Geometry and Geometric Programming III

CMSC425.01 fall 2019

Daniel Brown: guest lecture, later

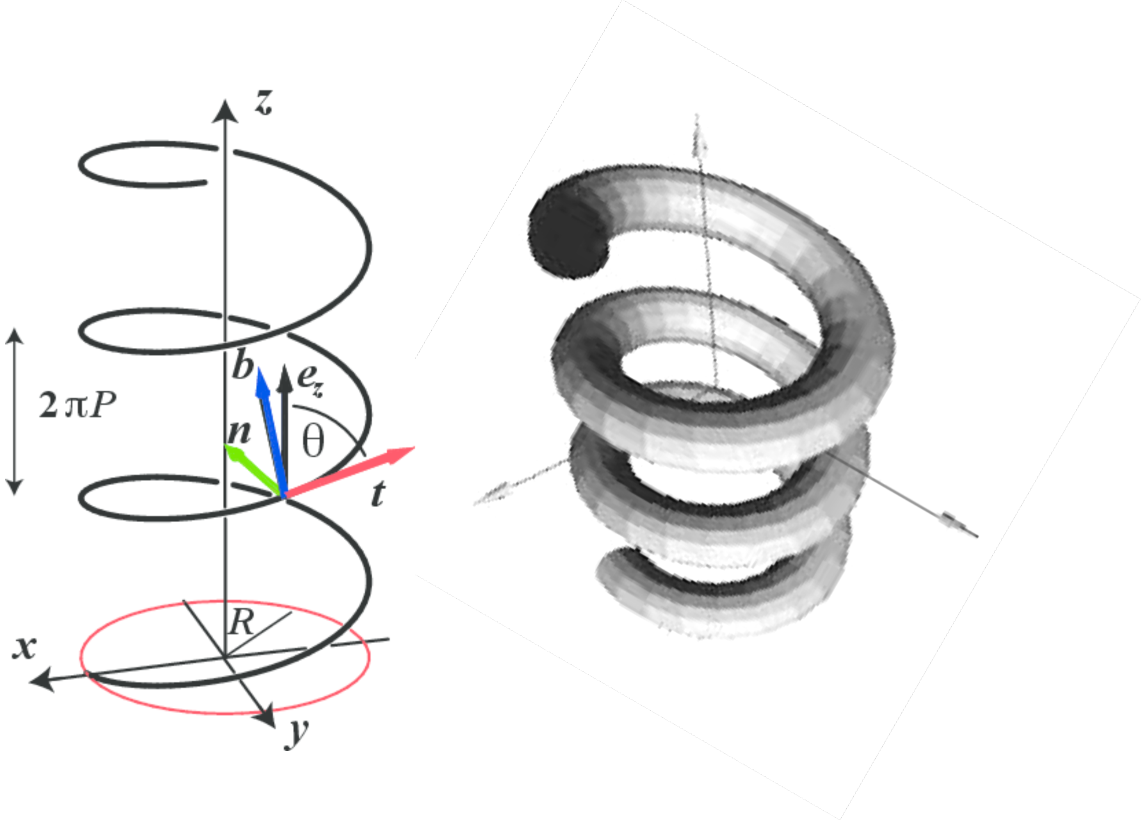
Administrivia

- Hw 1 out
- Practice Hw 1 out with solutions available
- Project 1a under grading
- Review Project 1b Thursday

Examples

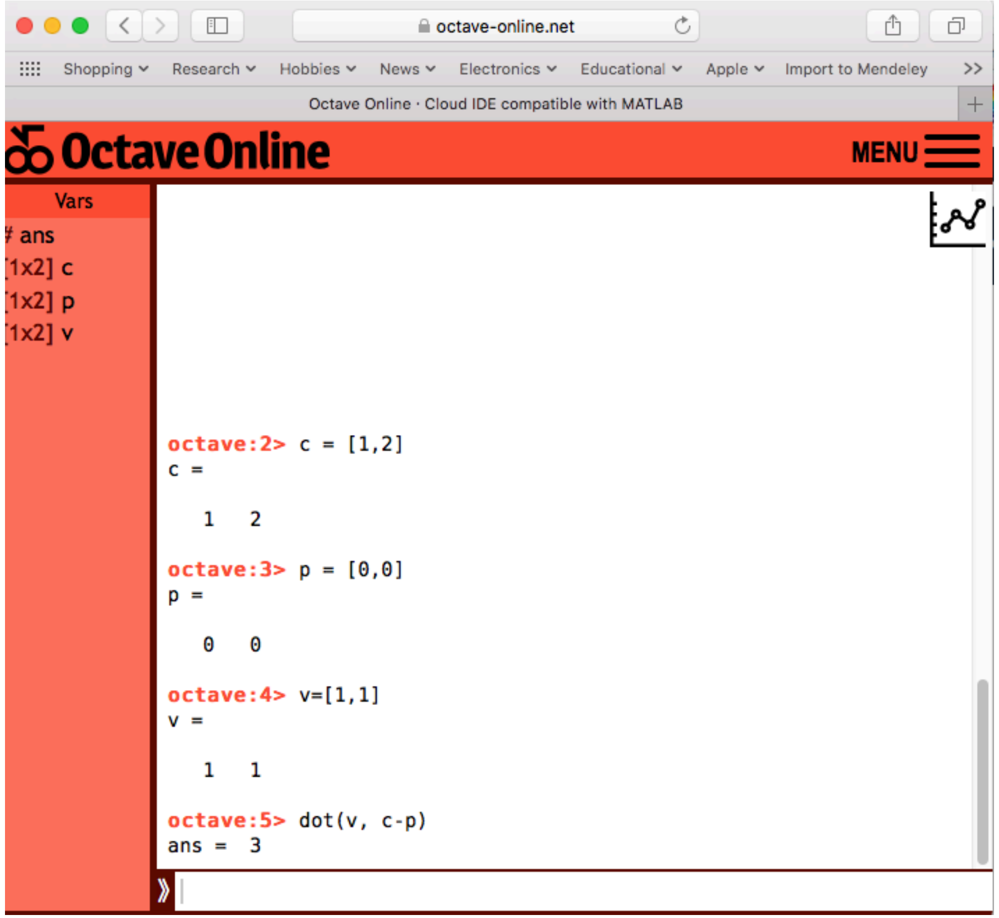
- Rotate moon around Earth around sun (multiple motions)

- Orient cylinder sections of 3D helix



Octave Online – working through examples

- Good for doing examples, verifying equations
- Vectors, Matrices, operations
- Open source version of Matlab
- Can also use app
- Or link Octave fcns externally to C or other languages

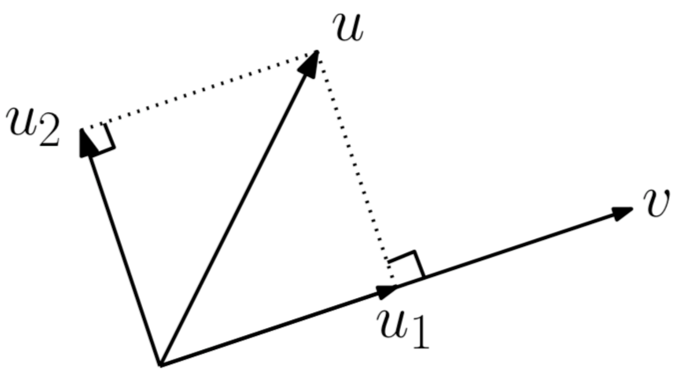


Back to orthogonal projection

Orthogonal projection: Given a vector $\vec{u}$ and a nonzero vector $\vec{v}$, it is often convenient to decompose $\vec{u}$ into the sum of two vectors $\vec{u} = \vec{u}_1 + \vec{u}_2$, such that $\vec{u}_1$ is parallel to $\vec{v}$ and $\vec{u}_2$ is orthogonal to $\vec{v}$.

$$\vec{u}_1 \leftarrow \frac{(\vec{u} \cdot \vec{v})}{(\vec{v} \cdot \vec{v})}\vec{v}, \qquad\qquad \vec{u}_2 \leftarrow \vec{u} - \vec{u}_1.$$

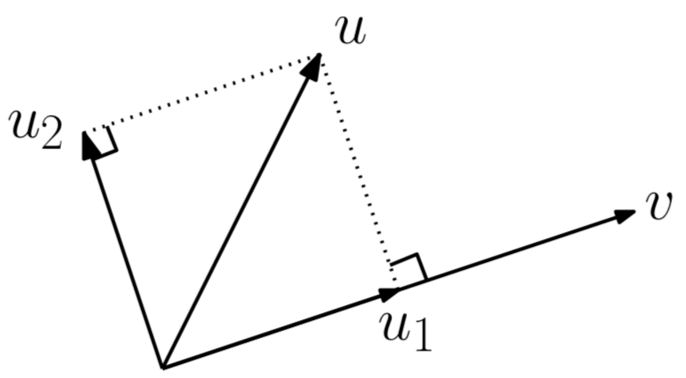
2D frame of reference



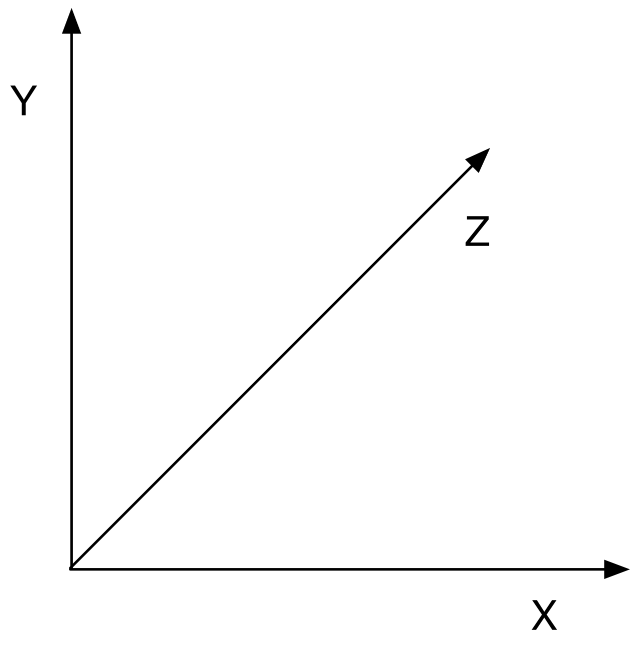
Big idea – frame of reference

Global or local coordinate system in which to define pts and vectors

- 2D



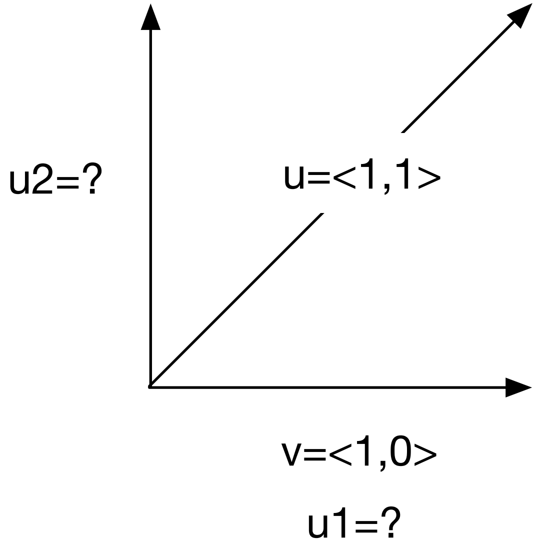
- 3D

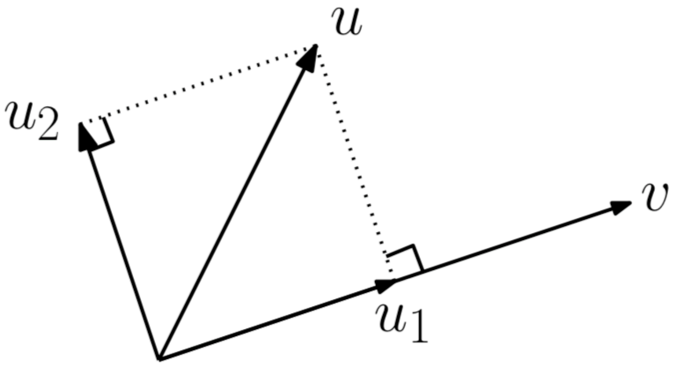


Understand: work through examples

- Start with obvious example
- u = <1,1>
- v = <1,0>

$$\vec{u}_1 \leftarrow \frac{(\vec{u}\cdot\vec{v})}{(\vec{u}\cdot\vec{v})}\vec{v}, \qquad \vec{u}_2 \leftarrow \vec{u} - \vec{u}_1$$



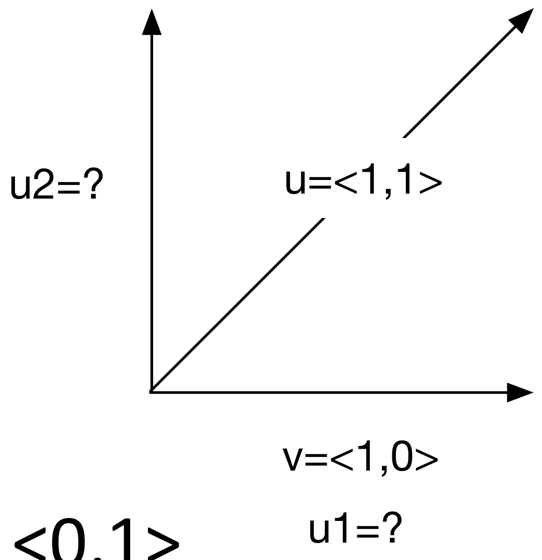


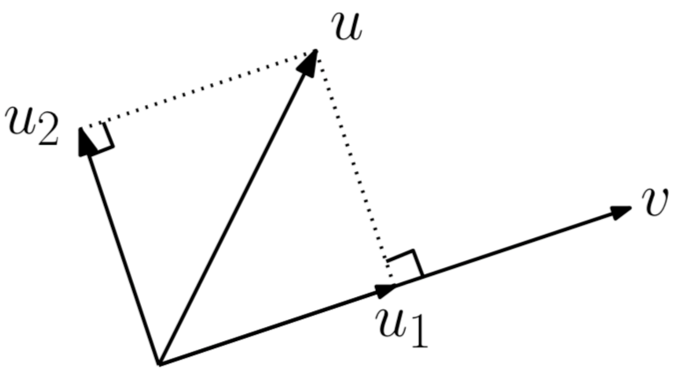
Understand: work through examples

- Start with obvious example
- u = <1,1>
- v = <1,0>

$$\vec{u}_1 \leftarrow \frac{(\vec{u}\cdot\vec{v})}{(\vec{u}\cdot\vec{v})}\vec{v}, \qquad \vec{u}_2 \leftarrow \vec{u} - \vec{u}_1$$

- u1 = 1/1*<1,0>
- u2 = <1,1>-<1,0>
  = <0,1>



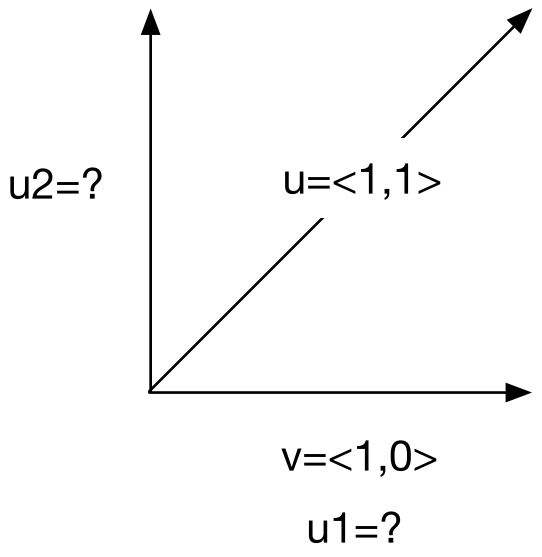


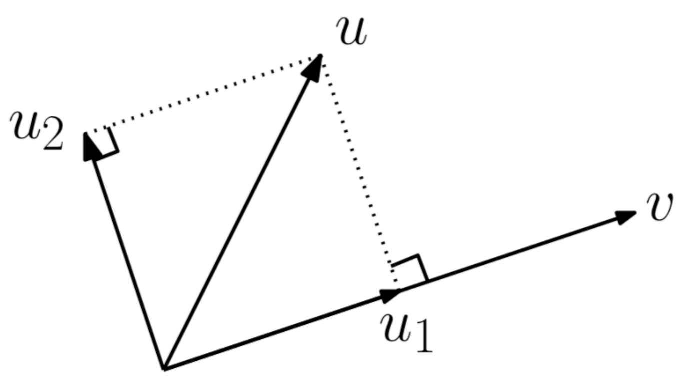
u projects onto <1,0>, <0,1>

Understand: work through examples

- Work slowly to complex
- u = <0,1>
- v = <1,1>

$$\vec{u}_1 \leftarrow \frac{(\vec{u}\cdot\vec{v})}{(\vec{u}\cdot\vec{v})}\vec{v}, \qquad \vec{u}_2 \leftarrow \vec{u} - \vec{u}_1$$



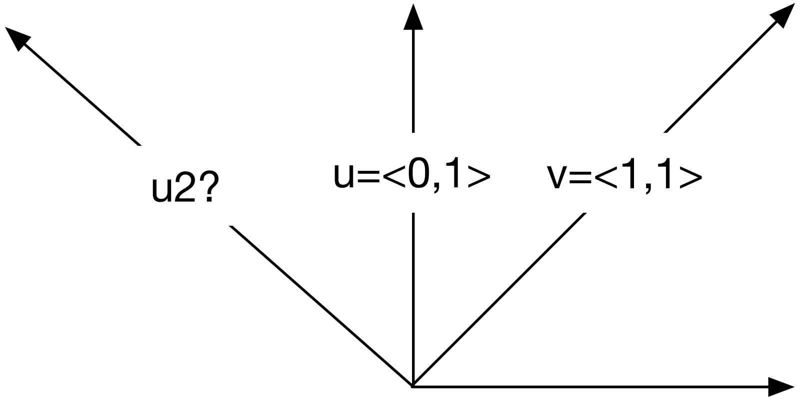


Understand: work through examples

- Work slowly to complex
- u = <0,1>
- v = <1,1>

- u1 = (u•v)/(v•v) v
  = ½ <1,1> = < ½, ½ >
- u2 = u – u1 = <0,1> - < ½, ½ >
  = < - ½ , ½ >

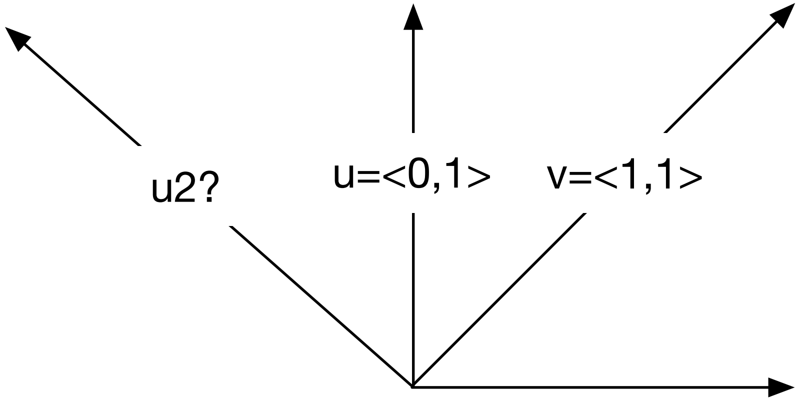
$$\vec{u}_1 \leftarrow \frac{(\vec{u}\cdot\vec{v})}{(\vec{u}\cdot\vec{v})}\vec{v}, \qquad \vec{u}_2 \leftarrow \vec{u} - \vec{u}_1$$



Observation: are u1, u2 normal vectors?

$$\vec{u}_1 \leftarrow \frac{(\vec{u}\cdot\vec{v})}{(\vec{u}\cdot\vec{v})}\vec{v}, \qquad \vec{u}_2 \leftarrow \vec{u} - \vec{u}_1$$

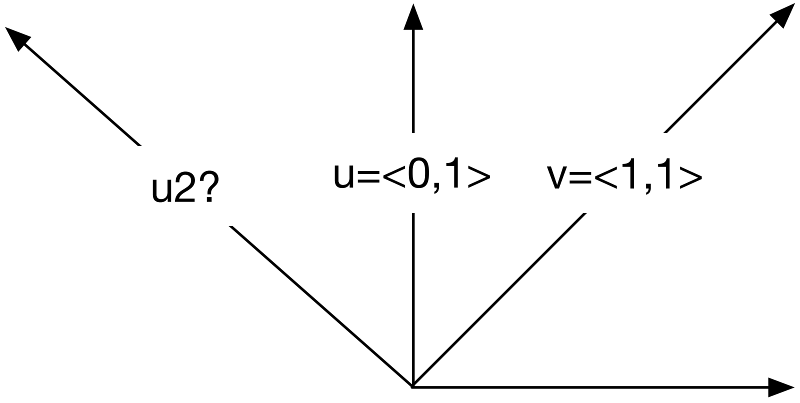
- u1 = < ½, ½ >
- u2 = < - ½ , ½ >



Observation: are u1, u2 normal vectors?

$$\vec{u}_1 \leftarrow \frac{(\vec{u}\cdot\vec{v})}{(\vec{u}\cdot\vec{v})}\vec{v}, \qquad \vec{u}_2 \leftarrow \vec{u} - \vec{u}_1$$

- u1 = < ½, ½ >
- u2 = < - ½ , ½ >



- |u1| = sqrt(¼ + ¼) = sqrt(½)

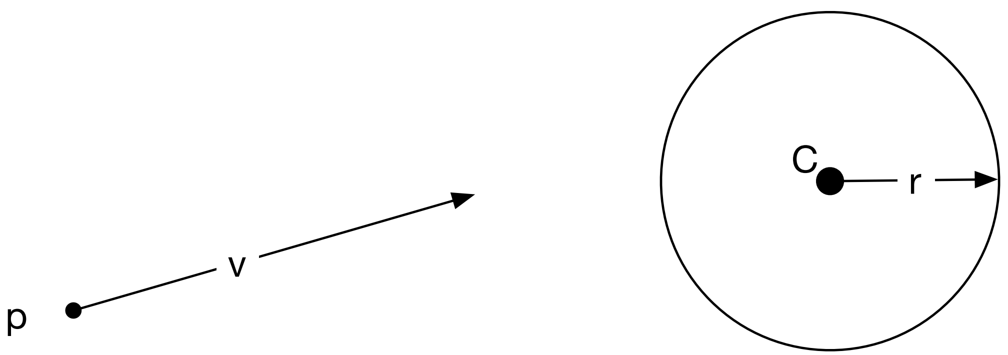
u1 = < ½, ½ > /sqrt(1/2)

= <sqrt(2)/2, sqrt(2)/2>

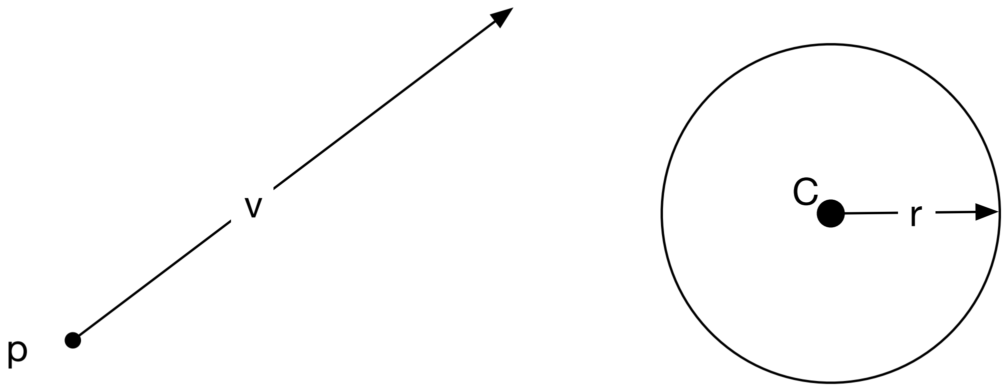
NO

Problem: Ray – circle intersection

- Does the ray defined by **p** and **v** intersect the circle defined by **c** and **r**?



True



False

Ray – circle intersection

- Does the ray defined by **p** and **v** intersect the circle defined by **c** and **r**?

- Solutions?

A) Do equations p(t) = p + tv and $(x\text{-}xc)^2 + (y\text{-}yc)^2$ = r^2 have solution?

B) Is sine of angle * length to circle less than radius?

C) Length of projection of normal less than radius?

Given vectors u, v, and w, all of type Vector3, the following operators are supported:

```
u = v + w; // vector addition
u = v - w; // vector subtraction
if (u == v || u != w) { ... } // vector comparison
u = v * 2.0f; // scalar multiplication
v = w / 2.0f; // scalar division
```

You can access the components of a Vector3 using as either using axis names, such as, u.x, u.y, and u.z, or through indexing, such as u[0], u[1], and u[2].

The Vector3 class also has the following members and static functions.

```
float x = v.magnitude; // length of v
Vector3 u = v.normalize; // unit vector in v's direction
float a = Vector3.Angle (u, v); // angle (degrees) between u and v
float b = Vector3.Dot (u, v); // dot product between u and v
Vector3 u1 = Vector3.Project (u, v); // orthog proj of u onto v
Vector3 u2 = Vector3.ProjectOnPlane (u, v); // orthogonal complement
```

Some of the Vector3 functions apply when the objects are interpreted as points. Let p and q be points declared to be of type Vector3. The function Vector3.Lerp is short for *linear interpolation*. It is essentially a two-point special case of a convex combination. (The combination parameter is assumed to lie between 0 and 1.)

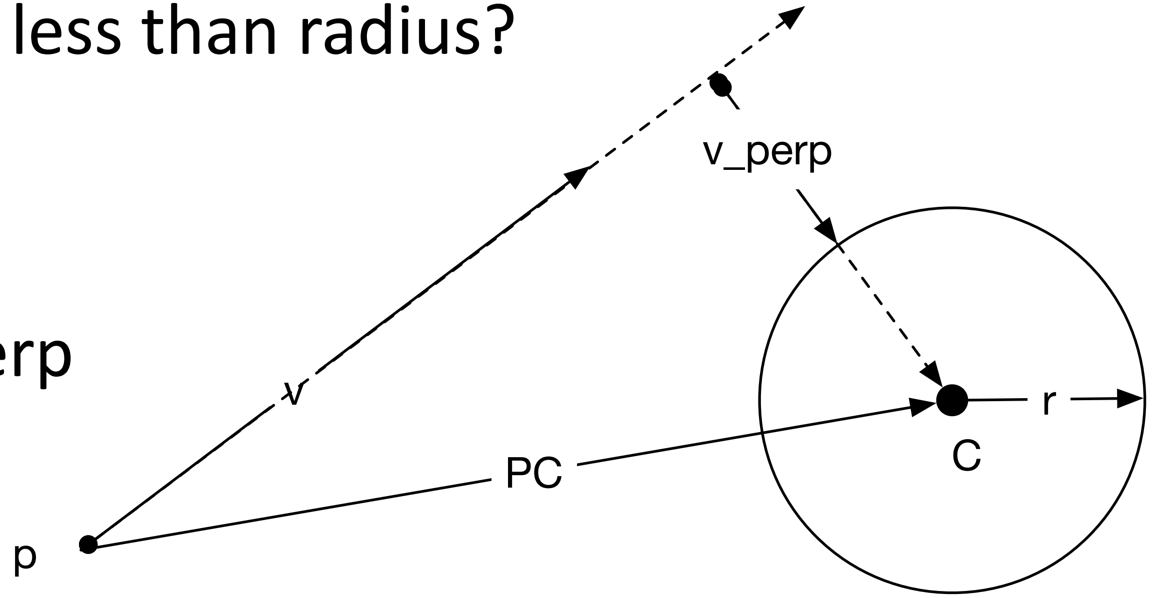
```
float b = Vector3.Distance (p, q); // distance between p and q
Vector3 midpoint = Vector3.Lerp(p, q, 0.5f); // convex combination
```

Instant Hw1 – Ray – circle intersection

- Does the ray defined by **p** and **v** intersect the circle defined by **c** and **r**?

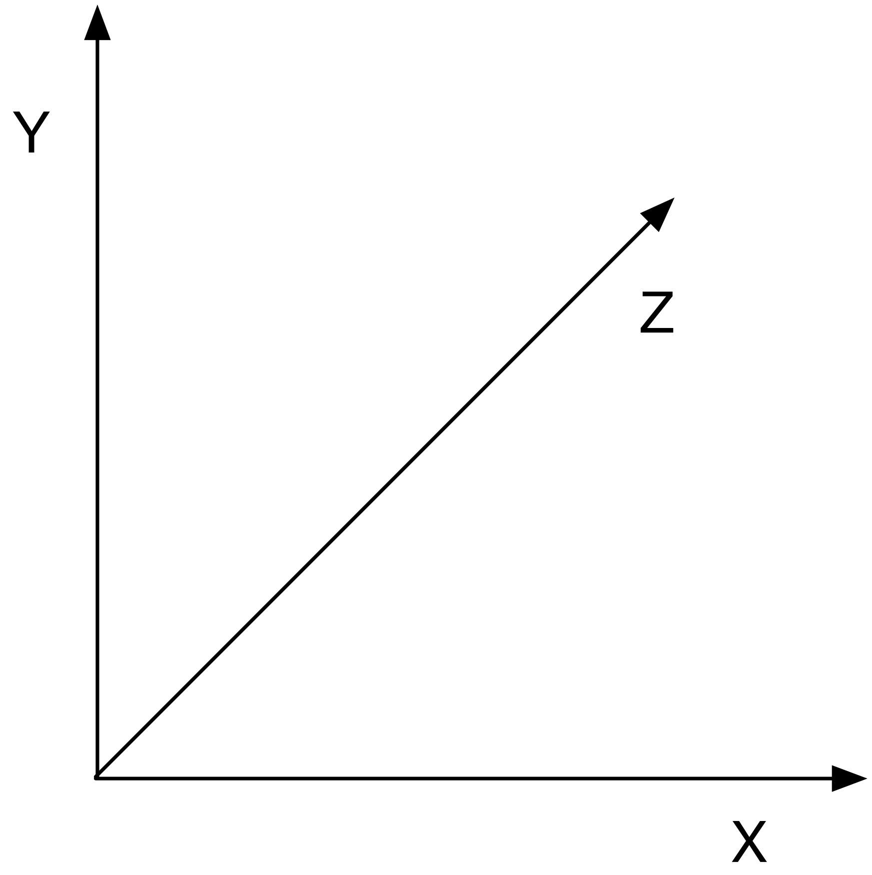
C) Length of projection of normal less than radius?

1) Compute v_perp
2) Normalize v_perp
3) Length of projection: PC•v_perp
4) Is PC•v_perp < r ?



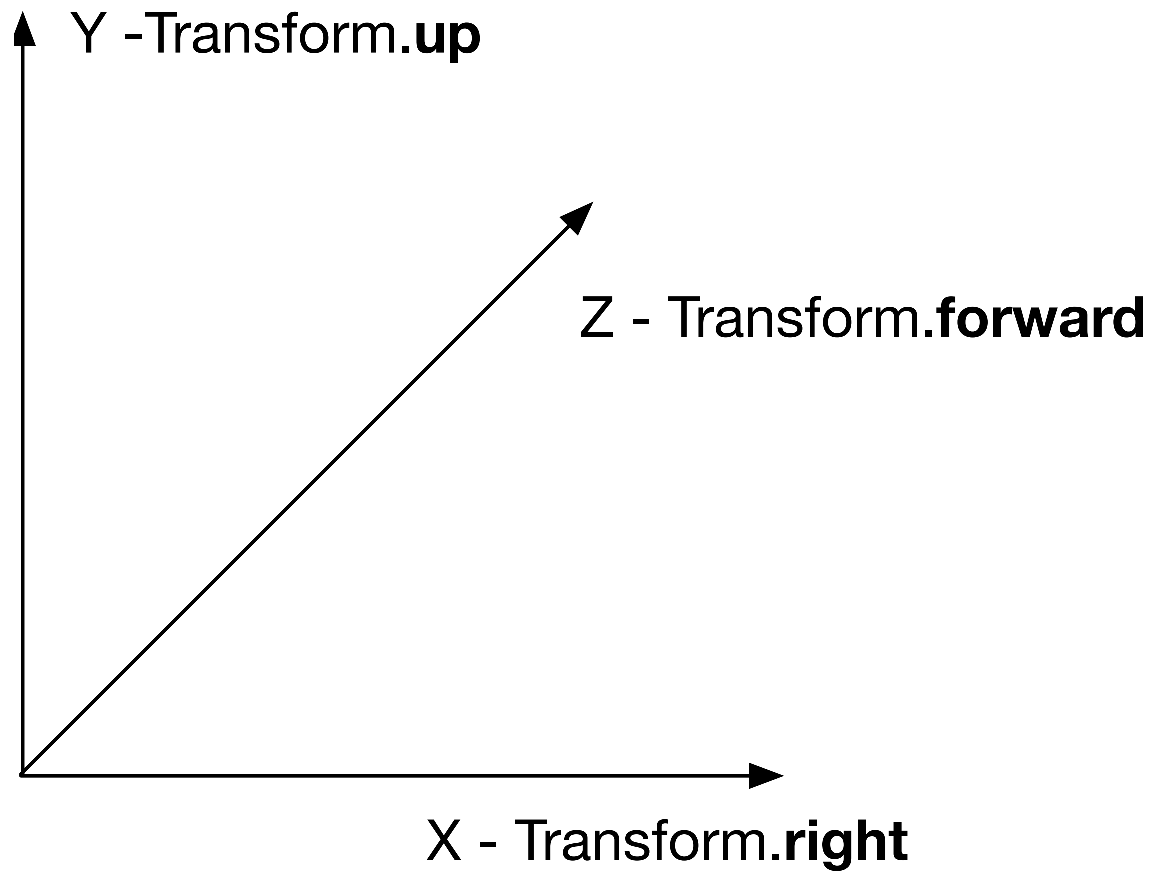
Moving to 3D – frame of reference

- Left handed system XYZ



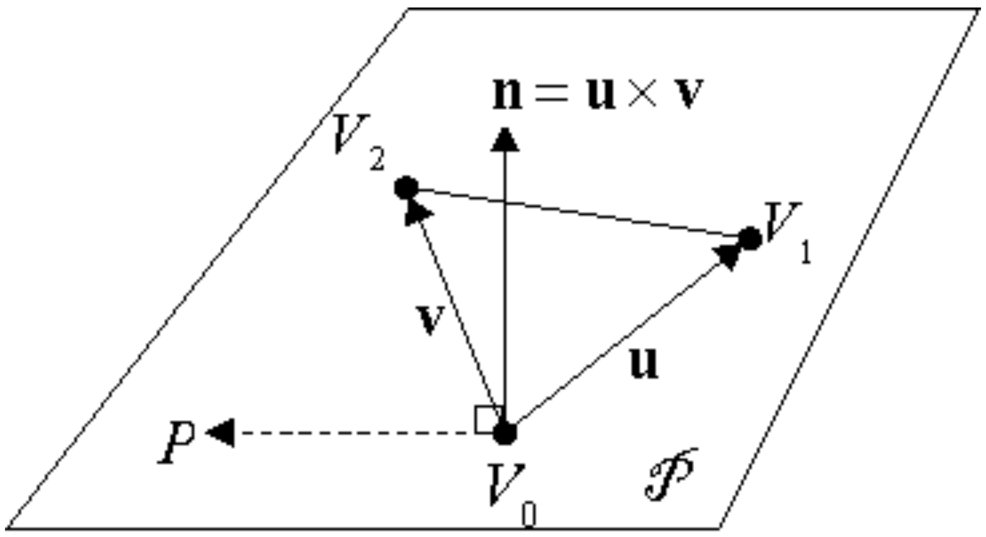
Moving to 3D – frame of reference

- In Unity – (right, up, forward)
- Forward – moving forward
- Up – a sense of gravity
- Right – turn direction



Applying cross product

- Computing normal vector
  - To triangle
  - To plane
- Computing local 3D orthonormal basis
- Point-normal form of plane
  - n•(p-v0) = 0 means p is on the plane

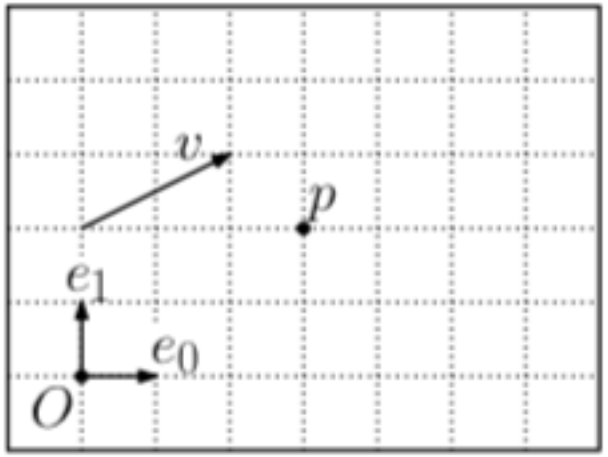


Homogeneous coordinates: points

- Step 2: Add origin to sum

$$p = \alpha_0 \vec{u}_0 + \alpha_1 \vec{u}_1 + O$$

- Now
  - point = <x,y,1>
  - vector = <x,y,0>



$$p = 3 \cdot \vec{e}_0 + 2 \cdot \vec{e}_1 + 1 \cdot O$$

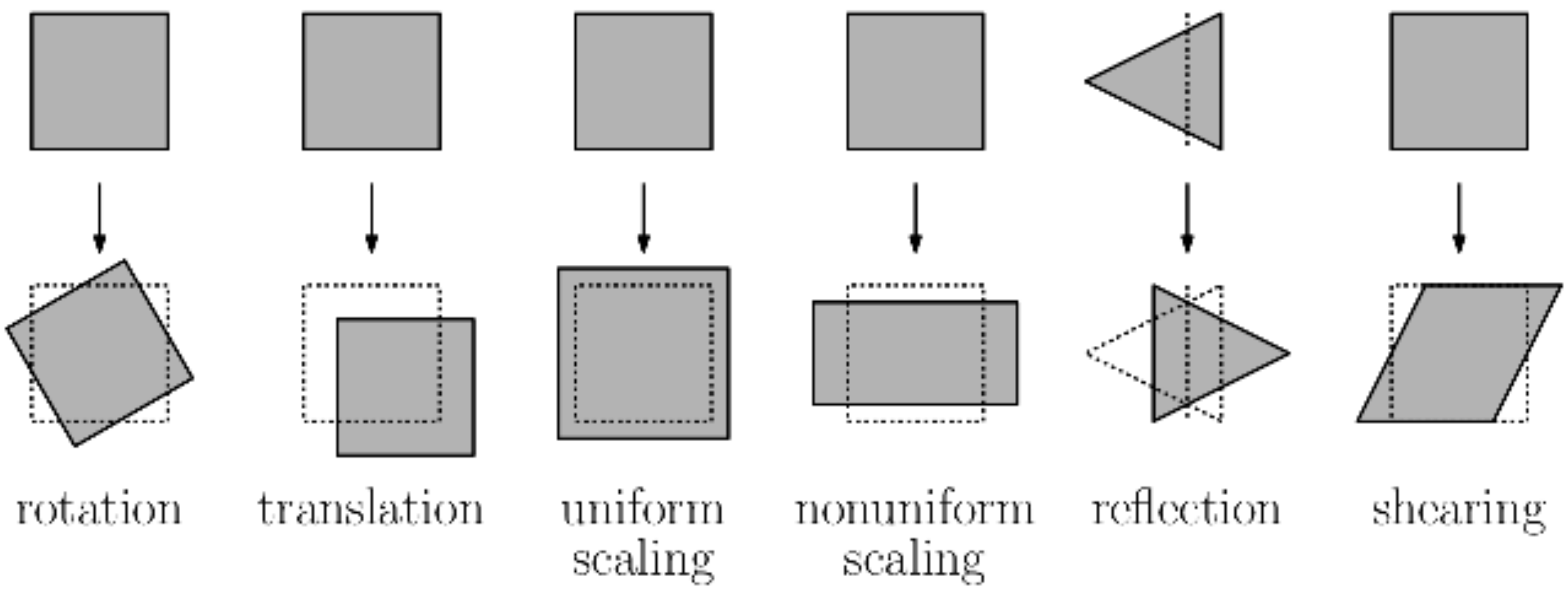
$$\Rightarrow p_{[F]} = (3, 2, 1)$$

$$v = 2 \cdot \vec{e}_0 + 1 \cdot \vec{e}_1 + 0 \cdot O$$

$$\Rightarrow v_{[F]} = (2, 1, 0)$$

Affine transformations

- Key: translation, rotation, scale



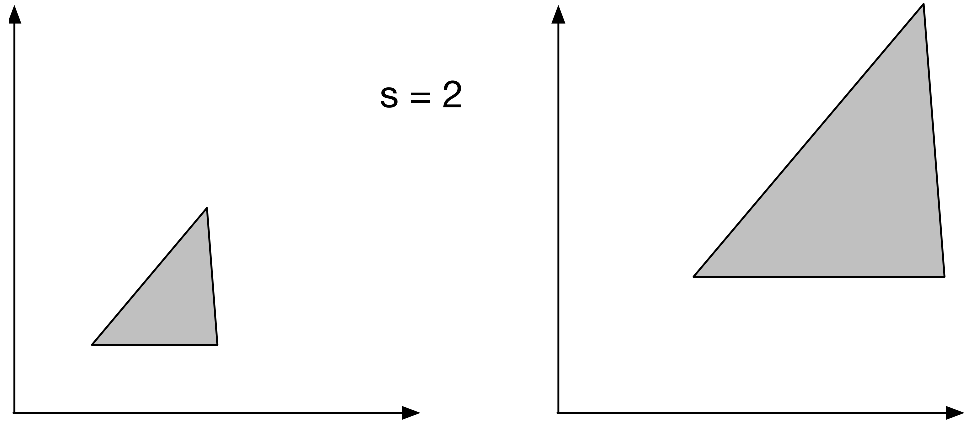
Scaling

- Coordinate free - uniform scale s

$$v = su$$

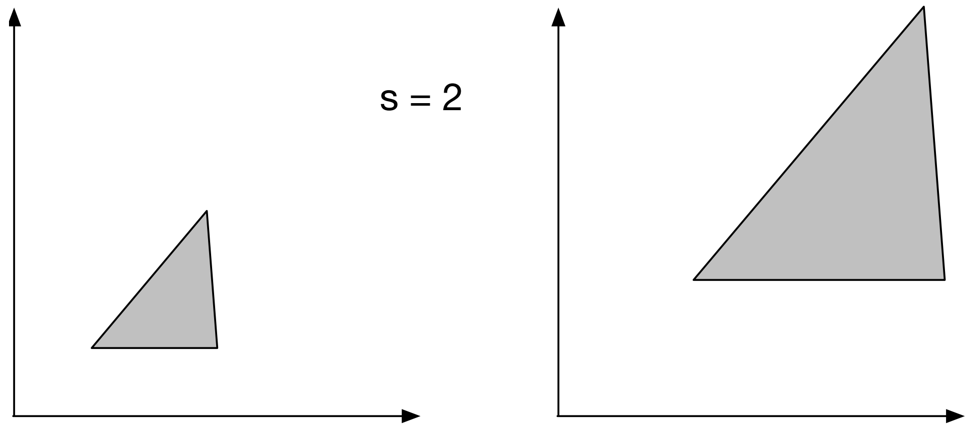
- Coordinate based

$$< v_x, v_y, v_z > = < su_x, su_y, su_z >$$



- Scaling sizes and moves

Scaling



- Coordinate free – uniform scale s

$$v = su$$

- Coordinate based

$$< v_x, v_y, v_z > = < su_x, su_y, su_z >$$

- Scaling sizes and moves

- Homogeneous coordinates – vector

$$< v_x, v_y, v_z\,, 0 > = < su_x, su_y, su_z\,, 0 >$$

- Homogeneous coordinates – points (simple scalar * doesn't work)

$$(v_x, v_y, v_z\,, 1) \ = (su_x, su_y, su_z\,, s)$$

Scaling

- Matrix form 2D

$$v^t = M_s u^t$$

$$M_s = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Vector
  $< v_x, v_y, 0 > =< su_x, su_y, 1 * 0 >$
- Point
  $(q_x, q_y, 1) \ =< sp_x, sp_y, 1 * 1 >$

- Matrix multiplication on the right with transpose of vector v^t
- Works for vectors and points
- Maintains homogeneous coordinate w

Translation

- Matrix form 2D
-

$$v = M_t u$$

$$M_t = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Translate point

$$(q_x, q_y, 1) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$

$$(q_x, q_y, 1) = \left(p_x + t_x, p_y + t_y, 1\right)$$

First version: coordinate based equations

- Translation by v: q = p + T(v) Add vector v
- Scale by a: q = a p Multiply by scalar a
- Rotate by t: (qx,qy) = <px*cos(t) – py*sin(t), px*sin(t) + py*cos(t)>

- Repeated scalings and translations:

- q = a (p + T(V)) = a ((a p +T(V)) + T(v)) = and so on …

- Complex

Second version: Homogeneous coordinates

- Unify all transformations in matrix notation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Identity Matrix

$$\begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glTranslatef(tx,ty,tz)

$$\begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glScalef(sx,sy,sz)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(d) & -\sin(d) & 0 \\ 0 & \sin(d) & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,1,0,0)

$$\begin{pmatrix} \cos(d) & 0 & \sin(d) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(d) & 0 & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,0,1,0)

$$\begin{pmatrix} \cos(d) & -\sin(d) & 0 & 0 \\ \sin(d) & \cos(d) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,0,0,1)

Defining rotations

- Euler angles
- Angle Axis
- Quaternions

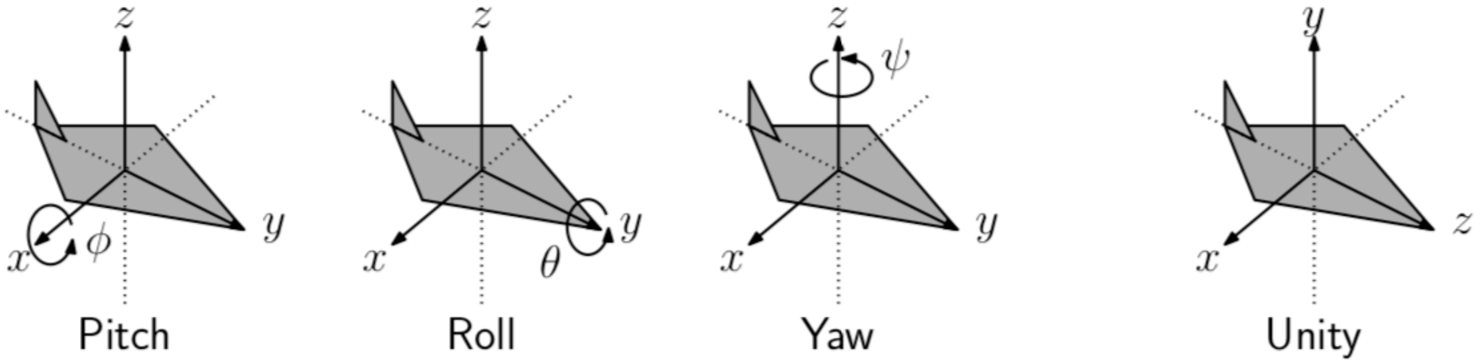
Roll – around forward direction

Pitch – around right direction

Yaw – around up direction

- In Unity
  transform.Rotate(x, y, z))

- Euler angles in order x,y,z



Defining rotations

- Euler angles
- Angle Axis
- Quaternions

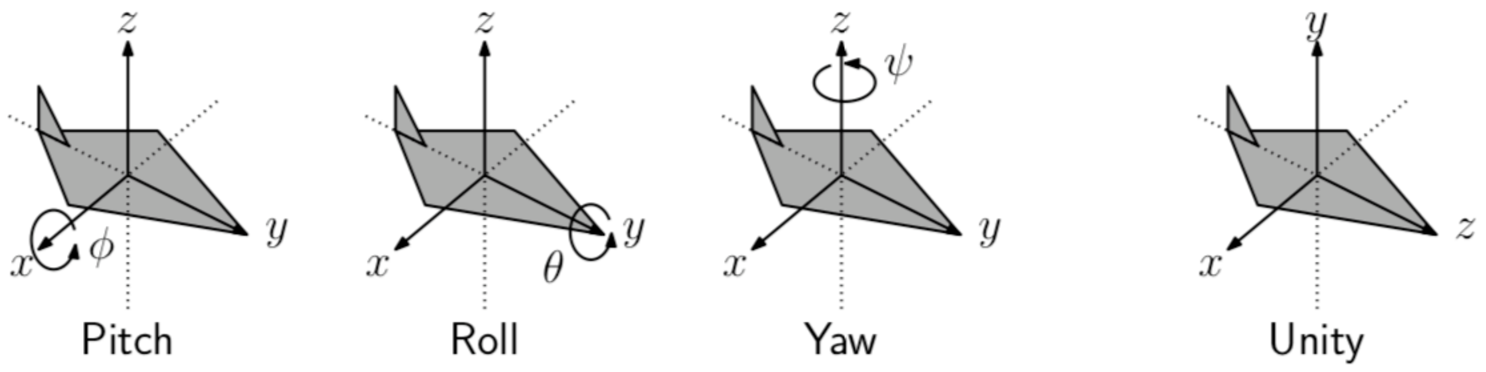
Roll – around forward direction

Pitch – around right direction

Yaw – around up direction

- In Unity
  transform.Rotate(x, y, z))

- Euler angles in order x,y,z



Defining rotations

- Angle Axis

Quaternion.AngleAxis

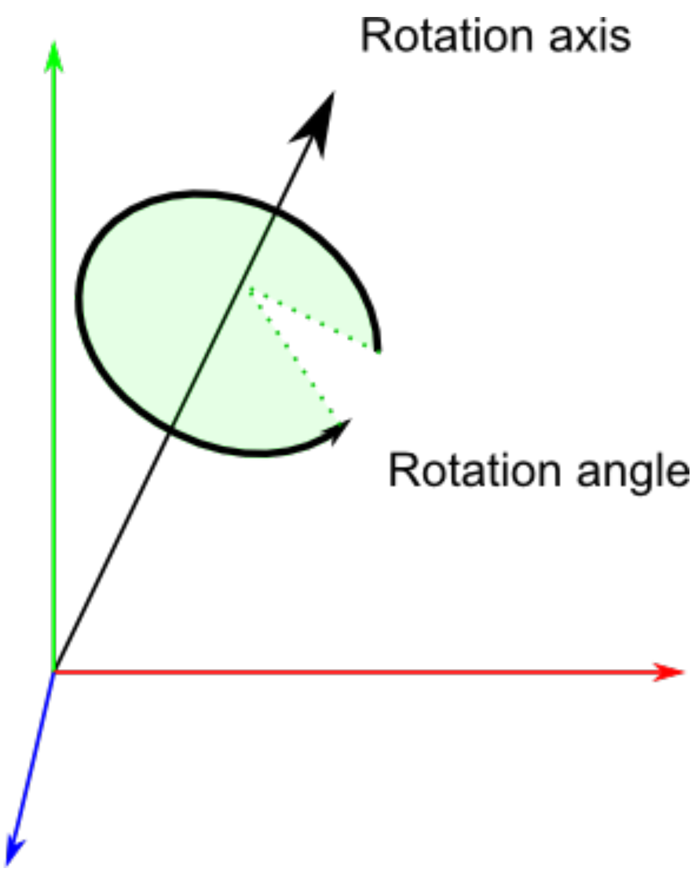
public static Quaternion **AngleAxis**(float **angle**, Vector3 **axis**);

Description

Creates a rotation which rotates `angle` degrees around `axis`.

```
using UnityEngine;

public class Example : MonoBehaviour
{
    void Start()
    {
        // Sets the transforms rotation to rotate 30 degrees around the y-axis
        transform.rotation = Quaternion.AngleAxis(30, Vector3.up);
    }
}
```



Interpolating transformations

- Translation. Easy – move v*dt each frame
- Scale. Easy – scale by s*dt each frame

- Interpolating rotations? Harder

  - Interpolate Euler angles? Doesn’t work well
  - Interpolate Axis Angle? Better
  - Interpolate Quaternions? Best Why Unity uses them.

Quaternion.Slerp

public static Quaternion **Slerp**(Quaternion **a**, Quaternion **b**, float **t**);

Description

Spherically interpolates between `a` and `b` by t. The parameter `t` is clamped to the range [0, 1].

```
// Interpolates rotation between the rotations "from" and "to"
// (Choose from and to not to be the same as
// the object you attach this script to)

using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    public Transform from;
    public Transform to;

    private float timeCount = 0.0f;

    void Update()
    {
        transform.rotation = Quaternion.Slerp(from.rotation, to.rotation, timeCount);
        timeCount = timeCount + Time.deltaTime;
    }
}
```

Defining rotations

- Angle Axis

Quaternion.AngleAxis

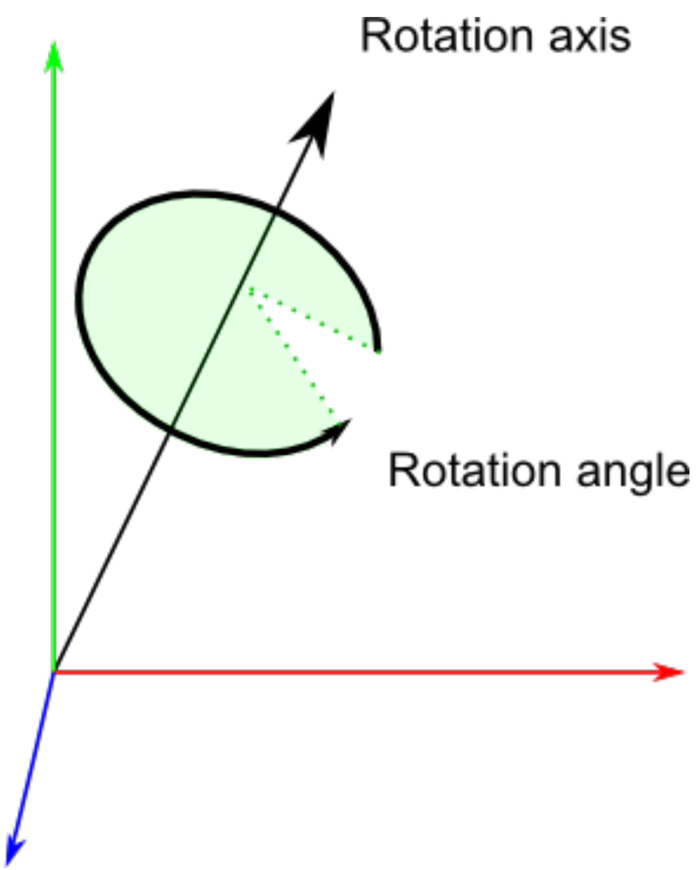
public static Quaternion **AngleAxis**(float **angle**, Vector3 **axis**);

Description

Creates a rotation which rotates `angle` degrees around `axis`.

```
using UnityEngine;

public class Example : MonoBehaviour
{
    void Start()
    {
        // Sets the transforms rotation to rotate 30 degrees around the y-axis
        transform.rotation = Quaternion.AngleAxis(30, Vector3.up);
    }
}
```



Interpolating transformations

- Translation. Easy – move v*dt each frame
- Scale. Easy – scale by s*dt each frame

- Interpolating rotations? Harder

  - Interpolate Euler angles? Doesn’t work well
  - Interpolate Axis Angle? Better
  - Interpolate Quaternions? Best Why Unity uses them.

Quaternion.Slerp

public static Quaternion **Slerp**(Quaternion **a**, Quaternion **b**, float **t**);

Description

Spherically interpolates between `a` and `b` by t. The parameter `t` is clamped to the range [0, 1].

```
// Interpolates rotation between the rotations "from" and "to"
// (Choose from and to not to be the same as
// the object you attach this script to)

using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    public Transform from;
    public Transform to;

    private float timeCount = 0.0f;

    void Update()
    {
        transform.rotation = Quaternion.Slerp(from.rotation, to.rotation, timeCount);
        timeCount = timeCount + Time.deltaTime;
    }
}
```

Readings

- David Mount's lectures on Geometry and Geometric Programming