3D Game Loop & Mathematics

(SENG 463 - **Game Programming)**

Dr.Çağatay ÜNDEĞER

Research and Innovation Director

SimBT Inc.

e-mail :

cagatay.undeger@simbt.com.tr

cagatay@undeger.com

Outline

- 3D Game Loop
- 3D Game Mathematics
  - Coordinate Systems
  - Vectors
  - Dot Product
  - Cross Product
  - Linear Interpolation
  - Quaternions

Game Loop

- Game composed of many interacting subsystems:
  - I/O, rendering, animation, collision detection, rigid body dynamics, multiplayer networking, audio, game objects, model importers, etc.
- Subsystems require periodic servicing with various rates
  - Rendering and Animation:
    - Desktop: 30 - 60 Hz
    - VR: 90 – 120 Hz
  - Dynamics simulations: 120 Hz
  - Higher-level systems such as AI : 1 or 2 times/second (not necessarily synch. with rendering)
- ⇒ Solution: a single “game loop” to update everything

```
while (true) {          //(need something to quit…)
    processInput();     //but don’t wait for input
    updateGameState();  //one step of the game simulation
    renderGame();       //generate outputs
}
```

Frame Rate (FPS)

- Frame rate / Frame Per Second
- Number of game loop renderings / second (FPS)
- Describes how rapidly the sequence of still 3D frames is presented to the viewer
- Frame time, Time delta, Delta time, Frame period means:
  - Amount of time elapsed between 2 successive frames (seconds)
  - Amount of time to process inputs, update game state and render image
  - E.g. 60 FPS requires 16,6 ms/frame delta time

Use of Delta Time

- Most game engines uses delta time in game loop
- Update of objects takes into account the amount of elapsed game time since last frame
- For instance to move a game object in a constant speed
  – Move (speed * elapsed time) meters in each frame

```
double lastTime = getCurrentTime(); //CPU's high resolution timer
while (true){
    double current = getCurrentTime();
    double elapsed = current - lastTime; //last frame duration
    processInput();
    update(elapsed);
    render();
    lastTime = current;
}
```

Variable and Constant Delta Time / Frame Rate

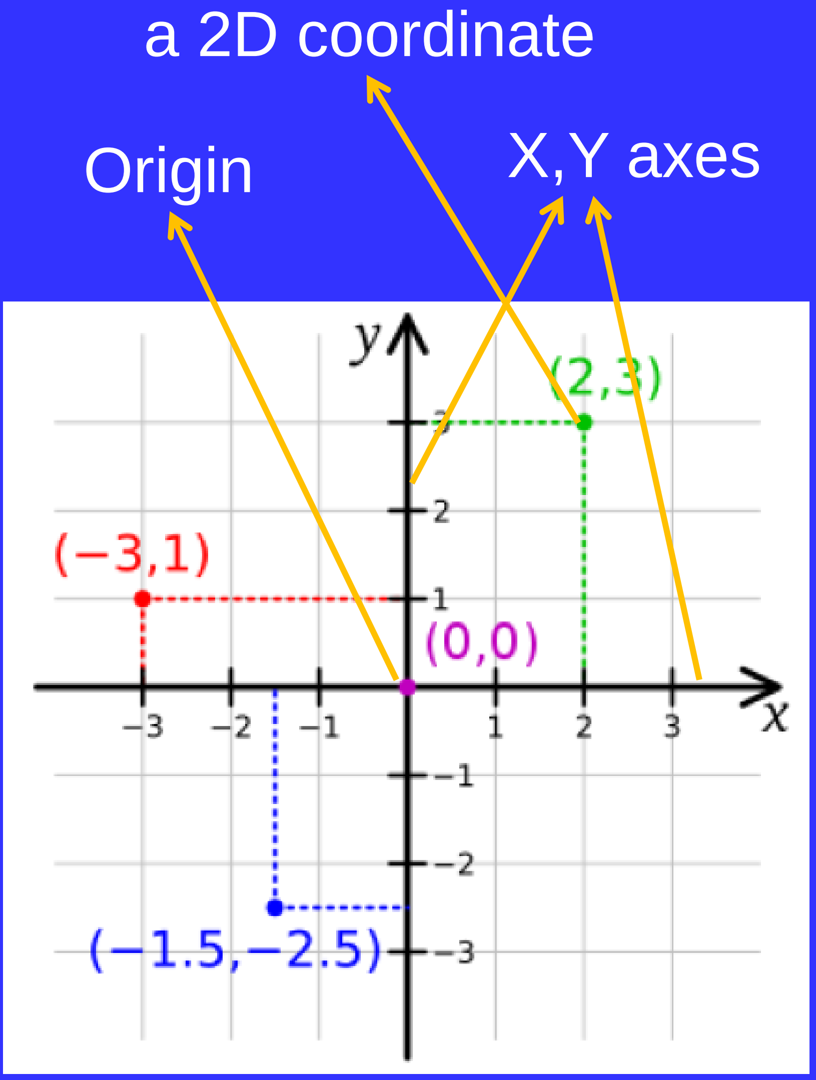
- Frame rate can be either variable or constant
- Variable frame rate means as fast as possible
  - Variable is undeterministic, sometimes very fast sometimes very slow
- In some applications a fixed rate may be prefered
- Constant frame rate means we require each frame to take a contant fixed delta time
  - So if frame time ise lower than contant fixed delta time,
    - Wait to reach contant fixed delta time
  - If frame time is higher than contant fixed delta time
    - Do not wait, go on

Fundamental Classes

- Game Objects
- Transform of Game Objects
  - Parent Transform
  - Position
  - Rotation
  - Scale
- Basic Mathematical Operations
  - Vectors
  - Rays
  - Bounds
  - Quaternions

2D Cartesian Coordinate System

- Specifies each point uniquely on a plane by
  – A pair of numerical coordinates,
  – Which are the signed (+/-) distances from the point to two fixed perpendicular directed lines (axes),
  – Measured in the same unit of length (e.g. meters).
  – Intersection point of directed lines is the origin.



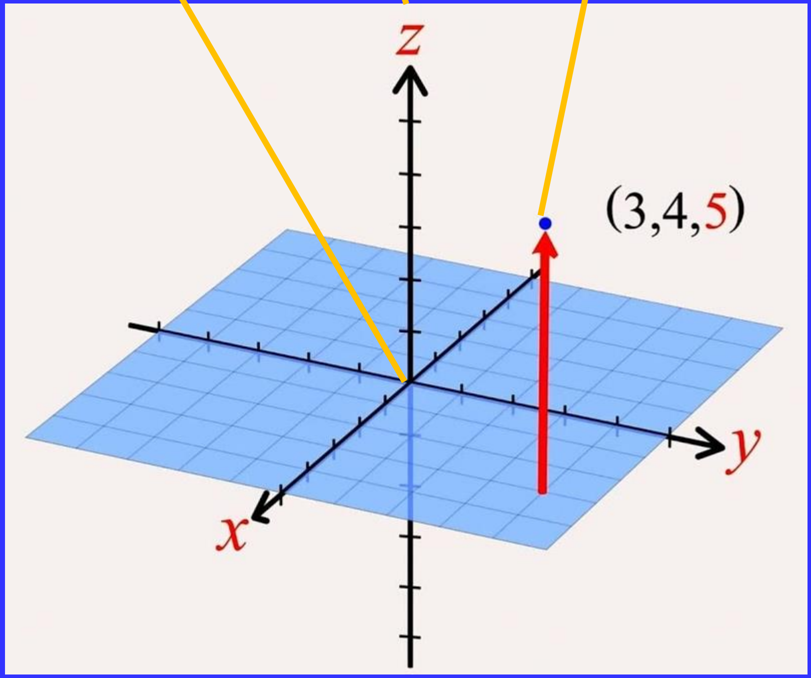
3D Cartesian Coordinate System

- 3D Cardesian Coordinate System Specifies each point uniquely in a volume by
  - With triple numerical coordinates,
  - Similar to 2D coordinate system, but a 3rd dimension (Z axis) is added with a directed line perpendicular to the 2D plane
  - Passing through the origin

a 3D coordinate

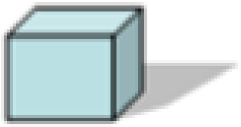
Z axis

Origin



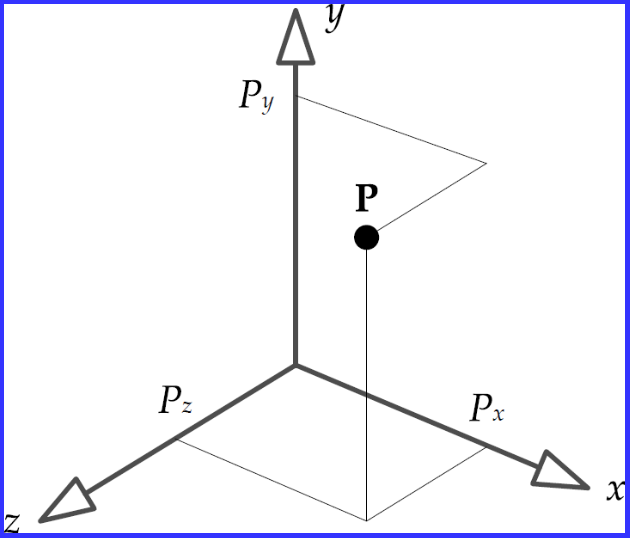
Position & Shape of Objects

- Position (coordinate) of objects in cartesian coordinate system are defined with respect to the dimension of the coordinate system (2D or 3D)
- Shape of objects are defined in zero, one, two or three dimensions

point	line	Plane	Solid
Zero dimensions	One dimension	Two dimensions	Three dimensions
			

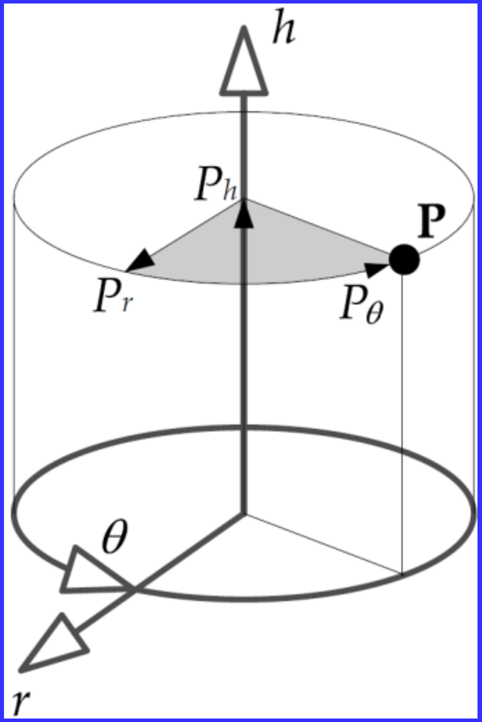
Point

- A point is a location in n-dimensional space
- Usually represented in Cartesian space
- Two or three mutually perpendicular axes
- A point in 3D is a triple of numbers (Px, Py, Pz)
- Usually a game object is defined by a point in cartesian coordinate system
- But on to that point,
  different type of shapes
  can be put to make advanced
  geometric objects

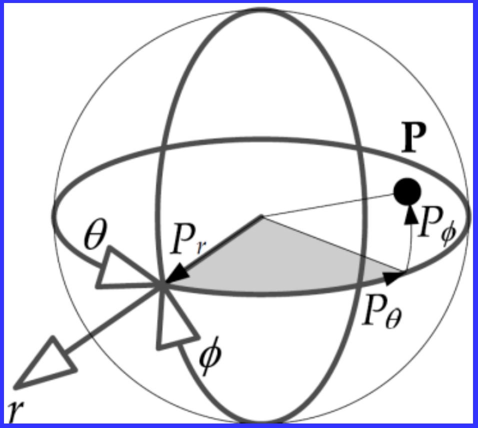


Some Other Coord. Systems

- Cylindrical Coordinate System
  - Employs a height axis (h), a radial axis (r), and a yaw angle (ϴ)
  - Points represented as (P_h,P_r,$P_ϴ$)



- Spherical Coordinate System
  - Pitch(Φ), yaw(ϴ), and radial (r)
  - Points represented as (P_r,$P_Φ$,$P_ϴ$)

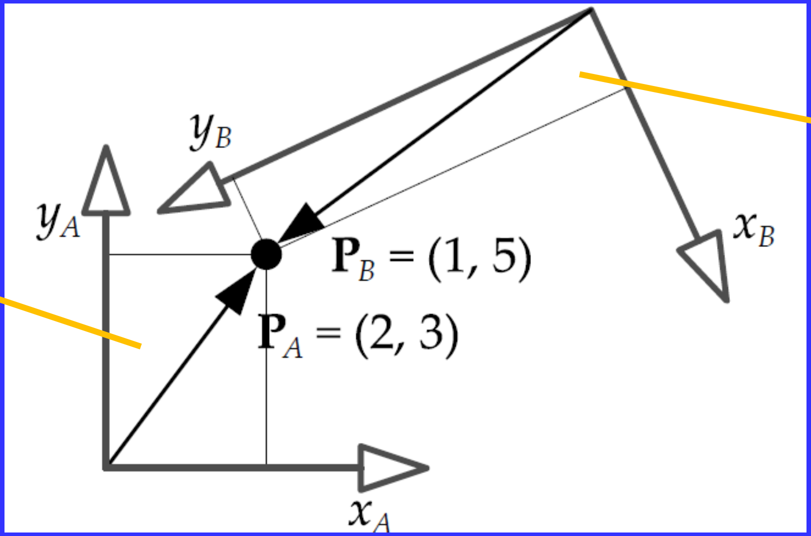


ϴ (theta) Φ (phi)

Coordinate Spaces

- We can think of a point as being a coordinate relative to a given set of axes
- The axes are just for a frame of reference and are referred to as a coordinate space
- Coordinate of a point can be defined in different coordinates spaces and can be converted from a coordinate space to another coordinate space

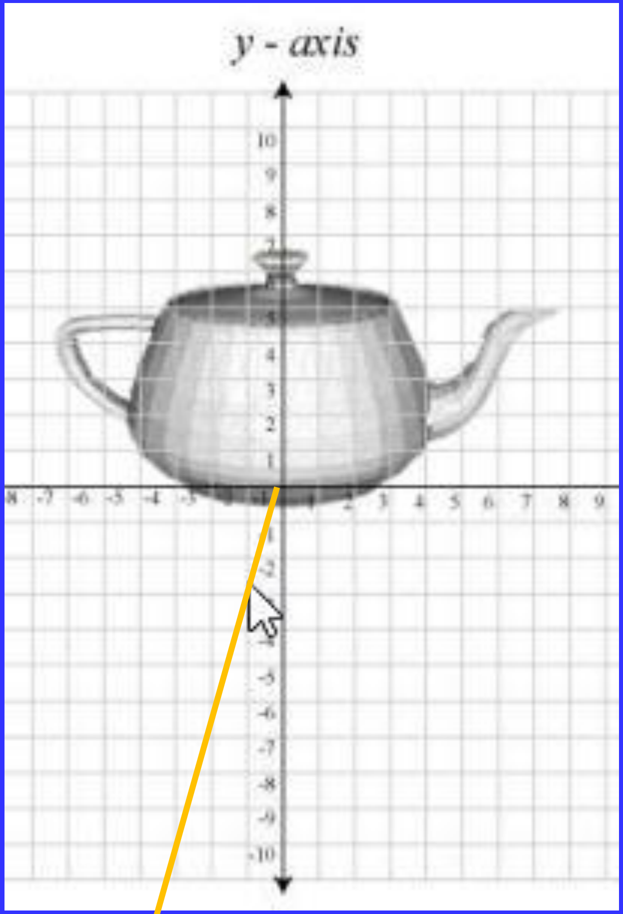
A coordinate space



Another coordinate space

Model Space

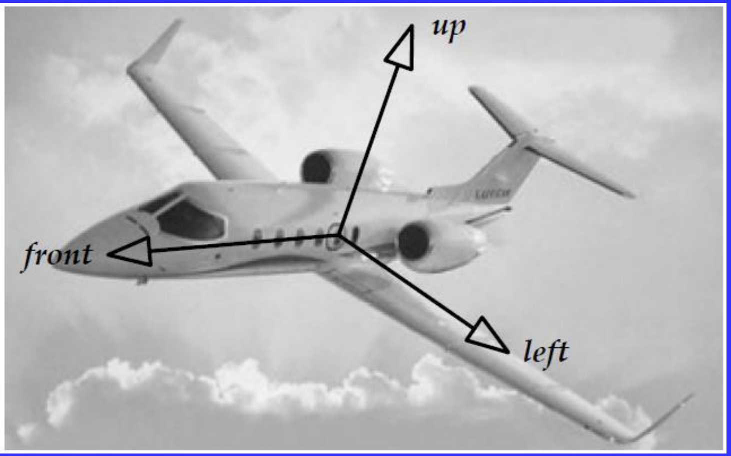
- Originally, an object is in “model space”, also known as “local space” or “object space”.
- In “model space” an object’s vertices are expressed relative to the object that they describe.
- That is, the way an artist models them.
- The image shows an example of an object in object space.
- As you can see from the image, the object is placed at it’s relative origin (for instance, at the bottom of cup)



Origin of local space

Model Space

- When a new model is created,
  - The vertices are relative to a coordinate system, which is the model space
- Model space origin is usually:
  - In the center of the object
  - Or where you would like to hold the object from
- Model space axes are usually named something like:
  - Front / Forward,
  - Left
  - Up

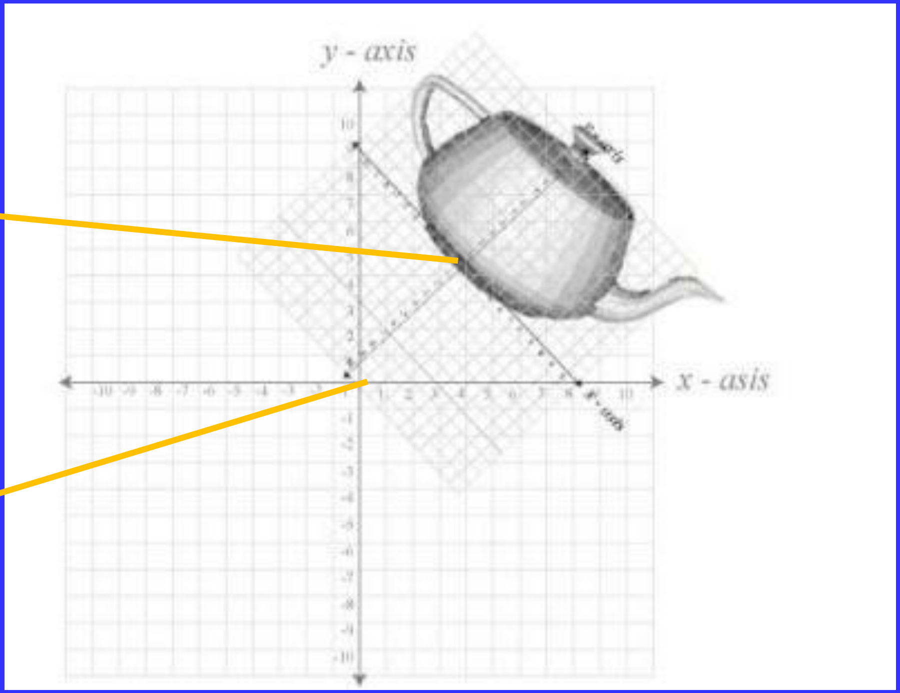


World Space

- A fixed global coordinate system of game engine where the objects, orientations, and scales are defined
- Origin usually placed at the center of the playable area
- The orientation is arbitrary but usually Y or Z is up
- All objects are located inside this world space with their world position

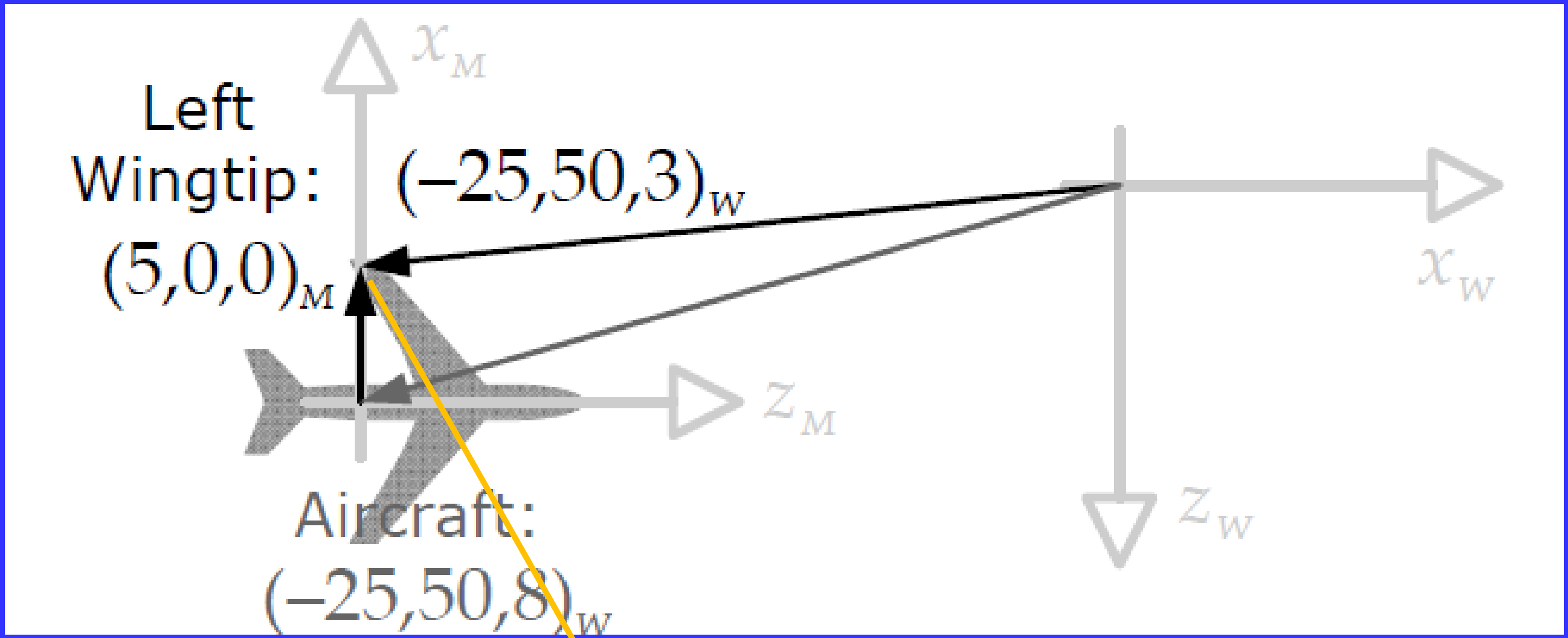
Origin of object space

Origin of world space



Model to World Space

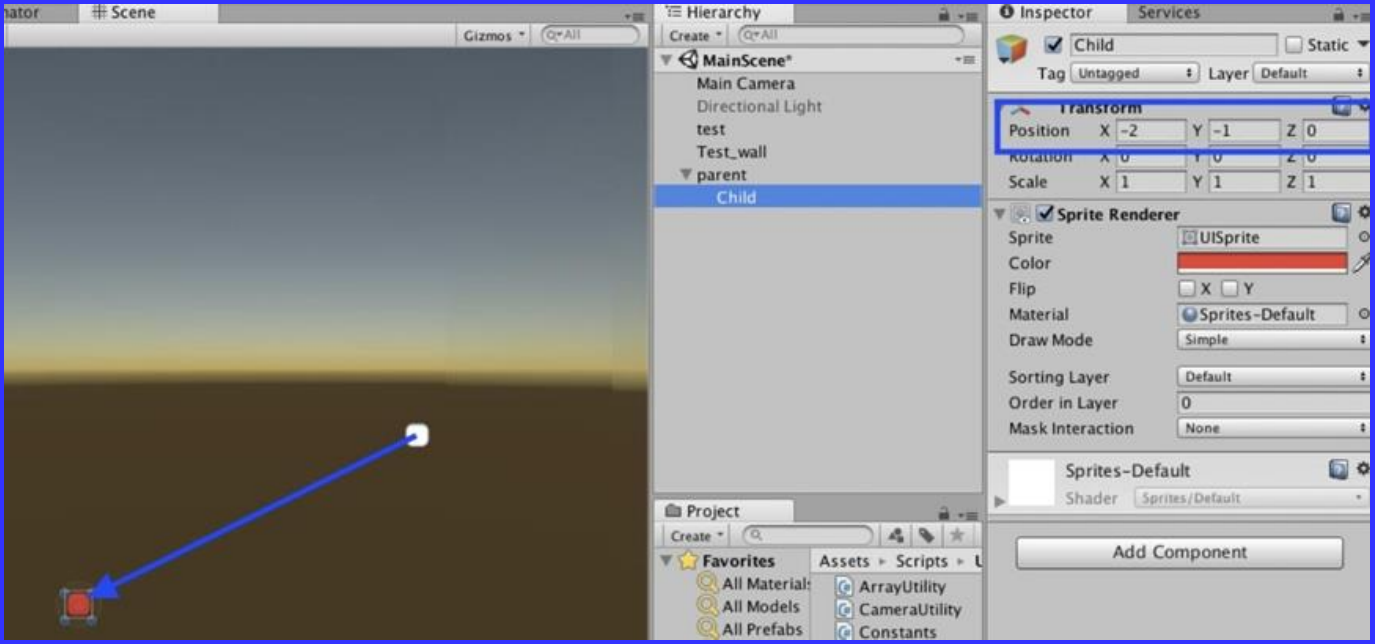
- A coordinate in model space can be converted to world space or vice versa



Wing tip is in (5,0,0) in Model space,
but in (-25,50,3) in World space

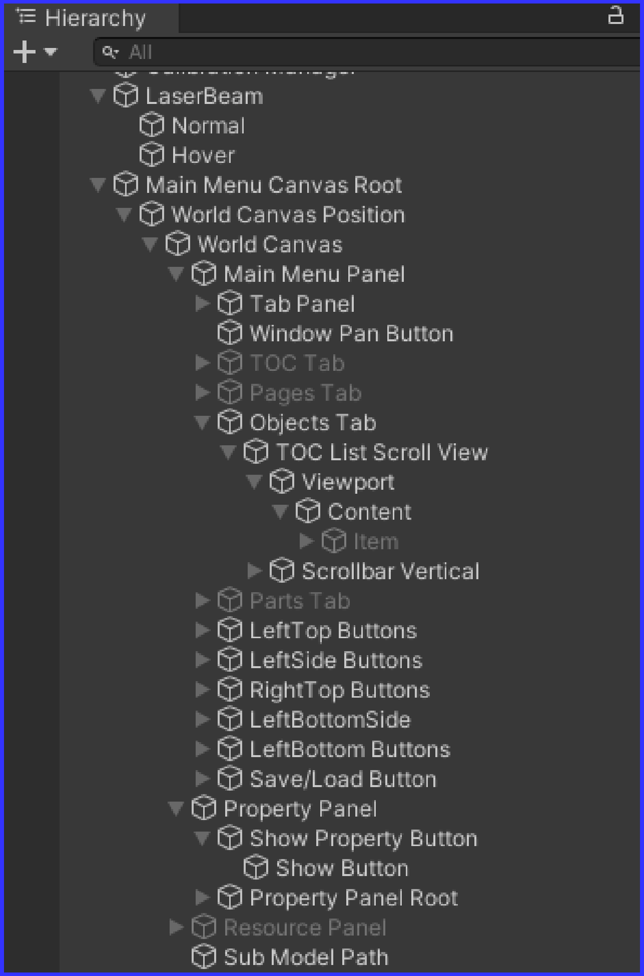
Coordinate Space Hierarchy

- A coordinate system that expresses its position based on its parent object.
- For example, assuming you have a character with an arm attached.
  - The vector that expresses the position of that arm is based on the coordinate system of its parent (shoulder).
- As you see from the example, the coordinates of the red box (child) are expressed as a vector from the white box (parent) position.



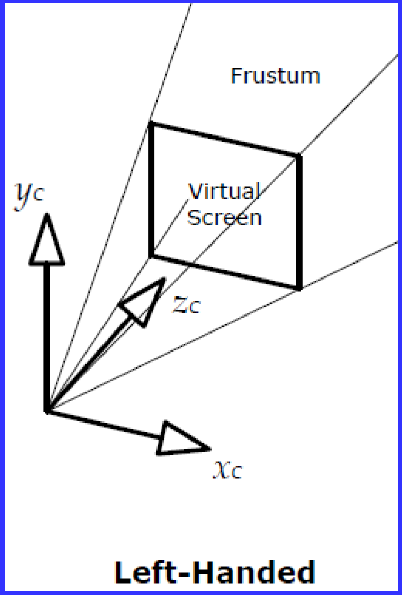
Coordinate Space Hierarchy

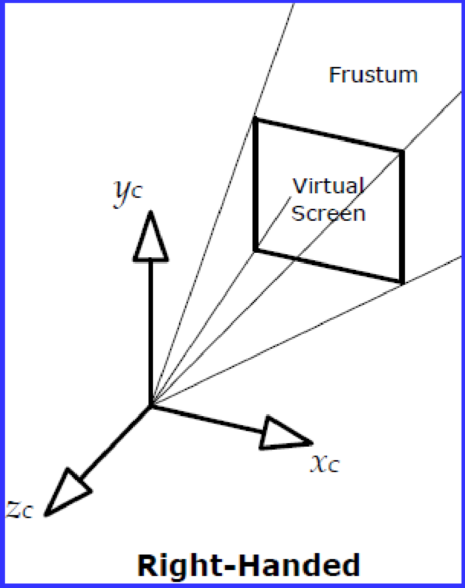
- In a coordinate space hierarchy, there may be multiple levels hierarchy like a tree



View / Camera Space

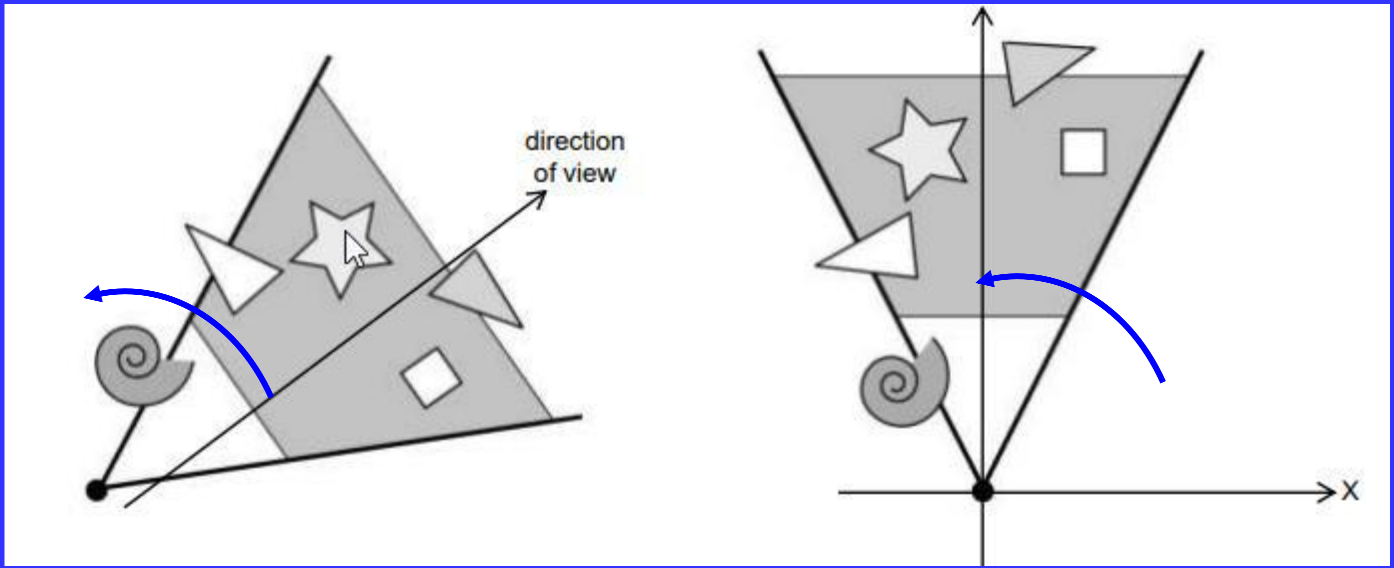
- The coordinate space that is associated with the observer
- View space is considered to be the origin and orientation of where we are looking at.
- The viewer's coordinate axis usually assume the positive axes pointing right and up, and in a left-handed coordinate system, the 3rd positive axis points forward
- In a right-handed coordinate system, the 3rd axis is reversed, so the negative axis points forward





World to View Space

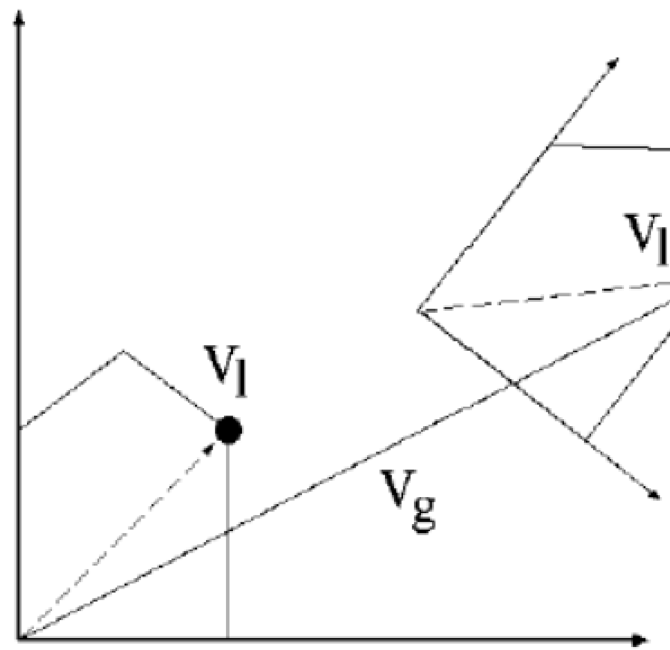
- To transform a world space into camera space
  - Apply inverse transform of view to objects
- First you move everything with an offset in inverse (negative) position of camera to move camera to origin
- Than rotate everything with a rotation in opposite (negative) direction of camera to align camera to axis



Transforming Between Different Coordinate Spaces

- Transformation matrices are used to convert location of vertices between different coordinate spaces

- $v_g = M v_l$
- $v_l = M^{-1} v^g$



Transforming Between Different Coordinate Spaces

- The transformation matrix converting from a child to its parent space is called $\mathbf{M}_{C\rightarrow P}$

$$\mathbf{M}_{C\rightarrow P} = \begin{bmatrix} \mathbf{i}_C & 0 \\ \mathbf{j}_C & 0 \\ \mathbf{k}_C & 0 \\ \mathbf{t}_C & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{i}_{Cx} & \mathbf{i}_{Cy} & \mathbf{i}_{Cz} & 0 \\ \mathbf{j}_{Cx} & \mathbf{j}_{Cy} & \mathbf{j}_{Cz} & 0 \\ \mathbf{k}_{Cx} & \mathbf{k}_{Cy} & \mathbf{k}_{Cz} & 0 \\ \mathbf{t}_{Cx} & \mathbf{t}_{Cy} & \mathbf{t}_{Cz} & 1 \end{bmatrix}$$

i_C unit x-axis basis vector of child in parent space

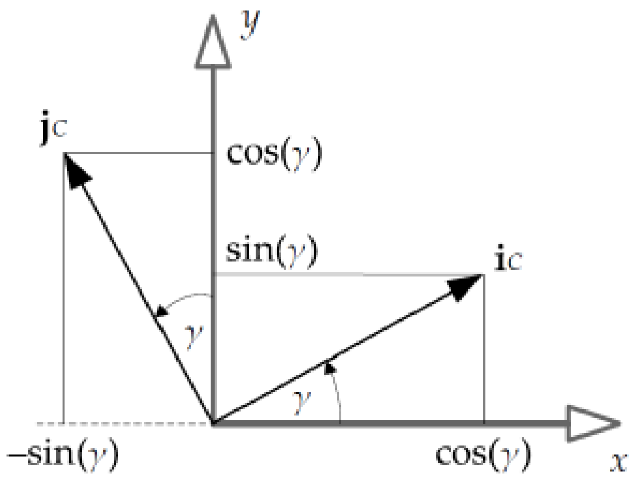
j_C unit y-axis basis vector of child in parent space

k_C unit z-axis basis vector of child in parent space

t_C translation of child relative to parent space

Transforming Between Different Coordinate Spaces

- A child space rotated by γ degree around Z axis



$$k_C = [0 \quad 0 \quad 1]$$

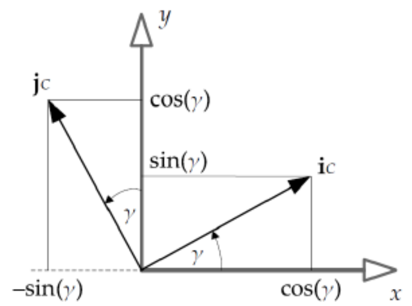
We can see that $i_C = [\cos\gamma \quad \sin\gamma \quad 0]$ and

$$j_C = [-\sin\gamma \quad \cos\gamma \quad 0]$$

Transforming Between Different Coordinate Spaces

- By puting these into our matrix we get local to world matrix

$k_C = [0 \quad 0 \quad 1]$



We can see that $i_C = [\cos\gamma \quad \sin\gamma \quad 0]$ and $j_C = [-\sin\gamma \quad \cos\gamma \quad 0]$

$$\mathbf{M}_{C\to P} = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = rotate_z(r, \gamma)$$

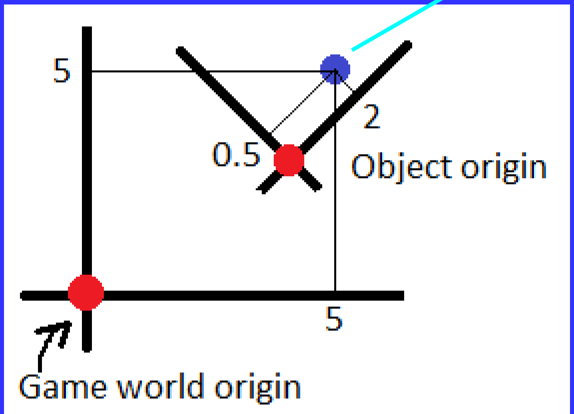
Transformations in Unity

- Do conversion with Respect to a GameObject's "Transform":
  - Local to World Transformations:
    - Transform.TransformPoint
    - Transform.TransformDirection
    - Transform.TransformVector
- Inverse computations are:
  - World to Local Transformations:
    - Transform.InverseTransformPoint
    - Transform. InverseTransformDirection
    - Transform. InverseTransformVector

TransformPoint in Unity

- **TransformPoint** transforms position from local space to world space.
- It is affected by local **position**, **rotation** and **scale** of game object that you call and also its parent game objects.

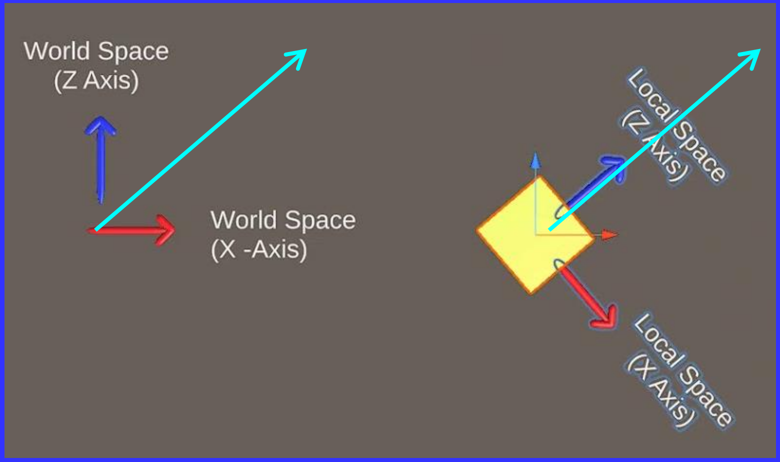
Local coordinate = (2, 0.5, 0)
World coordinate = (5, 5, 0)



TransformDirection in Unity

- **TransformDirection** is used to transform a direction from local space to world space.
- TransformDirection is not affected by position and scale. It is **only affected by rotation**
- And magnitude is preserved.

World direction = (0.7, 0, 0.7)   Local direction = (0, 0, 1)

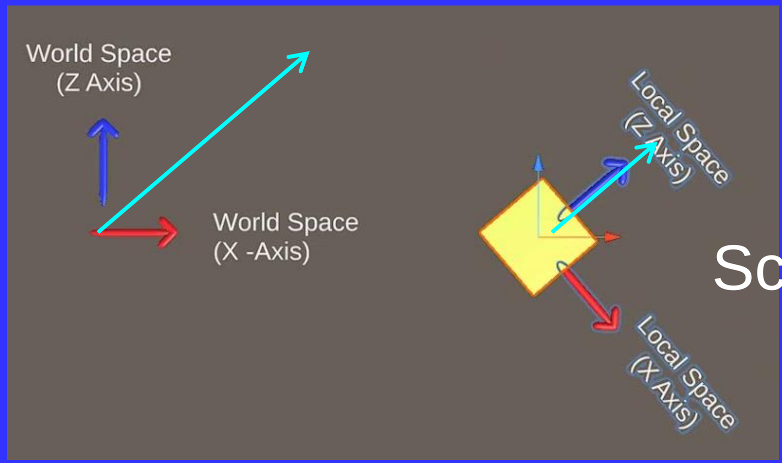


TransformVector in Unity

- **TransformVector** is used to transform a direction from local space to world space.
- **TransformVector** is not affected by position.
- But It is **affected by scale**
- And magnitude is changed.

World vector = (1.4, 0, 1.4)

Local vector = (0, 0, 1)



Scale = (2, 2, 2)
when scaled

TransformDirection Sample

```
RaycastHit hit;
// Does the ray intersect any objects excluding the player layer
if (Physics.Raycast(transform.position, transform.TransformDirection(Vector3.forward), out hit, Mathf.Infinity, layerMask))
{
    Debug.DrawRay(transform.position, transform.TransformDirection(Vector3.forward) * hit.distance, Color.yellow);
    Debug.Log("Did Hit");
}
```

Declaration

public Vector3 **TransformDirection**(Vector3 **direction**);

Description

Transforms `direction` from local space to world space.

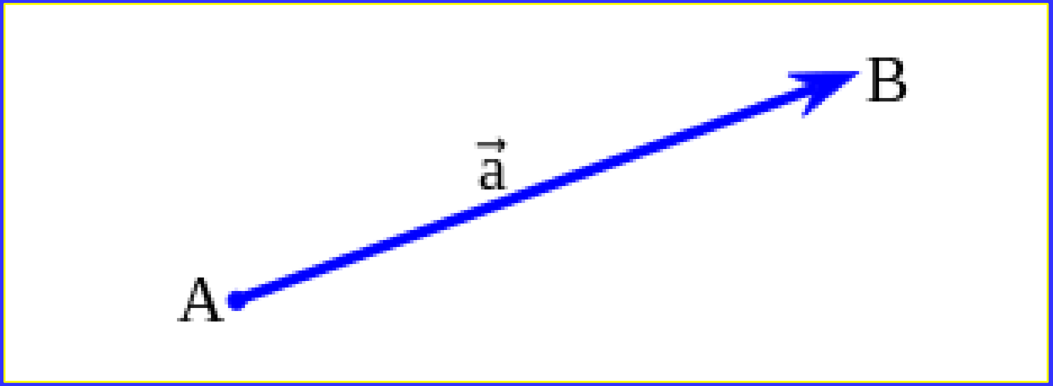
This operation is not affected by scale or position of the transform. The returned vector has the same length as `direction`.

You should use Transform.TransformPoint for the conversion if the vector represents a position rather than a direction.

What is a vector?

- A vector is a geometric object that has a magnitude (or length) and a direction.

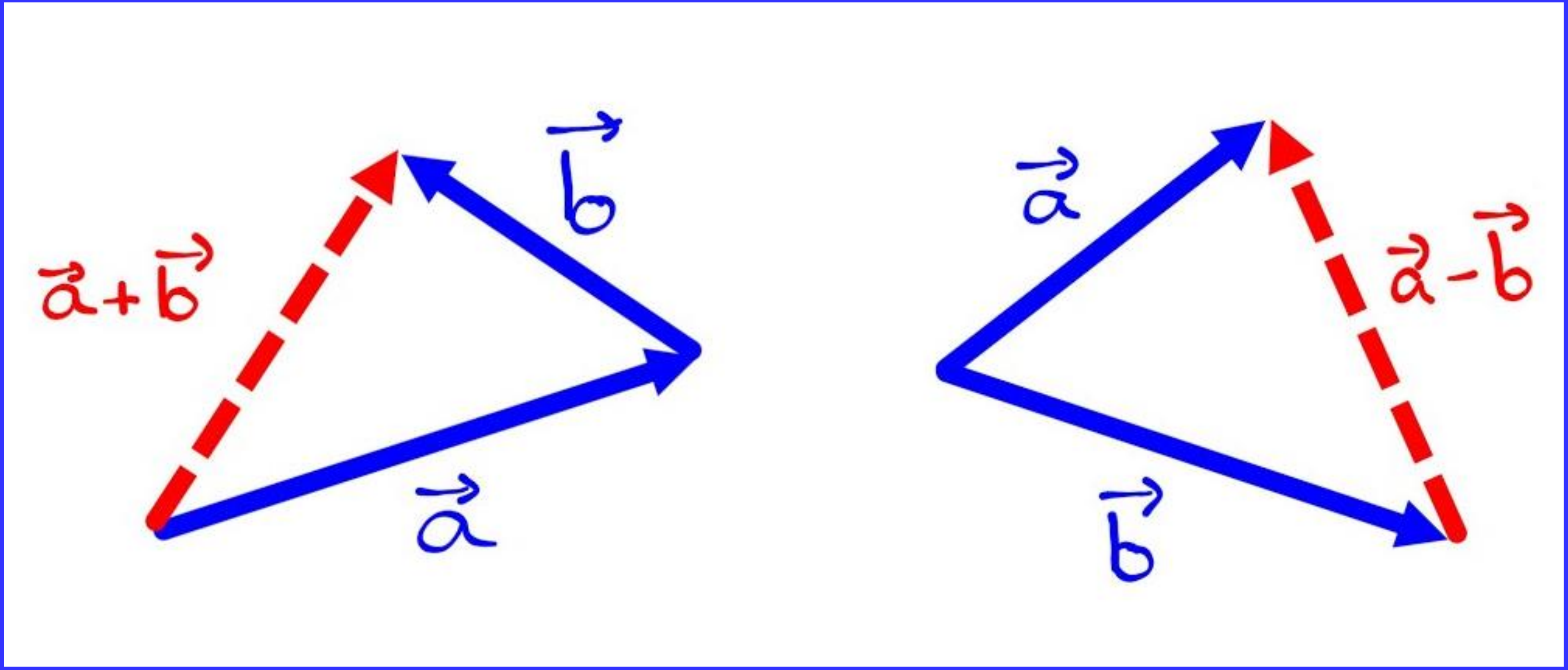
- Two vectors are said to be equal if they have the same magnitude and direction.
- Equivalently they will be equal if their coordinates are equal.



A vector pointing from *A* to *B*

Vector Addition & Subtraction

- A vector is a geometric object that has a magnitude (or length) and a direction.



Addition of a and b

Subtraction of b from a

Magnitude / Length of a Vector

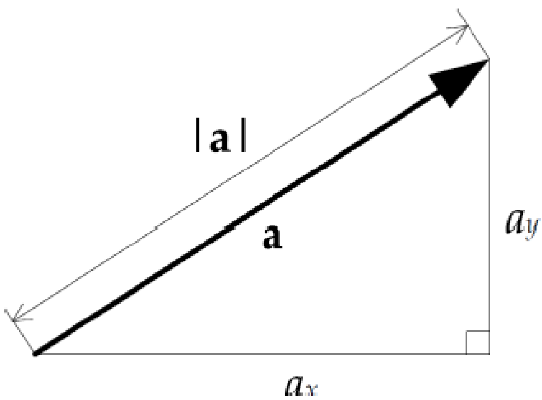
- The length of the vector a can be computed with the Euclidean norm,
  - Which is a consequence of the Pythagorean theorem.

$$\|\mathbf{a}\| = \sqrt{{a_1}^2 + {a_2}^2 + {a_3}^2}$$

$$\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$$

- This is equal to the square root of the dot product of the vector with itself

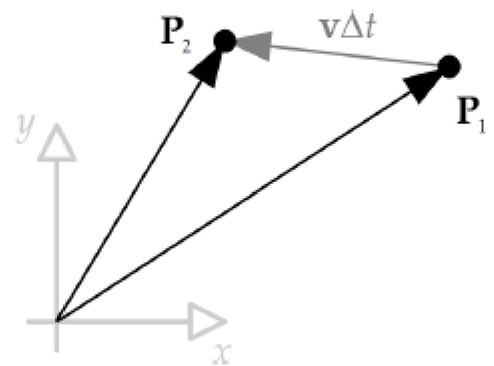
$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$



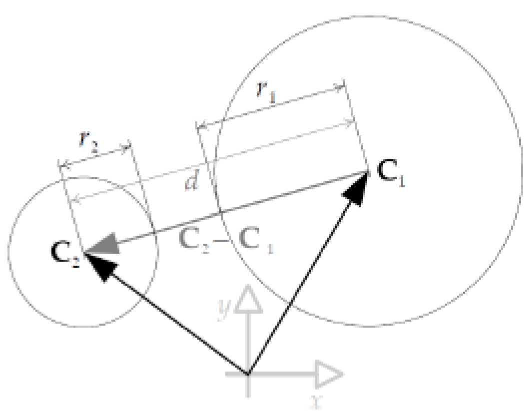
Use of Vector Operations

- Can be used in many places such as moving objects, collision tests, etc.

Moving objects
- $\mathbf{P}_2 = \mathbf{P}_1 + \mathbf{v}\Delta t$



Object collision
- if $d < r_1 + r_2$ then they collide
- Faster to compare $d^2 < (r_1 + r_2)^2$



Dot Product of Vectors

- A mathematical operation that can be performed on any two vectors with the same number of elements.
- The result is a scalar number equal to the magnitude of the first vector, times the magnitude of the second vector, times the cosine of the angle between the two vectors.

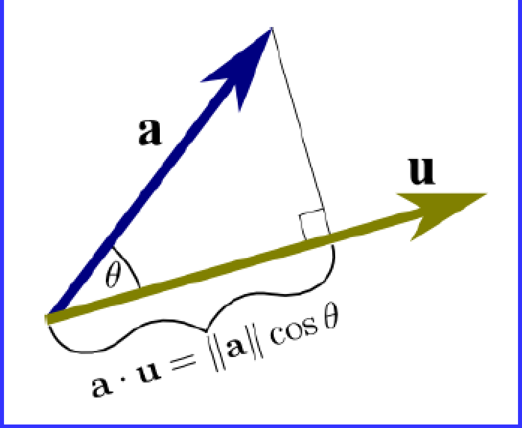
$$\mathbf{a} \cdot \mathbf{b} = |a||b| \cos(\theta)$$

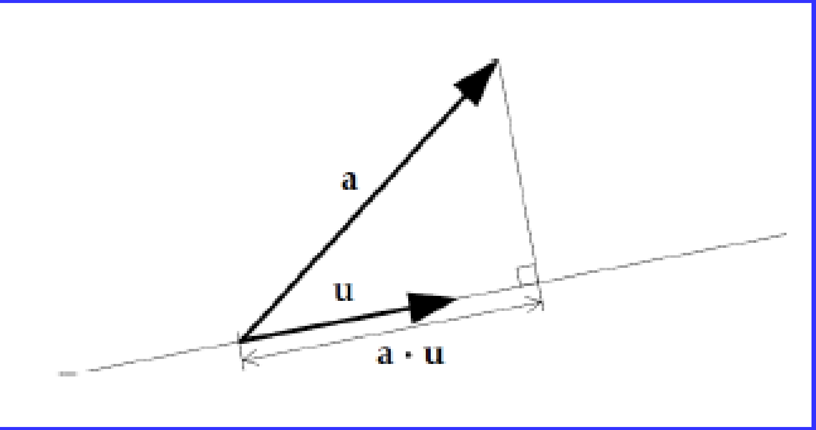
- Another way is to add components of the vector

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

Dot Product of Vectors

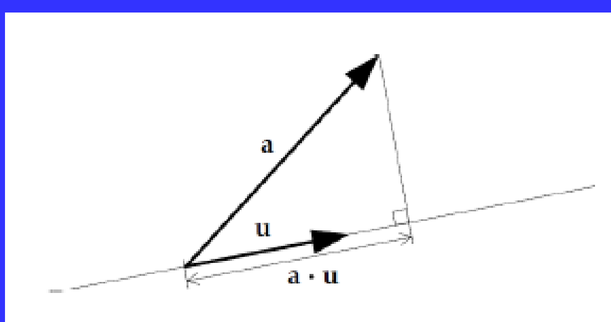
- Dot product is very commonly used for computing projection of one vector on to another vector
- If u is a unit vector (having length 1)
  - Then the dot product of a and u represents the length of the projection of a onto u
  - Or the amount that a is pointing in the same direction as unit vector u



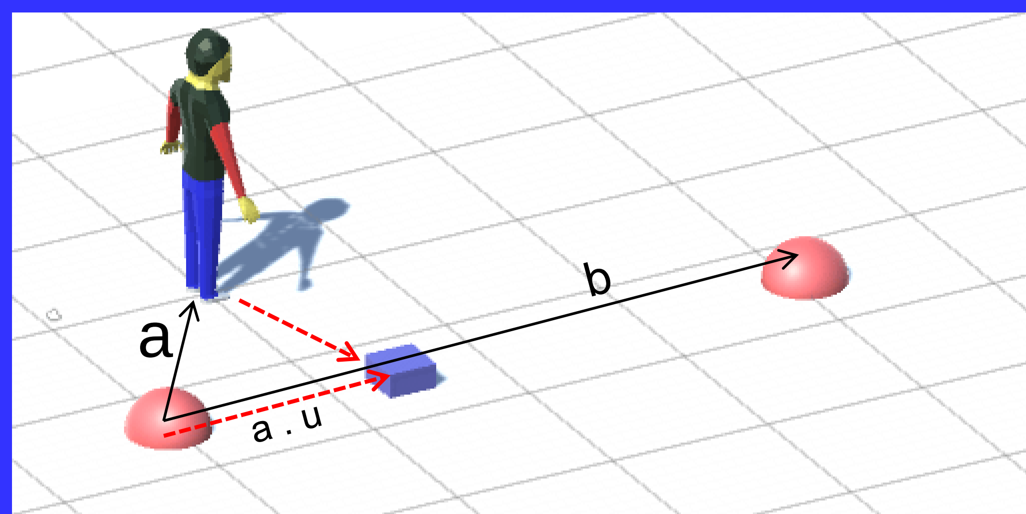


Use of Dot Product

- Use to tell if you passed a way point or not
- Need at least 2 waypoints: where you are going from and the waypoint you want to reach.
- Find the point on the line between the waypoints that's the closest to the character with the help of Vector Projection.
- For projection we need to normalize vector b to find unit vector u,
  - To find ratio of the way we completed we divide to length of b
- if it's above 1, then we know the character has passed the waypoint.

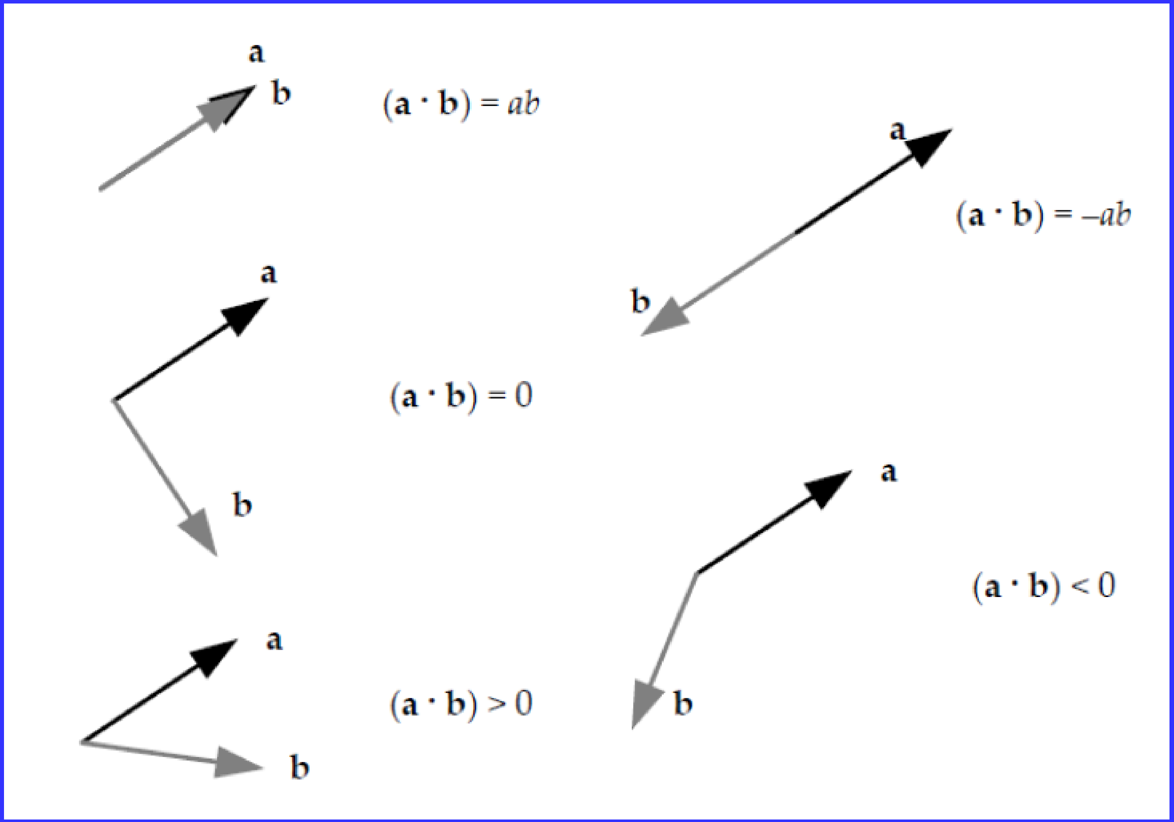


$$\mathbf{a} \cdot \mathbf{b} = \frac{|a||b|\cos(\theta)}{|b||b|}$$



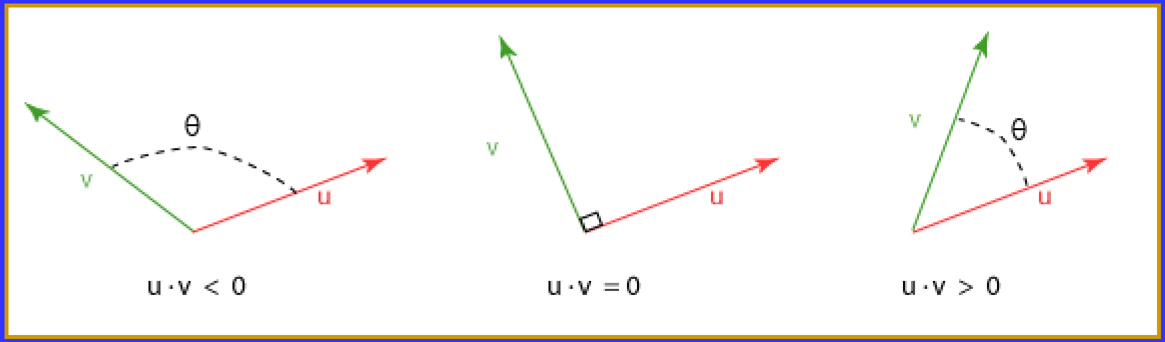
Use of Dot Product

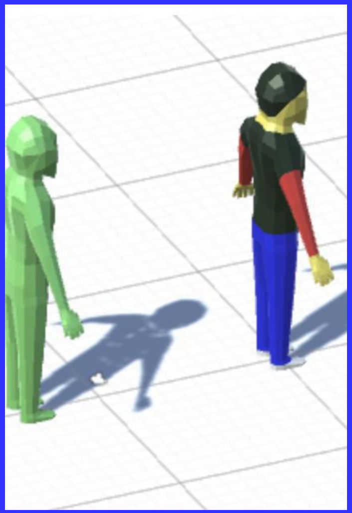
- Can be used to test different conditions such as being parallel forward, parallel opposite, perpendicular, same direction, inverse direction,...

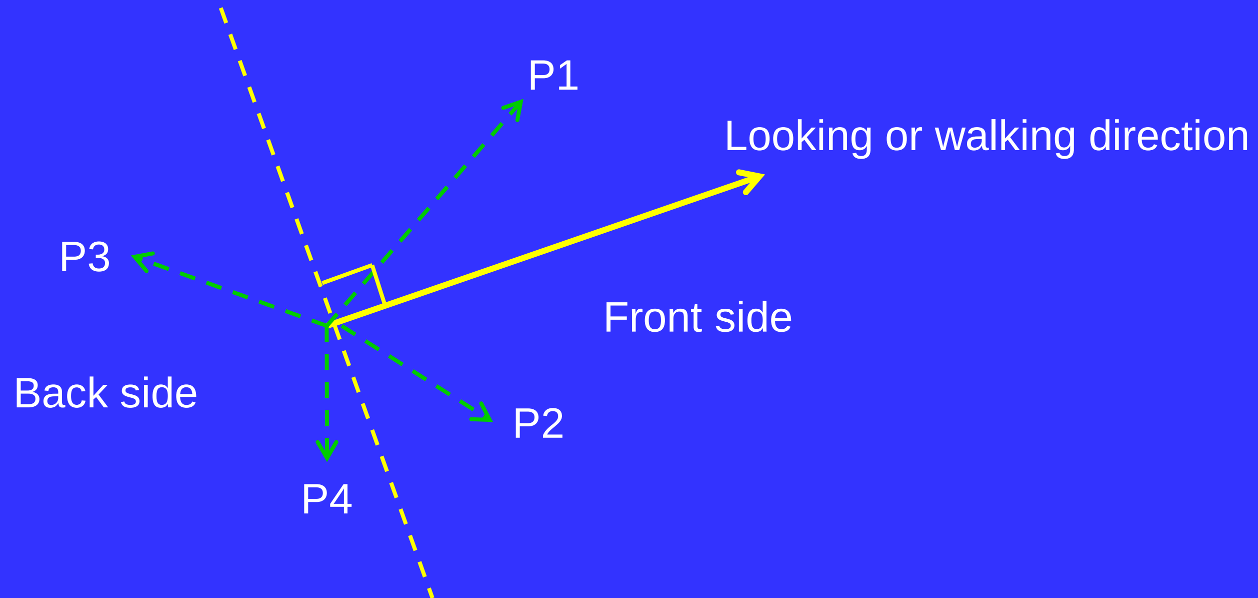


Use of Dot Product

- Can be used to understand if someone P is in front of you or behind you with respect to your face

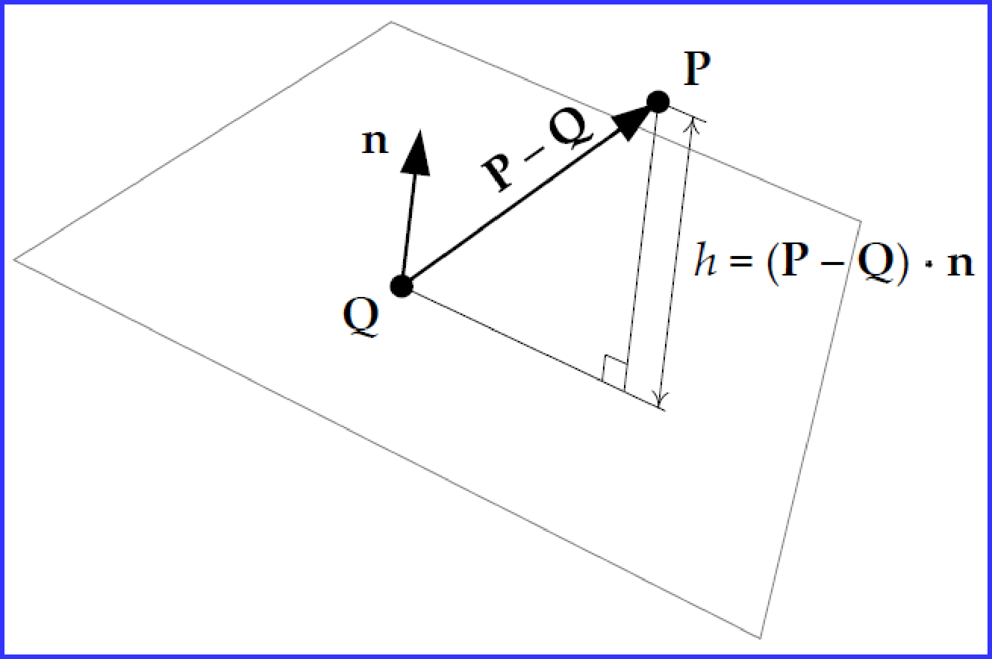






Use of Dot Product

- Can be used to calculate height of a point P on a plane
- If we define a plane as a point Q and a normal n
  - Then we can find the height h of a point P above the plane using projection

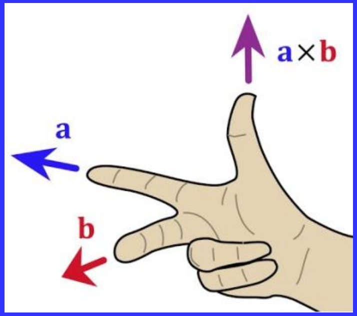


Cross Product of Vectors

- The cross product differs from the dot product primarily in that
  - The result of the cross product of two vectors is a vector again.
- The cross product, denoted a×b, is a vector
  - Perpendicular to both a and b and is defined as:

$$\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \sin(\theta) \, \mathbf{n}$$

right-handed coordinate system

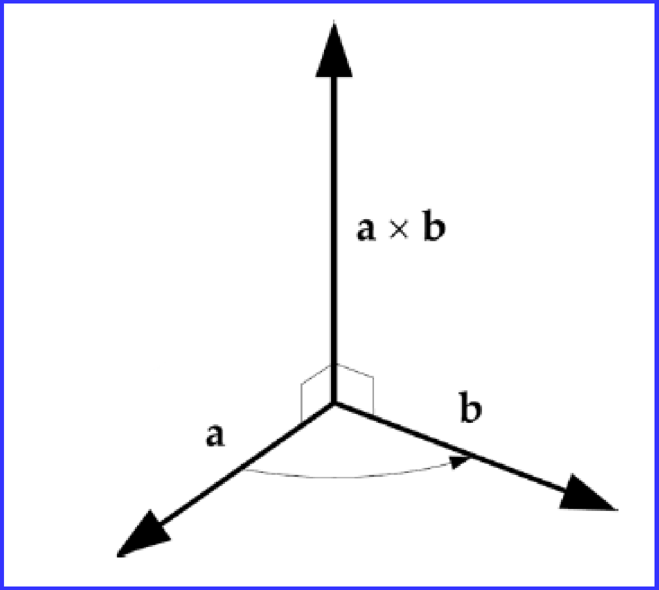


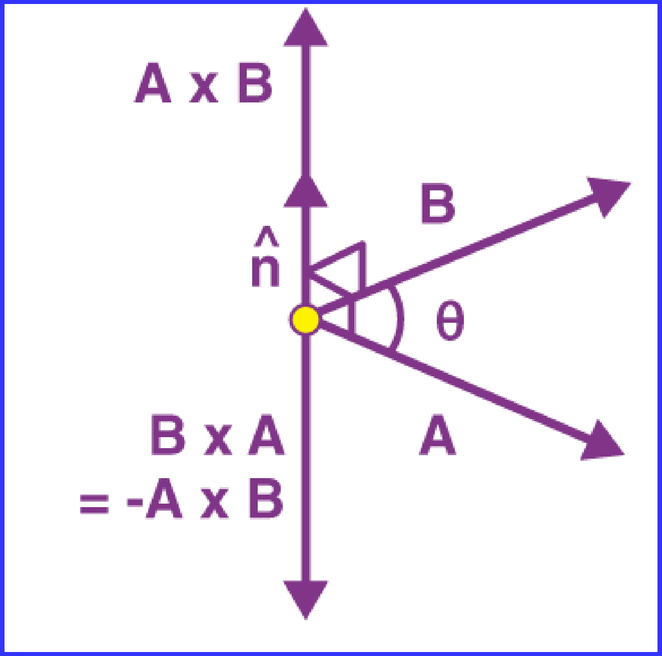
- Where θ is the measure of the angle between a and b,
- n is a unit vector perpendicular to both a and b
  - Like in figure in a right-handed coordinate system.
- If coordinate sistem is left handed, cross direction will be inverse

Cross Product of Vectors

- Results in another vector that is perpendicular to the vectors being multiplied
-

In a right-handed coordinate system





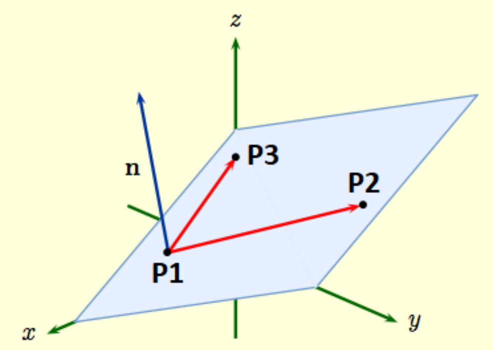
$$\mathbf{a} \times \mathbf{b} = [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)]$$

Use of Cross Product

- Finding a vector that is perpendicular to two other vectors
  - Finding the normal vector to a plane

$$\mathbf{n} = normalize\big((\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1)\big)$$

In a right-handed coordinate system



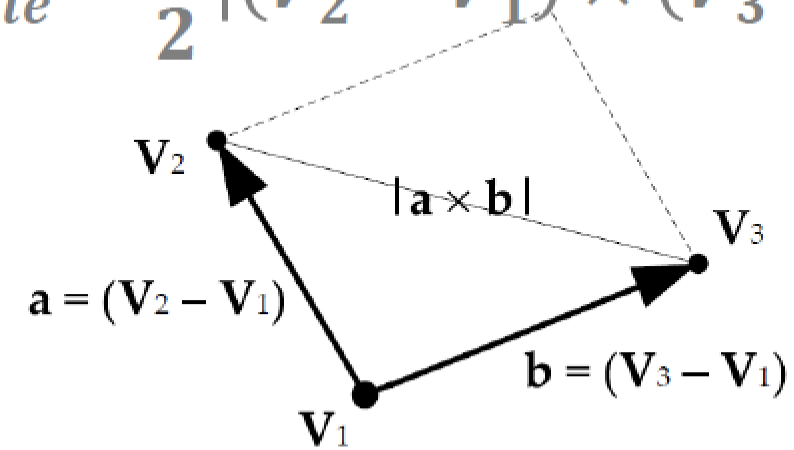
- Calculate torque
  - Given a force **F and a vector r from the center of mass the torque is**

$$\mathbf{N} = \mathbf{r} \times \mathbf{F}$$

Use of Cross Product

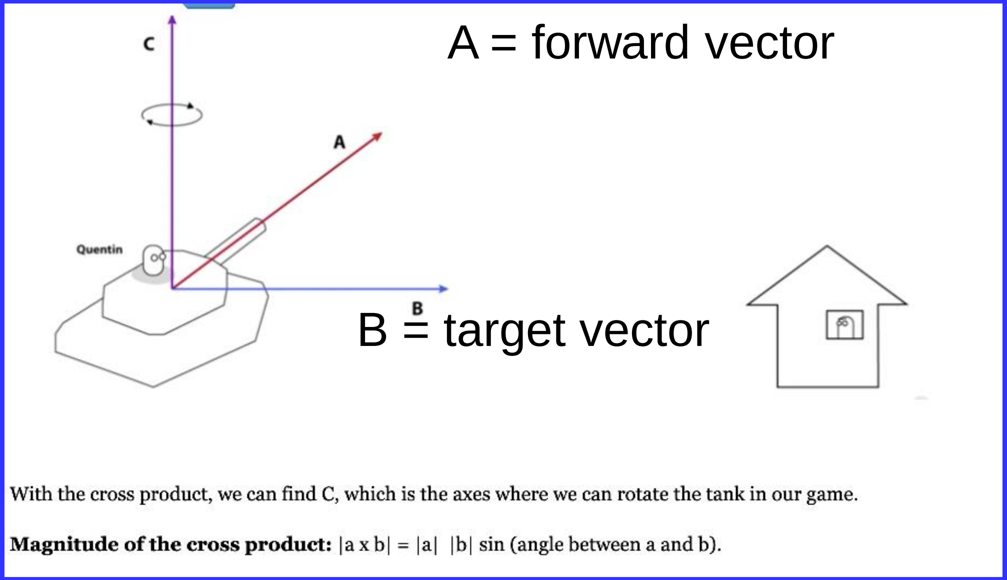
- Area of the triangle formed by the vertices can be computed by cross product
- Magnitude of the cross product is the area of the parallelogram formed by the two vectors

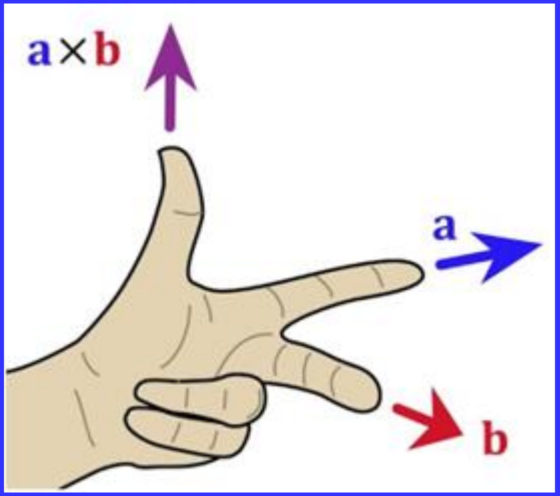
$$A_{triangle} = \frac{1}{2}\left|(V_2 - V_1) \times (V_3 - V_1)\right|$$



Use of Cross Product

- Using sign of the cross product you can find which direction to turn to move to a target point
- If direction of normal C (AxB) is up (left handed coordinate system, X axis to right)
  - Than B is on the right else B is on the left



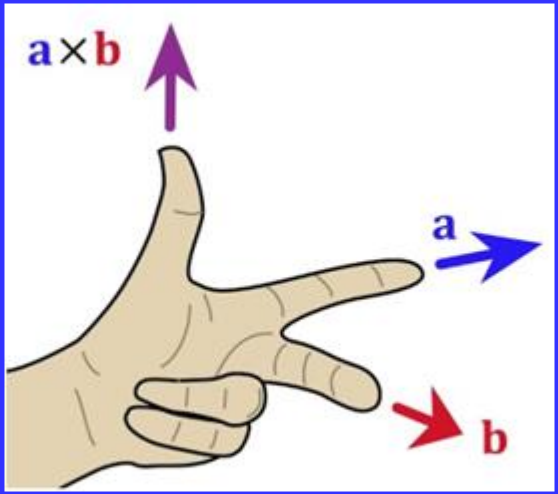


left-handed coordinate system

Use of Cross & Dot Product

- Which direction to turn?

```
using UnityEngine;
using System.Collections;

namespace LinearAlgebra
{
    //Figure out if you should turn left or right to reach a waypoint
    public class LeftOrRight : MonoBehaviour
    {
        public Transform youTrans;
        public Transform wayPointTrans;

        void Update()
        {
            //The direction you are facing
            Vector3 youDir = youTrans.forward;

            //The direction from you to the waypoint
            Vector3 waypointDir = wayPointTrans.position - youTrans.position;

            //The cross product between these vectors
            Vector3 crossProduct = Vector3.Cross(youDir, waypointDir);

            //The dot product between the your up vector and the cross product
            //This can be said to be a volume that can be negative
            float dotProduct = Vector3.Dot(crossProduct, youTrans.up);

            //Now we can decide if we should turn left or right
            if (dotProduct > 0f)
            {
                Debug.Log("Turn right");
            }
            else
            {
                Debug.Log("Turn left");
            }
        }
    }
}
```



left-handed coordinate system

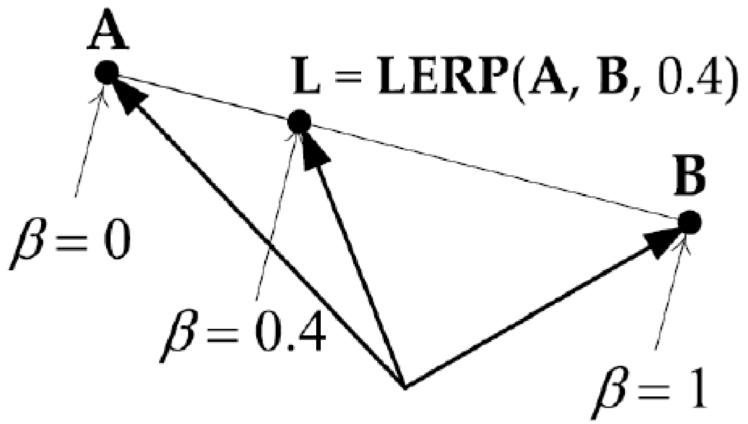
Interpolation Between Points

- The linear interpolation (LERP) is one of the most common operations used in game development.
- For example,
  - If we want to smoothly animate from point A to point B over the course of two seconds at 30 frames per seconds,
  - We would need to find 60 intermediate positions between A and B.
- A linear interpolation is a mathematical operation to find an intermediate point between two known points.

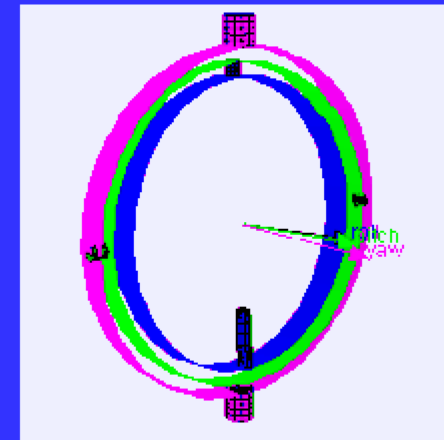
Interpolation Between Points

- Use LERP to find intermediate point L
  - A simple linear interpolation between 2 points
  - β ranges from 0 to 1
  - β = 0 is on A, β = 1 is on B

$$\mathbf{L} = \mathrm{LERP}(\mathbf{A}, \mathbf{B}, \beta) = (1 - \beta)\mathbf{A} + \beta\mathbf{B}$$
$$= \left[(1 - \beta)\mathbf{A}_x + \beta\mathbf{B}_x, (1 - \beta)\mathbf{A}_y + \beta\mathbf{B}_y, (1 - \beta)\mathbf{A}_z + \beta\mathbf{B}_z\right]$$

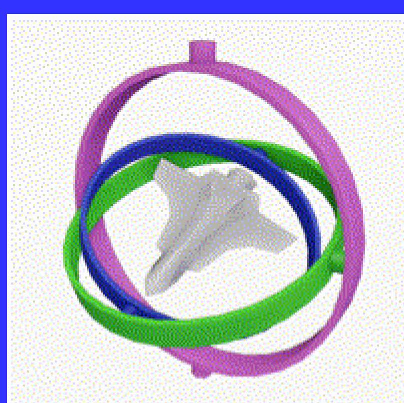


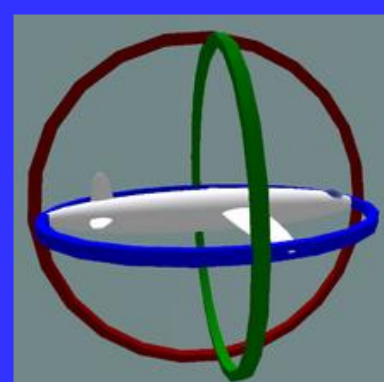
Gimbal Lock

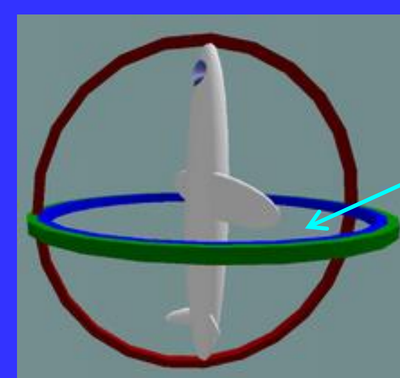


- Euler angles suffer from Gimbal Lock
  - Loss of one degree of freedom in 3D,
    - On a 3-gimbal mechanism
  - Occurs when the axes of two of the 3 gimbals become into a parallel configuration, "locking" the system into rotation in a degenerated 2D space.

When the pitch (green) and yaw (magenta) gimbals become aligned, changes to roll (blue) and yaw apply the same rotation to the airplane.







2 axes locked

you cannot rotate in 3 axes

Quaternions

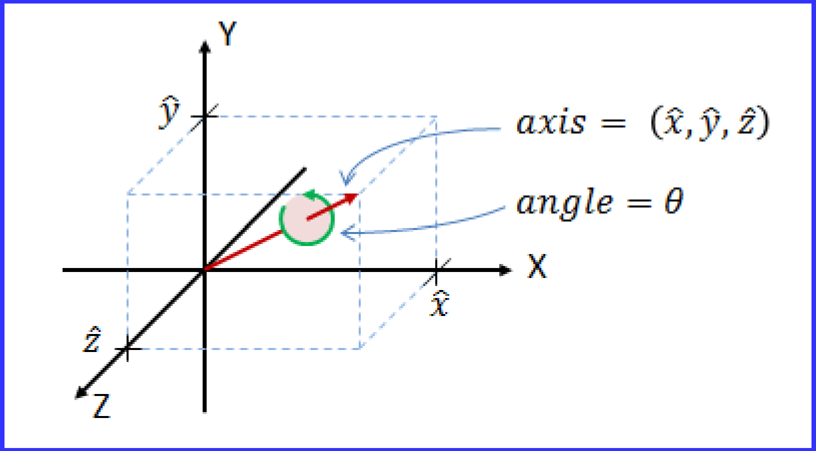
- Quaternion number system extends the complex numbers
- Used to define orientation of objects
- A better alternative to euler angles and solves Gimbal Lock
- Also more easy to interpolated between angles

- A quaternion is represented by four elements

$$\boldsymbol{q} = q_0 + \boldsymbol{i}q_1 + \boldsymbol{j}q_2 + \boldsymbol{k}q_3$$

Axis-Angle Representation

- Rotation Quaternions are closely related to the axis-angle representation of rotation.
- According to Euler's rotation theorem,
- Any 3D rotation can be specified with 2 parameters
  - A unit vector that defines an axis of rotation
  - An angle θ describing the magnitude of the rotation about that axis.

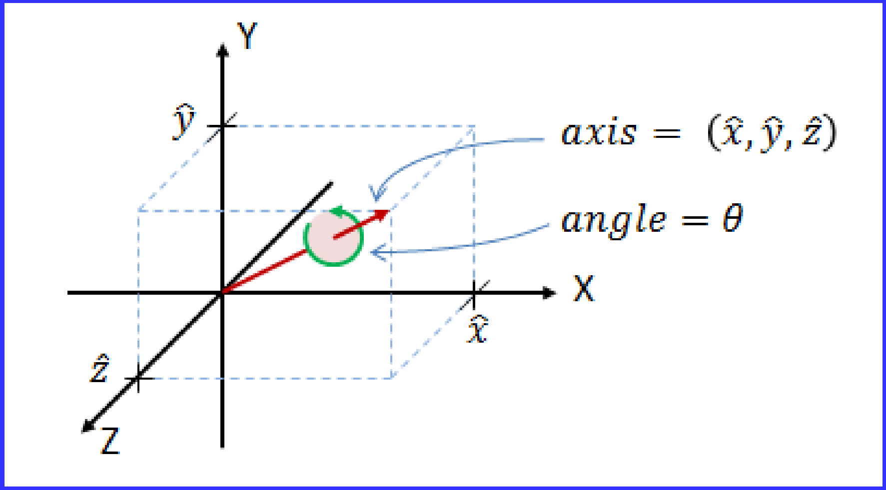


Axis-Angle Representation

- An axis-angle rotation can be represented by four numbers as:

  $(\theta, \hat{x}, \hat{y}, \hat{z})$

- where:
  - $(\hat{x}, \hat{y}, \hat{z})$ is unit vector defining the axis of rotation
  - θ is the amount of rotation around $(\hat{x}, \hat{y}, \hat{z})$



Quaternions

- A rotation quaternion is similar to the axis-angle representation.
- If we know the axis-angle components (θ, $\hat{x}$, $\hat{y}$, $\hat{z}$),
  - We can convert to a rotation quaternion q as follows:

magnitude of a rotation quaternion (the sum of the squares of all 4 components) is always equal to 1

$$\boldsymbol{q} = (q_0, q_1, q_2, q_3)$$

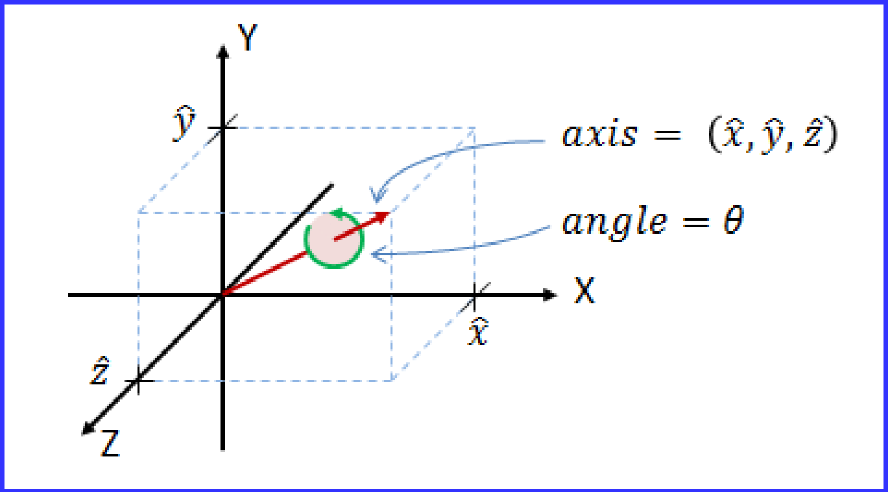
$$q_0 = \cos\left(\frac{\theta}{2}\right)$$

$$q_1 = \hat{x}\sin\left(\frac{\theta}{2}\right)$$

$$q_2 = \hat{y}\sin\left(\frac{\theta}{2}\right)$$

$$q_3 = \hat{z}\sin\left(\frac{\theta}{2}\right)$$

($\hat{x}$, $\hat{y}$, $\hat{z}$) related



Quaternions

- Since axis-angle and quaternion representations contain exactly the same information,
- We may ask why we would bother with quaternion?
- The answer is that to do anything useful with an axis-angle quantity such as rotate a set of points
- Have to perform these trigonometric operations anyway.
- Performing beforehand means that most quaternion operations can be accomplished using only multiplication/division and addition/subtraction
- So saving valuable computer performance.

Convert Quaternion to Axis-Angle

- Given the quaternion $\boldsymbol{q} = (q_0, q_1, q_2, q_3)$
- We can convert back to an axis-angle representation as follows.
- First, we find the rotation angle from q_0:

$$\theta = 2\cos^{-1}(q_0)$$

- If θ is not zero,
  – we can then find the rotation axis unit vector as follows:

$$(\hat{x}, \hat{y}, \hat{z}) = \left(\frac{q_1}{\sin\left(\frac{\theta}{2}\right)}, \frac{q_2}{\sin\left(\frac{\theta}{2}\right)}, \frac{q_3}{\sin\left(\frac{\theta}{2}\right)}\right)$$

Convert Quaternion to Axis-Angle

- One special case in which equation will fail.
- A quaternion with the value ***q*** = (1,0,0,0) is known as the ***identity quaternion***, and will produce no rotation.
- rotation angle (θ) will be zero
- Equation will generate a divide-by-zero error.
- Need to test whether q_0 equals 1.0
  - In case, set $\theta = 0$, and $(\hat{x}, \hat{y}, \hat{z}) = (1, 0, 0)$.

$$\theta = 2\cos^{-1}(q_0)$$

$$(\hat{x}, \hat{y}, \hat{z}) = \left(\frac{q_1}{\sin\left(\frac{\theta}{2}\right)}, \frac{q_2}{\sin\left(\frac{\theta}{2}\right)}, \frac{q_3}{\sin\left(\frac{\theta}{2}\right)}\right)$$

https://danceswithcode.net/engineeringnotes/quaternions/quaternions.html

Properties of Rot. Quaternions

- A quaternion is a "unit" quaternion if $|q| = 1$.
- All rotation quaternions must be unit quaternions

- The quaternion $\boldsymbol{q} = (1, 0, 0, 0)$ is the ***identity quaternion***. It represents no rotation.

- Inverse of a quaternion is $q^* = (q_0, -q_1, -q_2, -q_3)$

Properties of Rot. Quaternions

- Any given rotation has 2 possible quaternion representations.
  - If one is known, the other is negative of all 4 terms, reversing both the rotation angle and the axis of rotation.
  - if $\boldsymbol{q}$ is a rotation quaternion, $\boldsymbol{q}$ and $-\boldsymbol{q}$ will produce the same rotation.
- Quaternion multiplication is associative: $(\boldsymbol{ab})\boldsymbol{c} = \boldsymbol{a}(\boldsymbol{bc})$
- Quaternion multiplication is not commutative:

  $\boldsymbol{ab} \neq \boldsymbol{ba}$