- In computer graphics, it is a very common operation to test whether two objects are intersecting (**collision detection**).
- For complex objects, comparing all the triangles that make up the objects is much too computationally expensive.
- Most real-time applications and games use simplified **bounding areas/volumes** for intersection tests.

  Refer to Chapter 22 of your textbook for more details.



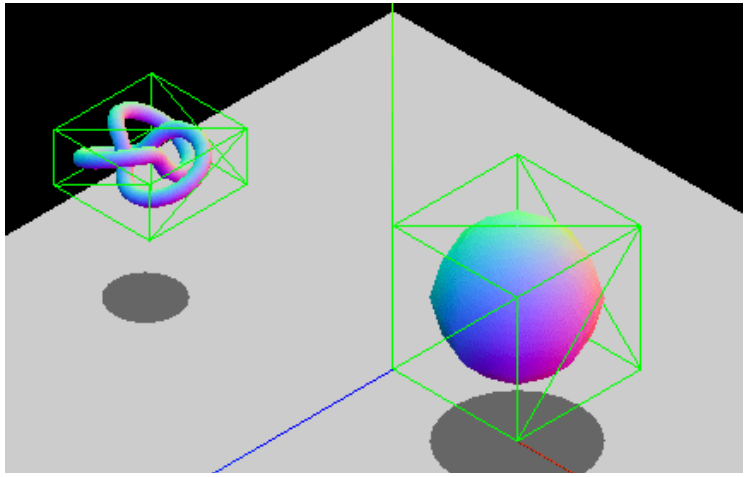


Image Credit: https://levelup.gitconnected.com/2d-collision-detection-8e50b6b8b5c0, https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection

Bounding Circle / Sphere

computed from corners of object's bounding box

computed based on most distant vertex from the center

smaller radius may be possible by moving the circle's center

Bounding Circle / Sphere

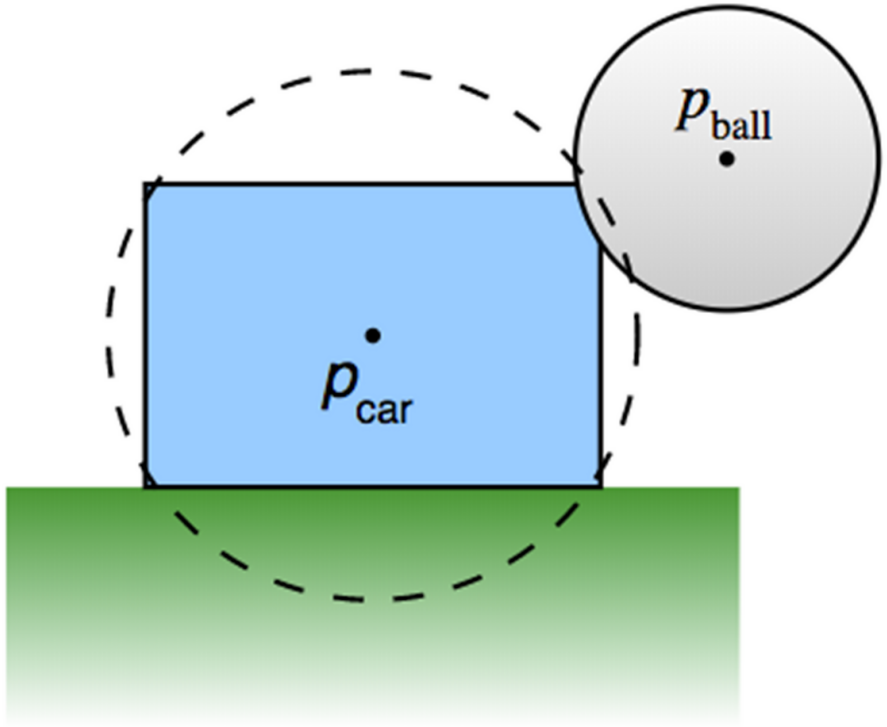
computed from corners of object's bounding box

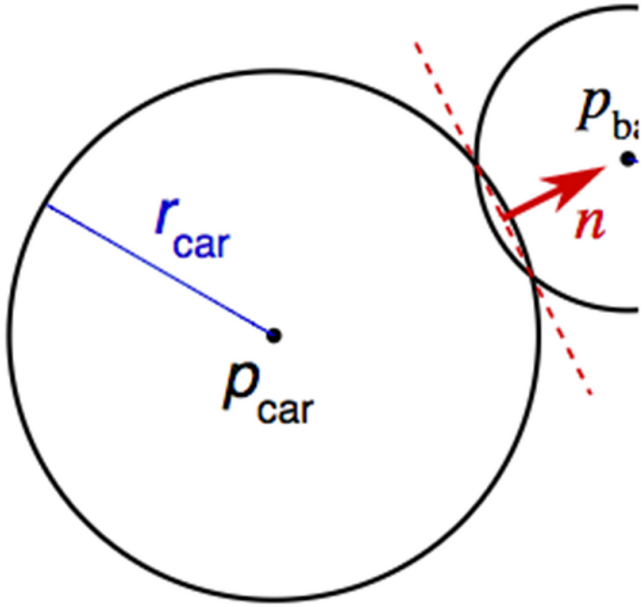
computed based on most distant vertex from the center

smaller radius may be possible by moving the circle's center

GopherGfx automatically computes the bounding circle/sphere using this approach.

Circle/Sphere Intersections

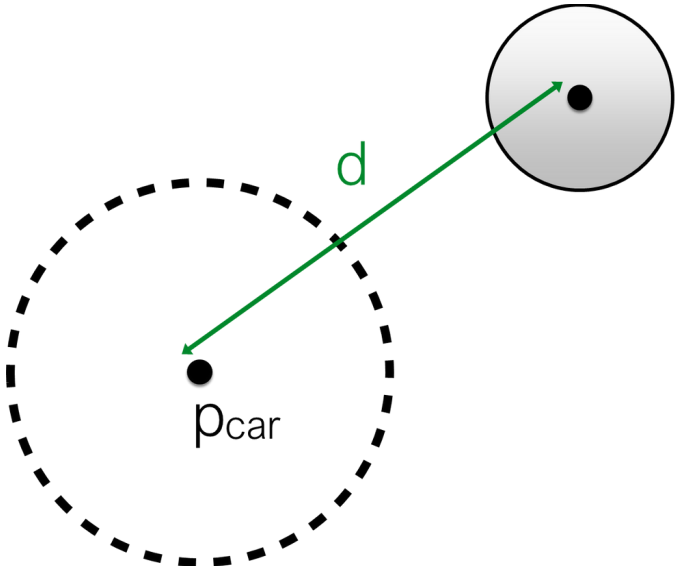




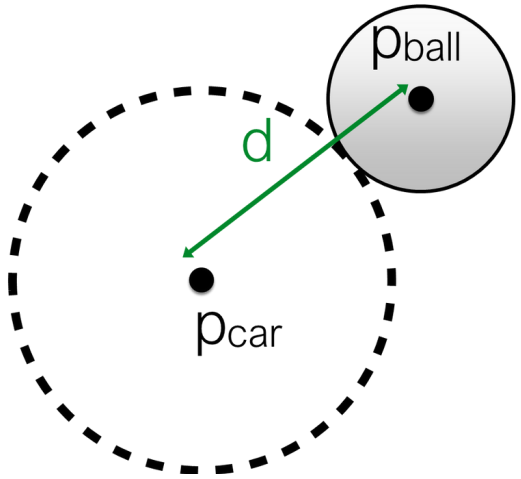
Circle/Sphere Intersections

What can you tell me about the distance (d) in these three cases?

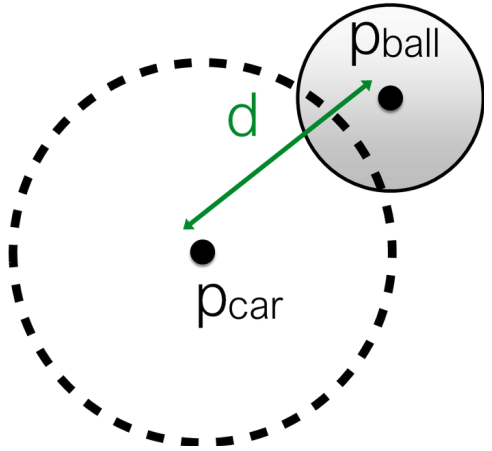
Case 1: Far Away



Case 2: Touching



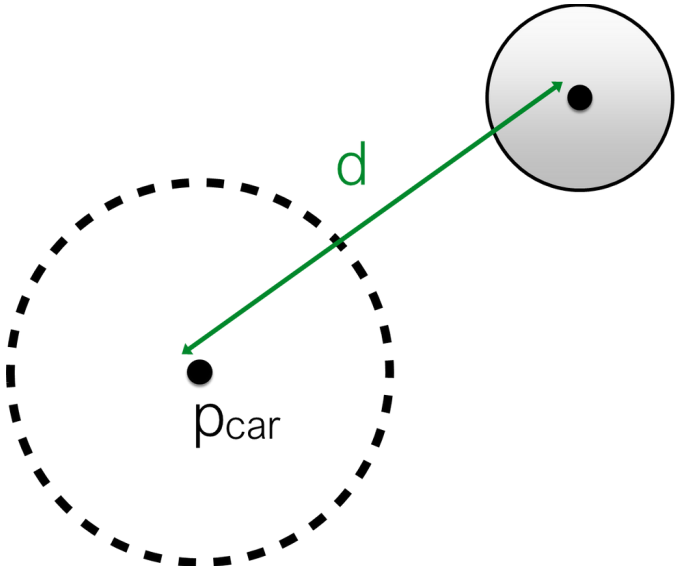
Case 3: Intersecting



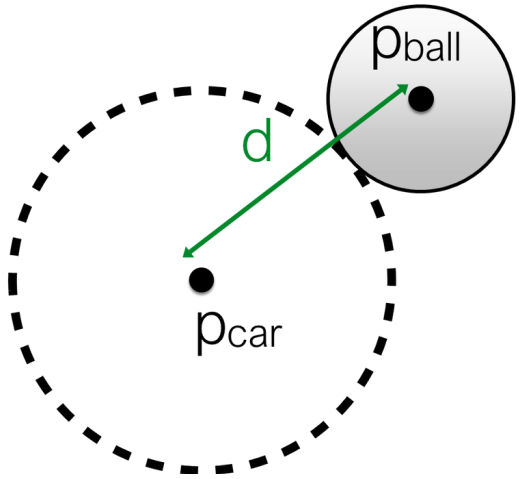
Circle/Sphere Intersections

What can you tell me about the distance (d) in these three cases?

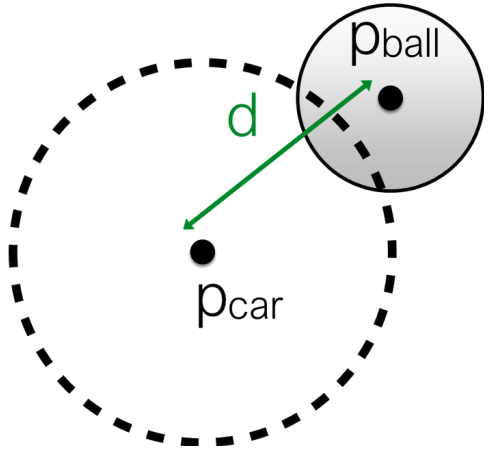
Case 1: Far Away



Case 2: Touching



Case 3: Intersecting

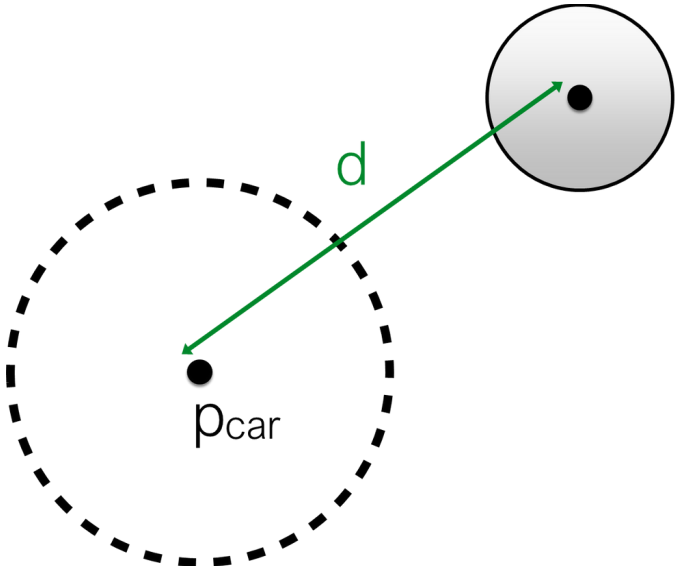


$d > (r_{car} + r_{ball})$

Circle/Sphere Intersections

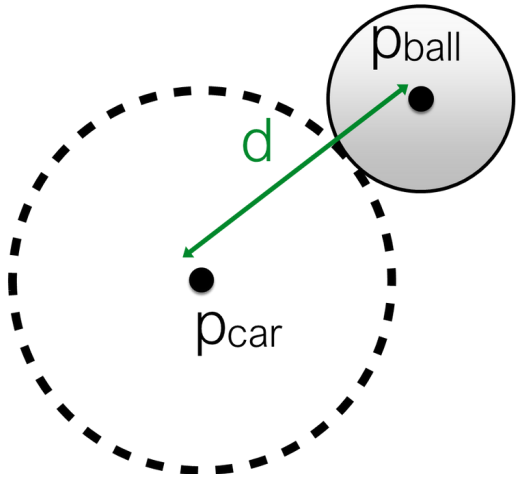
What can you tell me about the distance (d) in these three cases?

Case 1: Far Away



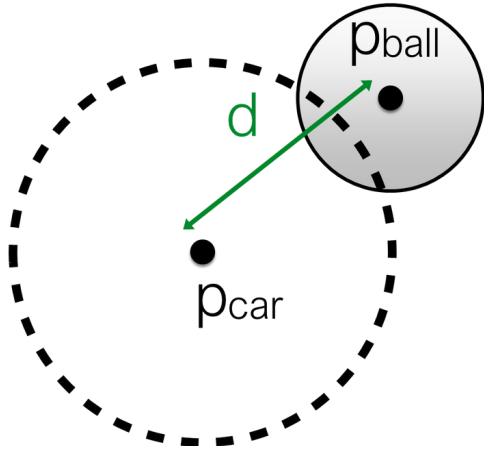
$d > (r_{car} + r_{ball})$

Case 2: Touching



$d == (r_{car} + r_{ball})$

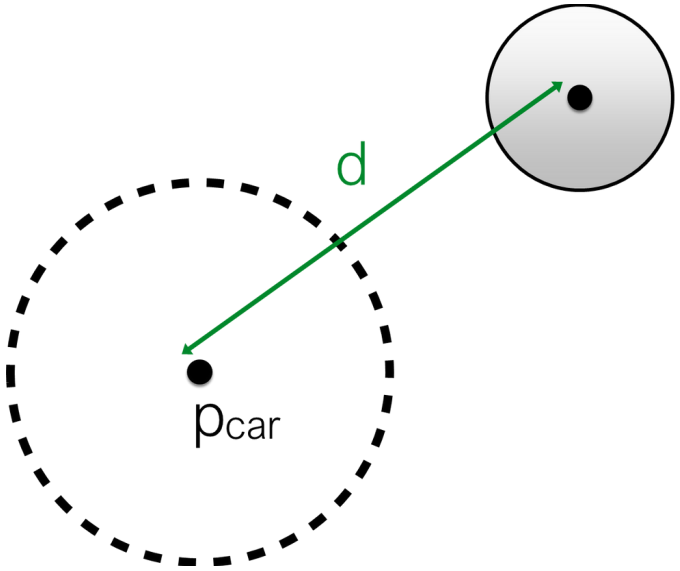
Case 3: Intersecting



Circle/Sphere Intersections

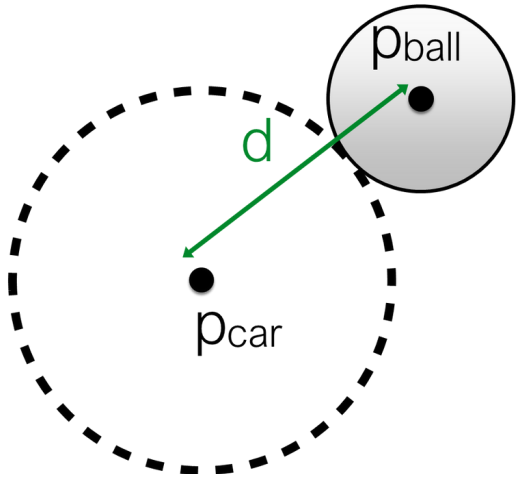
What can you tell me about the distance (d) in these three cases?

Case 1: Far Away



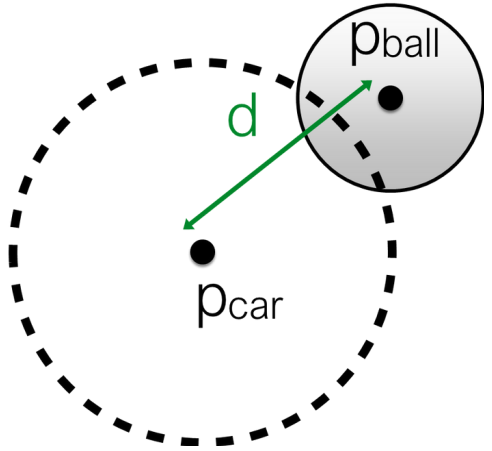
$d > (r_{car} + r_{ball})$

Case 2: Touching



$d == (r_{car} + r_{ball})$

Case 3: Intersecting



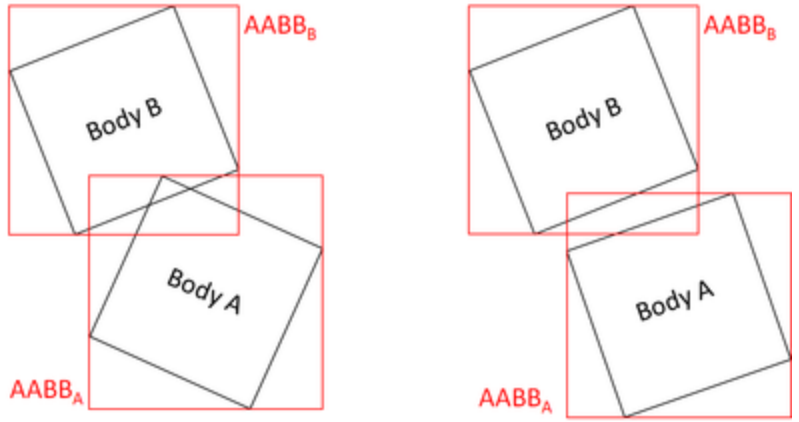
$d < (r_{car} + r_{ball})$

Axis-Aligned Bounding Box (AABB)

- Axis-aligned bounding boxes are another efficient method when a circle/sphere provides a poor approximation.

- Objects are encapsulated within **non-rotated** boxes. The axis-aligned constraint exists for performance reasons.

- Note that the AABB dimensions may change when objects are rotated relative to the world axes.



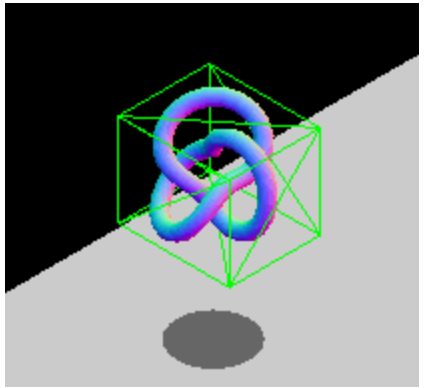


Image Credit: http://myselph.de/gamePhysics/collisionDetection.html, https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection

AABB Intersections

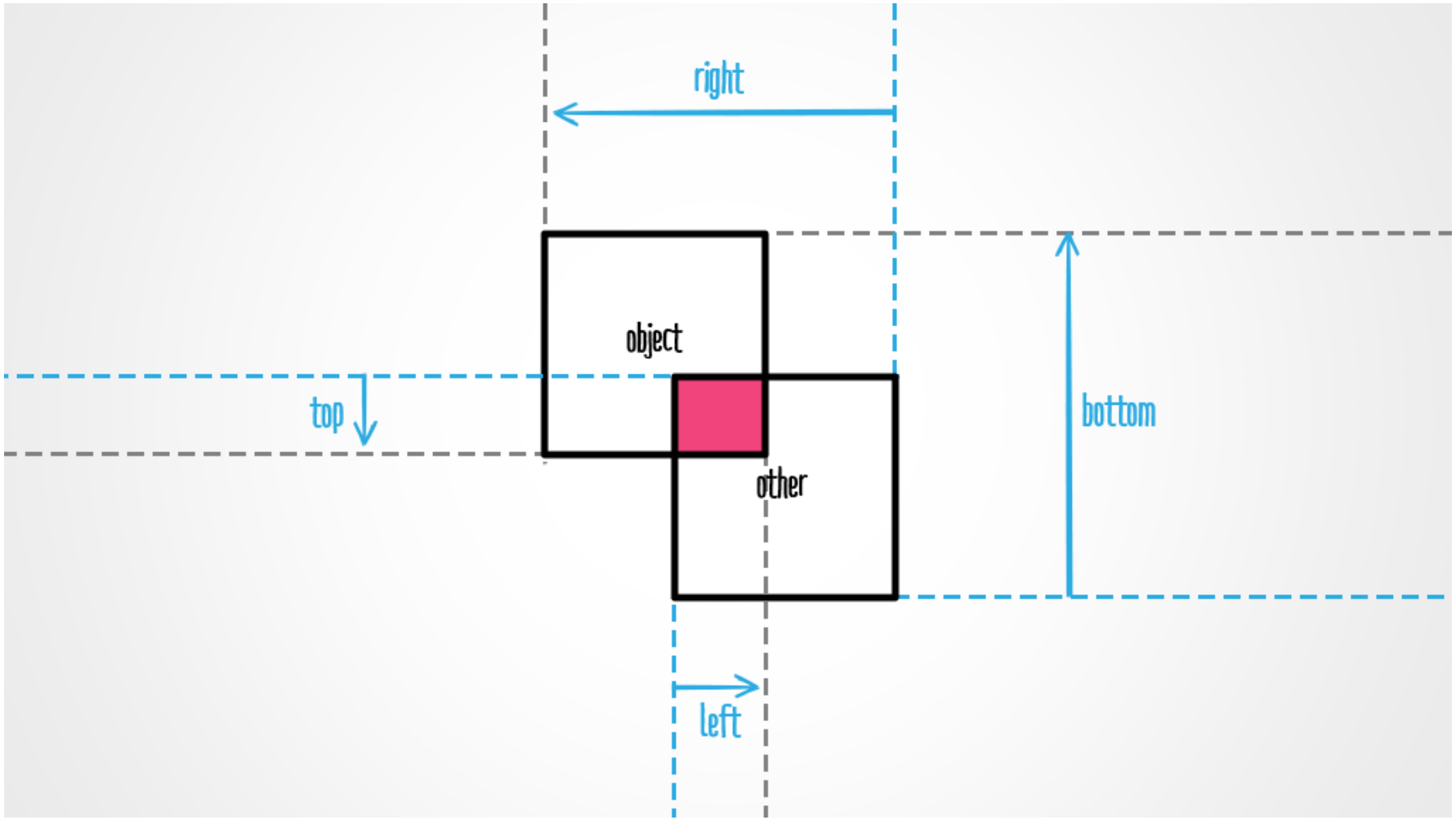
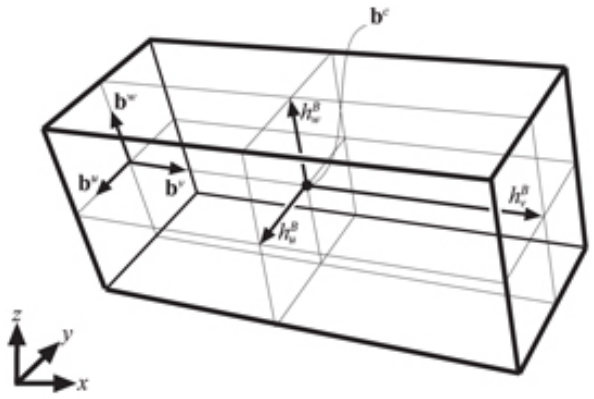
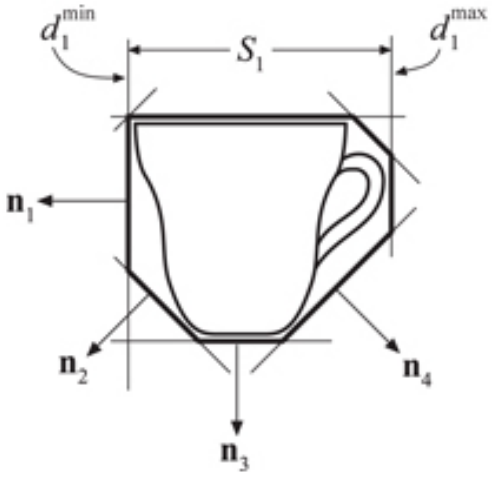


Image Credit: https://www.amanotes.com/post/using-swept-aabb-to-detect-and-process-collision

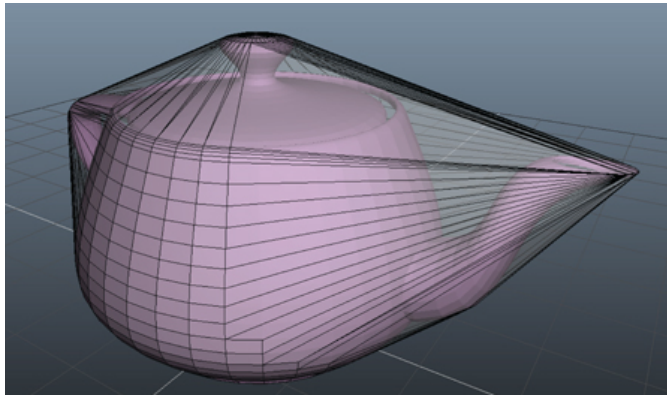
More Complex Bounding Volumes



Oriented Bounding Box (OBB)



Discrete Oriented Polytope (k-DOP)



Convex Hull

Summary

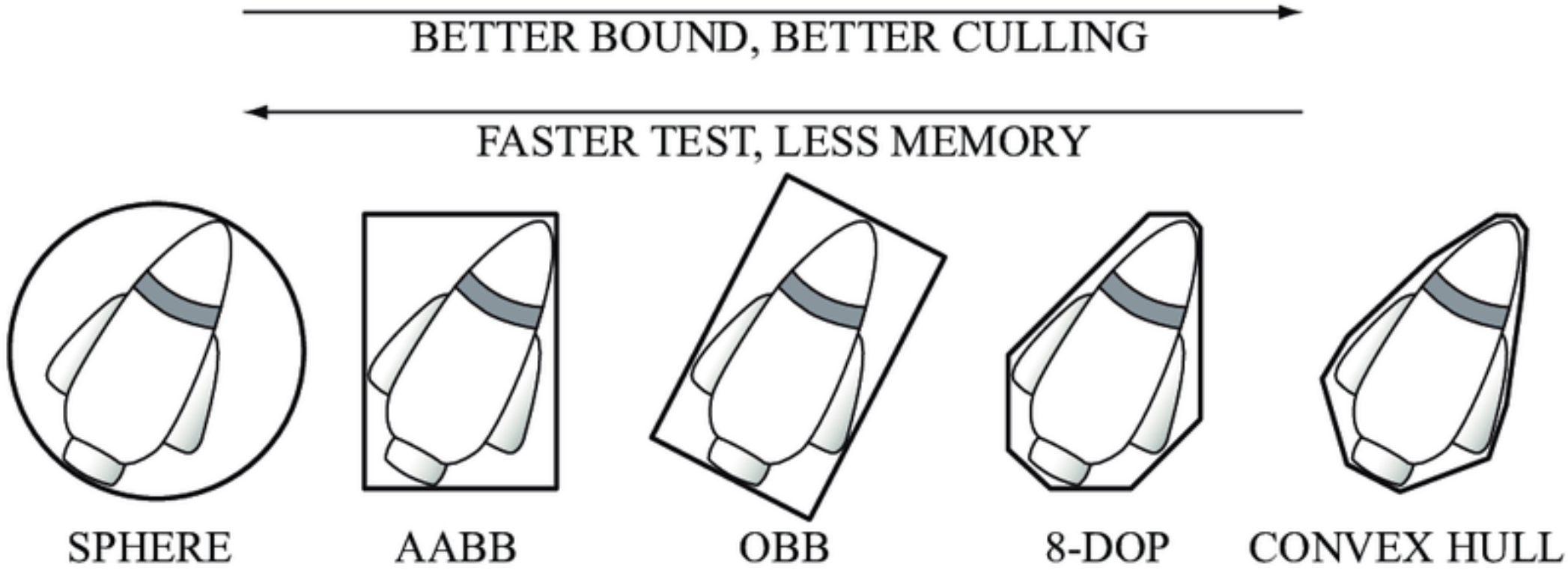


Image Credit: https://www.researchgate.net/figure/Bounding-volumes-sphere-axis-aligned-bounding-box-AABB-oriented-bounding-box_fig9_272093426