of Earth

$$\frac{\vec{v}_i + \vec{v}_f}{2}$$

$$\frac{\vec{v}_i + \overbrace{\vec{v}_i + \vec{a}\cdot\Delta t}^{v_f}}{2}$$

$$\vec{s} = \left(\frac{\cancel{2}\vec{v}_i}{\cancel{2}} + \frac{\vec{a}\cdot\Delta t}{2}\right)\Delta t$$

$$\boxed{\vec{s} = \vec{v}_i\,\Delta t + \frac{1}{2}\vec{a}\,(\Delta t)^2}$$

$$19.6\,m/s - 9.8\,m/s^2\cdot\Delta t$$

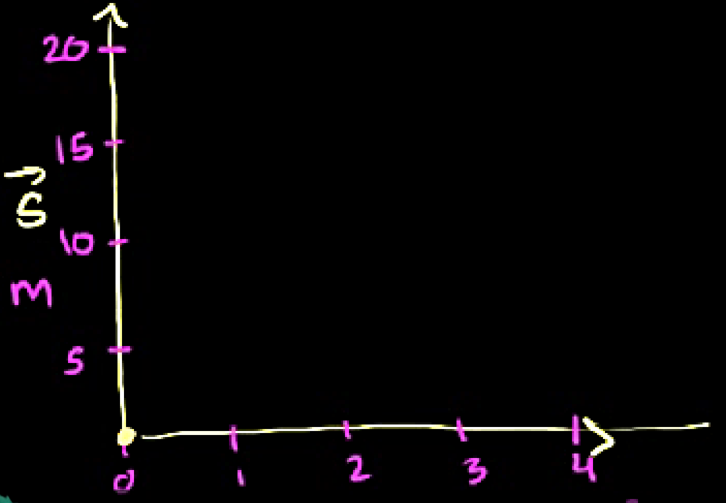
$$\vec{v}_f = \vec{v}_i + \vec{a}\cdot\Delta t$$

$$\boxed{\vec{a}_g = -9.8\,m/s^2}$$

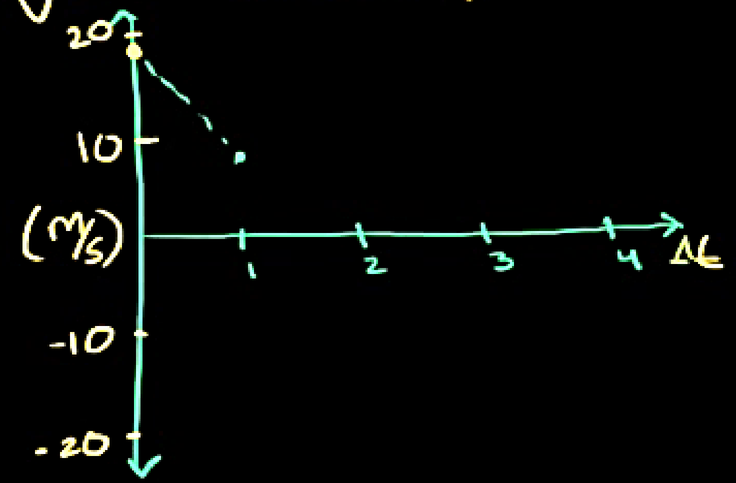
$$\vec{s} = 19.6\,\Delta t - 4.9$$

Δt	$\vec{v}_f$	$\vec{s}$
0	19.6	0
1	9.8	
2		
3		
4		

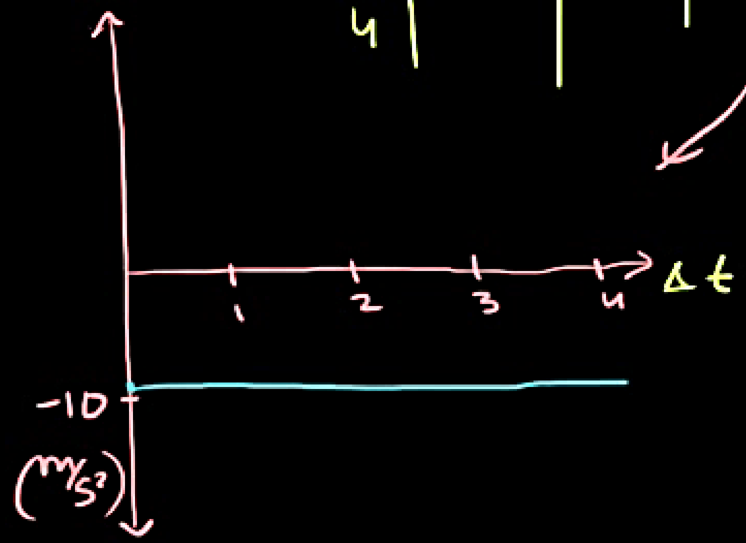
Displacement



velocity

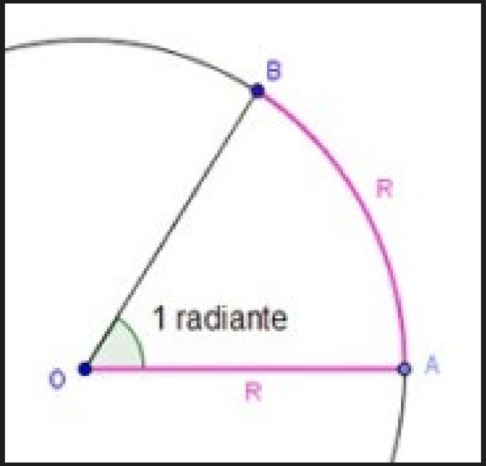


Acceler



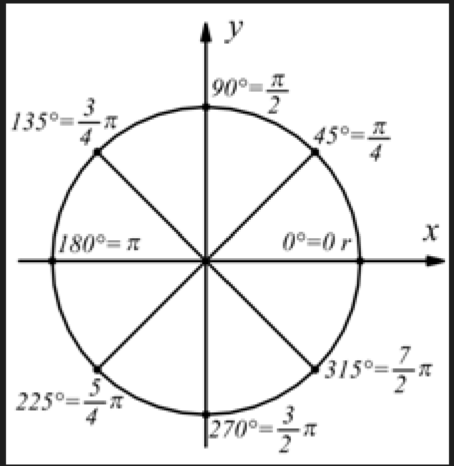
Sin/Cos

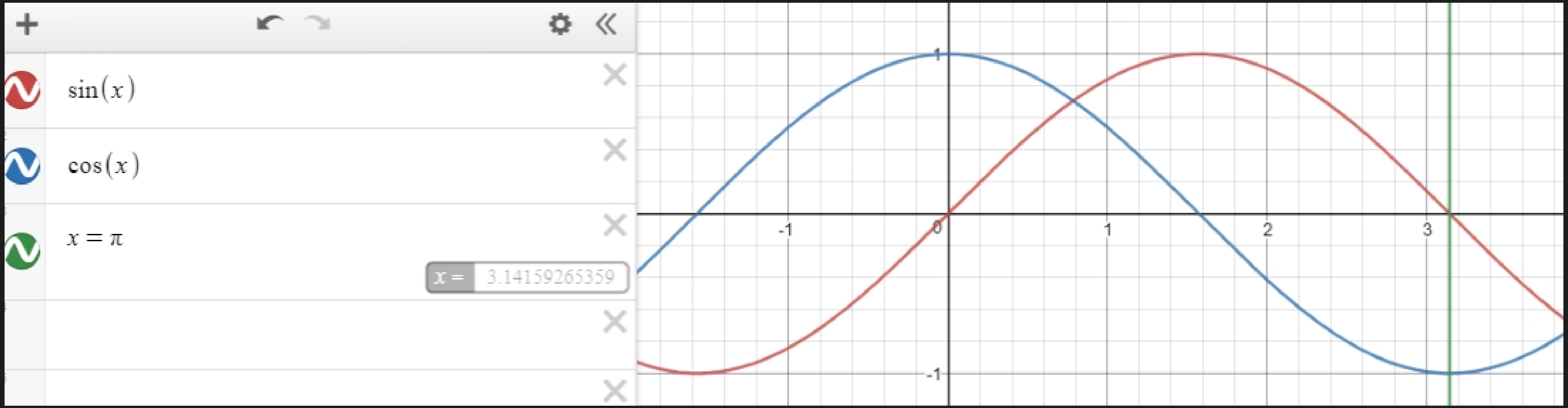
- Degrees/Radiants
- Sin/cos



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

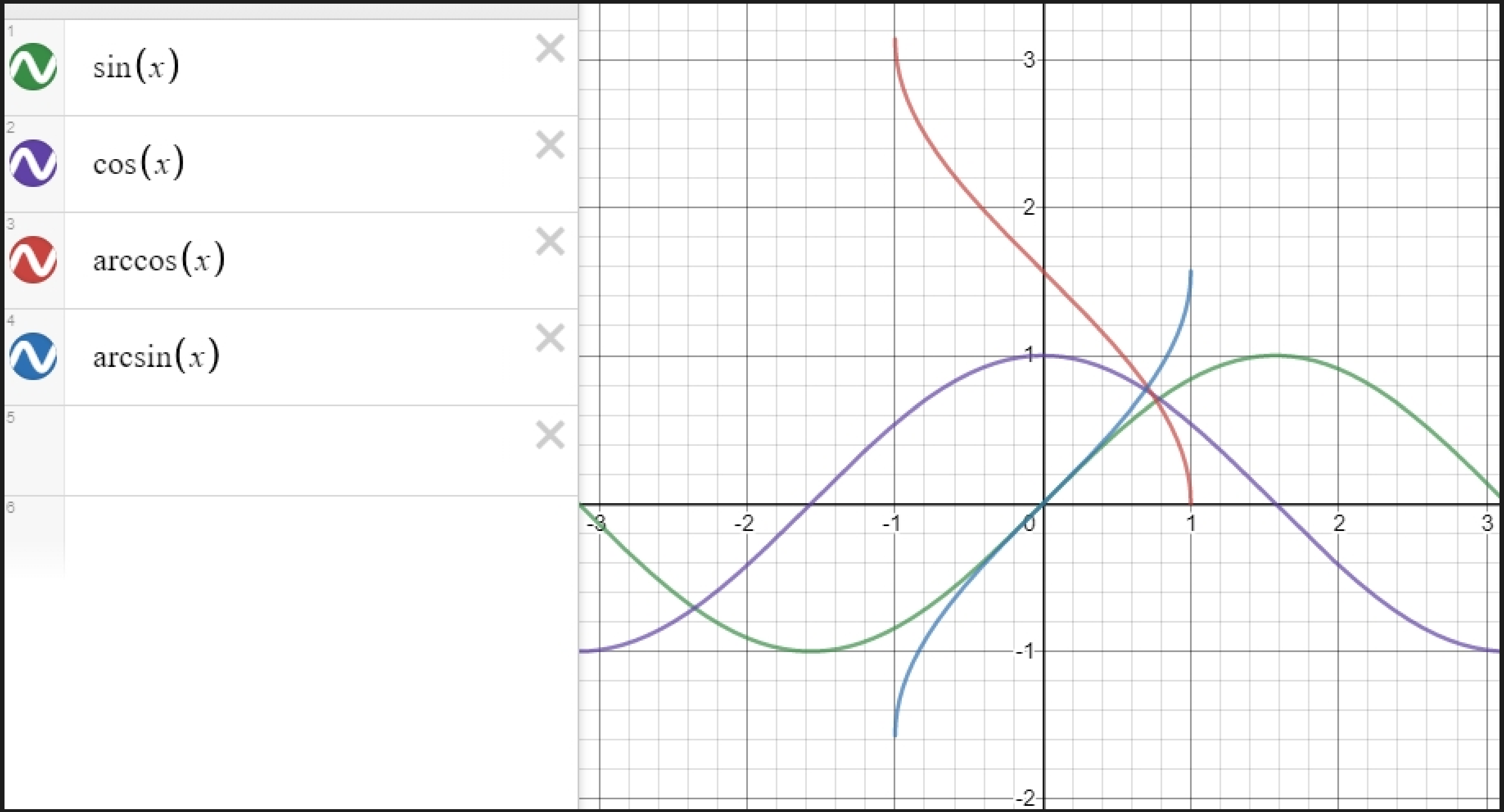
$$r \text{ rad} = \frac{g^\circ \times \pi \text{ rad}}{180^\circ}$$





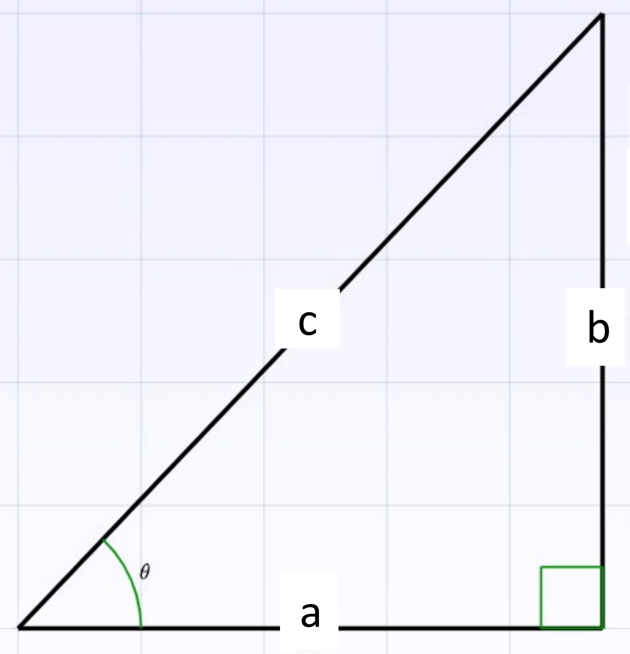
ArcSin/Cos

- 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:

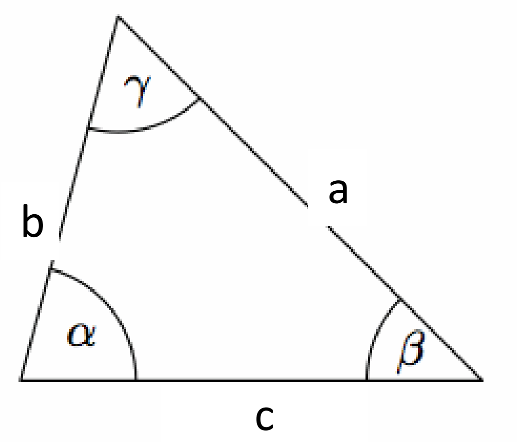
$$\cos^2\theta + \sin^2\theta = 1.$$

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

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

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

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



Regola dei seni
(per trovare i lati)

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

Regola dei coseni
(per trovare i lati)

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

Regola dei seni
(per trovare gli angoli)

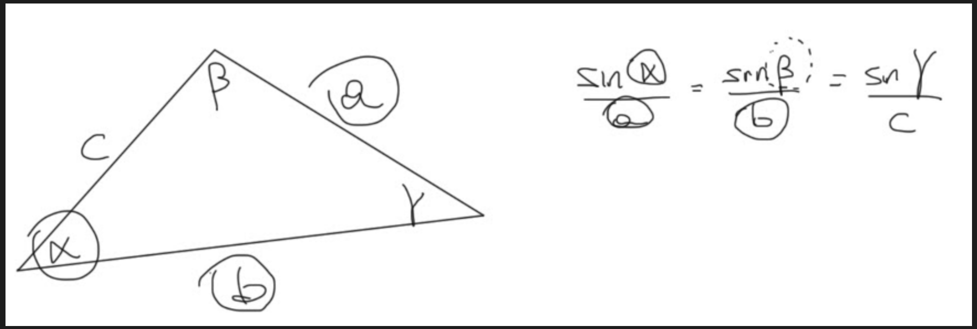
$$\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c}$$

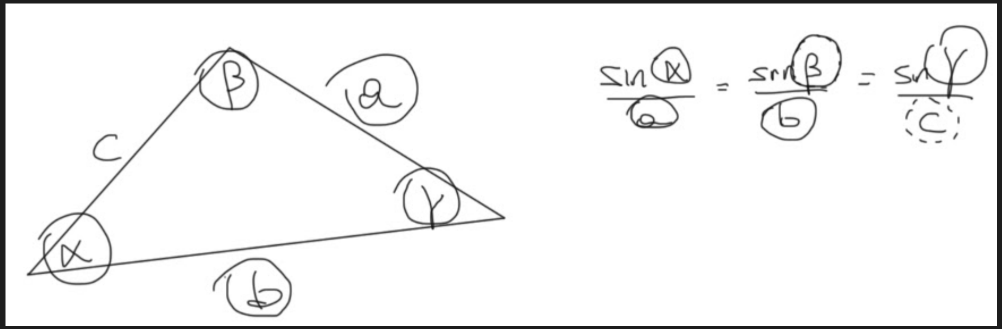
Regola dei coseni
(per trovare gli angoli)

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

Triangles

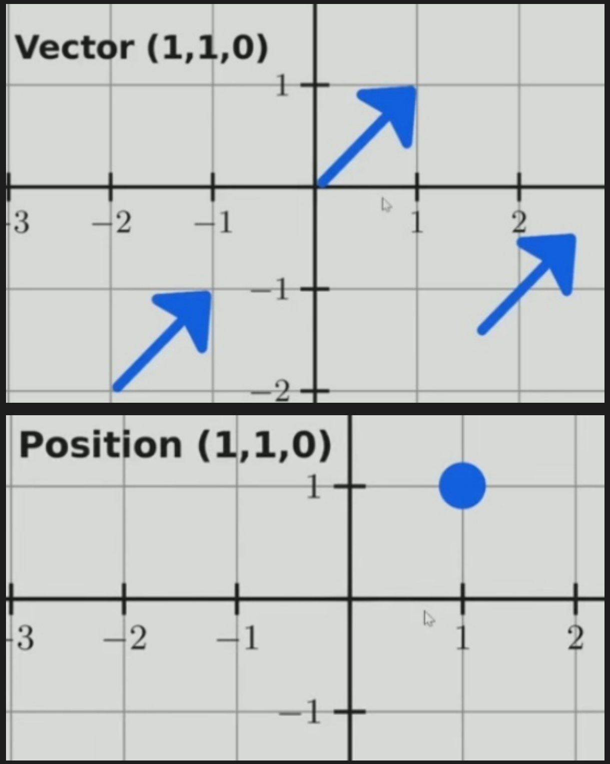
- Pythagorean theorem
- The sum of all triangles angles is 180
- Law of Sines
    - How to resolve a triangle if you have a, alfa, b => find beta => the sum of alfa, beta, gamma is 180 => find gamma => find c
- Law of Cosines





Vector math

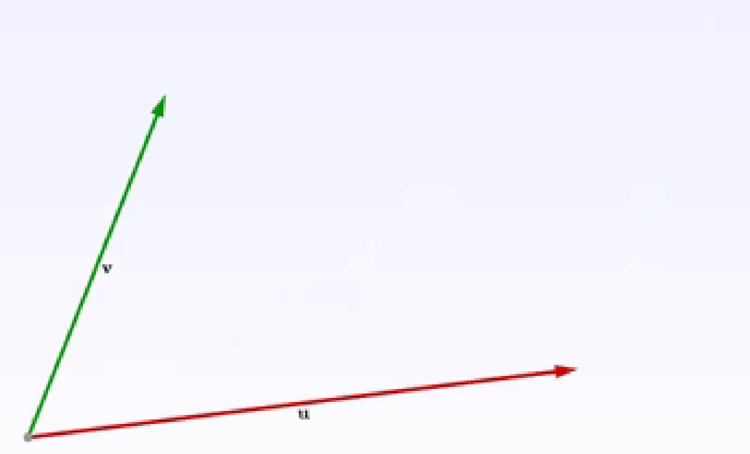
- A Vector is defined by:
  - Direction
  - Length (Magnitude)
- Easily confused with a position, it is NOT a position in Math
- But, in cg, we use to refer an obj pos with a Vector3: this time there is only ONE vector for position p: the one that starts from the origin



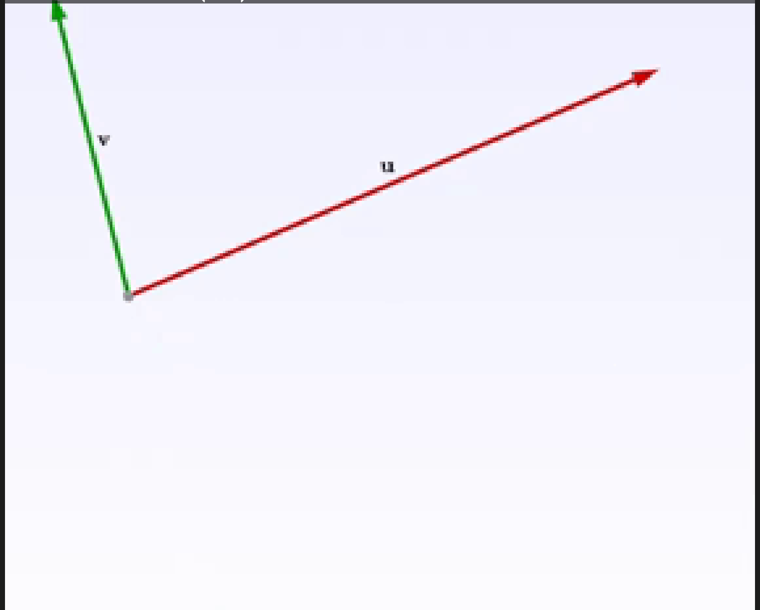
Vector math

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

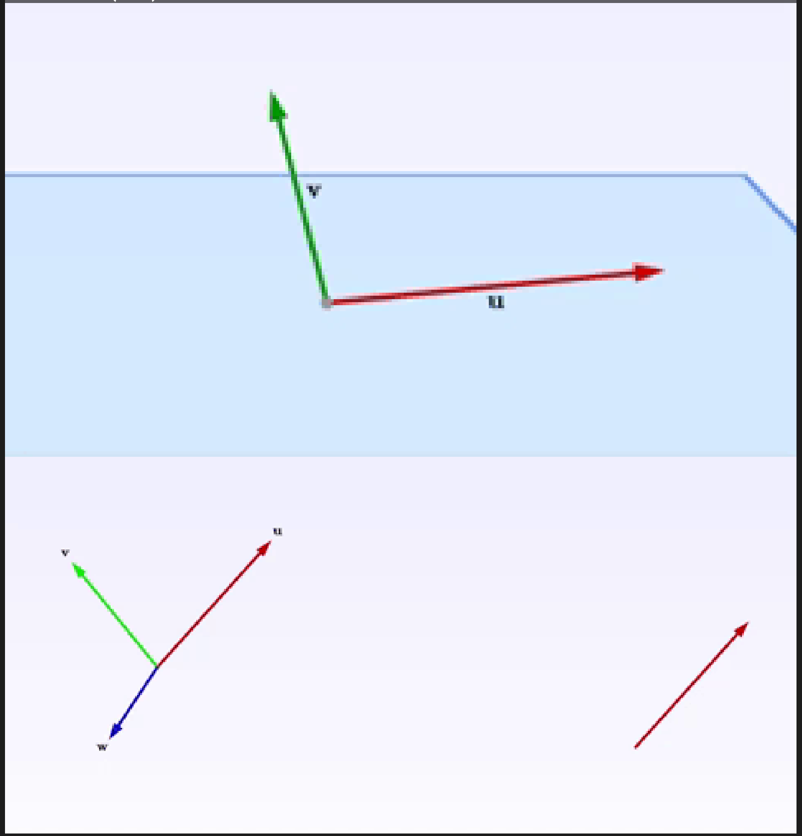
Sum (2D)

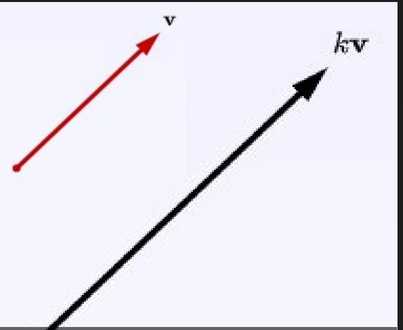


Subtraction (2D)



Sum (3D)





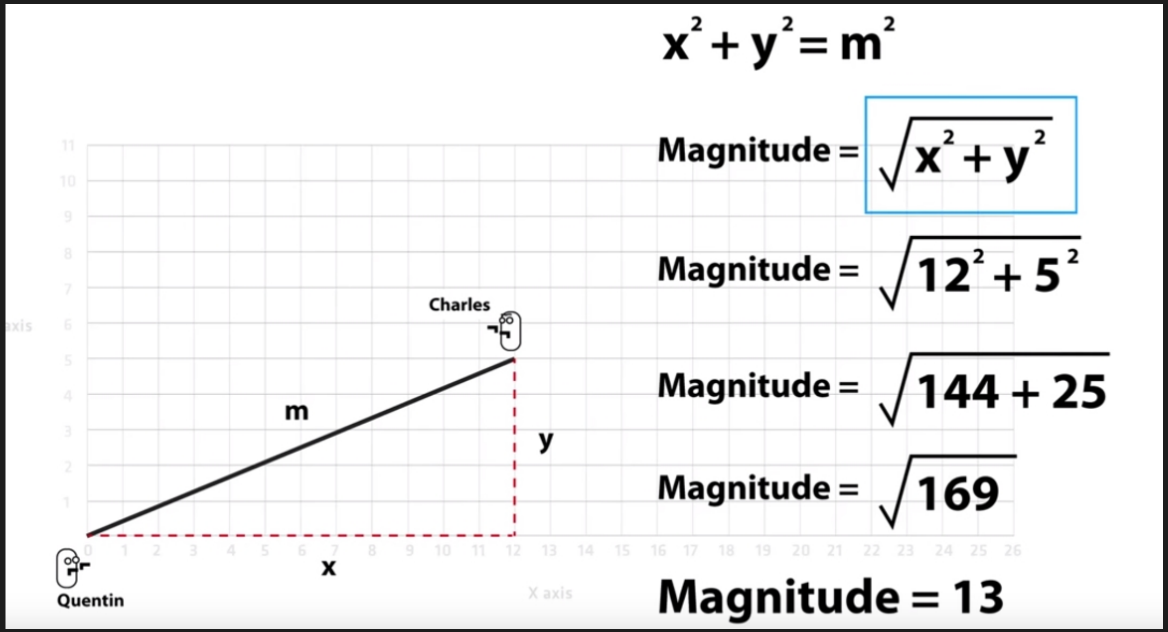
Multiplication

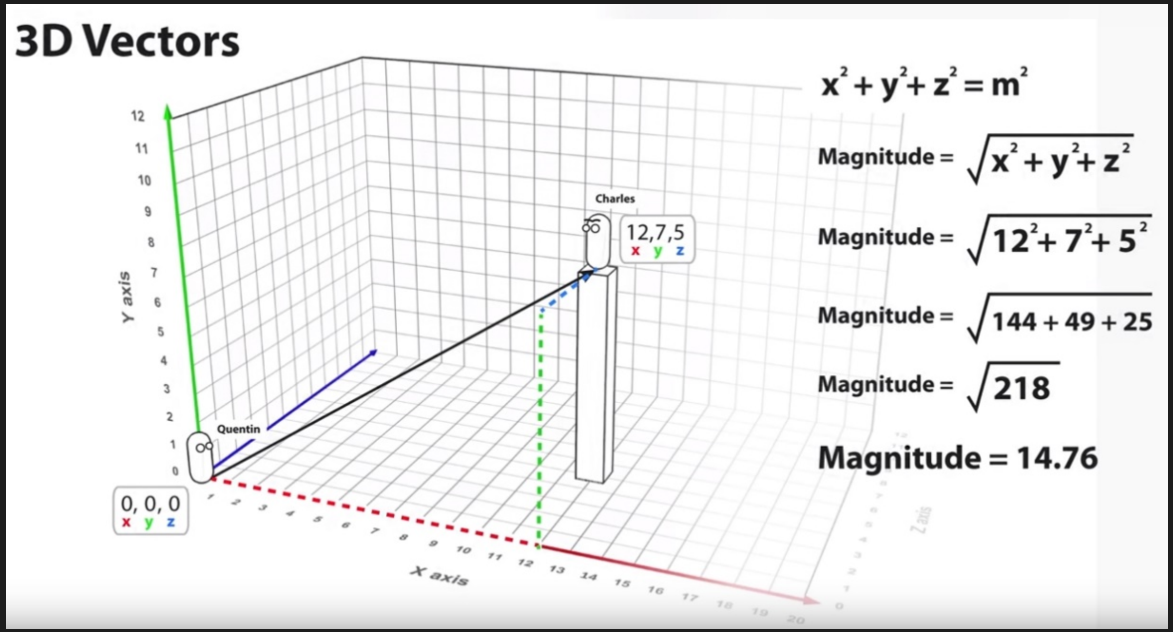
Vector math

- You can add and subtract direction vectors freely. However, technically speaking, points cannot be added to one another—you can only add a direction vector to a point, the result of which is another point. Likewise, you can take the difference between two points, resulting in a direction vector
- These operations are summarized below:
    - direction + direction = direction
    - direction – direction = direction
    - point + direction = point
    - point – point = direction • point + point = nonsense

Vector math

- Magnitude – vector length, always positive
- Distance between A and B: d = Vector3.Distance(A.position, B.position) or (B.position – A.position).magnitude
- If C is a vector from A to B or from B to A: d = C.Magnitude
- Given a vector V(x,y) with magnitude |V| = 3 its normalized vector Vn is (x/3, y/3), where |Vn| = 1

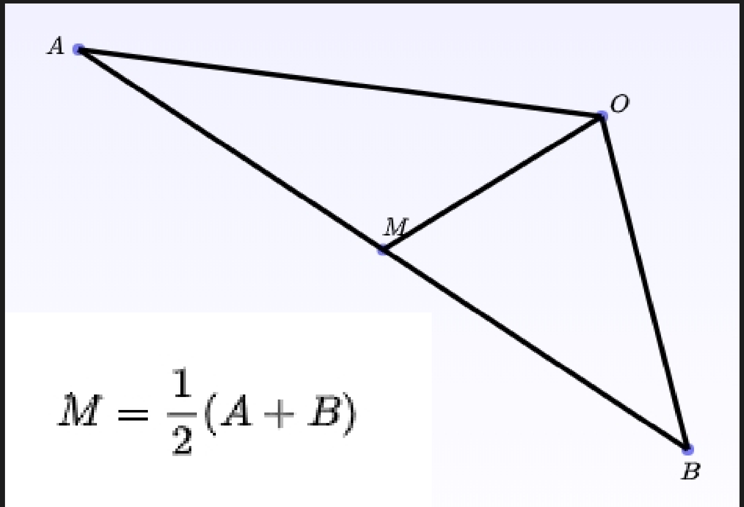


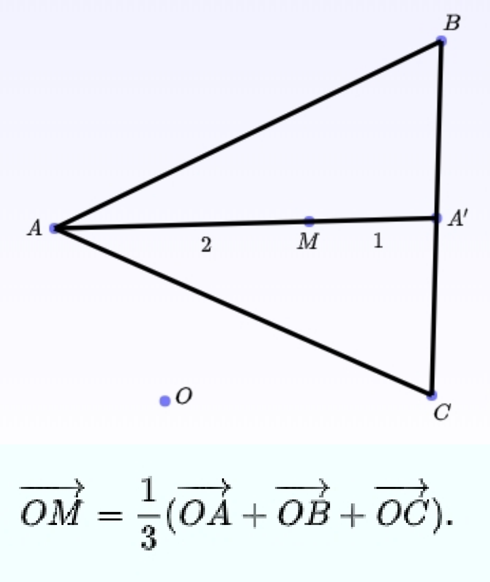


Images from Unity3D official guide

Vector math

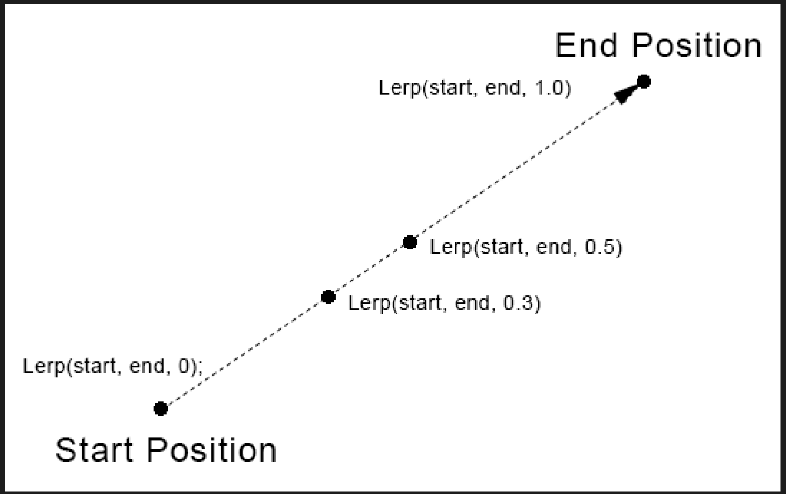
- Middle Point
- Center of mass

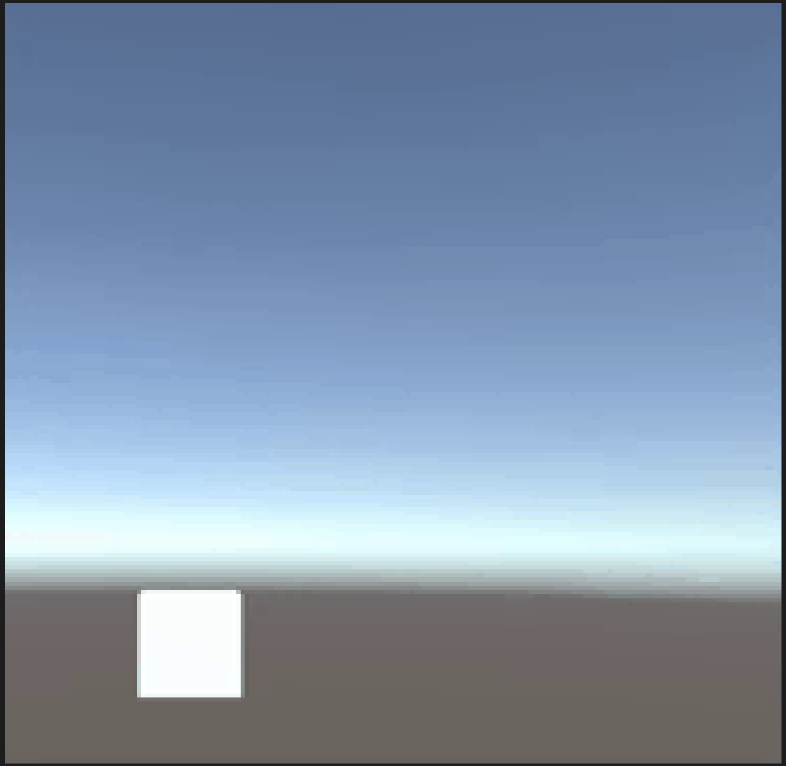




Linear interpolation

- 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);



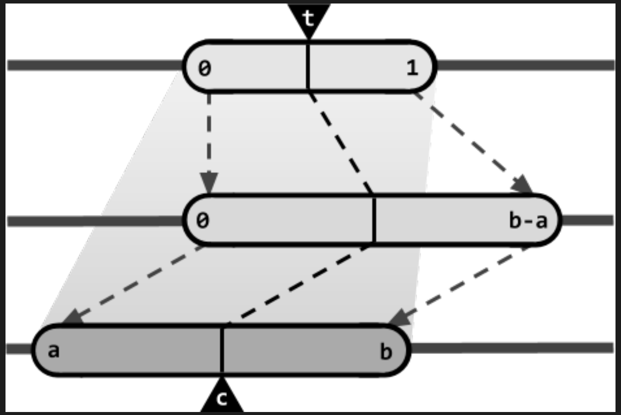


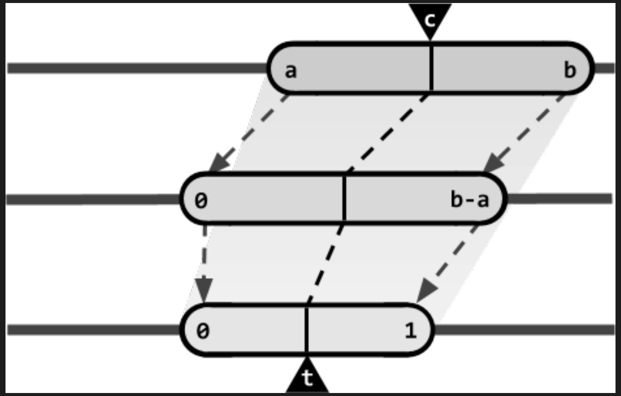
[SimpleLerp_Start_00, Simple_Lerp_Start_00.scene, SimpleLerp.cs]

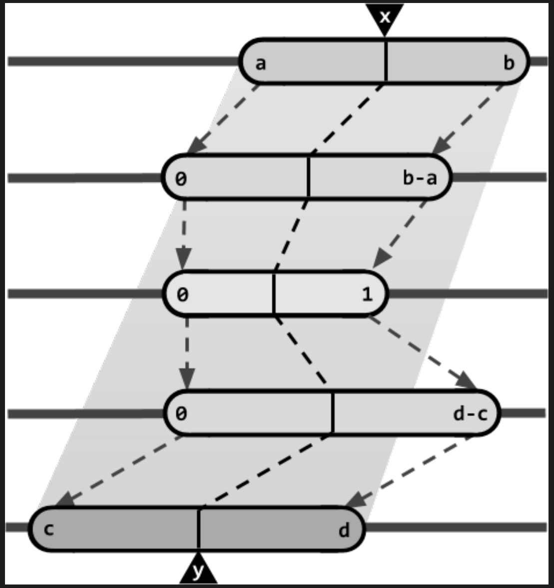
Linear interpolation

- Lerp
- InverseLerp
- Map

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



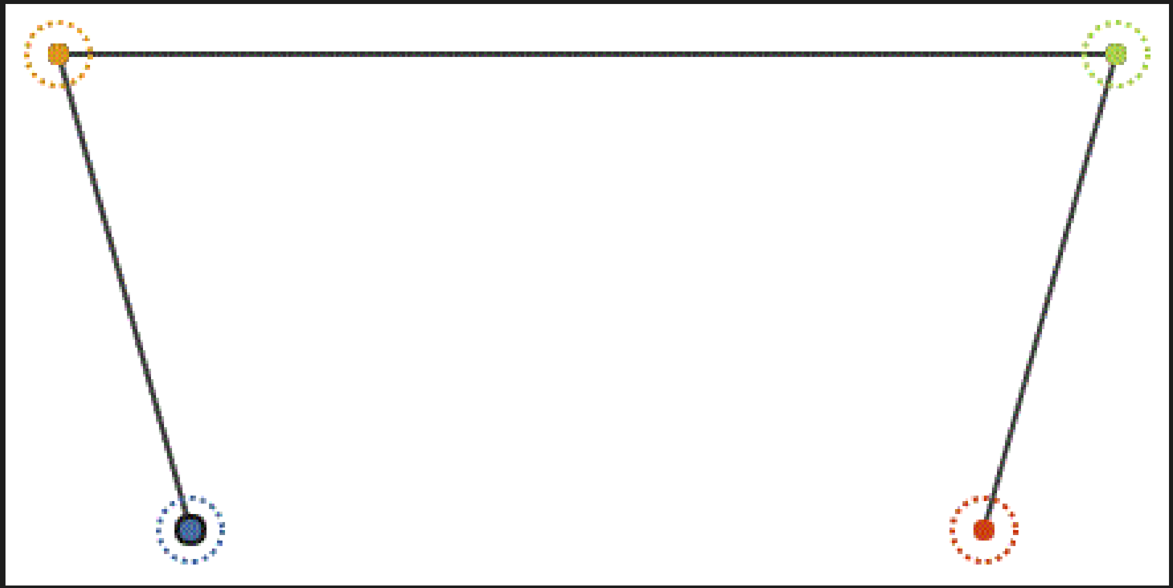




Linear interpolation

- Use interpolation to create a BezierCurve
- https://acegikmo.com/bezier/

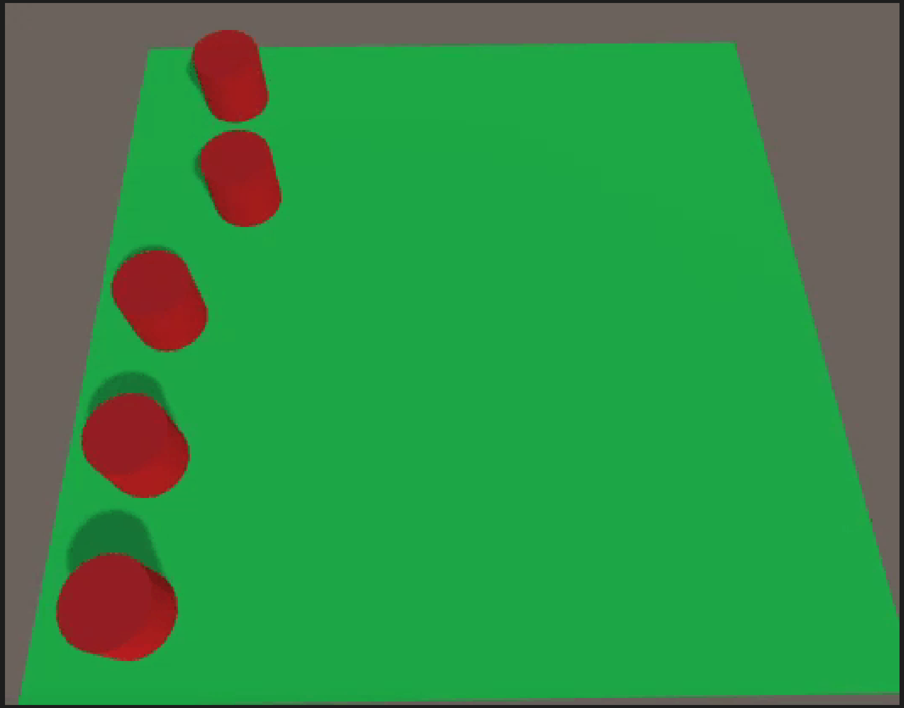
[LerpBezier_End, LerpBezier.cs]

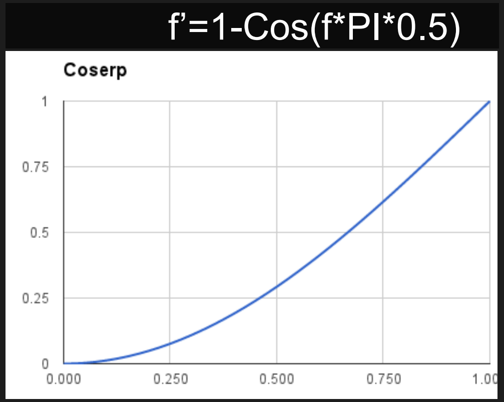


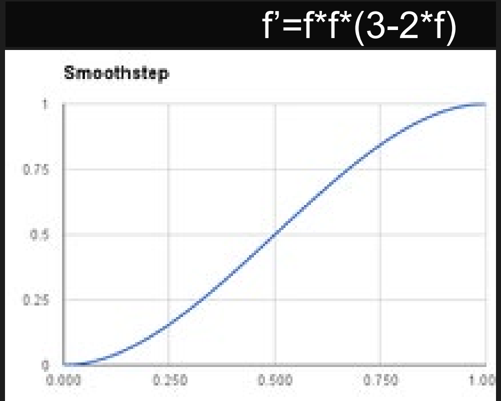
```
Vector2 Bezier (Vector2 p0, Vector2 p1, Vector2 p2, Vector2 p3, float t)
{
    // Lerp between the control points
    Vector2 a = Vector2.Lerp(p0, p1, t);
    Vector2 b = Vector2.Lerp(p1, p2, t);
    Vector2 c = Vector2.Lerp(p2, p3, t);

    // Lerp between the lerped points
    Vector2 d = Vector2.Lerp(a, b, t);
    Vector2 e = Vector2.Lerp(b, c, t);

    // Lerped between the lerped points (again!)
    return Vector2.Lerp(d, e, t);
}
```

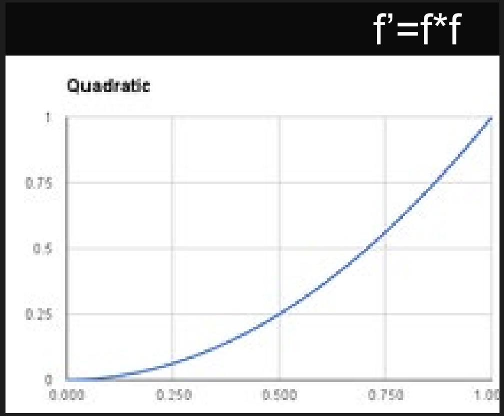
Linear interpolation - Advanced

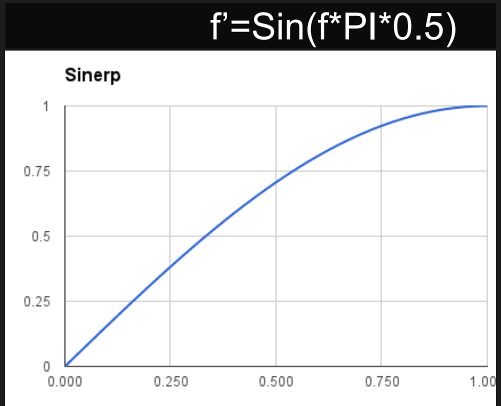
- Different types of interpolations maps [0,1] into [0,1], not in a linear way
- Start from [Lerp_Advanced_Start_00]
  - Each obj has LerpPingPong.cs: allows to switch between FORWARD/BACKWARD state, while customLerp.cs allows other kind of lerps, different from linear
- Duplicate LerpPingPong.cs and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



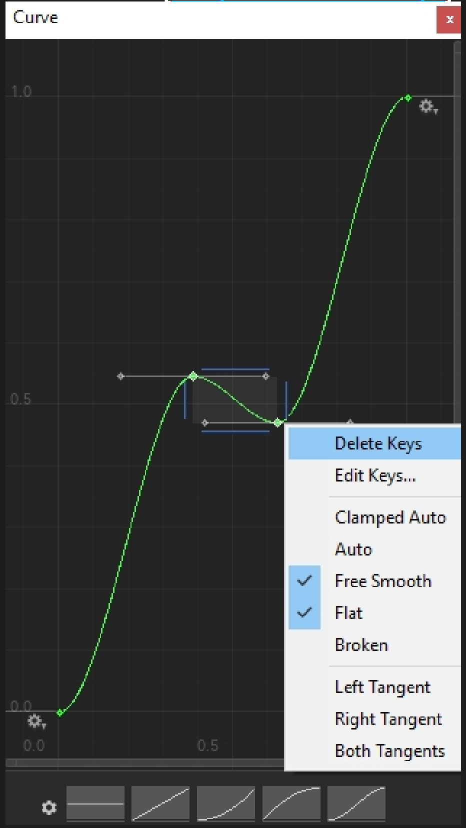








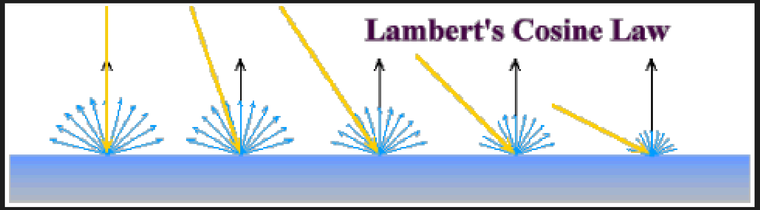
- [LerpPro.UPackage]



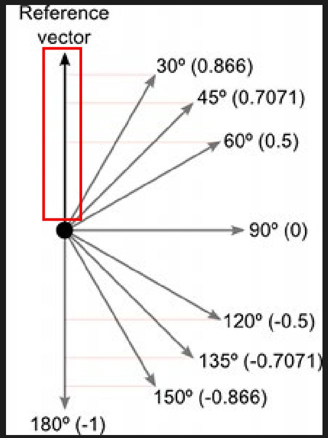
Dot Product / Projection

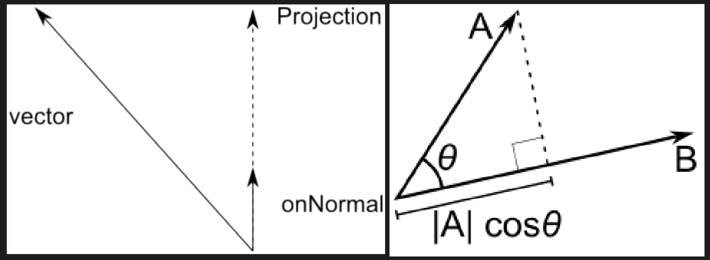
- Dot(A,B) = |A| |B| cos(α)
- To know the angle between A,B: cos(α) = Dot(A,B) / |A| |B|
- The angle between A and B doen't change if A,B are normalized, then:
    - cos(α) = Dot(A.normalized,B.normalized)
- alphaInRads = Mathf.Acos(cos_alpha)

- Lambert's law



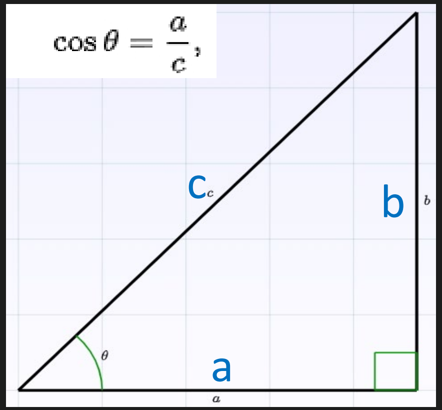
$$\mathbf{u} \bullet \mathbf{v} = |\mathbf{u}||\mathbf{v}|\cos(\theta) = x_1 \times x_2 + y_1 \times y_2 = \mathbf{u}\mathbf{v}^T$$





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

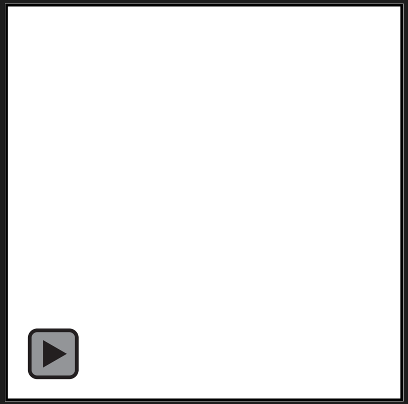
- In a right triangle cos(α) = a/c [1]
    - If a and c are normalized: Dot(a,c) = cos(α) [2]
    - From [1] we know that
        - a = c * cos(α) = c * Dot(a,c)
    - Then, projection of c on a = c * dot(a,c)

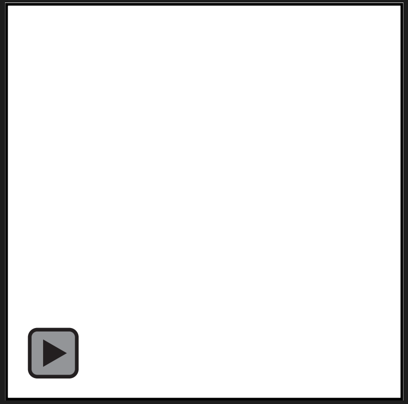


[dotProduct_Projection_01.UPackage, dotProduct_Projection.scene, vectorProjection.cs]

Dot Product / Projection

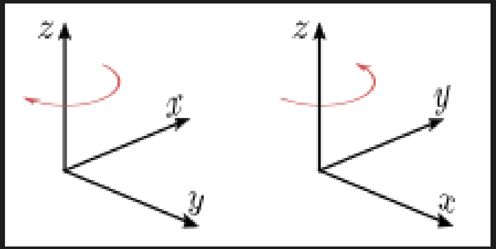
- Dot product visualization



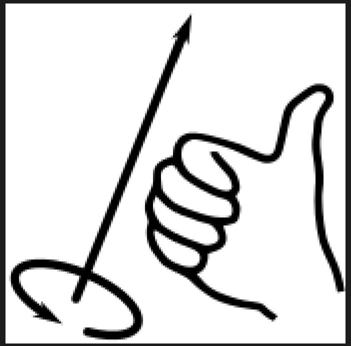


Cross Product

- 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

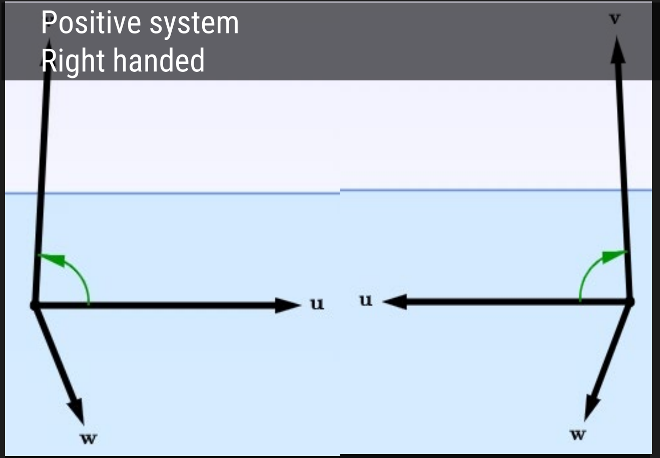


Left-handed coordinates on the left, right-handed coordinates on the right



Right-hand rule for curve orientation

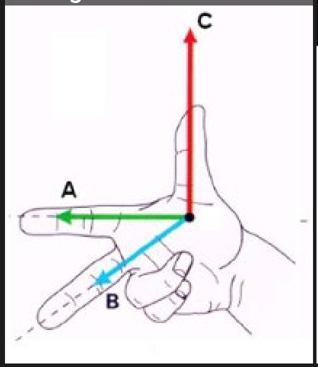
Positive system
Right handed



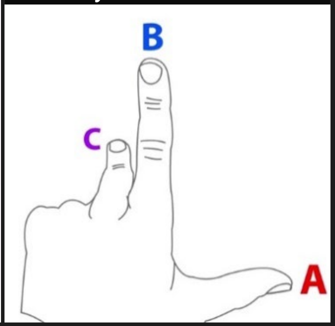
Positive rotation on w axis

Negative rotation on w axis

Positive system
Right handed



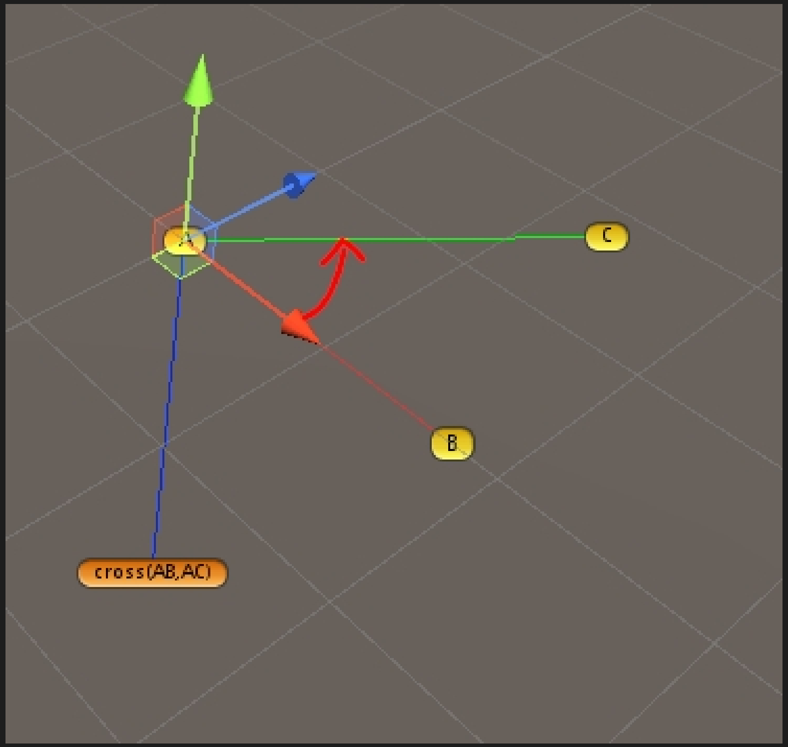
Negative system
Left handed
Unity



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]

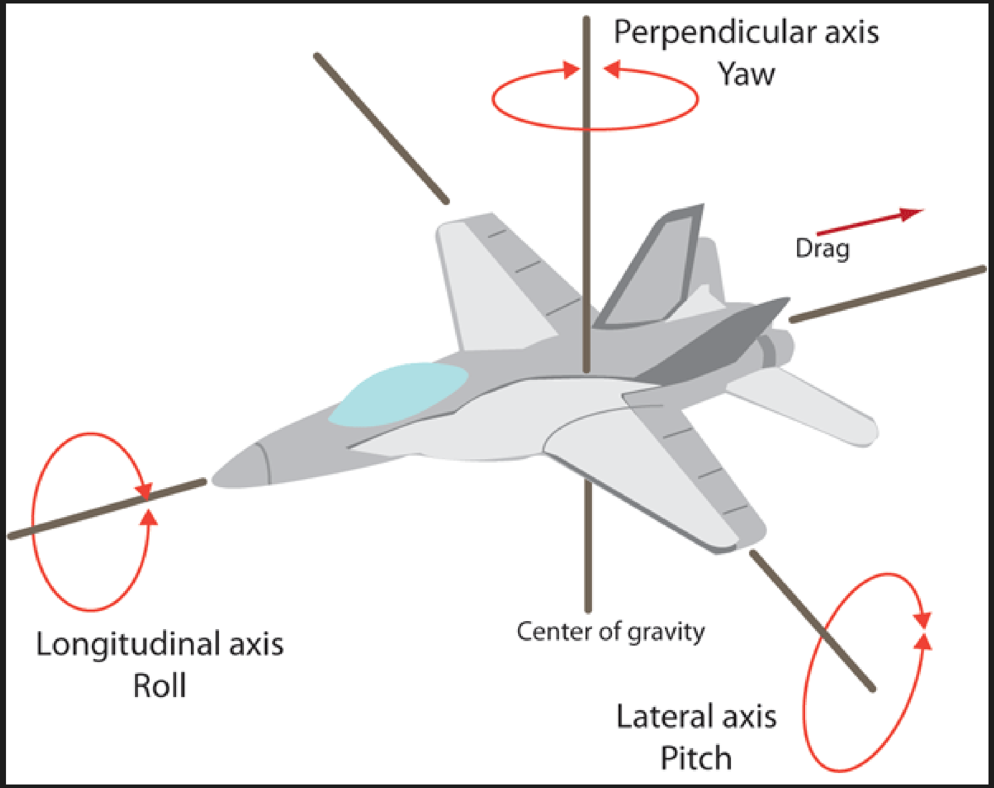


Unity is LeftHanded: the angle between AB and AC is negative (counterclockwise, because we need to orient the left thumb towards the floor)

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

Orientation

- Position – Vecotr3
- Scale – Vector3
- Velocity – Vector3
- Force accumulator – Vector3
- Orientation – Vector3 (?)

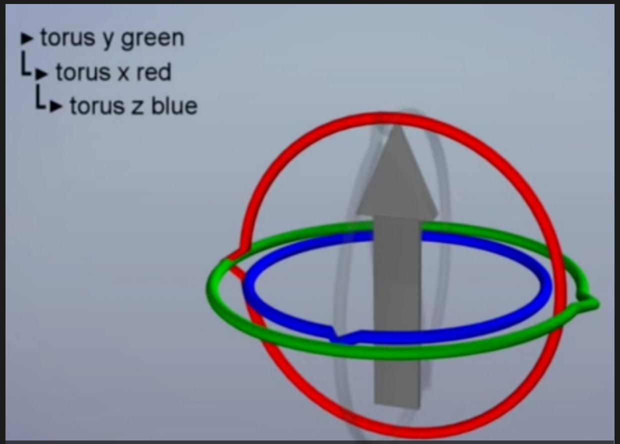


Orientation

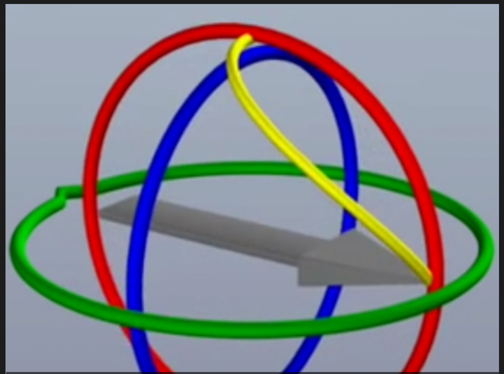
- Vector3 is not a good data structure for orientation
  - Combine multiple rotations
  - Same resulting orientation for more than one PYR combinations
  - Interpolation
  - Gimbal Lock

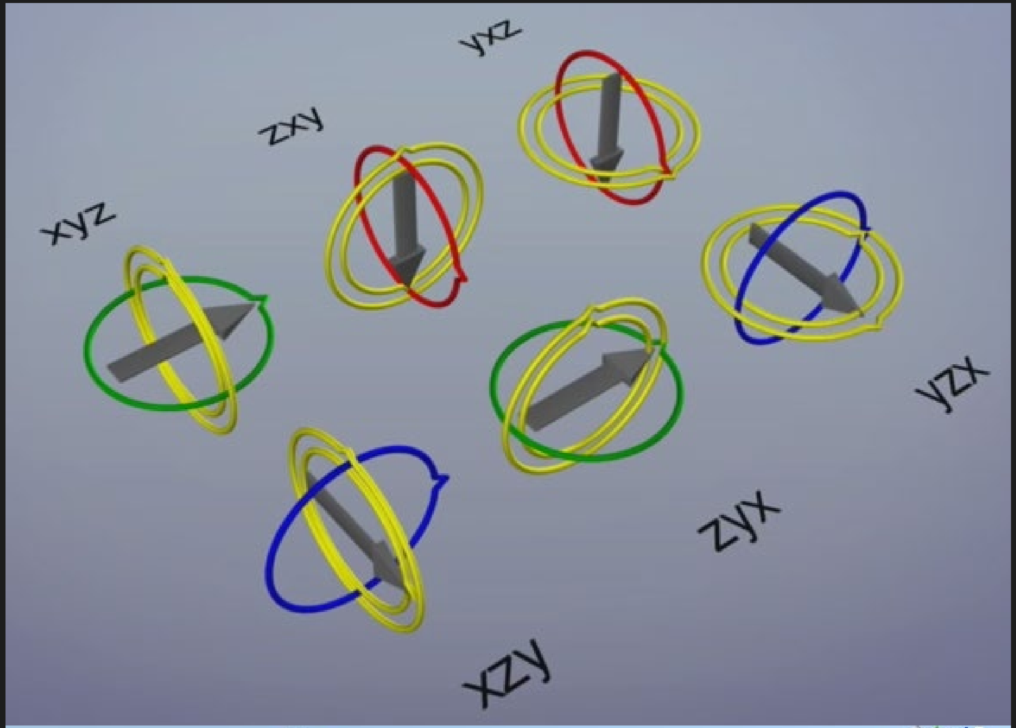
Unity Rotation Order: Z, X, Y

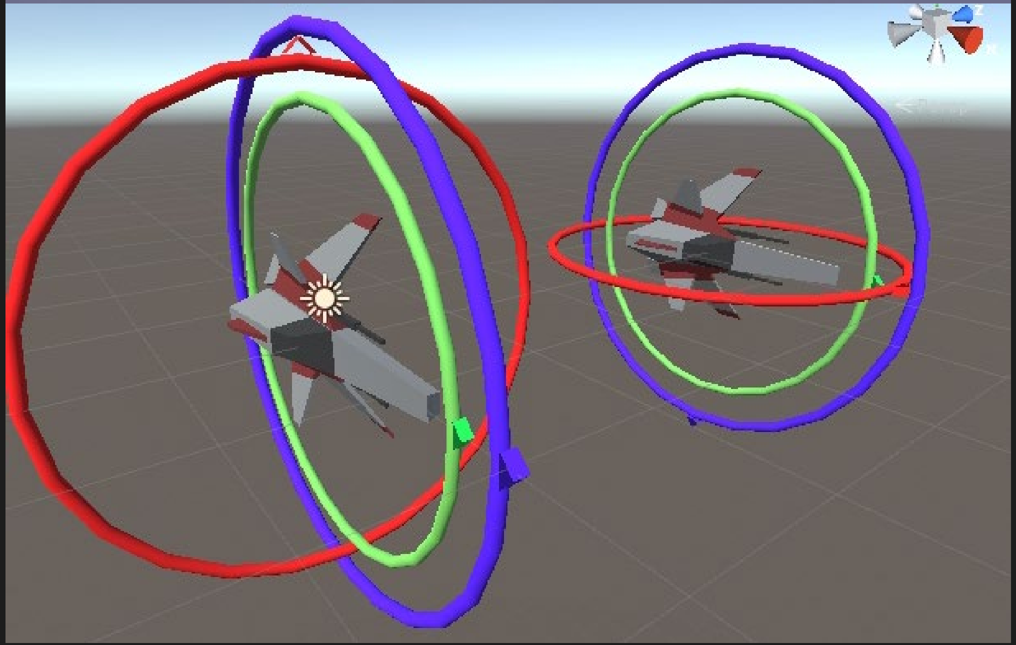
[GimbalLock_01.UPackage]



The gray axis is missing

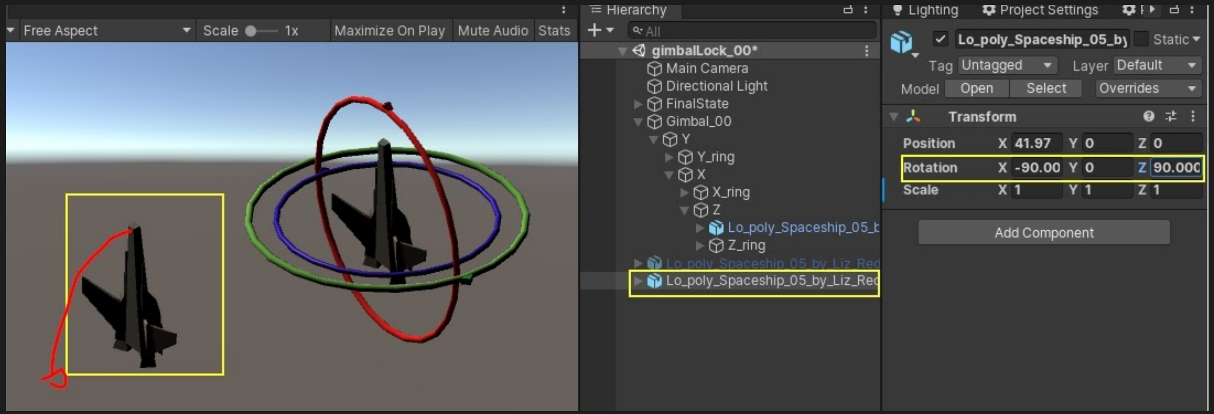

The object can move unexpectedly in between keyframes





Orientation

- Gimbal Lock
  - Try to rotate Gimbal_00 (-90,0,90) AND (-90,90,0): they end with the same orientation
  - If we move the Spaceship outside the hierarchy, we find that we can't rotate it along the missing axis using Transform.rotation Vector3 values



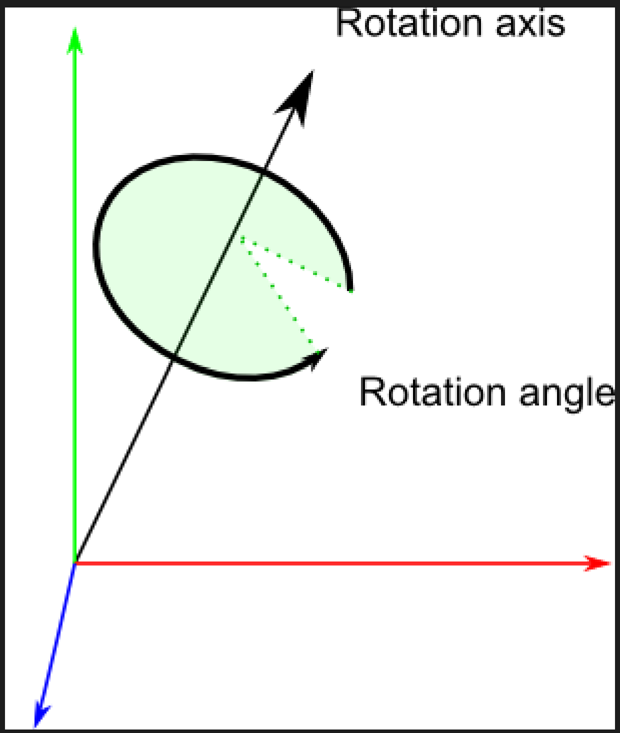
[GimbalLock_01.UPackage]

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

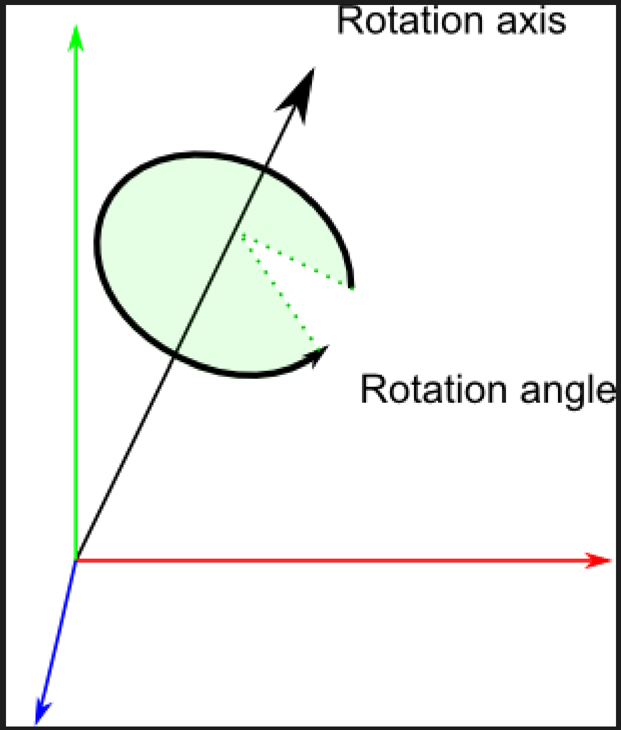
- Essentially the relationship of these numbers mirrors the relationship of the three dimensions to each other! If you rotate 180 degrees in two dimensions (like x and y), it is essentially the same as rotating that same amount in the third dimension (z). It is this relationship that allows quaternions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

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

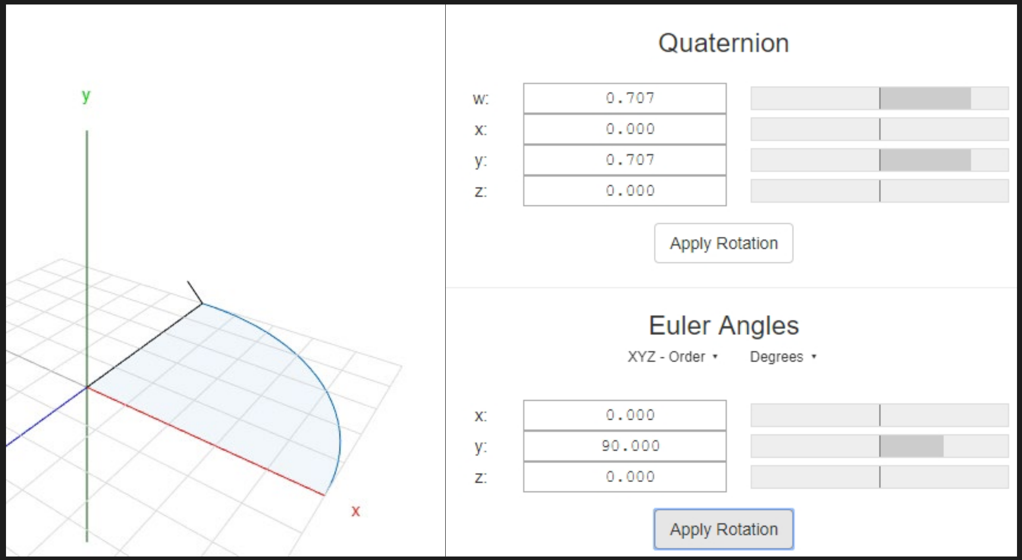


Quaternions

- 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
- www.andre-gaschler.com/rotationconverter



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]

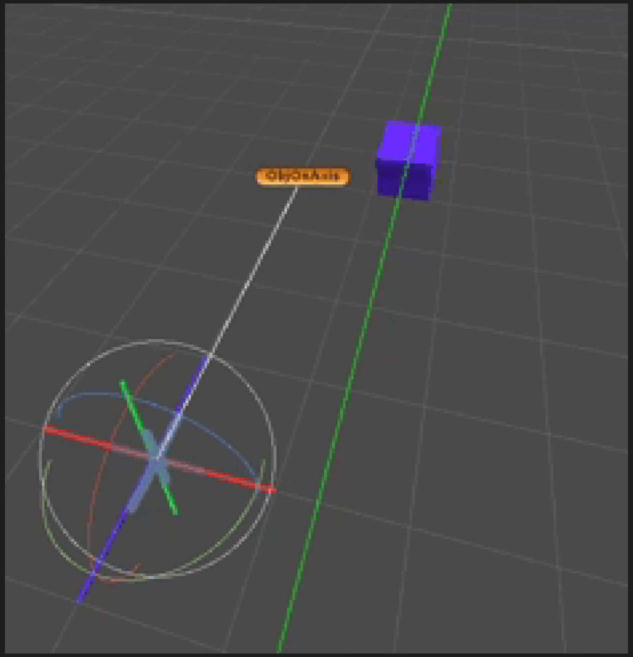
- Quaternion.Angle(Quaternion q1, Quaternion q2)
  - 2 objs AngleA & AngleB must have a quaternion with the same rotation axis. Starting from this situation, this is useful to know the difference between the 2 quaternions' spin
- Concatenation
  - Obj.rotation = Aq*Bq is the same as set the hierarchy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

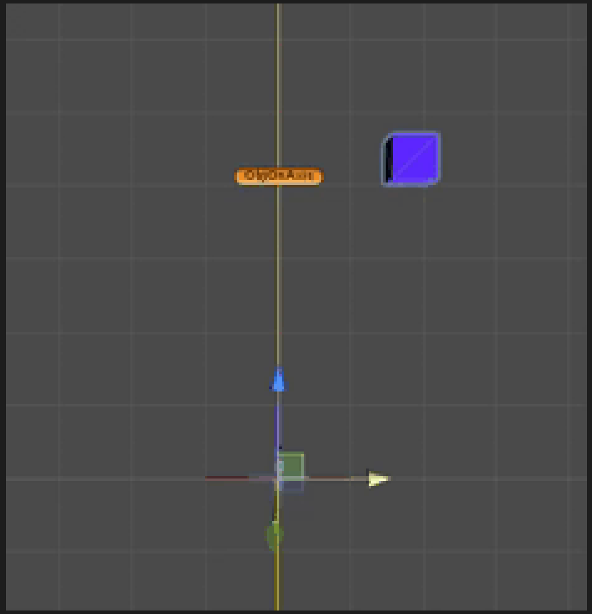
Rotate around custom axis

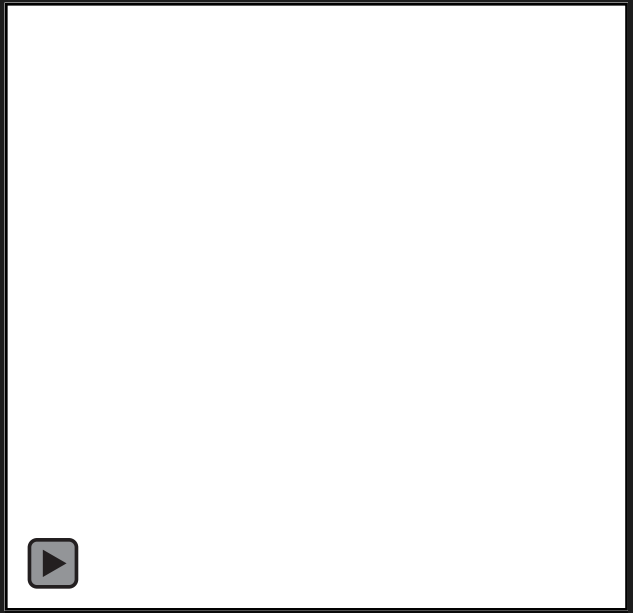


- Position update: Quaternion * Vector3
  - We need: Axis of rotation, degrees, transform.position
  - Rotate around a custom Axis passing <u>through the origin</u>
- Rotation update: Quaternion * Quaternion
  - We need: Axis of rotation, degrees, transform.rotation
  - Rotate around a custom Axis passing <u>through the ObjToRotate PIVOT</u>

- Interpolation Slerp()

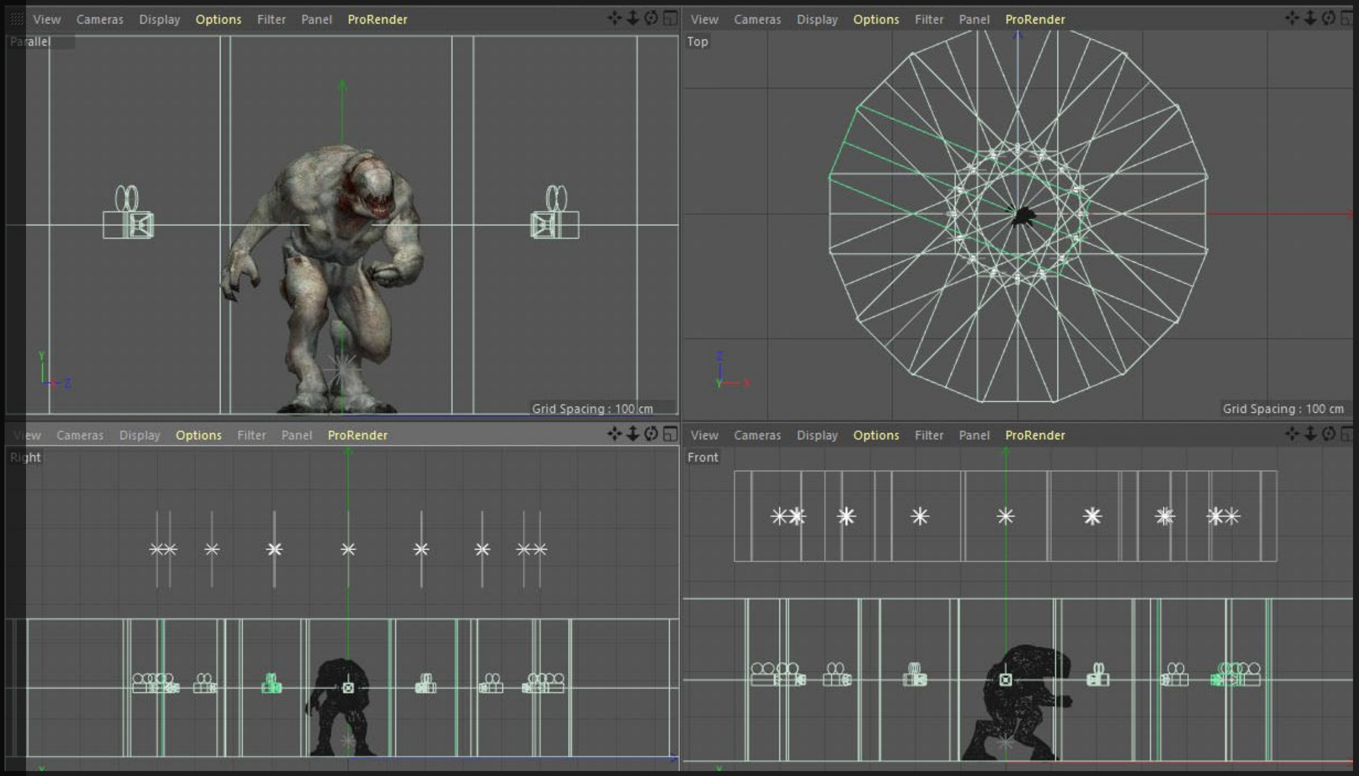
[Quaternions_02]





Linear vs Spherical Interpolation

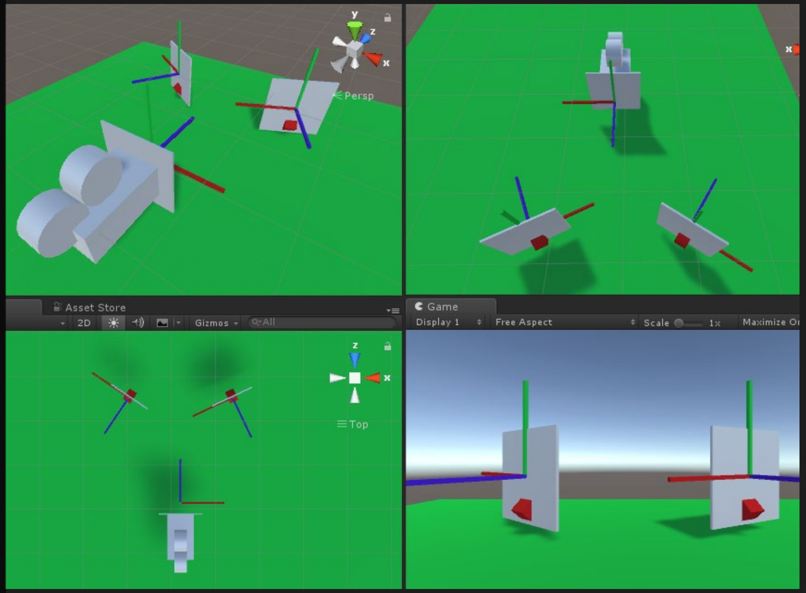
Billboard facing camera



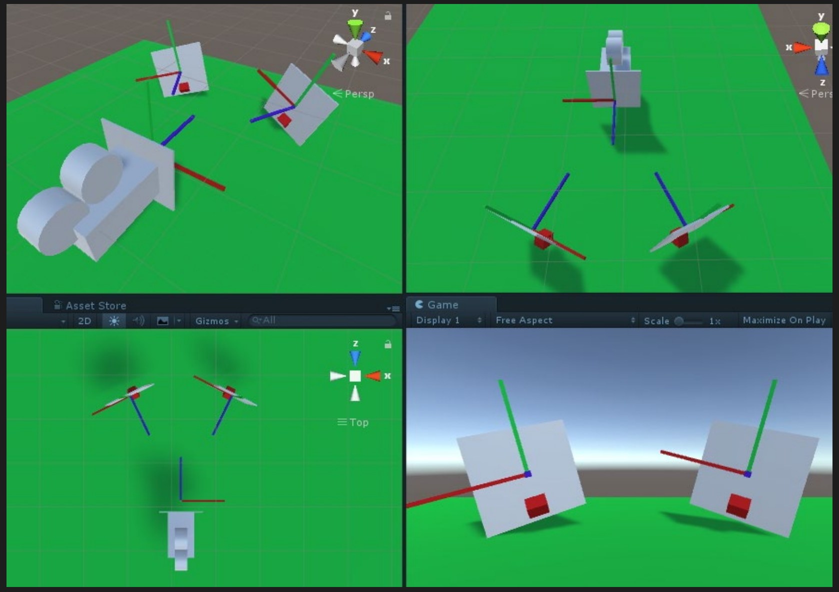
Billboard facing camera

[Billboard_03, CameraFacingBillboard.cs]

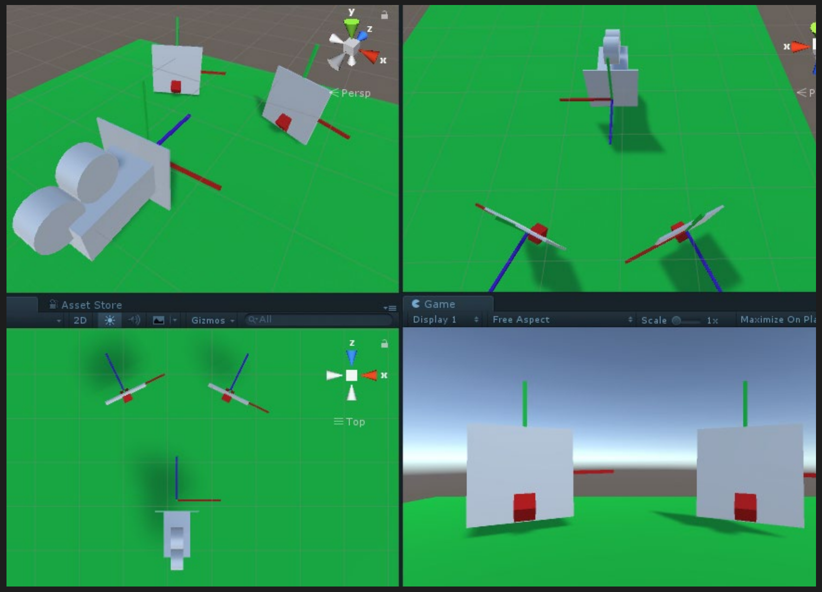
No billboard effect



FromToRotation / RotateTowards (smooth)

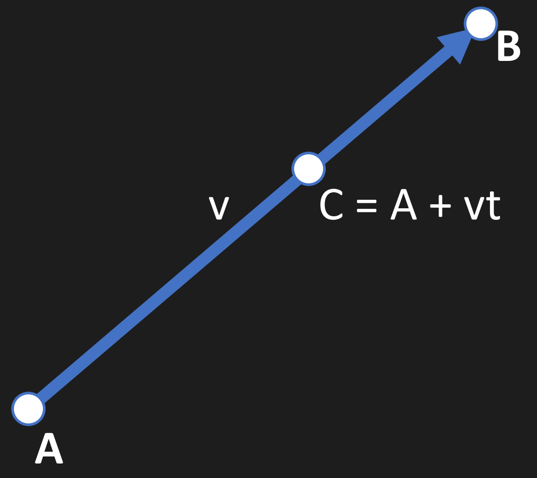


LookRotation (upVector locked)



Parametric Line

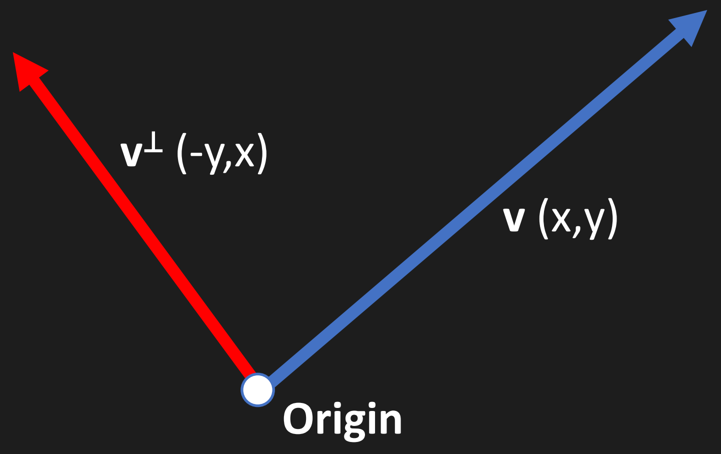
- We can see a line between A and B like a sequence of points $C = A + vt$, where t is in range $[-\infty, +\infty]$



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)

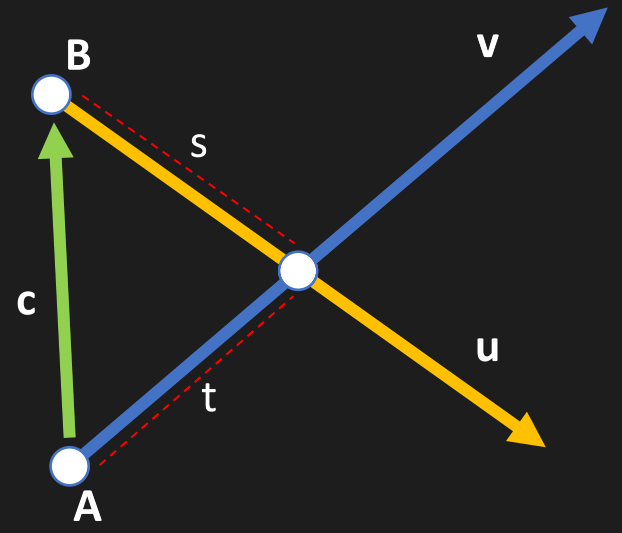
- We should solve the equation
  - $A + vt = B + us$
  - Since $B - A = c$ we have:
  - $vt = us + c$
- to know t and s. We have to isolate t or s:

$vt = us + c$

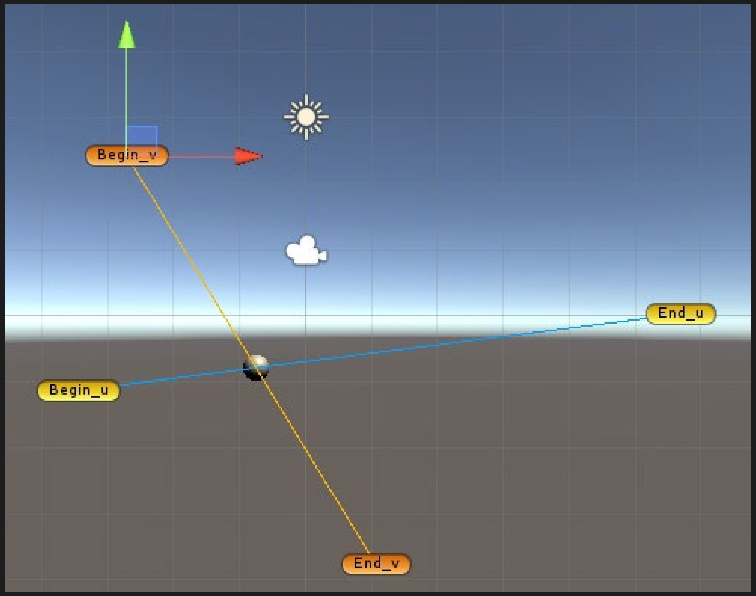
$Dot(u^{\perp},vt) = \underline{Dot(u^{\perp},us)} + Dot(u^{\perp},c)$ // remember that $Dot(u^{\perp},u) = 0$

$Dot(u^{\perp},vt) = Dot(u^{\perp}, c)$

$t = Dot(u^{\perp}, c) / Dot(u^{\perp}, v)$



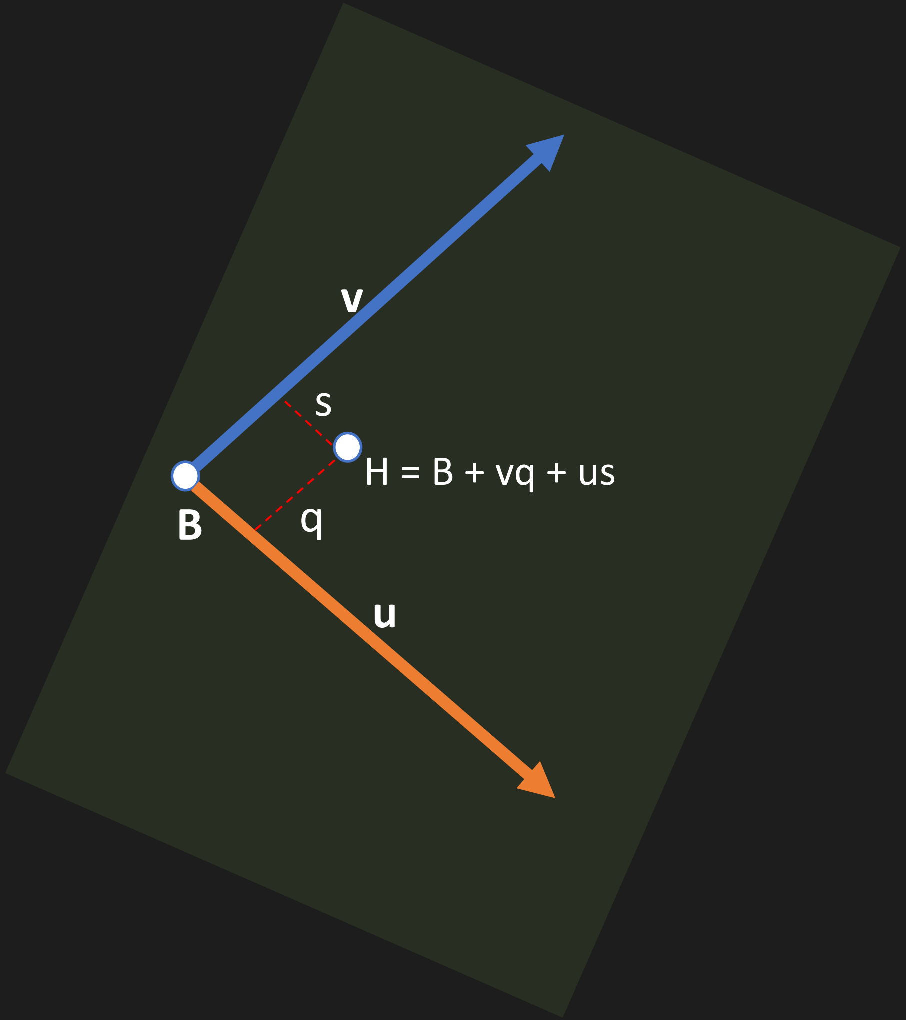
- See how the implementation of VectorUtils.GetSegmentIntersection(), in LineLineIntersection.cs works



[LinesPlanesOps_03, VectorUtils_End.cs]

Parametric plane

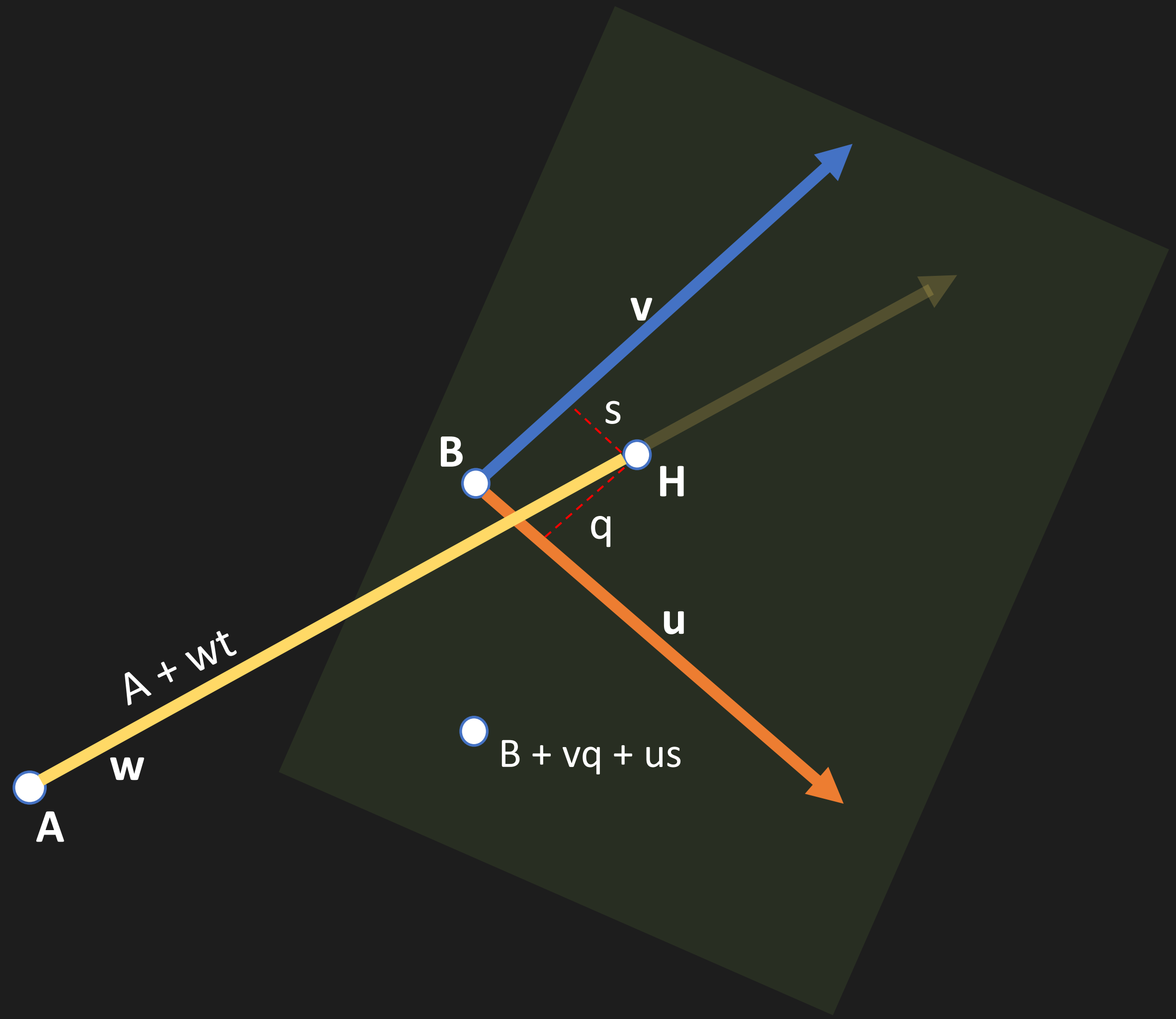
- Any point on a plane can be defined as the sum of various lengths of two given vectors
- See DrawPlane/DottedPlaneDrawerStart.cs



[LinesPlanesOps_03]

Line Plane Intersection

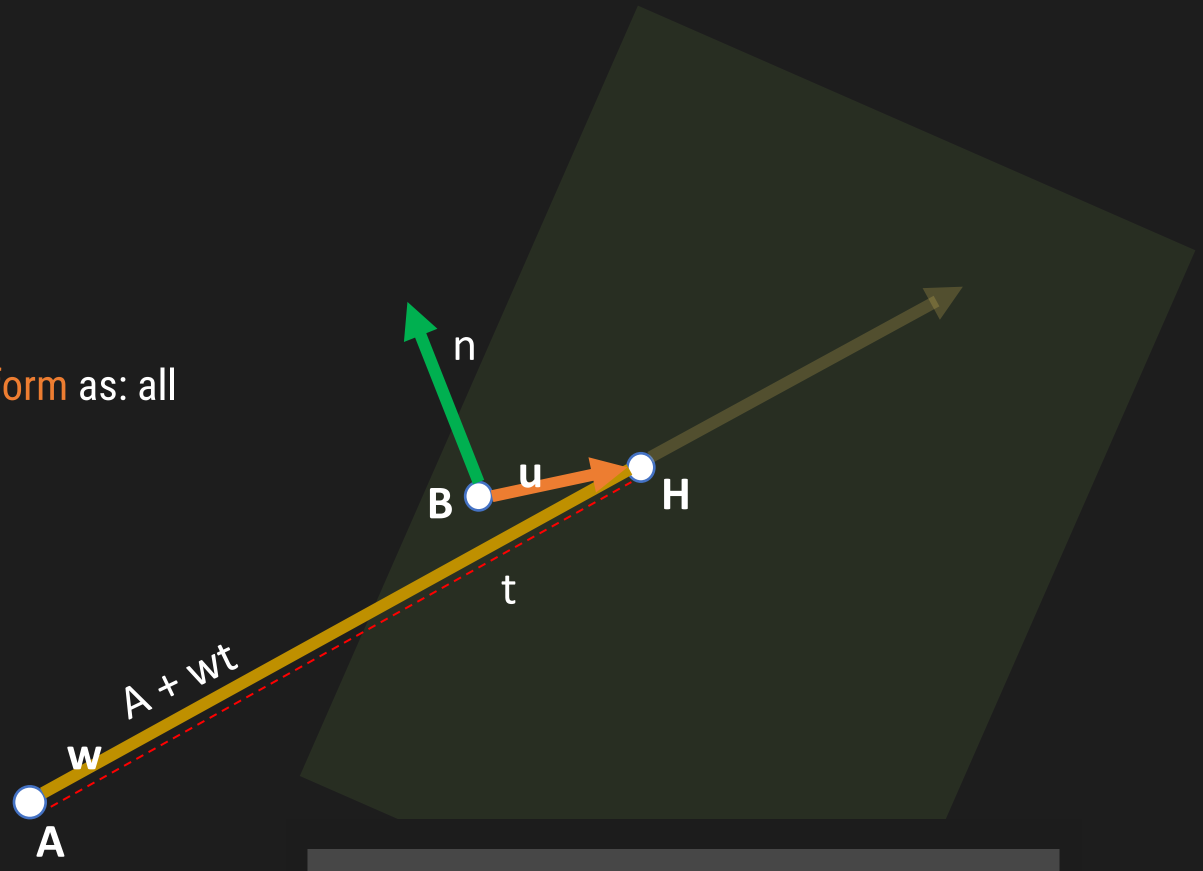
- We should solve the equation $A + wt = B + vq + us$



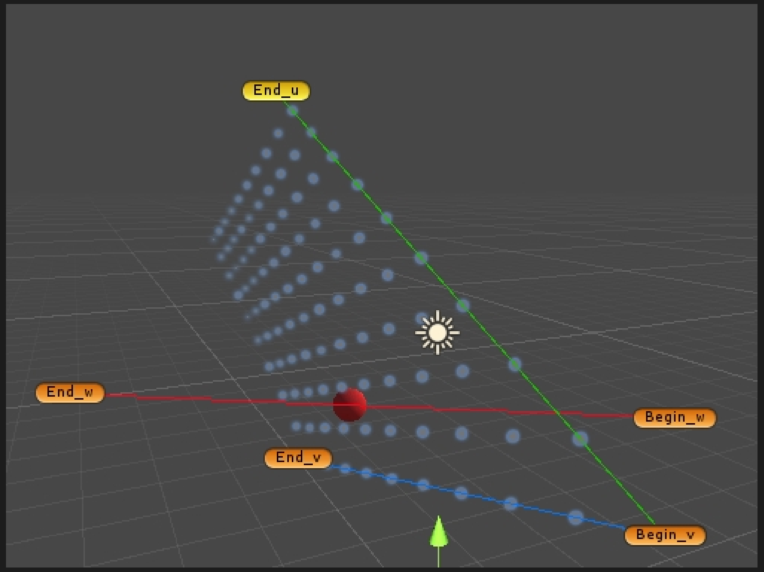
[LinesPlanesOps_03]

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)



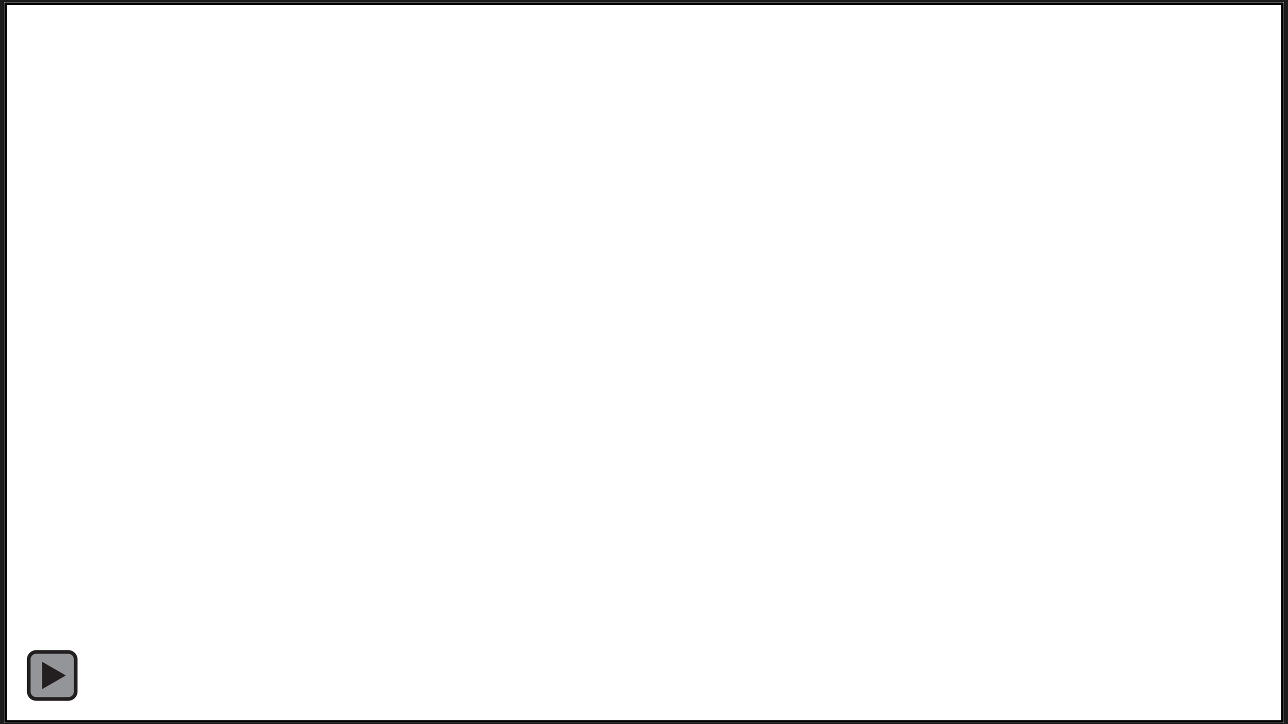
- Activate DrawPlane GObj to draw plane using DottedPlaneDrawer.cs
- Activate LinePlaneIntersection and finish the implementation of VectorUtils.GetLinePlaneIntersection(), used in LinePlaneIntersection.cs



[LinesPlanesOps_03, vectorUtils_End_02.cs]

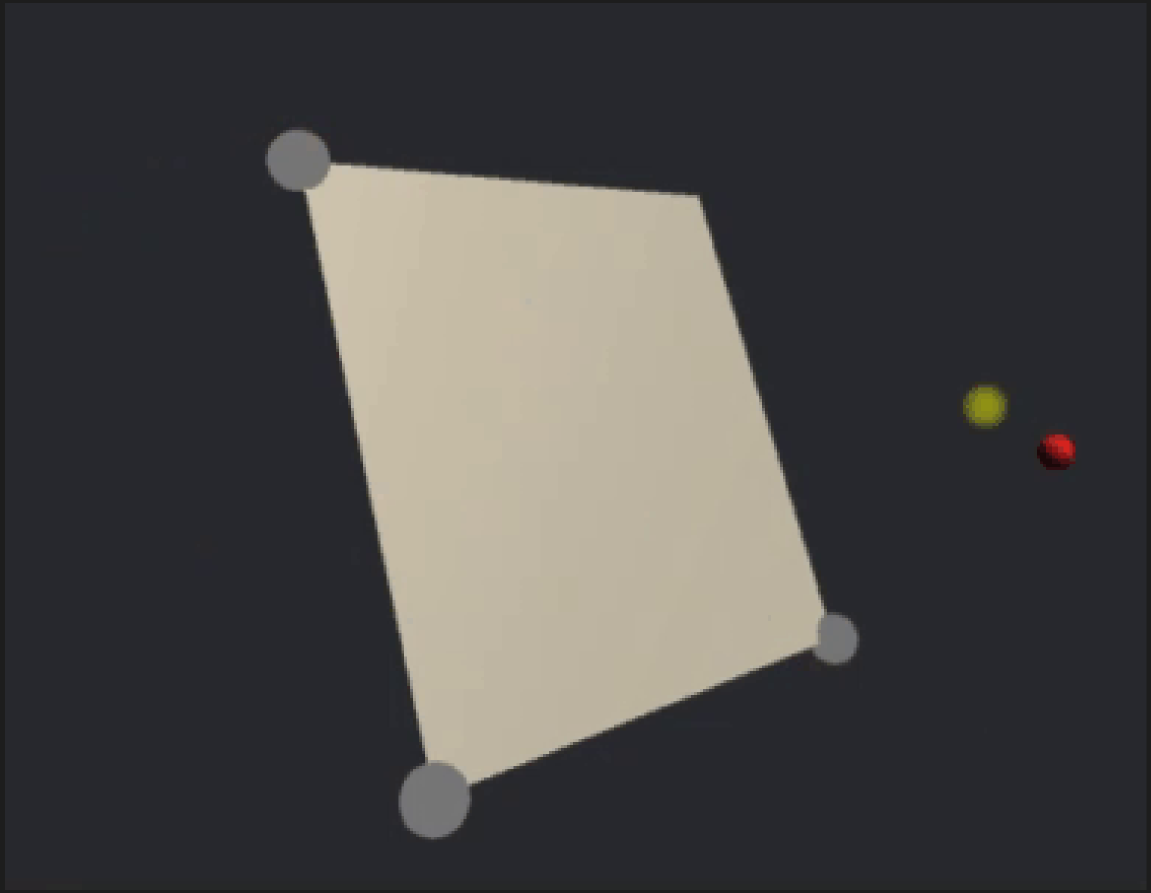
Line Rectangle Intersection

- the [line] x [rectangle] intersection test can be reduced to a [line] x [line segment] test, by placing a line segment in the rectangle's diagonal based on the line direction



Unity Ray-Plane Intersection

- Extract Quad vertices using Quad.GetComponent<MeshFilter>().mesh.vertices but they are in LocalSpace!
- Use Quad.transform.TransformPoint(vertices[0]) to transform from LocalSpace to WorldSpace, using Transform T,R,S Matrices Quad.transform.InverseTransformPoint(vertices[0]) to transform from WorldSpace to LocalSpace
  - NB: Quad LocalSpace coords are in range [-0.5, 0.5]
- Create a Plane primitive using new Plane(v0, v1, v2)
- Ray ray = Camera.ScreenPointToRay(Input.mousePosition) Returns a ray going from camera through a screen point
- plane.Raycast(ray, out t) Intersects a ray with the plane. t is the distance along the ray. Returns true if there is intersection
- Ray.GetPoint(t) to know 3D cords of the point P at distance t on Ray
- Start from RayPlaneIntersectionStart.cs and use Transform.InverseTransformPoint() to check if the hitPoint is inside the Quad

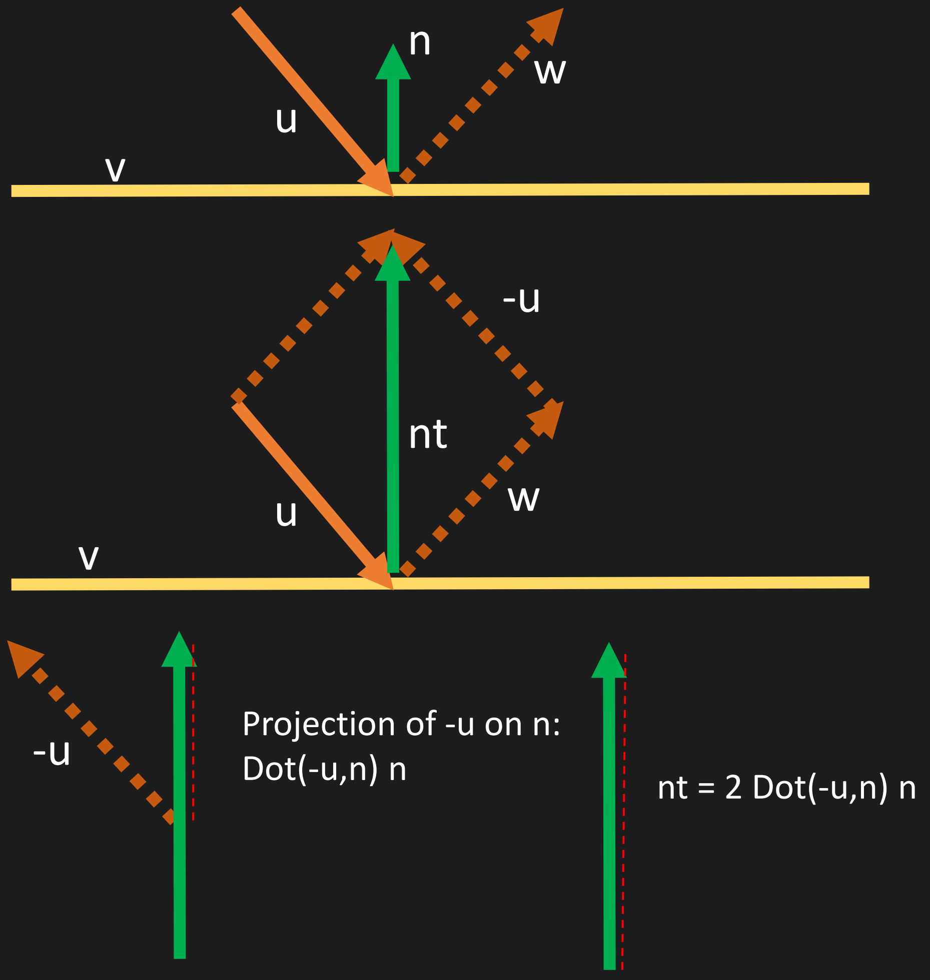


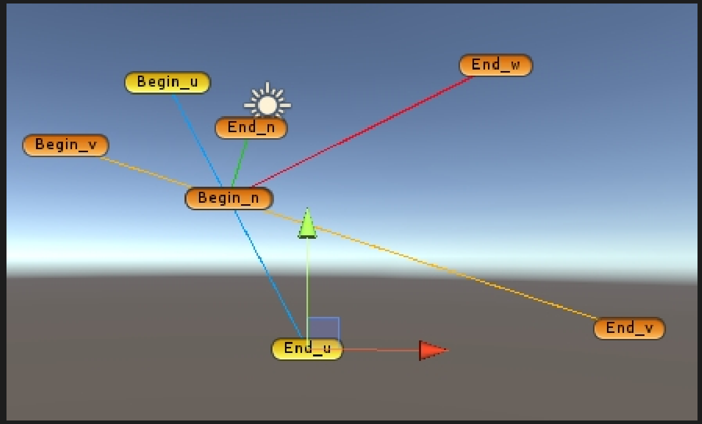
[LinesPlanesOps_03, RayPlaneIntersection.cs]

Reflection 2D

- If u is the incoming vector bouncing on v, n is the v normal, we can say that
  nt = w - u [1]
- Notice that the half of nt is the projection of −u on n. This projection is
  Vector3.Project(-U, n.normalized)
- Then
  nt = 2 Vector3.Project(-U, n.normalized)
- Hence, from [1]
  - w − u = 2 Vector3.Project(-U, n.normalized)
  - w = 2 Vector3.Project(-U, n.normalized) + u

- Finish the implementation ofVectorUtils.GetReflection() in VectorUtils.cs
- NB: Begin_u must be ABOVE the line v, and End_u must be UNDER the line v

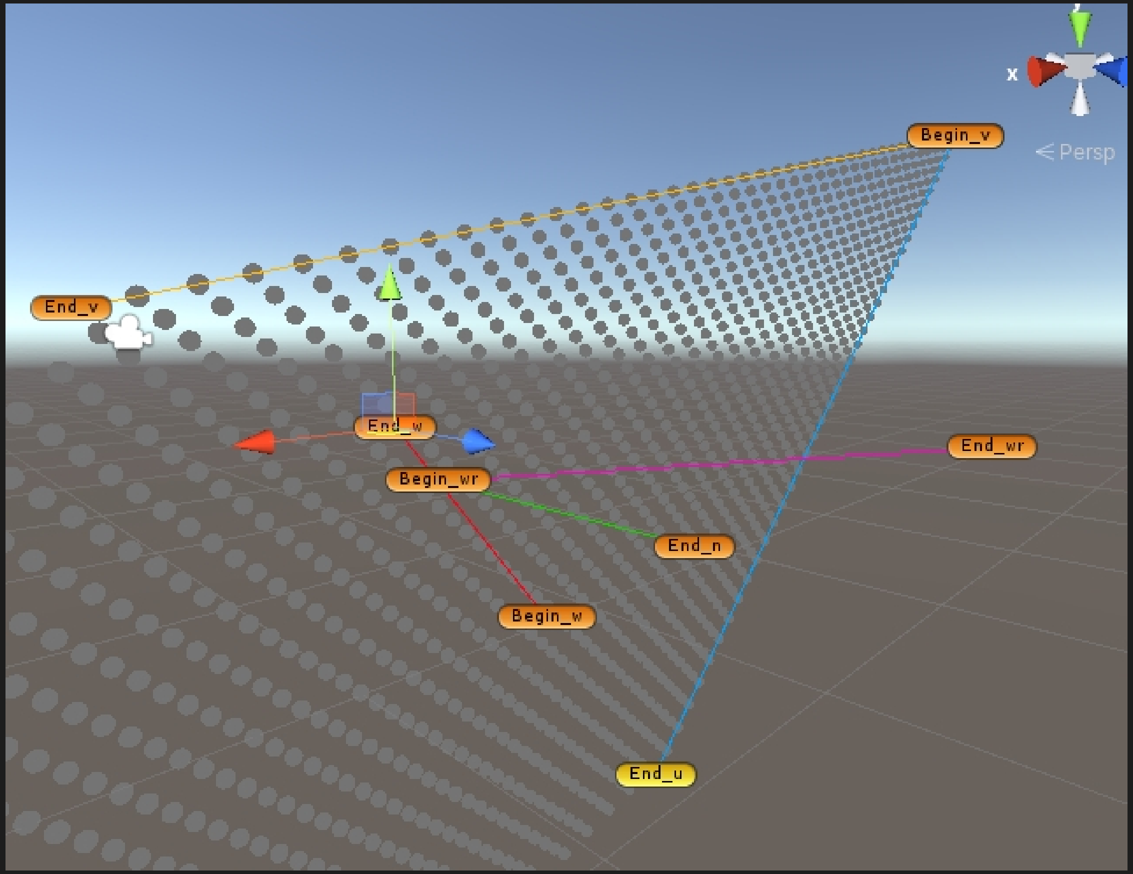




[LinesPlanesOps_03]

Reflection 3D

- The logic is the same as 2D
- Use
  - u and v to draw plane points
  - w is the incoming vector
  - wr is the reflected vector
- planeNormal = Cross(u,v)



[LinesPlanesOps_03]