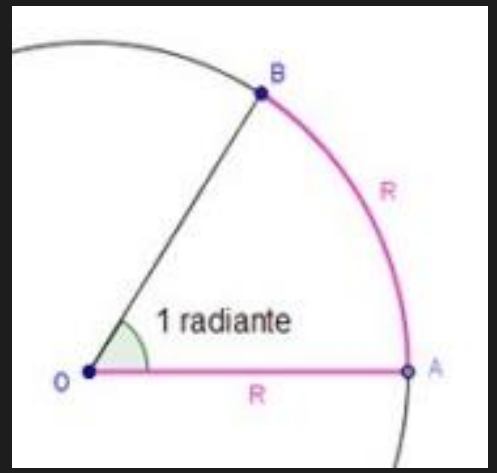
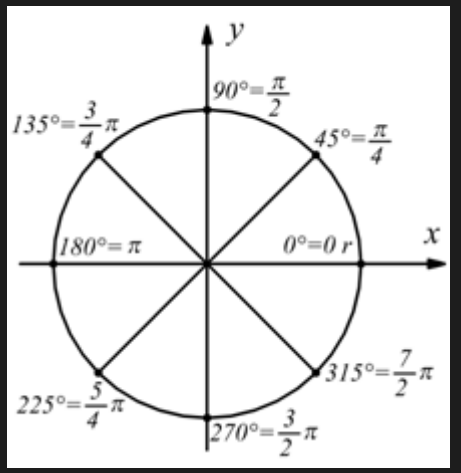
Sin/Cos

- Degrees/Radiants
- Sin/cos



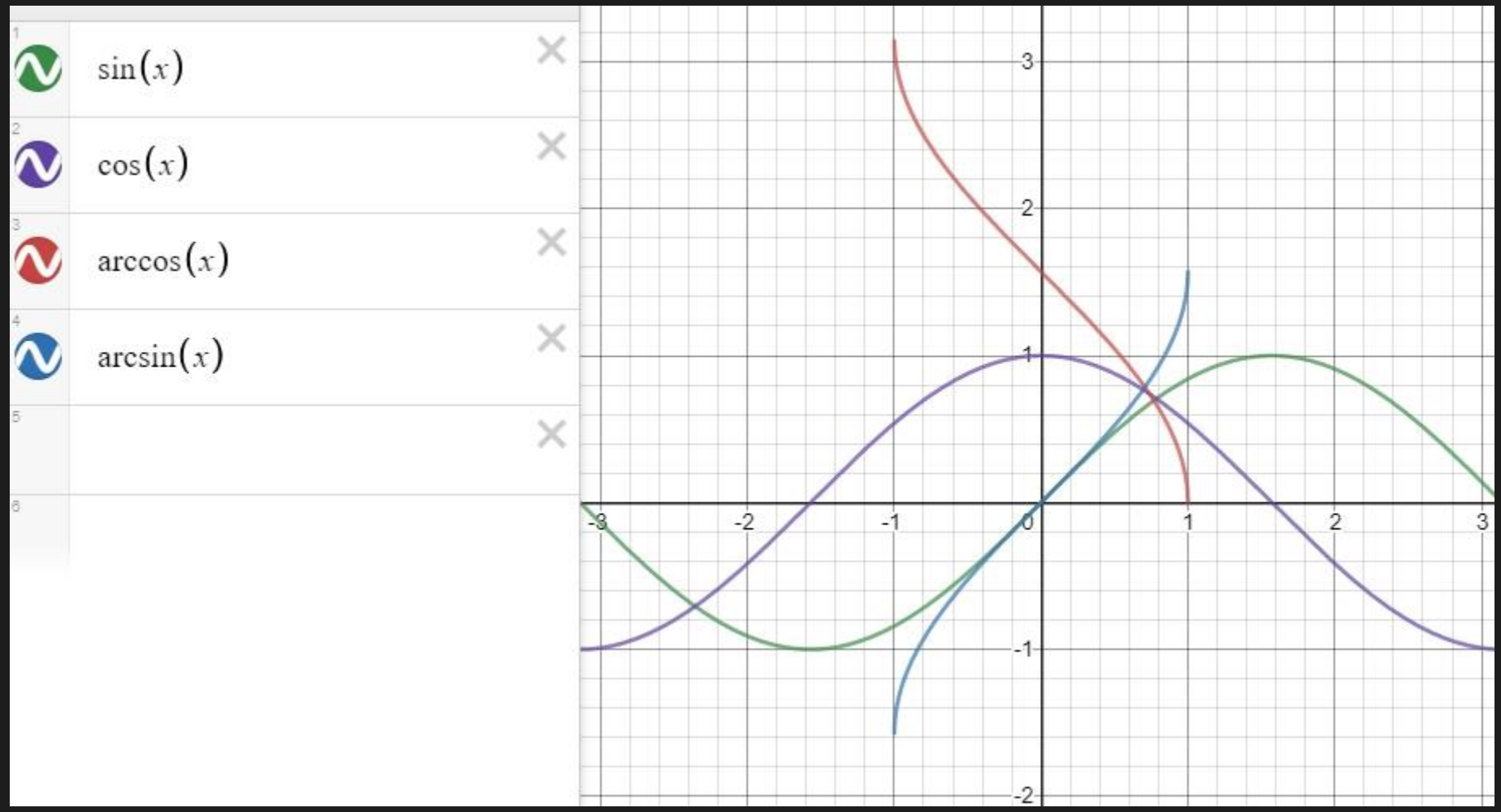
$$g^\circ = \frac{r \text{ rad} \times 180^\circ}{\pi \text{ rad}}$$

$$r \text{ rad} = \frac{g^\circ \times \pi \text{ rad}}{180^\circ}$$



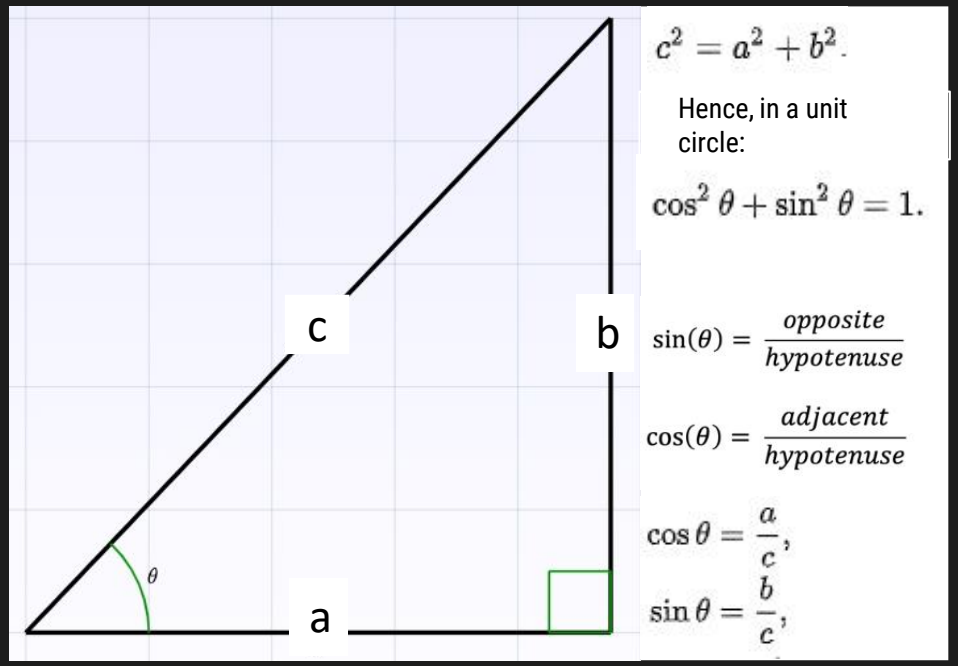
ArcSin/Cos

- Mathf.asin/acos()



Triangles

- Pythagorean theorem
- The sum of all triangles angles is 180
- Law of Sines
  - How to resolve a triangle if you have a, alfa, b => find beta => the sum of alfa, beta, gamma is 180 => find gamma => find c
- Law of Cosines



$$c^2 = a^2 + b^2.$$

Hence, in a unit circle:

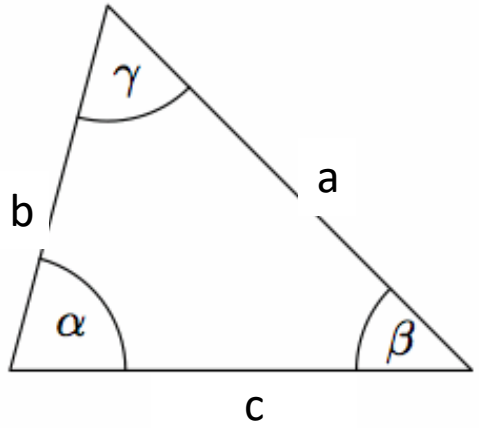
$$\cos^2\theta + \sin^2\theta = 1.$$

$$\sin(\theta) = \frac{opposite}{hypotenuse}$$

$$\cos(\theta) = \frac{adjacent}{hypotenuse}$$

$$\cos\theta = \frac{a}{c},$$

$$\sin\theta = \frac{b}{c},$$



Regola dei seni
(per trovare i lati)

$$\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma}$$

Regola dei seni
(per trovare gli angoli)

$$\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c}$$

Regola dei coseni
(per trovare i lati)

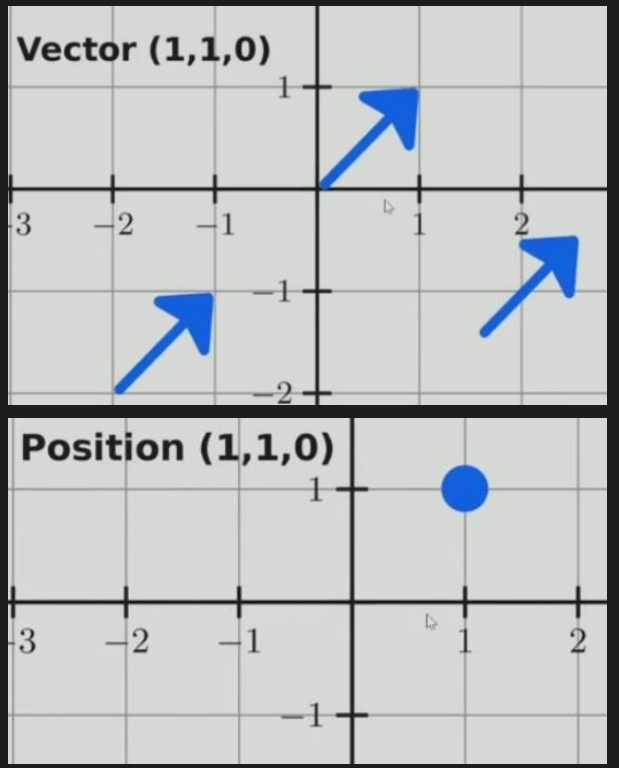
$$a^2 = b^2 + c^2 - 2bc\cos\alpha$$

Regola dei coseni
(per trovare gli angoli)

$$\cos\alpha = \frac{b^2+c^2-a^2}{2bc}$$

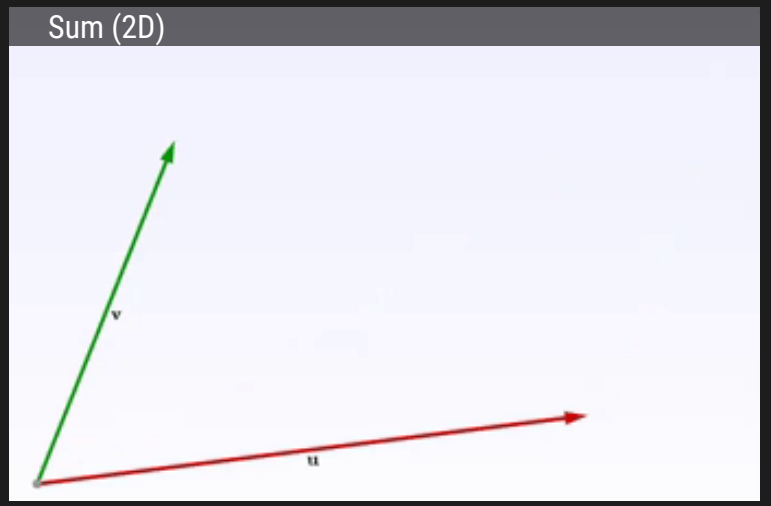
Vector math

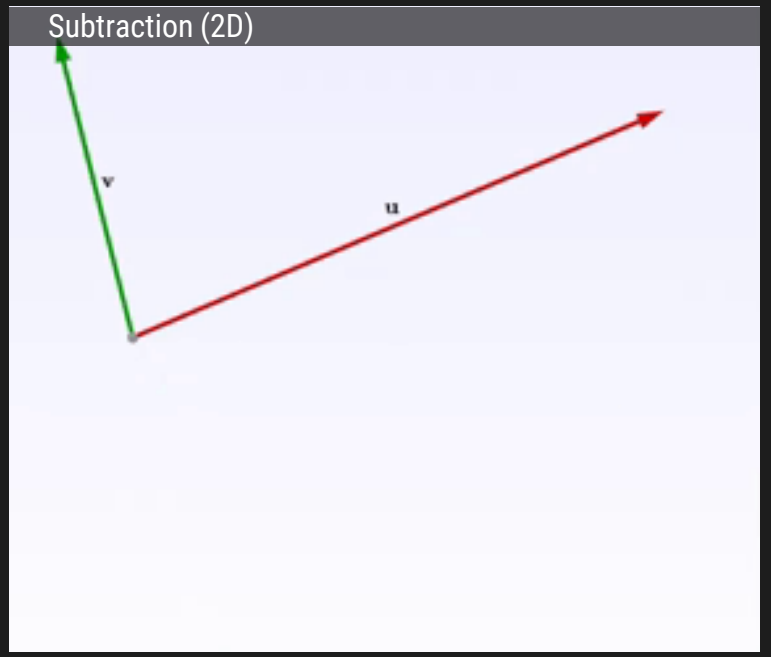
- A Vector is defined by:
  - Direction
  - Length (Magnitude)
- Easily confused with a position, it is NOT a position in Math
- But, in cg, we use to refer an obj pos with a Vector3: this time there is only ONE vector for position p: the one that starts from the origin

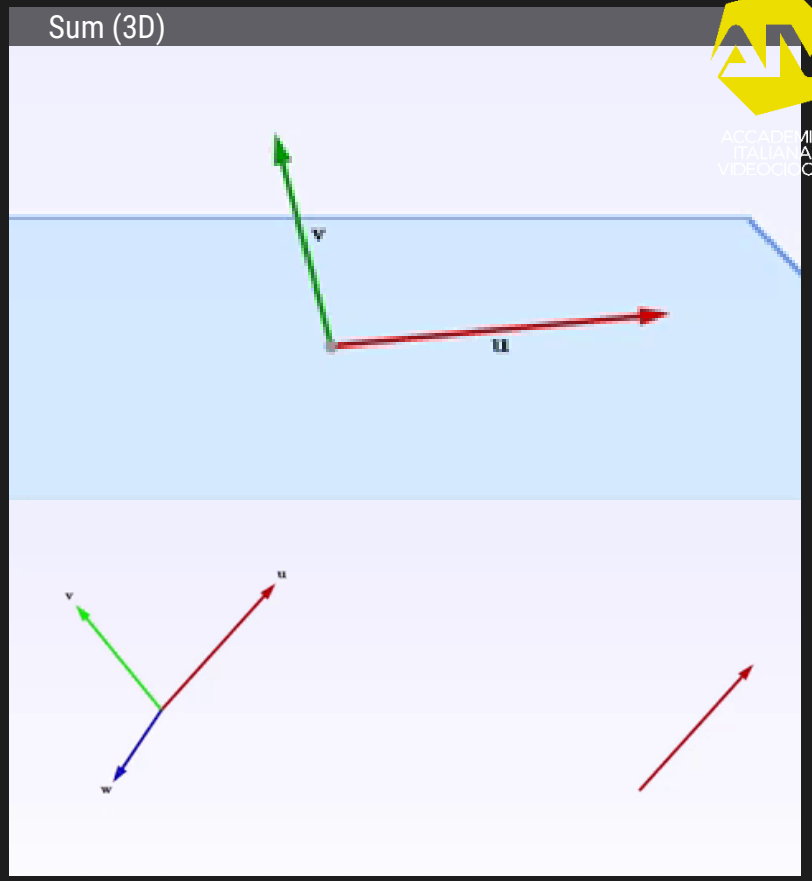


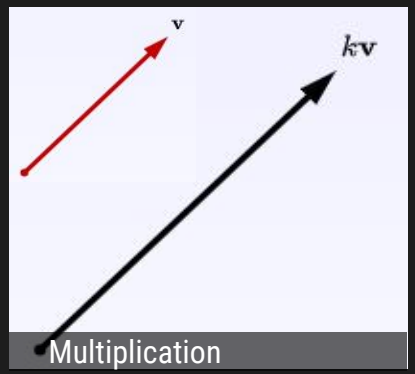
Vector math

- Sum, Mul, Sub
- Associativity
- To know vector C from A to B
  - use difference: C = B – A



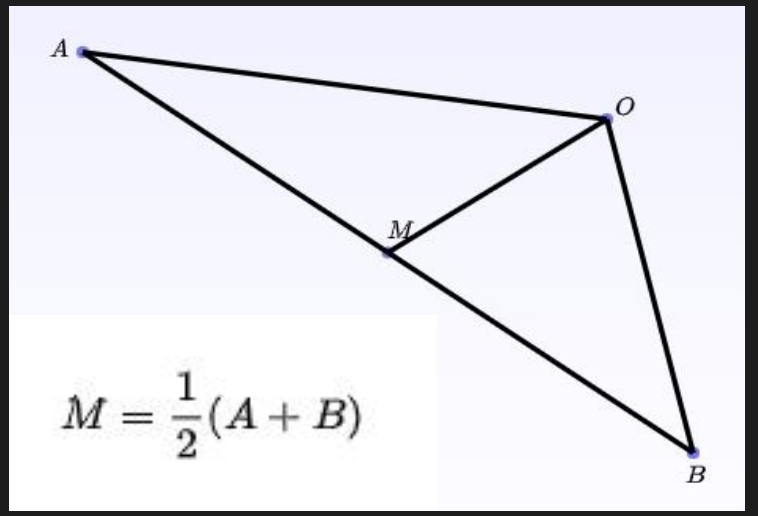


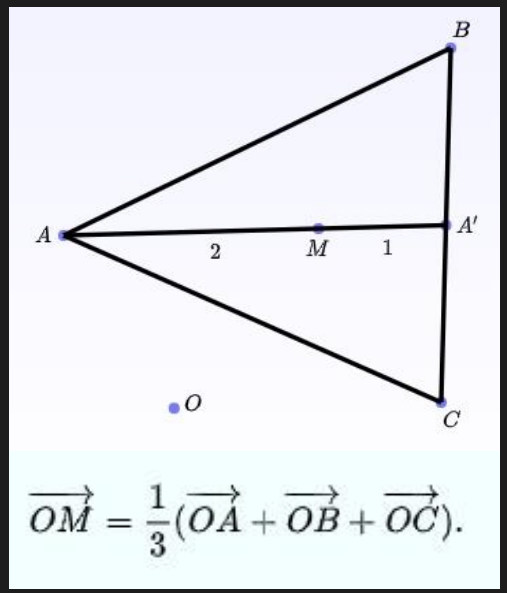




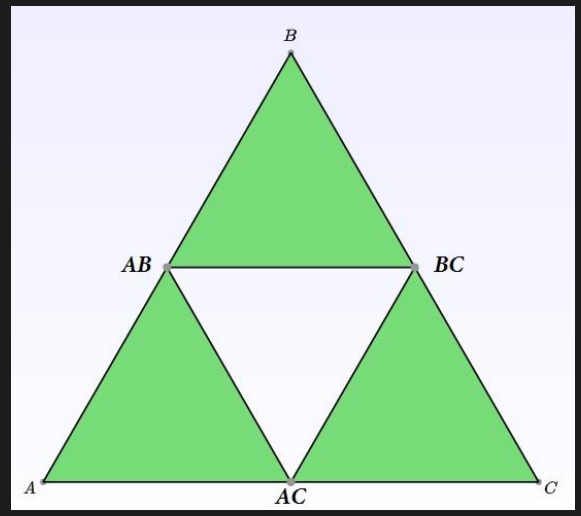
Vector math

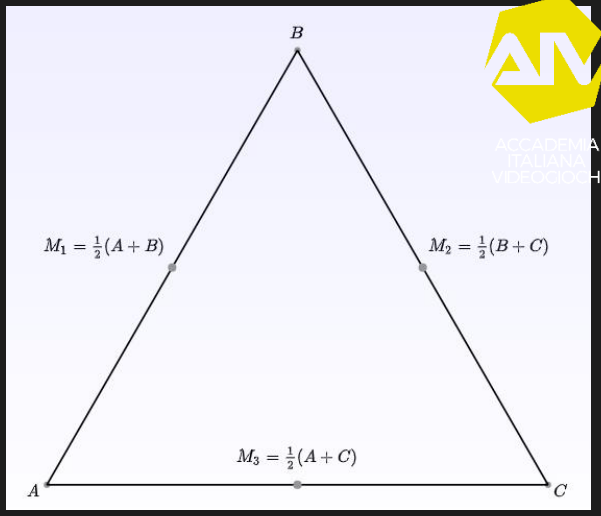
- Middle Point
- Center of mass





Ex // Sierpinski Triangle with center of mass





Put 6 3D Objs in the Scene Hierarchy:

- A [0,0,0], B[3,0,0], C[1.5,0,3]
- AB, BC, CA [at the origin: they will be placed during runtime]

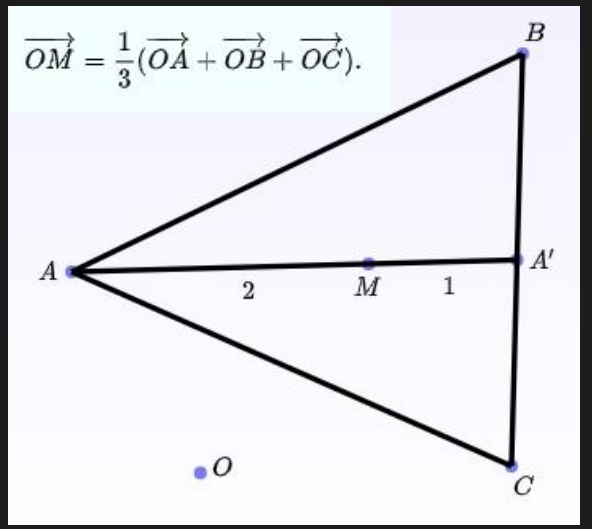
During Runtime

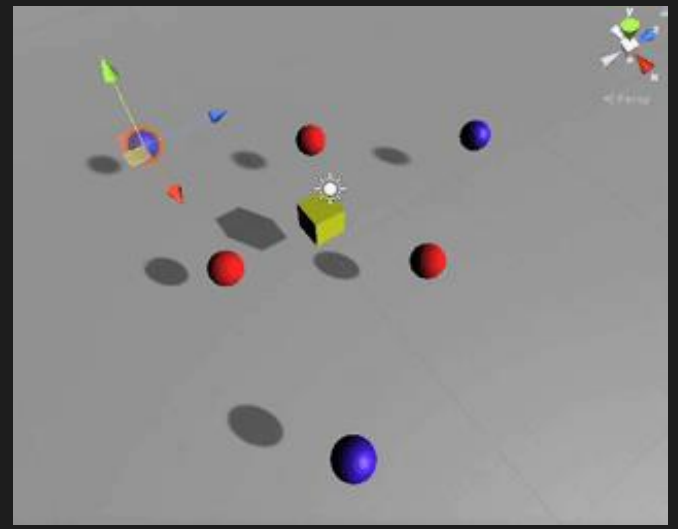
- You can move manually (in SceneView) A, B, C objs, while AB, BC, CA objs will change their position automatically
- AB will always be between A and B (M1)
- BC will always be between B and C (M2)
- CA will always be between C and A (M3)
- Create [StayBetween2Objs.cs] and use it to update AB, BC, CA positions

| Add Obj D. During Runtime

- | D will be always in the CenterOfMass of A,B,C
- | Create [StayOnCOM.cs] and use it to update D position

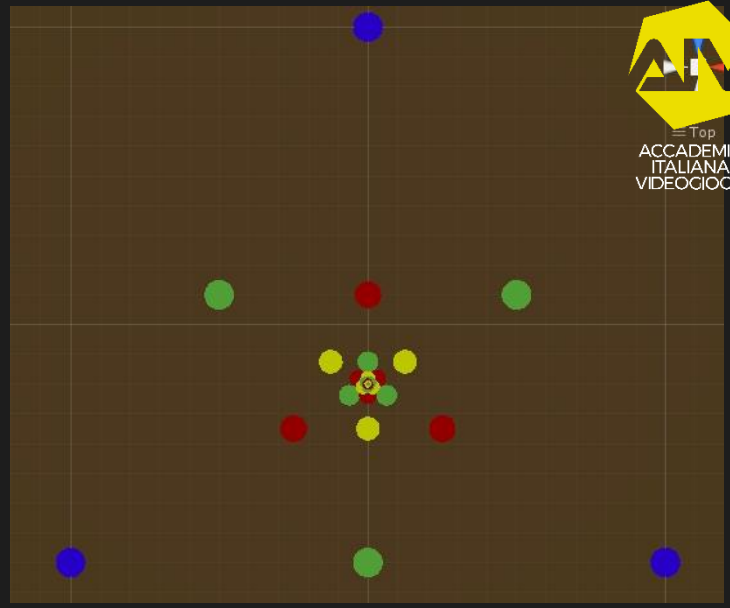
[Sierpinski_Triangle_End_00, StayBetween2Objs.cs, StayOnCOM.cs]

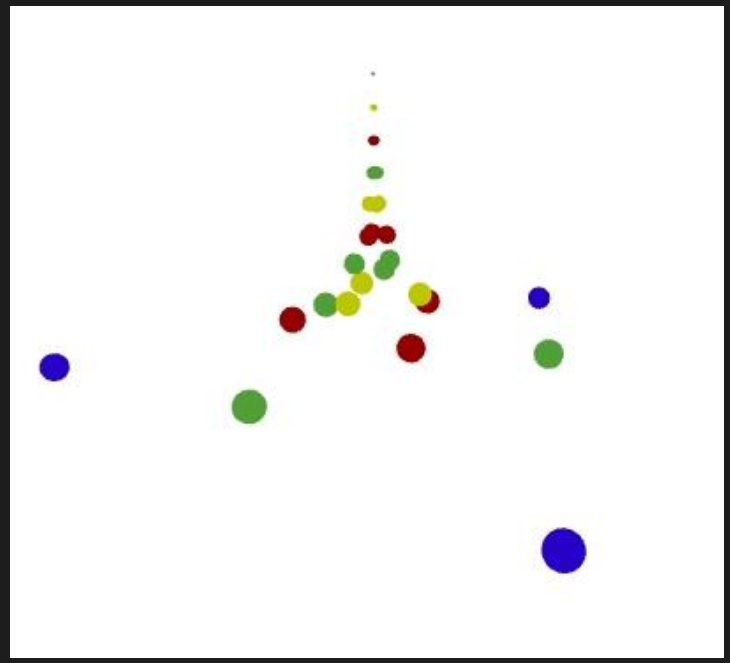




Ex // Iterative Sierpinski Triangle

- Iterate Sierpinski triangle construction for n times
- Start with 3 objects, place them on the triangle corners
- Each triangle is inscribed into the previous one, and has an Y offset
- | Use a different material for each layer
  - | PrefabObj.GetComponent<MeshRenderer>().material = currLevelMaterial



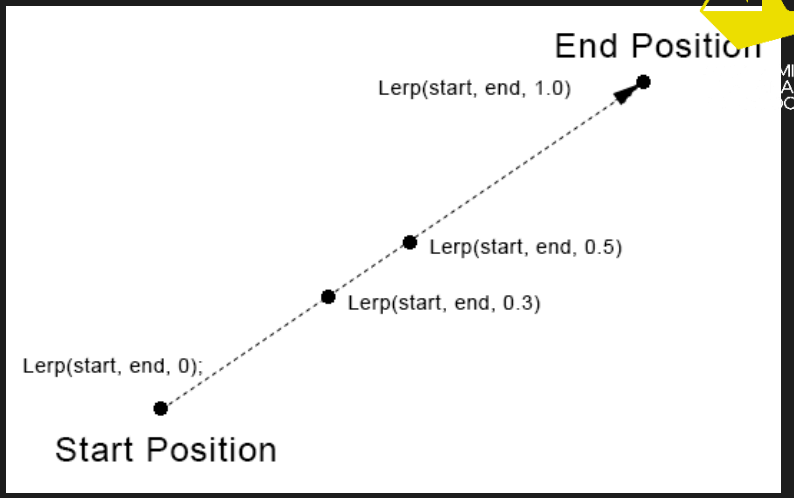


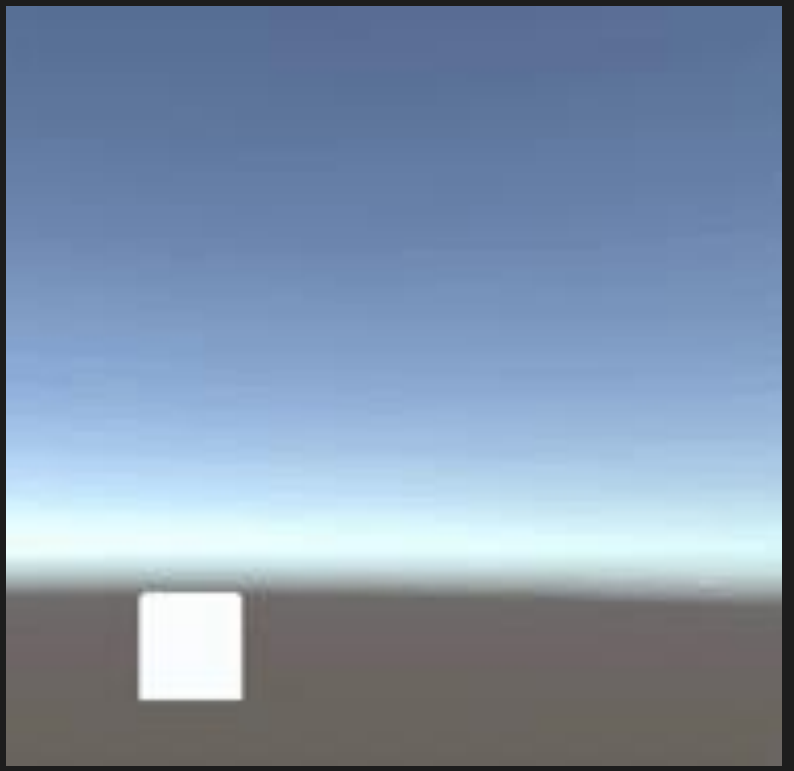
[Sierpinski_Triangle_Iterations_End_00, SierpinskiCreator.cs]

Linear interpolation

- It allows to go from A to B smoothly (noise reduction, filters, etc)
- Scalar interpolation
  - What do we need to go from 5 to 15?
  - If A=5 and B=15, we can say: C=A+(B-A)t, where 0<t<1
  - What is the value of C if
    - t=0
    - t=1
- We can interpolate scalars, vectors, colors, orientations
- Mathf/Vector3/Quaternion/Color.Lerp()
- Lerping between Materials: In Unity, you can "blend" between two different materials using Material.Lerp. This will lerp all properties with the same name
  - renderer.material.Lerp(material1, material2, t);

[SimpleLerp_Start_00, Simple_Lerp_Start_00.scene, SimpleLerp.cs]





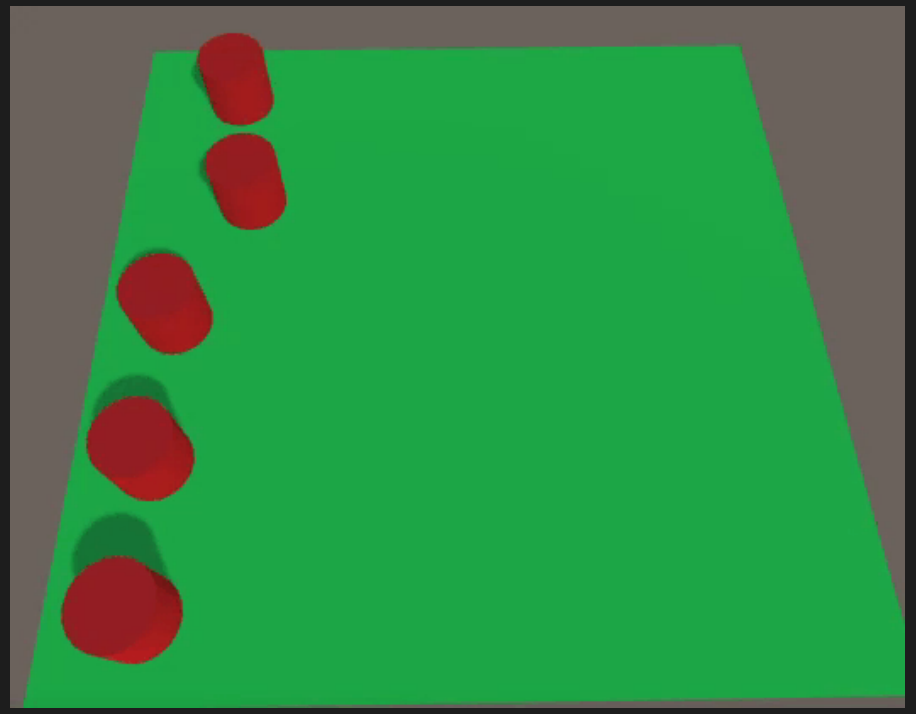
Linear interpolation

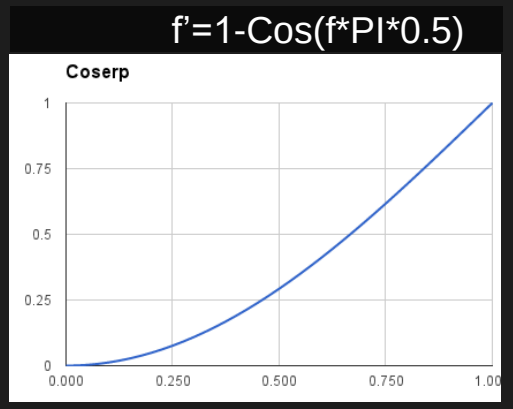
- Lerp
- InverseLerp
- Map

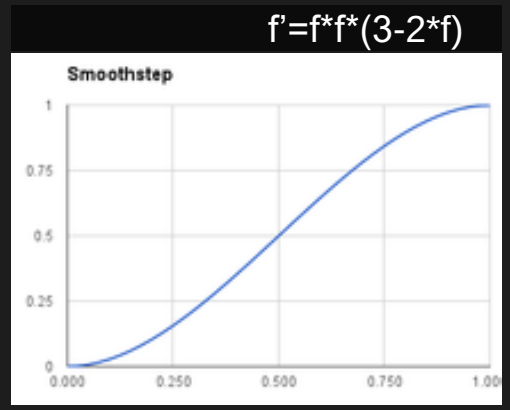
```
float y = Mathf.Lerp(c, d, Mathf.InverseLerp(x, a, b));
```

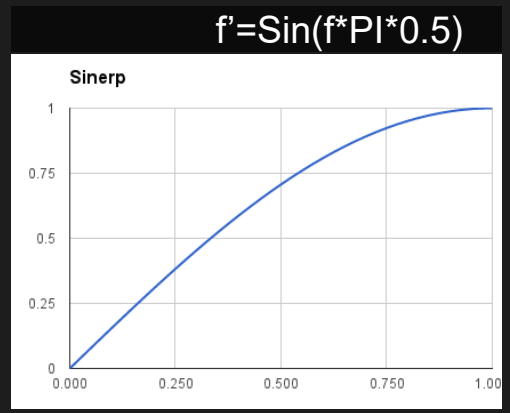
Linear interpolation - Advanced

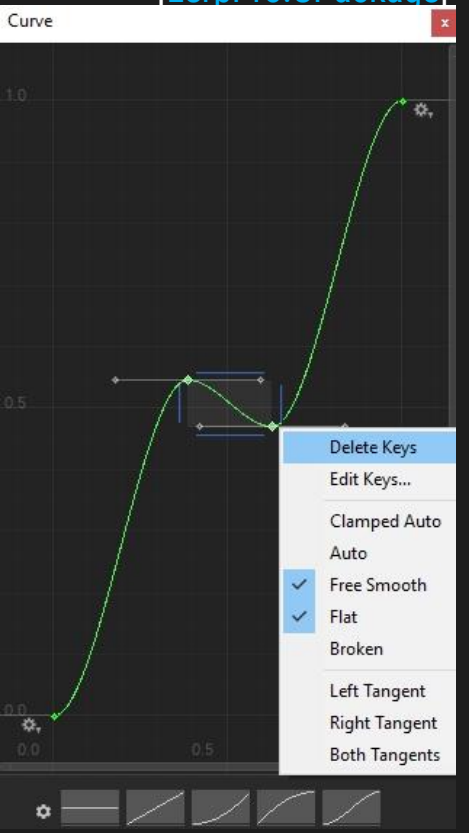
- Different types of interpolations maps [0,1] into [0,1], not in a linear way
- Start from [Lerp_Advanced_Start_00]
  - Each obj has LerpPingPong.cs: allows to switch between FORWARD/BACKWARD state, while customLerp.cs allows other kind of lerps, different from linear
- Duplicate LerpPingPong.cs and create LerpPingPongEvCurve.cs using AnimationCurve and AnimationCurve.Evaluate()
- Visit easings.net to know more easing functions
- [LerpPro.UPackage]











Ex // Live Columns

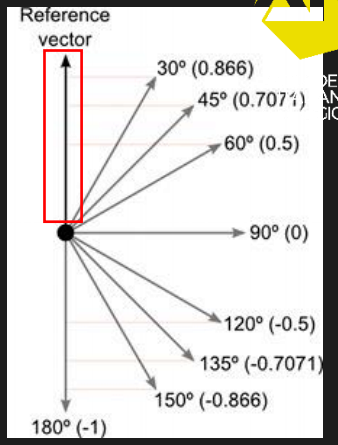
- Create RootInstantiator.cs
  - Instantiate N Root prefabs within a circle of radius R (use your circleInstantiator)
    - Vector2 point = Random.insideUnitCircle;
- Root prefab has InstantiateColumn.cs, which instantiate N ColumnPart prefabs above itself, at distance OffsetPos
  - | The ColumnPart at the column top has a scale of EndScale: while instantiating cubes above itself, Root reduces their scale accordingly, starting from its scale
- ColumnPart prefab has CubeColumnFollower.cs, which every frame follows the movements of the PartToFollow (the ColumnPart below it): every time it moves, ColumnPart is always at the same distance above it
  - | ColumnPart adjust its position using a Lerp function
- | Be creative with these Live Columns (Eg. Instantiate them around a circle and rotate around a statue, like a spell)

You can see the final result here: http://bit.ly/EX_LIVECOLS_RESULT_FK

[LiveColumns_End_00]

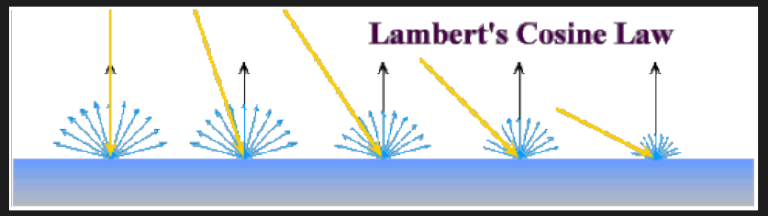
Dot Product / Projection

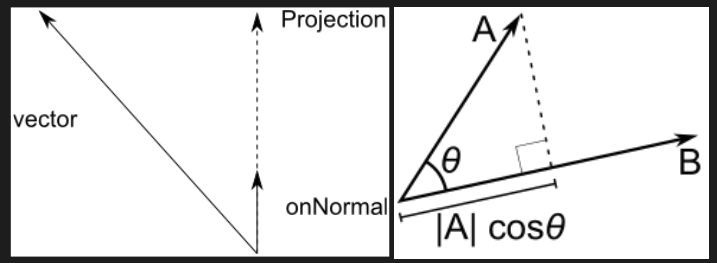
$$\mathbf{u} \bullet \mathbf{v} = |\mathbf{u}||\mathbf{v}|\cos(\theta)$$
$$= x_1 \times x_2 + y_1 \times y_2$$
$$= \mathbf{u}\mathbf{v}^T$$



- Dot(A,B) = |A| |B| cos(α)
- To know the angle between A,B: cos(α) = Dot(A,B) / |A| |B|
- The angle between A and B doen't change if A,B are normalized, then:
  - cos(α) = Dot(A.normalized,B.normalized)
- alphaInRads = Mathf.Acos(cos_alpha)

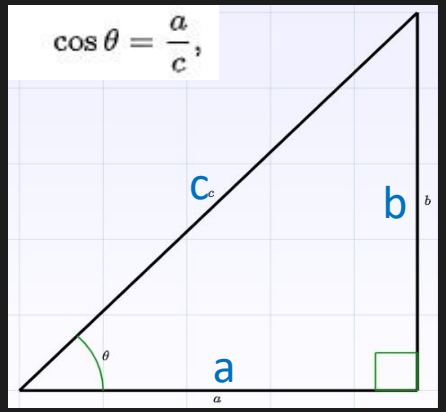
- Lambert's law





If we have 2 vectors c and a, how we calculate projection of c on a?

- In a right triangle cos(α) = a/c [1]
  - If a and c are normalized: Dot(a,c) = cos(α) [2]
  - From [1] we know that
    - a = c * cos(α) = c * Dot(a,c)
  - Then, projection of c on a = c * dot(a,c)



[dotProduct_Projection_01.UPackage, dotProduct_Projection.scene, vectorProjection.cs]

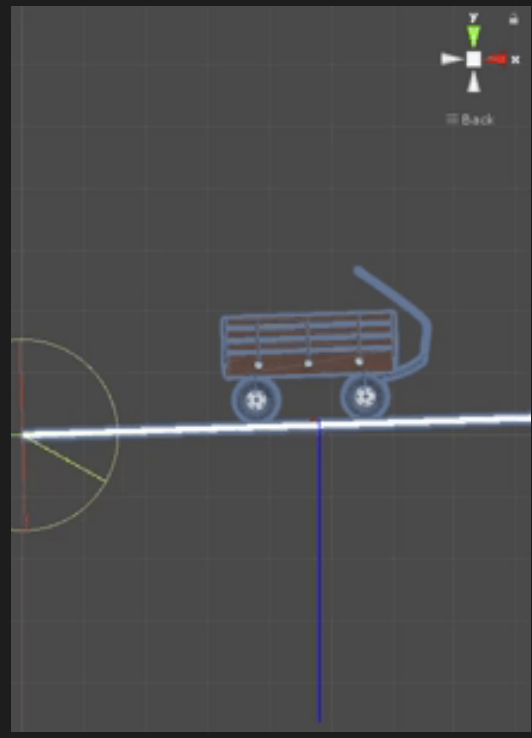
Ex // Weight Projection

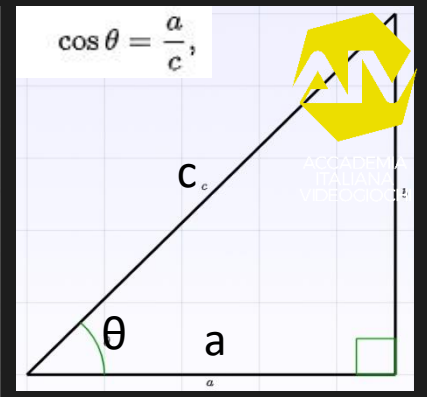
- On WagonRoot you will find [CalculateForce.cs]
- It draws weight vector W (in blue) based on its mass (W = m*g)
- Calculate the projection P of W on WagonRoot.forward and draw it in red: this is the amount of force that is needed to keep the wagon stable
- If you rotate Root on Z axis, P should increase its magnitude
- | Move the wagon in the P direction (Translation of P * Time.deltaTime)
- | Rotate the wheels according to the Wagon direction and speed
  - | NB: Wheels have wrong pivot (See FixedWheel GameObj)

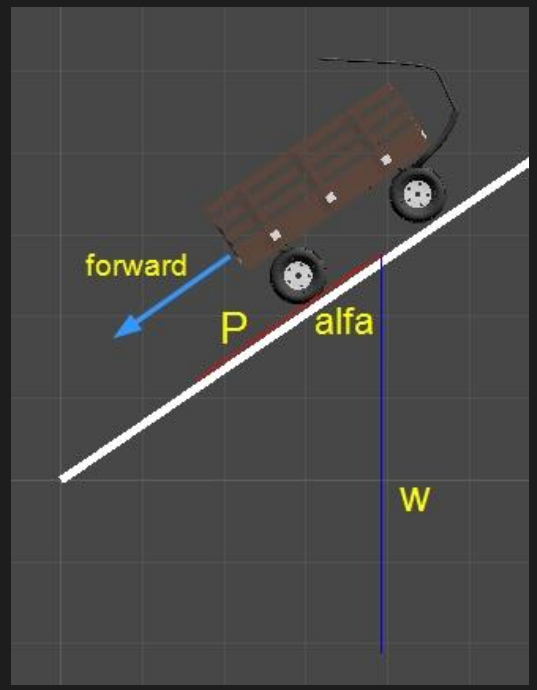
HINTS

- Calculate WN, which is the normalized vector W
- Calculate cos(alfa) using Dot product between WN and transform.forward
- Calculate the length (magnitude) of P, using the projection formula (on the right, cos(theta) = a/c)
- c in the triangle image is our W, and we need to find a
- Calculate P = Pmagnitude * transform.forward

[Projection_Wagon_Start_02, Projection_Wagon_End_02, CalculateForceEnd.cs]

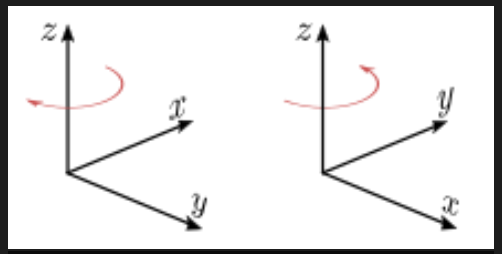




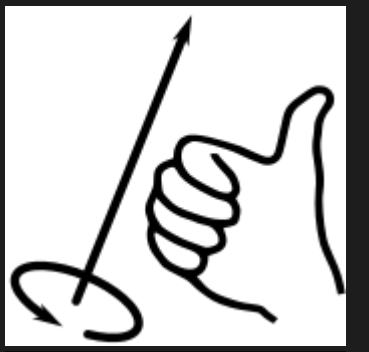


Cross Product

- Left-handed coordinate system. the positive x, y and z axes point right, up and forward, respectively. Positive rotation is clockwise about the axis of rotation
- Right-handed coordinate system. the positive x and y axes point right and up, and the negative z axis points forward. Positive rotation is counterclockwise about the axis of rotation
- A vector space with a fixed orientation is called an oriented vector space
  - Unity is left handed based

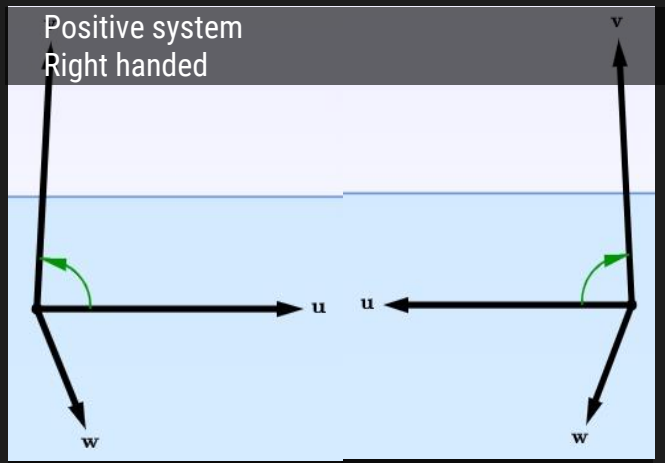


Left-handed coordinates on the left, right-handed coordinates on the right



Right-hand rule for curve orientation

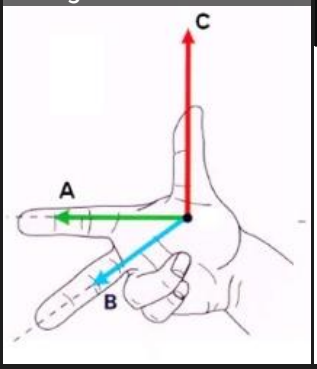
Positive system
Right handed



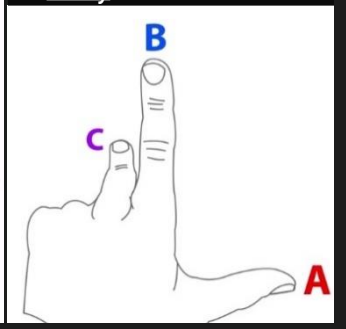
Positive rotation on w axis

Negative rotation on w axis

Positive system
Right handed



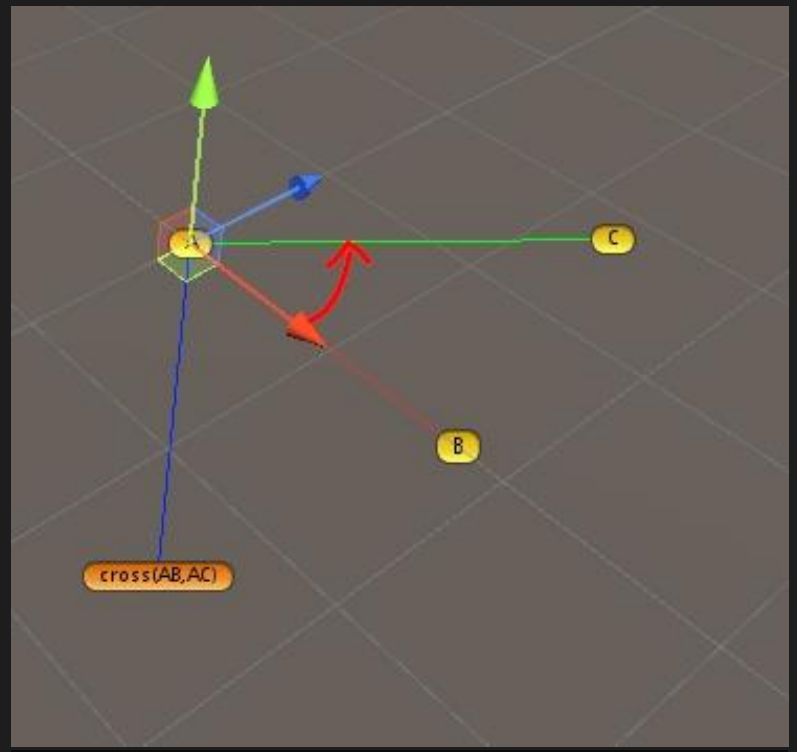
Negative system
Left handed
Unity



Cross Product

- perpV = Vector3.Cross(A.normalized,B.normalized)
- Open CrossProduct.scene and try to move AB vector around on XZ plane. The result vector
  - will have positive/negative z values depending on Sin(α)
  - Will have a magnitude depending on AB, AC magnitude

[crossProduct_01.UPackage, CrossProduct.scene, crossCalculator.cs]



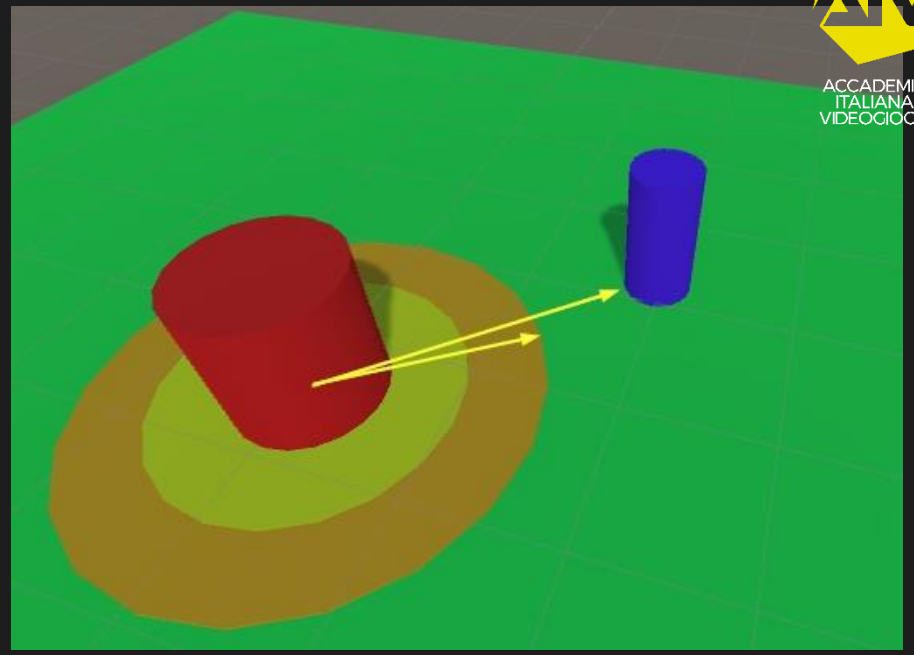
Unity is LeftHanded: the angle between AB and AC is negative (counterclockwise, because we need to orient the left thumb towards the floor)

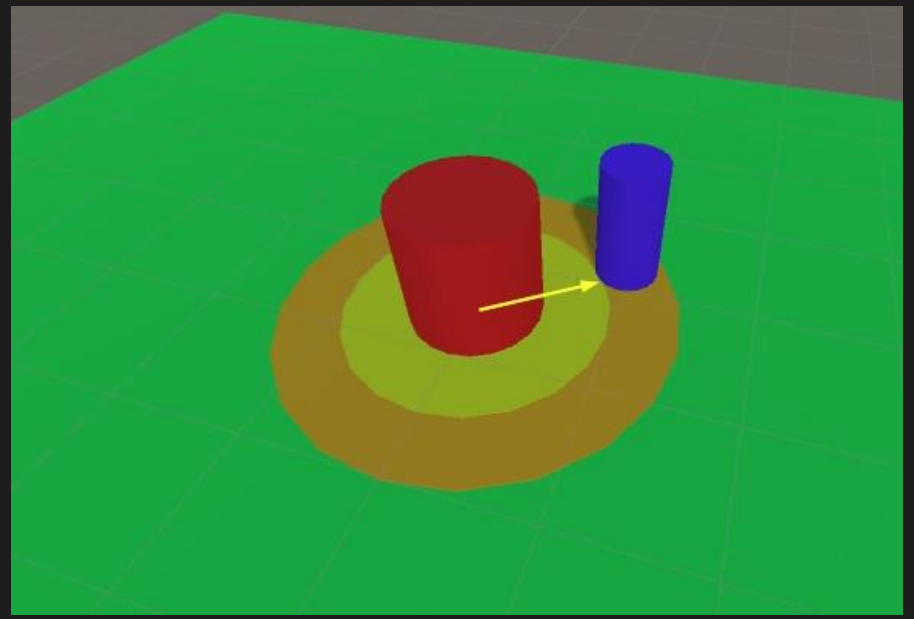
(1) $\mathbf{u} \times \mathbf{v}$ is orthogonal to both $\mathbf{u}$ and $\mathbf{v}$.
(2) $\|\mathbf{u} \times \mathbf{v}\| = \|\mathbf{u}\| \, \|\mathbf{v}\| \sin[\mathbf{u}, \mathbf{v}]$.

Ex // Hunter & Hunted

- Create a Hunter (red) and a Hunted (blue) object
- The blueObj moves with a [ObjMoveTranslation.cs] script
- The hunter has a [lerpHunter.cs] script, that allows it to follow the blue obj
- In playmode, if blue moves, red should behave like this:
    - As soon as the distance between the 2 is greater than the orange radius FollowDistance => Red moves towards Blue, until their distance is less or equal to the yellow radius ReachDistance

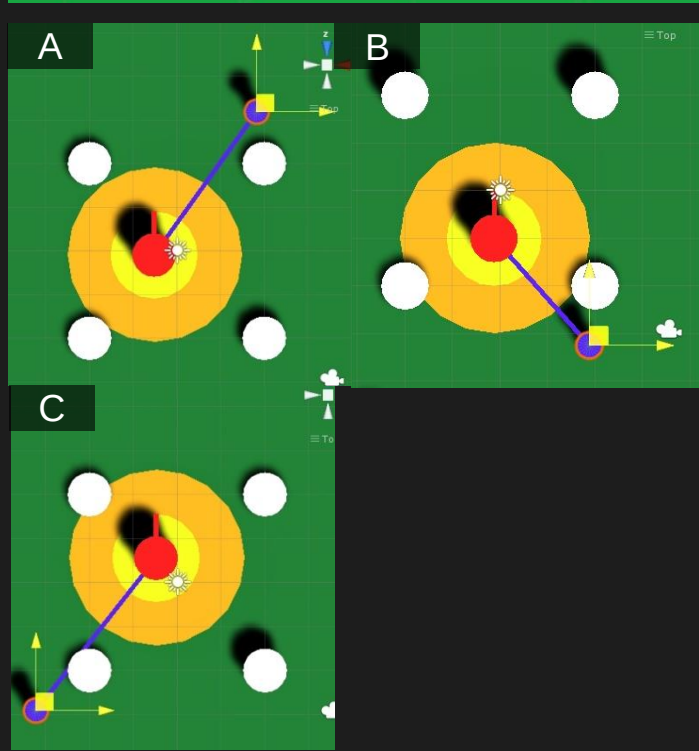
[HunterHunted_Start]





Ex // Hunter & Hunted

- You will find a visual representation of 2 vectors (they are simply scaled cubes):
  - the Hunter forward vector f ForwardRoot (the red one).
  - the vector distance d between the Hunter and the Hunted Hunter2Hunted (the blue one)
    - At runtime, while the Hunter moves, ForwardRoot will stay oriented towards the Z positive axis (The Hunter doesn't rotates)
    - Put a script [HunterHuntedVector.cs] on Hunter2Hunted, that rotates it by an angle a, which represents the angle between ForwardRoot and the Hunted
    - Pay attention: what happens when a is > 180° (case c)?
- | Add evenly distributed cylinders on the floor
  - Use a single prefab and instantiate N x N instances of that prefab at RunTime, inside a Start() function (Use your 2DInstantiator). This means that if you are not in play mode, there are no cylinders in the scene.
  - Each cylinder checks every frame if the Hunter or the Hunted are near them, in [HunterHuntedWatcher.cs]. If their distance is less than a chosen value, then the cylinder scales itself down (1), to let the hunter and the hunted pass over them
    - Use a Lerp() function for the scaling (pay attention to the Column pivot!)

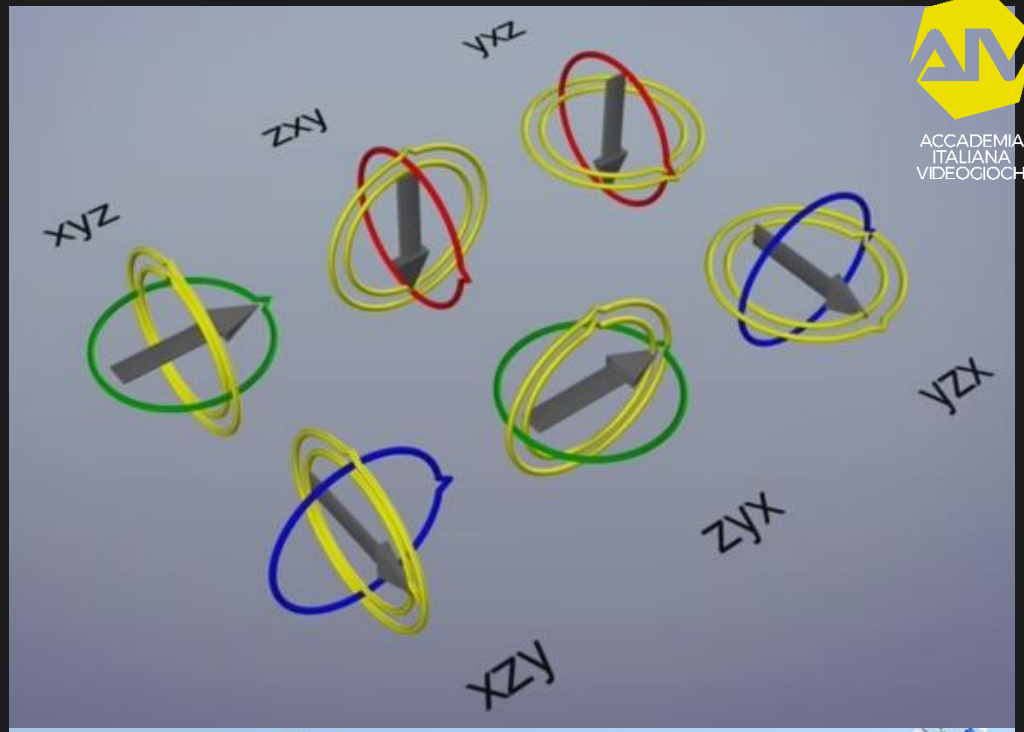


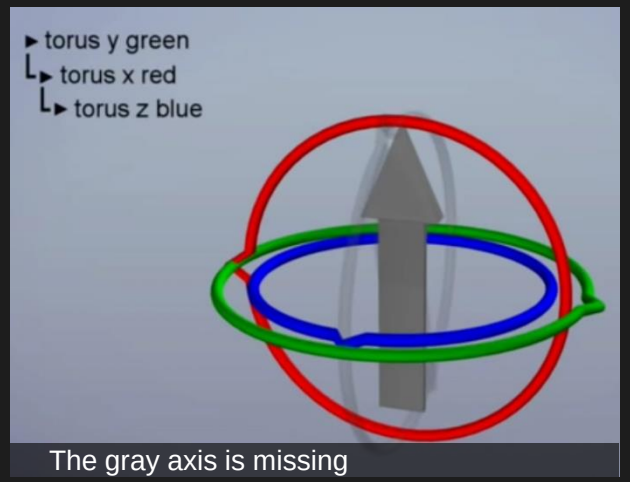
Orientation

- Vector3 is not a good data structure for orientation
  - Combine multiple rotations
  - Same resulting orientation for more than one PYR combinations
  - Interpolation
  - Gimbal Lock

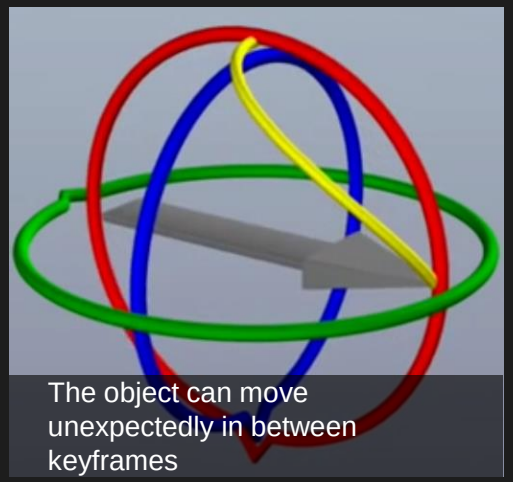
Unity Rotation Order: Z, X, Y

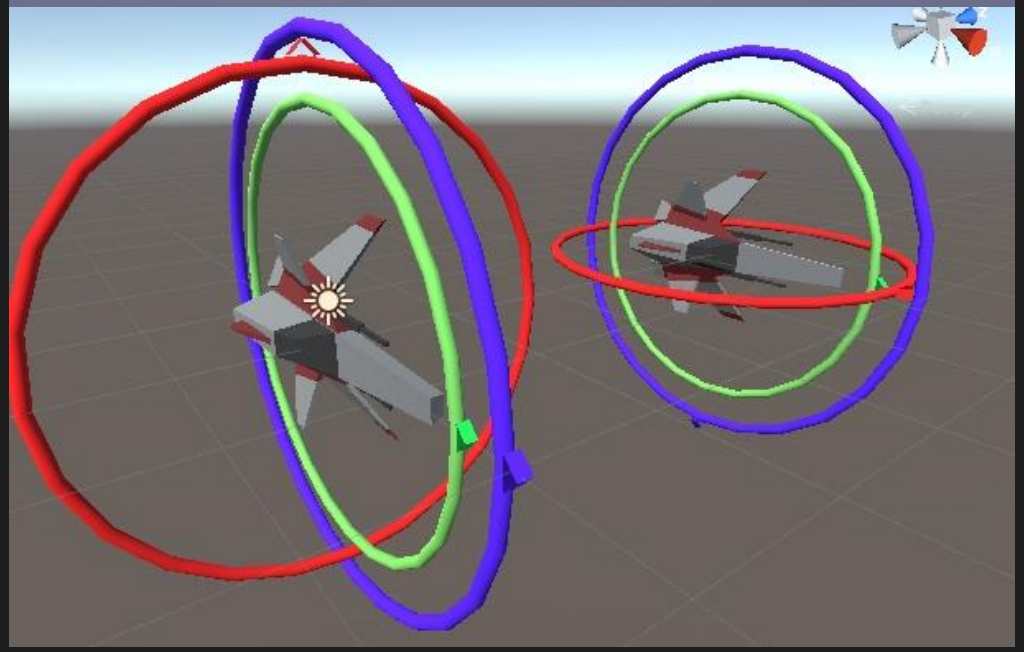
[GimbalLock_01.UPackage]



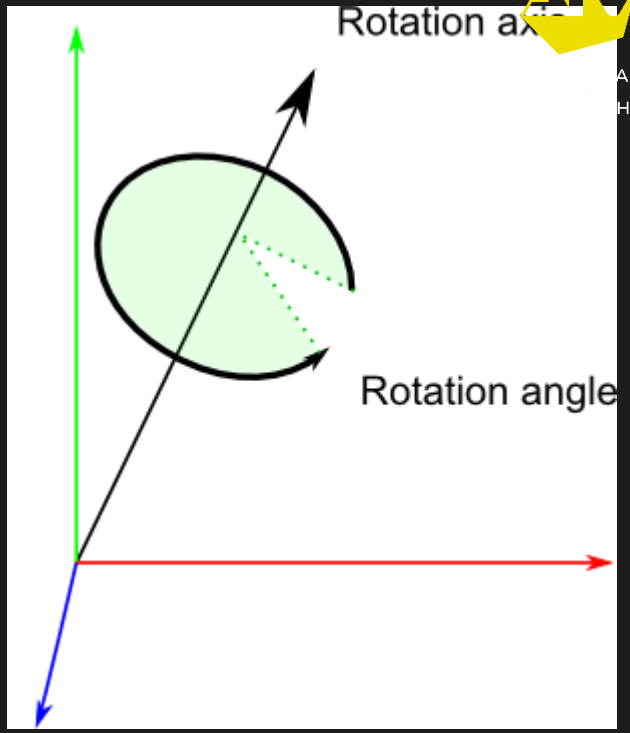


The gray axis is missing


The object can move unexpectedly in between keyframes



Quaternions



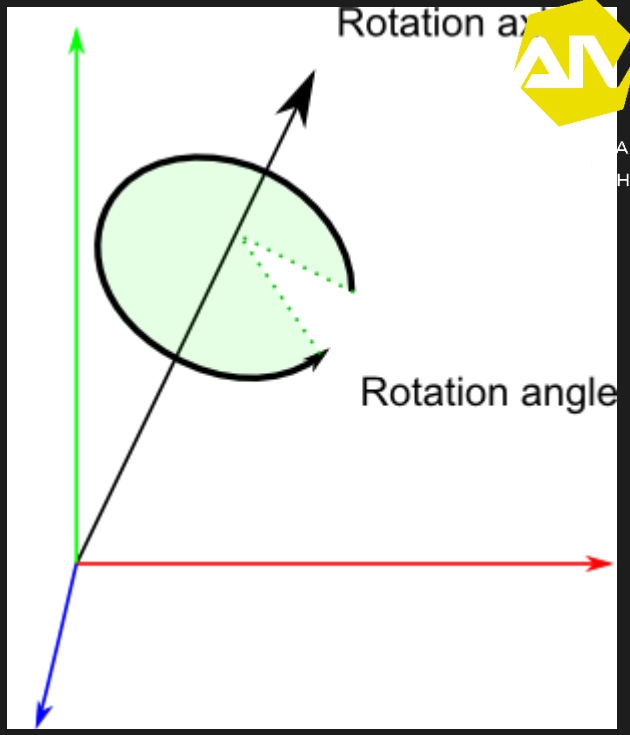
- A quaternion q is defined by: axis of rotation & angle of rotation
  - If Obj.rotation = q, this means that
    - Start from Identity rotation (the orientation with Rotation = (0,0,0) of the object, that depends on how it was exported from 3D Software)
    - Rotate the Obj around q.axis of q.angle degrees
    - This is the final orientation of Obj
- $q = xi + yj + zk + w$, where $i^2 = j^2 = k^2 = ijk = -1$
- xyz is the vector part, w the scalar part

- Essentially the relationship of these numbers mirrors the relationship of the three dimensions to each other! If you rotate 180 degrees in two dimensions (like x and y), it is essentially the same as rotating that same amount in the third dimension (z). It is this relationship that allows quaternions to represent true rotation coordinates, simplifying beyond Euler rings, and avoiding Gimbal lock as a side effect

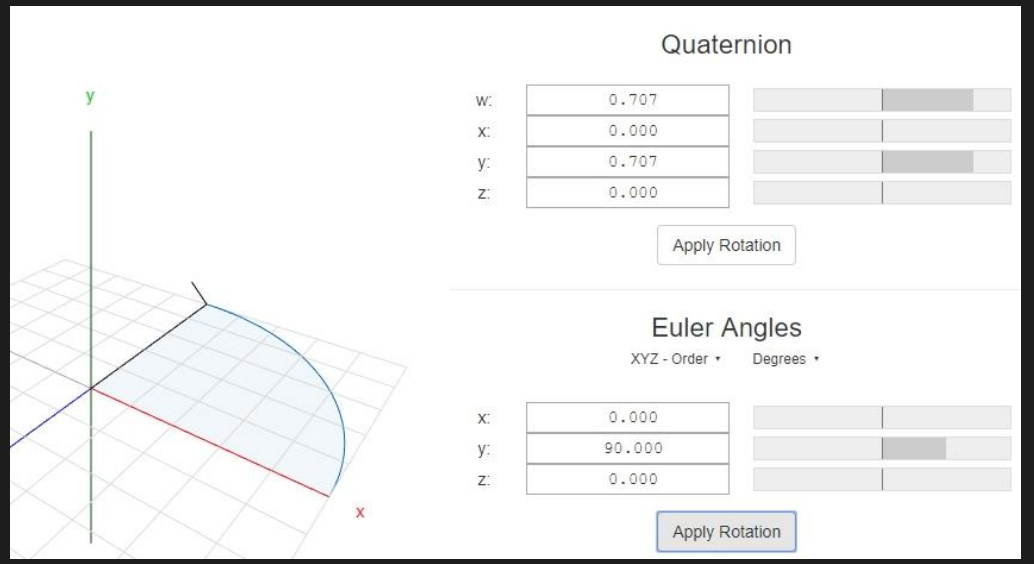
- Magnitude of $q = \sqrt{x^2 + y^2 + z^2 + w^2}$
- If $|q| \neq 1$, q is not a valid quaternion

Quaternions

- To rotate a 3D point V(x,y,z): q * V
- Problem: V is 3D, q is 4D => We need to express V in 4D
  - To build q(V) from V: xi + yj + zk + 0
- To build q, we need: a Rotation axis (a direction) V(x,y,z) & angle of rotations in rads Θ
  - q.x = axis.x * sin(Θ/2)
    q.y = axis.y * sin(Θ/2)
    q.z = axis.z * sin(Θ/2)
    q.w = cos(Θ/2)



- Online quaternion / Euler angles simulation: [quaternions.online]
  - Try to construct by hand the quaternion to rotate a point V on Y axis by 90 degrees, and then check the result on the website
    - Cos(pi/4) = Sin(pi/4) = 0.707



Quaternions

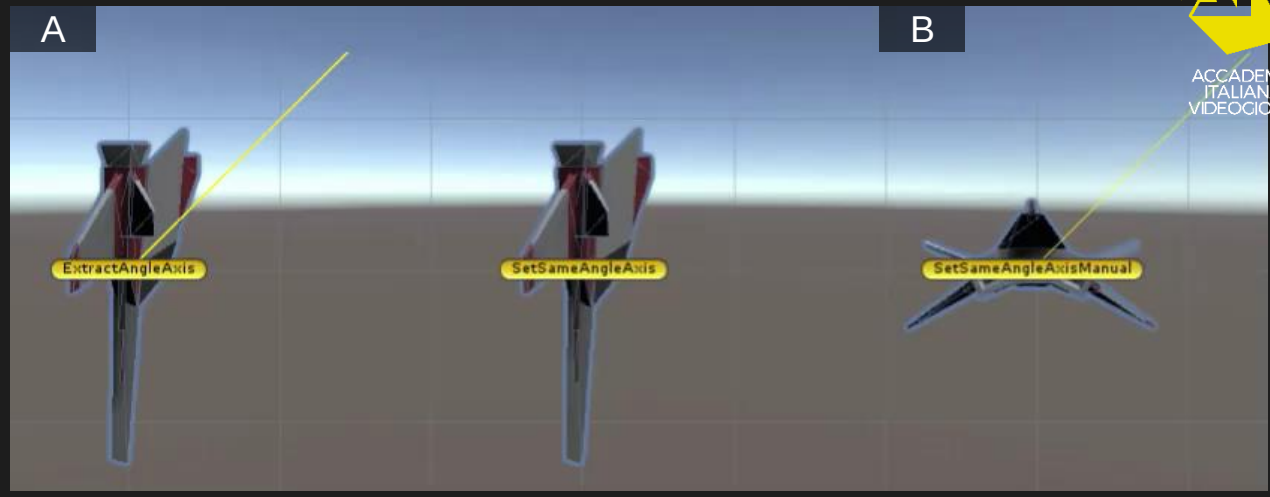
- PROs
  - Data structure size (3x3Matrix, 9 scalars vs 4 scalars)
  - Easy interpolation between quaternions
  - Floating point normalization for quaternions suffers from fewer rounding defects than matrix representations
  - Gimbal Lock
- CONs
  - Less intuitive
  - Doesn’t contain translation & scale info

Quaternions

Basic operations [QuaternionsOps.cs]

- Quaternion.identity
- Quaternion.Euler(Vector3 eulerRotation)
- Quaternion.ToAngleAxis(out float alpha, out Vector3 axis)
  - A is rotated (90,90,0). ToAngleAxis() extracts the yellow axis and an angle of rotation of 120. To highlight the equivalence, B Object rotates from Identity to 120 degrees, around the yellow axis (use QuaternionsOps.ManualAlpha)
- Quaternion.AngleAxis(float alpha, Vector3 axis)
  - If we set obj.rotation = q or increment the rotation with obj.rotation *= q, the obj position is the same: we are changing its orientation. The q rotation axis passes in obj local cords
- Quaternion.Inverse(Quaternion source)
  - If an Obj has a rotation Q1, its inverse rotation is Q2, where Q2*Q1 = Identity. Hence, the inverse of identity is… Identity.

[Quaternions_02.UPackage]



- Quaternion.Angle(Quaternion q1, Quaternion q2)
  - 2 objs AngleA & AngleB must have a quaternion with the same rotation axis. Starting from this situation, this is useful to know the difference between the 2 quaternions' spin
- Concatenation
  - Obj.rotation = Aq*Bq is the same as set the hierarchy in this way: Parent1 (with rotation Aq) / Child1 (with rotation Bq) / Child2 (with rotation Identity)
  - Hence, Obj.rotation = Aq*Bq is different from Obj.rotation = Bq*Aq

Quaternions

Basic operations [QuaternionsOps.cs]

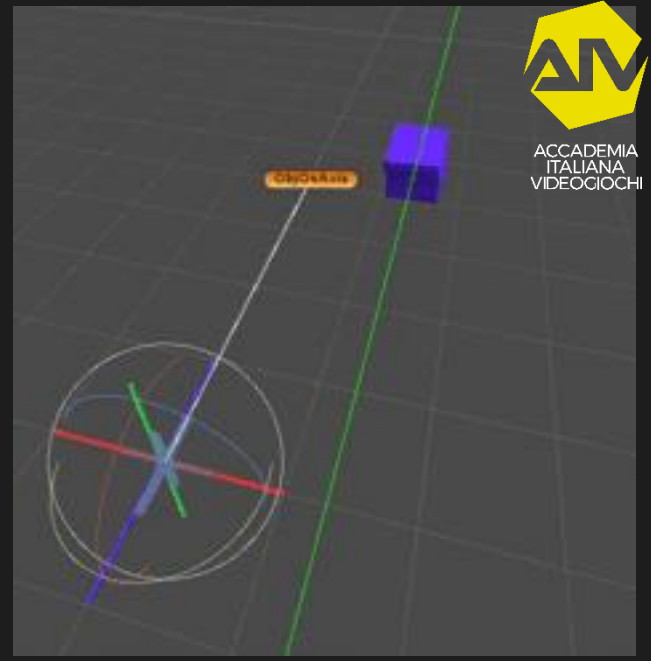
- Quaternion.Dot(Quaternion q1, Quaternion q2)
  - The result takes into account also the spin on the Rotating axis
  - The result range is [0,1] instead of [-1,1]
  - Hence, if A has a rotation of (0,0,0), B has (0,0,90), C has (0,0,180), we got:
    - Dot(A.rotation, B.rotation) = cos(pi/4) //instead of cos(pi/2)
    - Dot(A.rotation, C.rotation) = cos(pi/2) //instead of cos(pi)
  - In other words, if we are rotating the obj only on the Z axis
    - If the Z angle is the same, the result is 1
    - If Z angles have a difference of 90 deg, the result is 0.707
    - if Z angles have a difference of 180 deg, the result is 0

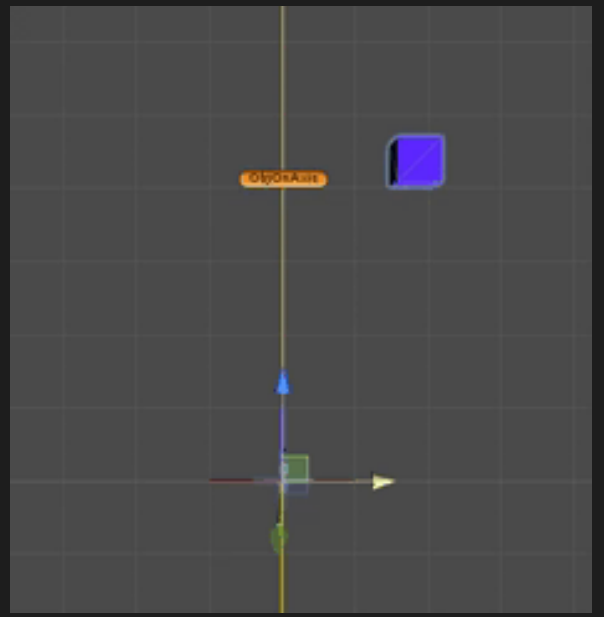
[Quaternions_02.UPackage]

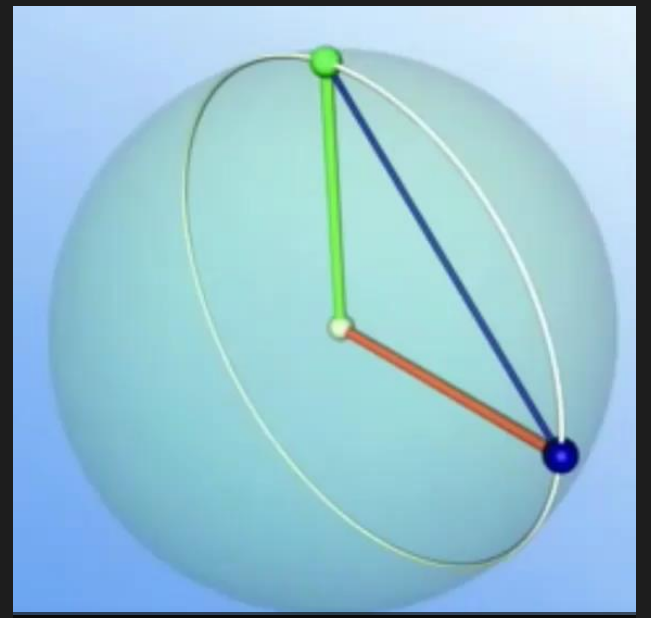
Rotate around custom axis

- Position update: Quaternion * Vector3
  - We need: Axis of rotation, degrees, transform.position
  - Rotate around a custom Axis passing through the origin
- Rotation update: Quaternion * Quaternion
  - We need: Axis of rotation, degrees, transform.rotation
  - Rotate around a custom Axis passing through the ObjToRotate PIVOT
- Interpolation Slerp()

[Quaternions_02.UPackage]





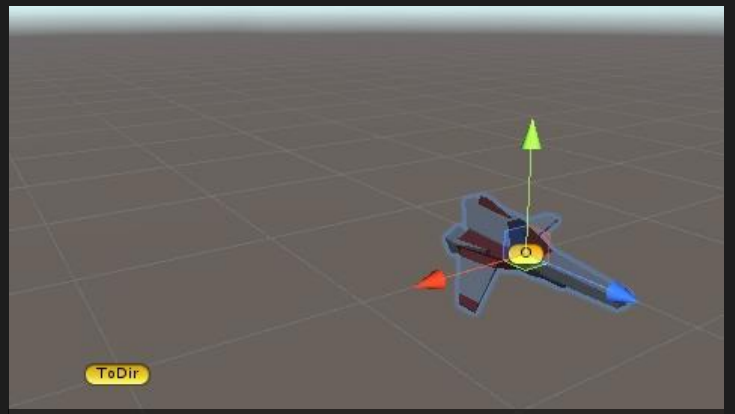


Linear vs Spherical Interpolation

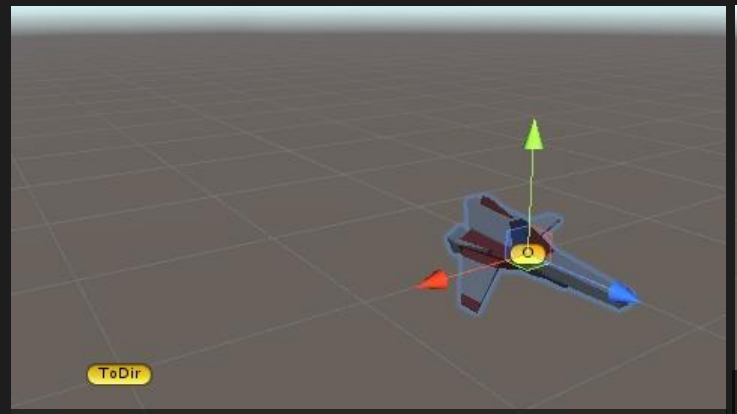
FromToRotation()

Why do we use Vector3.forward instead of transform.forward in FromToRotation()?

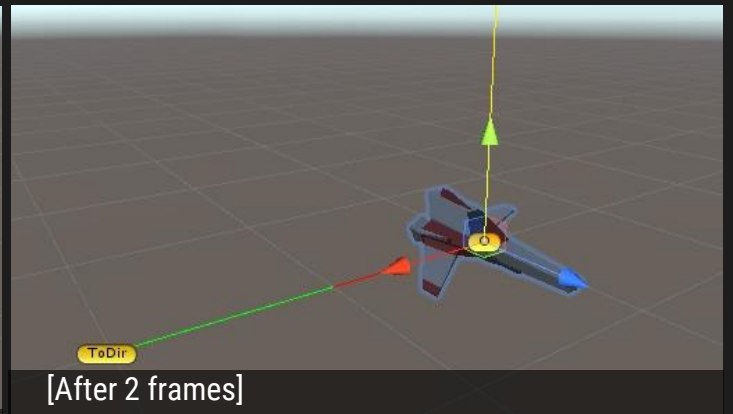
Because we constantly assign Obj.rotation = q in the Update() method. This means that every frame we calculate the rotation needed from transform.forward to dir2Cam. But: transform.forward changes every frame, so q == Identity in frame 1, this leds to a rotation of (0,0,0) in frame 2, and again in a correct rotation in frame 3, and so on, continuously rotating the Obj between frames.



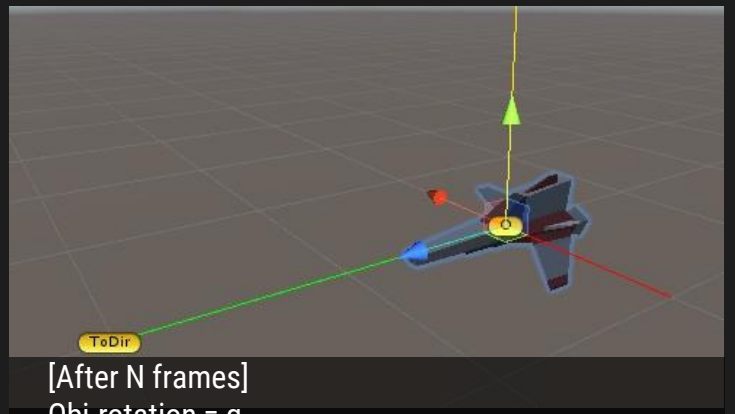
q = FromToRotation(Vector3.forward, dir2Cam)



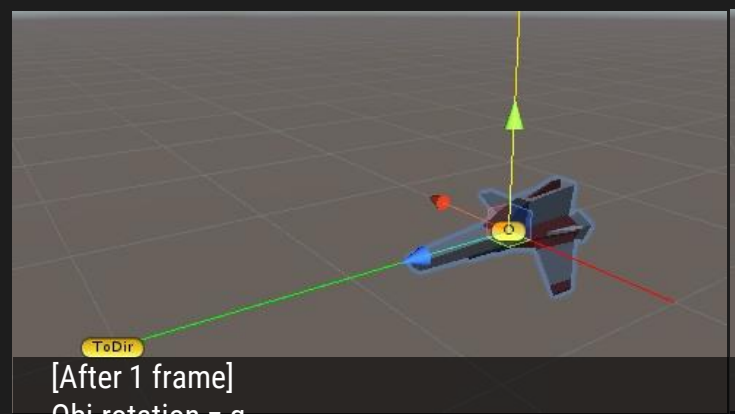
q = FromToRotation(transform.forward, dir2Cam)



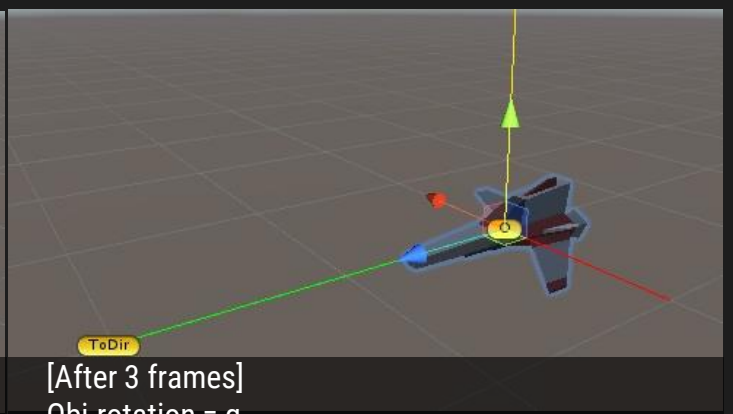
[After 2 frames]
Obj.rotation = q



[After N frames]
Obj.rotation = q



[After 1 frame]
Obj.rotation = q



[After 3 frames]
Obj.rotation = q

Ex // Doom is back

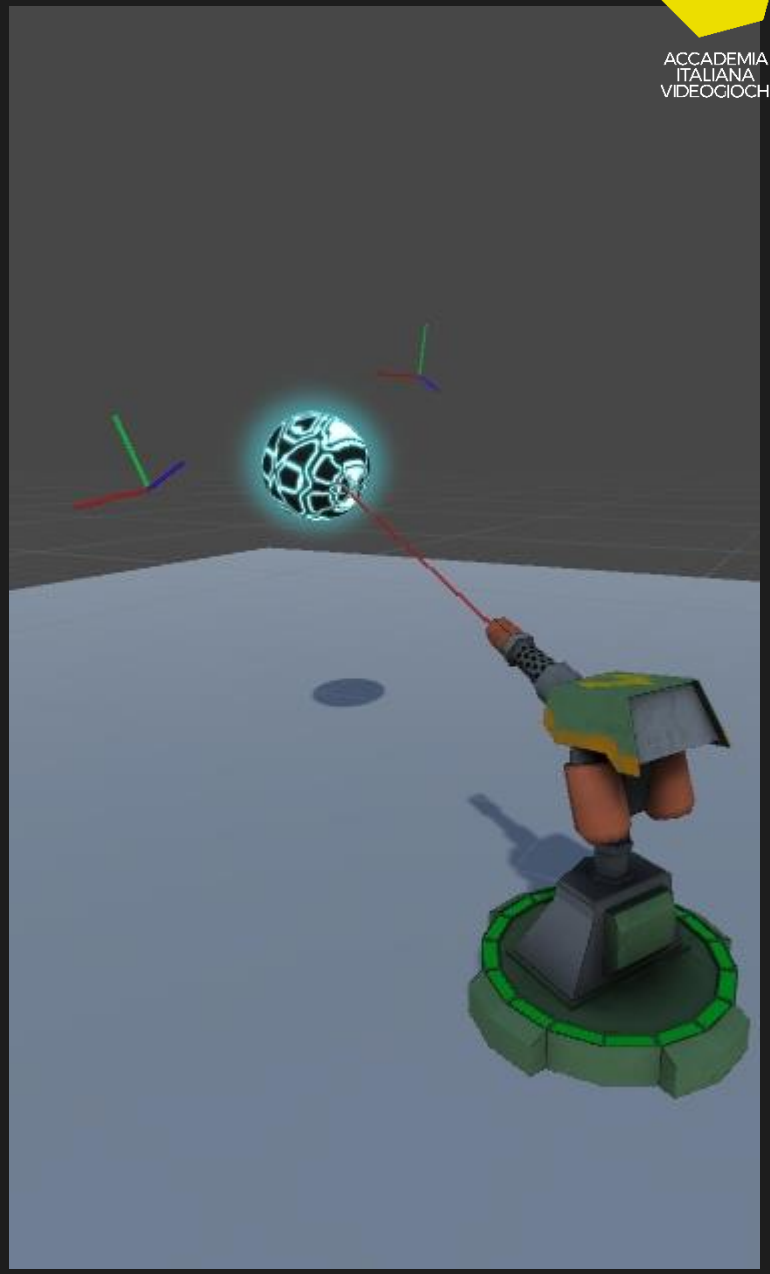
- Start from [Doom_Start_00.UPackage] and [DoomSprites] folder
- Duplicate quad 00 under Enemy/Sprites 8 times: each quad should have a different material with a different Textures/DoomSprites texture
- Setup texture import settings and materials to use alpha
- Use CameraFacingBillboard.cs script to orient quads in the right direction
  - Pay attention: where is the best place to put CameraFacingBillboard.cs script?
- At runtime
  - Calculate the angle alpha between the enemy forward vector and the camera
  - Create SpriteSelector.cs on Enemy, that based on alpha enables only one quad under Enemy/Sprites and disable all the others

[Doom_Start_01, Doom_End_00]

Ex // Spot that Bot!

- Do not place the turret on (0,0,0)
- Create [turretDirection.cs]. Our goal is to orient the turret towards the target. Two orientations involved here:
  - The turret obj should rotate around its Y axis. Use LookRotation()
  - The gun obj should rotate around its X axis. Use Vector3.Dot()
- Add a quad inside the target; assign to it a glow texture and a billboard facing camera script [cameraFacingBillboard.cs]
- | Create a laser gun. It should go from the turret gun to the target (no longer)
- | If the target is under the gun Y, the gun should stay parallel to the ground and no laser should appear
- | Add a camera switch script [cameraSwitch.cs]: choose 4 orientation in 3D space and let the user choose the right camera orientation using keys 1,2,3,4. The switch must use a slerp interpolation
- | Use SunGlow.png texture on a billboard facing camera quad to create the Sphere Glowing effect

[SpotThatBot_Start_02.UPackage]

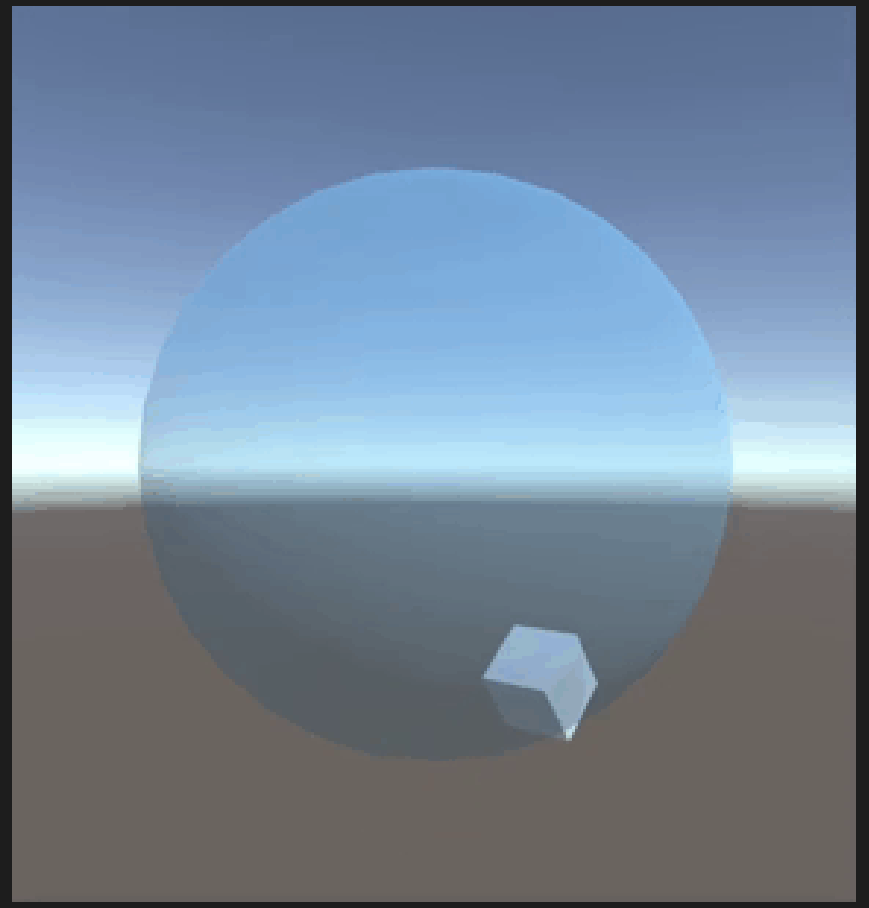


Ex // GiftPicker

[Prototype – this is not the final exercise, but it is a first step]

- We want to rotate the cube around the center of the sphere
- Create quaternion qy that has y axis and a degree alfa based on Input.HorizontalAxis
- Create quaternion qx that has x axis and a degree beta based on Input.VerticalAxis
- FinalRotation = qx * qy
- Calculate FinalPosition based on
  - Sphere pivot position
  - Cube FinalRotation
  - Vector3.forward
  - Radius
    - FinalPosition = origin - (FinalRotation * Vector3.forward * radius)

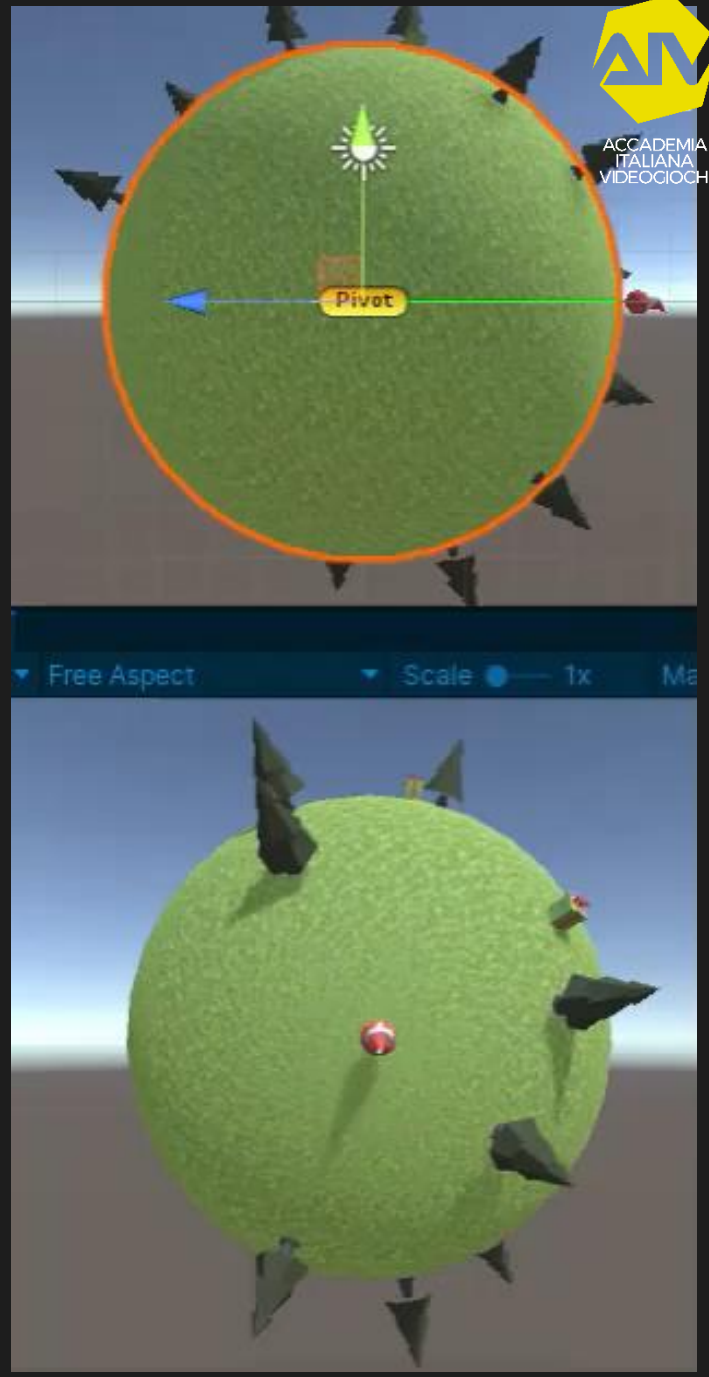
[Orbiting_Start.UPackage, Orbiting.cs, Orbiting_End]



Ex // GiftPicker

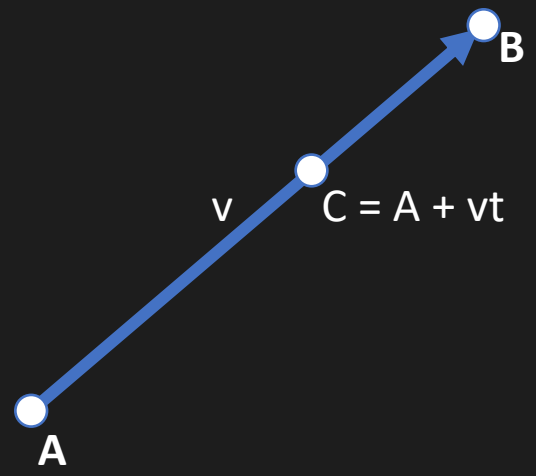
- Use 3D Objects in XMas folder. They have wrong Pivots, adjust them.
- Tree and Gift prefab has Pivot on Origin and pos.y = sphere radius.
  - Autodestroy after N seconds
  - PointCheck: if Tree/Gift distance from Player is < d, then Adds/Remove points to the player
    - Call Points.AddPoint(val)
- Autorotate.cs Constantly rotates the sphere on X axis
- Points.cs Print the current score with IMGUI
- Orbiting.cs Move the player like the previous exercise
- TreesAndGiftInstantiator.cs
  - Instantiate Tree and Gift prefabs with Euler rotation (90, [range], 0)
    - Eg. Range [-40,40]
- Add what you want to improve the game experience (using only what we have done so far)

[GiftPicker_Start_01, GiftPicker_End]



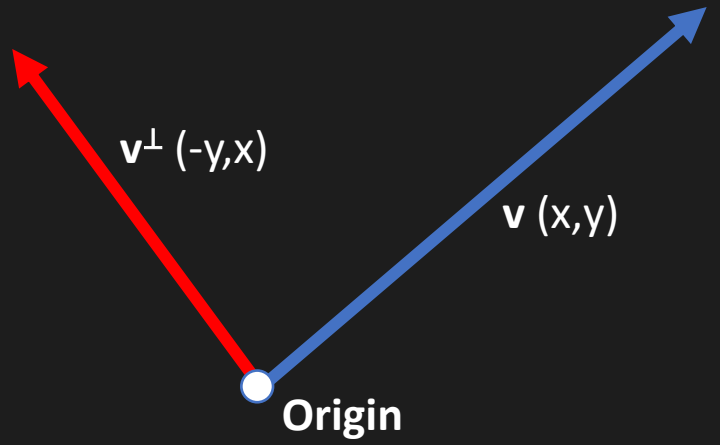
Parametric Line

- We can see a line between A and B like a sequence of points C = A + vt, where t is in range [-∞, +∞]



Perp Vector

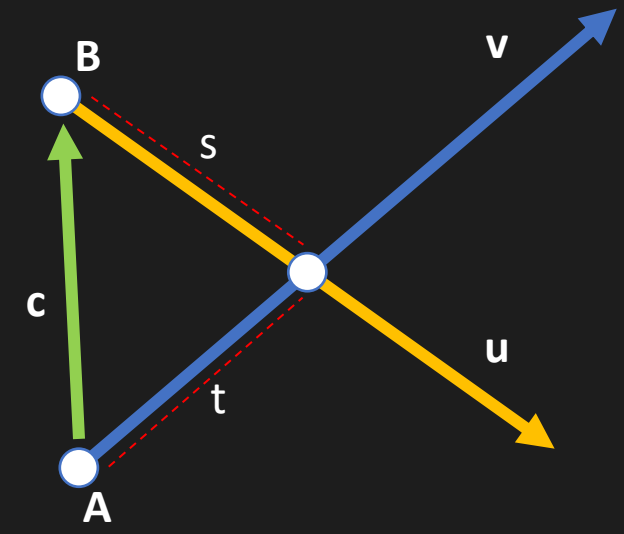
- Dot($v^{\perp}$, v) = 0
- Dot($v^{\perp}$, u) = Dot($u^{\perp}$, v)
- For each point P(x, y), its perpendicular point (on the line that passes through the origin) is (-y,x)



- See DrawLine/DottedLineDrawer.cs

[LinesPlanesOps_03]

Line-Line intersections (2D)



- We should solve the equation
  - A + vt = B + us
  - Since B – A = c we have:
  - vt = us + c
- to know t and s. We have to isolate t or s:

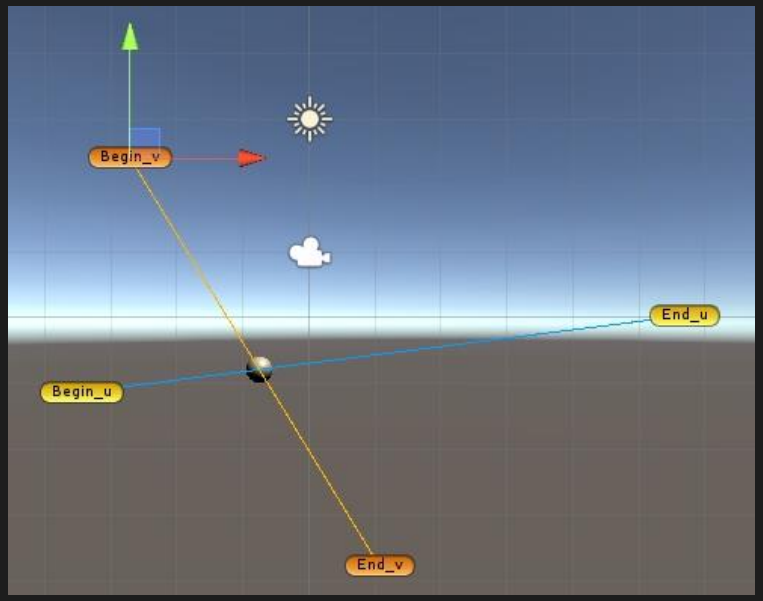
vt = us + c

Dot(u⊥,vt) = Dot(u⊥,us) + Dot(u⊥,c) // remember that Dot(u⊥,u) = 0

Dot(u⊥,vt) = Dot(u⊥, c)

t = Dot(u⊥, c) / Dot(u⊥, v)

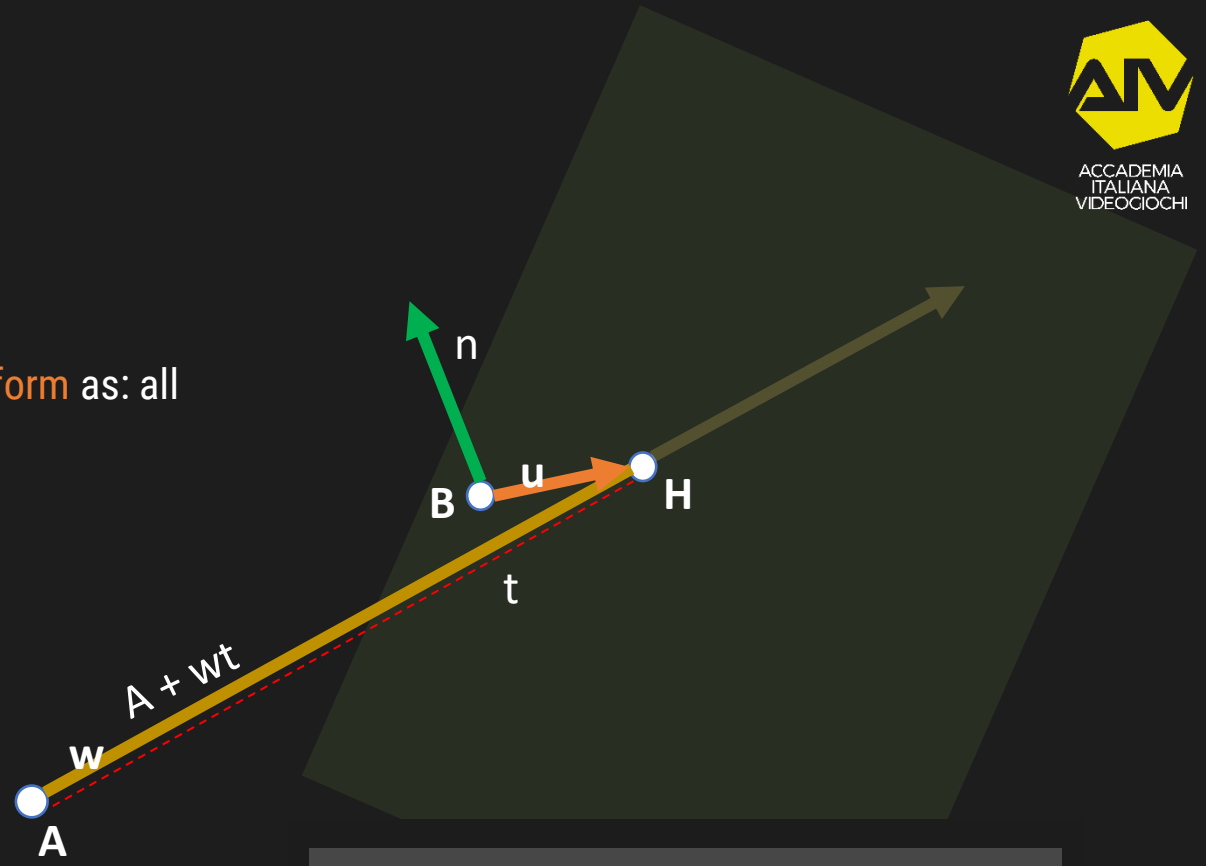
- See how the implementation of VectorUtils.GetSegmentIntersection(), in LineLineIntersection.cs works



[LinesPlanesOps_03, VectorUtils_End.cs]

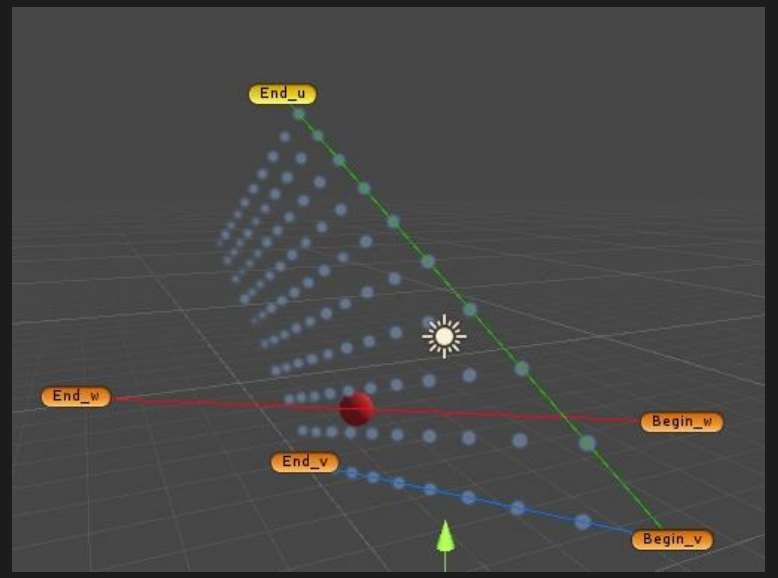
Line Plane Intersection

- We should solve the equation A + wt = B + vq + us
- If B is a point on the plane, we can express the plane in Point Normal form as: all points H that satisfy this condition: Dot(n, (H-B)) = 0
- We also have that H = A + wt
- Then:
  - Dot(n, (A + wt - B)) = 0
  - Dot(n, (A–B)) + Dot(n,wt) = 0
  - t = -Dot(n,(A-B) / Dot(n,w)



- Activate DrawPlane GObj to draw plane using DottedPlaneDrawer.cs
- Finish the implementation of VectorUtils.GetLinePlaneIntersection(), used in LinePlaneIntersection.cs

[LinesPlanesOps_03]



Ex // Inside Walls

- Start from Unity Ray-Plane intersection methods and from [LinesPlanesOps_00/EX_InsideWalls_Start]
- Finish the implementation of InsideWalls_Start.cs
- Check every frame if the hitPoint is inside the Walls using Dot product between C and N (draw these vectors), where
  - C is the vector from the hitPoint to the Wall
  - N is the wall normal
  - Use Dot(C,N) to know if hitPoint is in the “normal” side of the wall (the same side the normal is pointing at)
- If hitPoint is inside the wall, then move the RedSphere
- Use GetComponent<MeshFilter>().mesh.normals To know plane normals (they are all the same)
  - Normals are in localSpace. Use Transform.Direction() to transform them from LocalSpace to WorldSpace
- NB - From Local to worldspace:
  - Transform.Direction(dir) not affected by scale or position of the transform. The returned vector has the same length as dir
  - Transform.TransformPoint(pos) the returned position is affected by scale
  - Transform.TransformVector(dir) not affected by position of the transform, but it is affected by scale. The returned vector may have a different length than vector

[LinesPlanesOps_03]