GEOMETRY

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

Basic Elements

- **Geometry** is the study of the relationships among objects in an n-dimensional space
  - In computer graphics, we are interested in objects that exist in 3 dimensions
- Want a minimum set of primitives from which we can build more sophisticated objects
- We will need three basic elements
  1. **Scalars**
  2. **Vectors**
  3. **Points**

Coordinate-Free Geometry

- When we learned simple geometry, most of us started with a Cartesian approach
  - Points at locations in space **p**=(x,y,z)
  - We derived results by algebraic manipulations involving these coordinates
- This approach is nonphysical
  - Physically, points exist regardless of the location of an arbitrary coordinate system
  - Most geometric results are independent of the coordinate system
    - Example: Euclidean geometry: two triangles are identical if two corresponding sides and the angle between them are identical

Scalars

- Scalars can be defined as members of sets
  - which can be combined by two operations: addition and multiplication
  - obeying some fundamental axioms: associativity, commutativity, inverses
- Examples include the **real** and **complex** number systems under the ordinary rules with which we are familiar.
- Scalars alone have no geometric properties

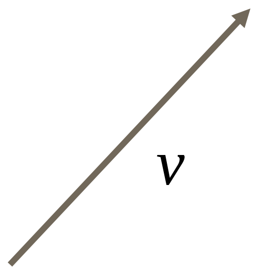
Vectors

- Physical definition:

a vector is a quantity with two attributes

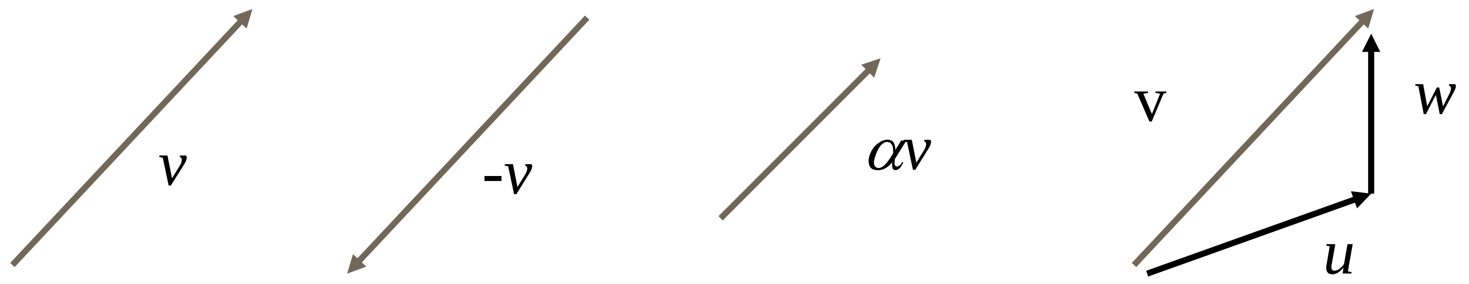
  - **Direction**
  - **Magnitude**

- Examples include
  - Force
  - Velocity
  - Directed line segments
    - Most important example for graphics
    - Can map to other types



Vector Operations

- Every vector has an inverse
  - Same magnitude but points in opposite direction
- Every vector can be multiplied by a scalar
- There is a zero vector
  - Zero magnitude, undefined orientation
- The sum of any two vectors is a vector
  - Use head-to-tail axiom



Linear Vector Spaces

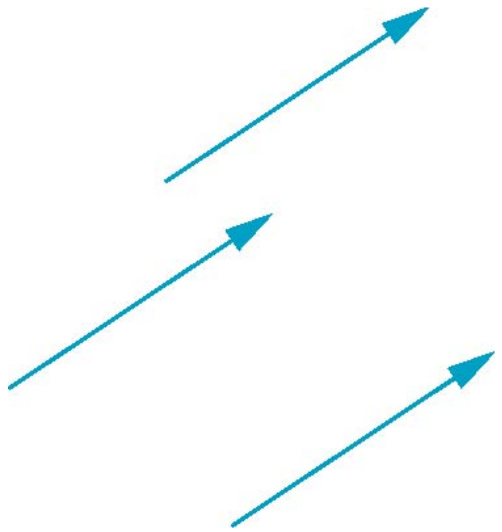
- Mathematical system for manipulating vectors
- Operations
  - scalar-vector multiplication $u = \alpha v$
  - vector-vector addition: $w = u + v$
- Expressions such as

  $v=u+2w-3r$

  make sense in a vector space

Vectors Lack Position

- These vectors are identical
  - Same length and magnitude

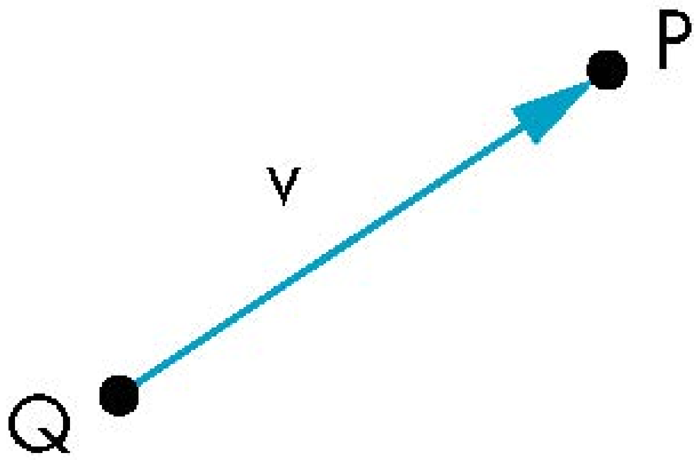


>> Vector spaces insufficient for geometry

>> Need points too

Points

- Location in space
- Operations allowed between points and vectors
  - Point-point subtraction >> yields a vector
    - Equivalent to point-vector addition



$$v = P - Q$$

$$P = v + Q$$

Affine Spaces

- Geometrically, curves and surfaces are usually considered to be sets of points with some special properties, living in a space consisting of “points.”
- Typically, one is also interested in geometric properties invariant under certain transformations, for example, translations, rotations, projections, etc.
- One could model the space of points as a vector space, but this is not very satisfactory for a number of reasons.
  - One reason is that the point corresponding to the zero vector (0), called the origin, plays a special role, when there is really no reason to have a privileged origin.
  - Another reason is that certain notions, such as parallelism, are handled in an awkward manner.
  - But the deeper reason is that vector spaces and affine spaces really have different geometries.
- <u>Affine spaces provide a better framework for doing geometry.</u>

Affine Spaces

- In particular, it is possible to deal with points, curves, surfaces, etc., in an intrinsic manner, that is, independently of any specific choice of a coordinate system.
  - As in physics, this is highly desirable to really understand what is going on.
  - Affine spaces are the right framework for dealing with motions, trajectories, and physical forces, among other things. Thus, affine geometry is crucial to a clean presentation of kinematics, dynamics, and other parts of physics (for example, elasticity).
- Also, given an m×n matrix A and a vector b ∈ R^m, the set U = {x ∈ R^n | Ax = b} of solutions of the system Ax = b is an affine space, but not a vector space (linear space) in general.

Affine Spaces

- no specific point that serves as an origin.
  - >> no vector has a fixed origin and no vector can be uniquely associated to a point.
- instead, there are displacement vectors between two points of the space.
  - Thus it makes sense to subtract two points of the space, giving a vector,
    - but it does not make sense to add two points of the space.
  - Likewise, it makes sense to add a vector to a point, resulting in a new point displaced from the starting point by that vector.

- Of course, coordinate systems have to be chosen to finally carry out computations, but one should learn to resist the temptation to resort to coordinate systems until it is really necessary.
- Should use coordinate systems only when needed.

Affine Spaces

- Points + a vector space
  - Points are typically used to position ourselves in space and vectors are use to move about in space.

- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations

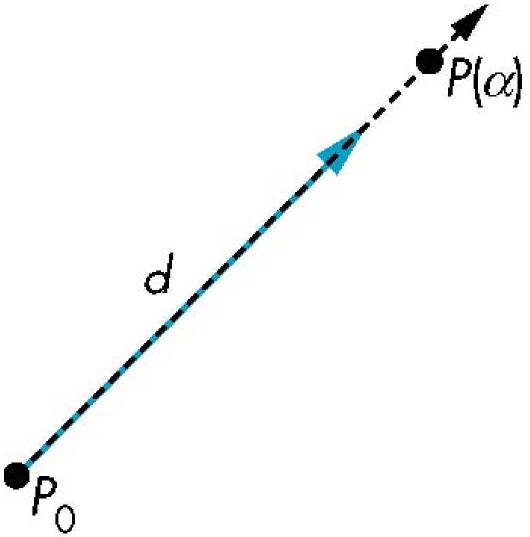
- For any point define
  - $1 \bullet P = P$
  - $0 \bullet P = \mathbf{0}$ (zero vector)

Lines

- Consider all points of the form

  $P(\alpha)=P_0 + \alpha\, \mathbf{d}$

  >> Set of all points that pass through P_0 in the direction of the vector $\mathbf{d}$



Parametric Form

- Two-dimensional forms
  1. Explicit form: $y = mx + h$
  2. Implicit form: $ax + by + c = 0$
  3. **Parametric form:**

     $x(\alpha) = \alpha x_0 + (1-\alpha)x_1$

     $y(\alpha) = \alpha y_0 + (1-\alpha)y_1$

- **parametric form of the line**
  - More robust and general than other forms
  - Extends to curves and surfaces

Rays and Line Segments

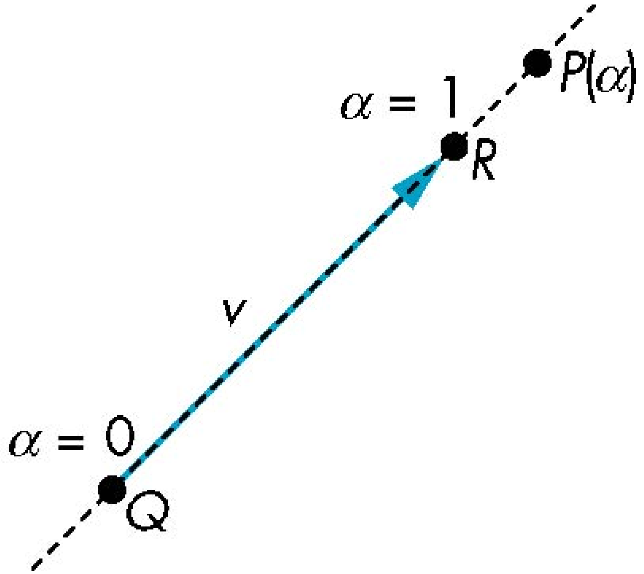
- If $\alpha >= 0$, then $P(\alpha)$ is the ***ray*** leaving P_0 in the direction $\mathbf{d}$

If we use two points to define v, then

$P(\alpha) = Q + \alpha (R-Q) = Q + \alpha v$

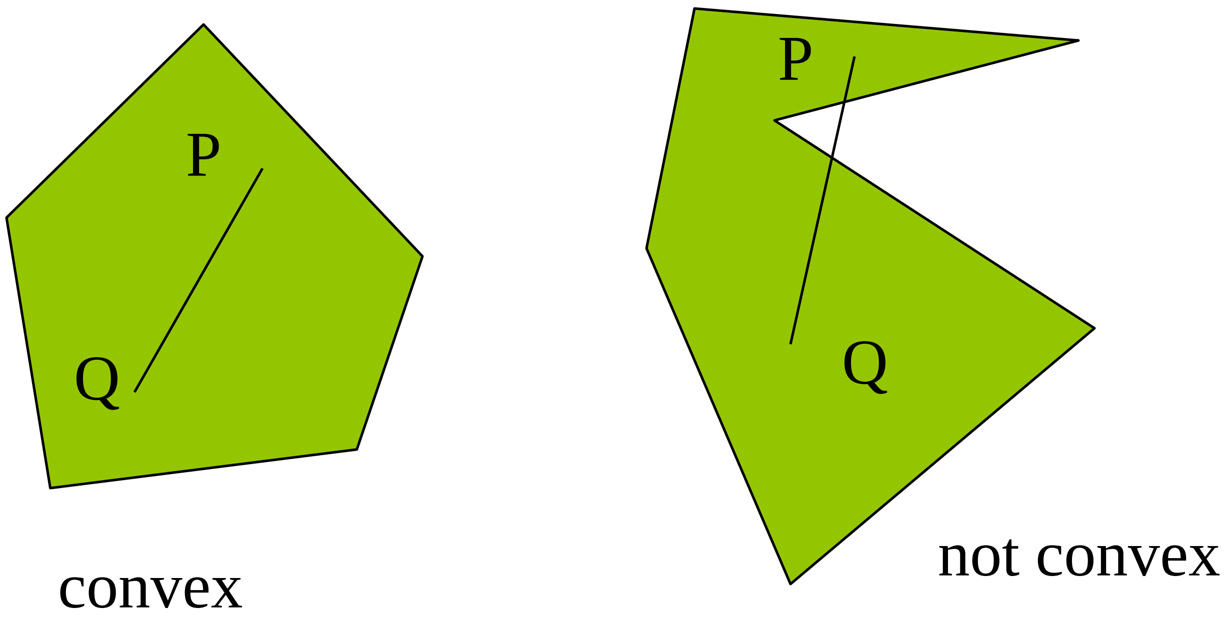
$= \alpha R + (1-\alpha)Q$

For $0<=\alpha<=1$ we get all the points on the ***line segment*** joining R and Q



Convexity

- An object is *convex* iff for any 2 points in the object all points on the line segment between these 2 points are also in the object



Affine Sums

- Consider the "sum"

$P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots\ldots + \alpha_n P_n$

Can show by induction that this sum makes sense iff

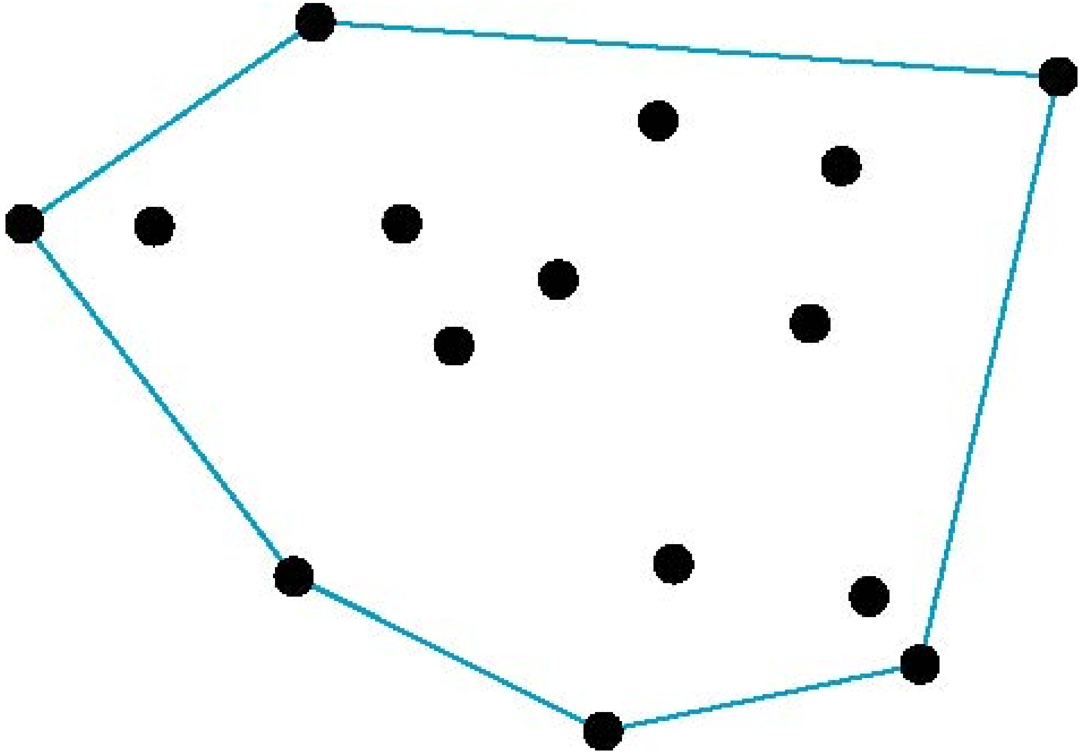
$\alpha_1 + \alpha_2 + \ldots\ldots \alpha_n = 1$

in which case we have the ***affine sum (affine combination)*** of the points $P_1, P_2, \ldots\ldots P_n$

- If, in addition, $\alpha_i >= 0$, then we have the ***convex hull*** of $P_1, P_2, \ldots\ldots P_n$
  - Convex combinations are simply affine combinations where the constants in the combination are limited to be in the interval [0,1].

Convex Hull

- Smallest convex object containing $P_1, P_2, \ldots\ldots P_n$
  - The set of all points P that can be written as convex combinations of $P_1, P_2, \ldots\ldots P_n$
- Formed by “shrink wrapping” points

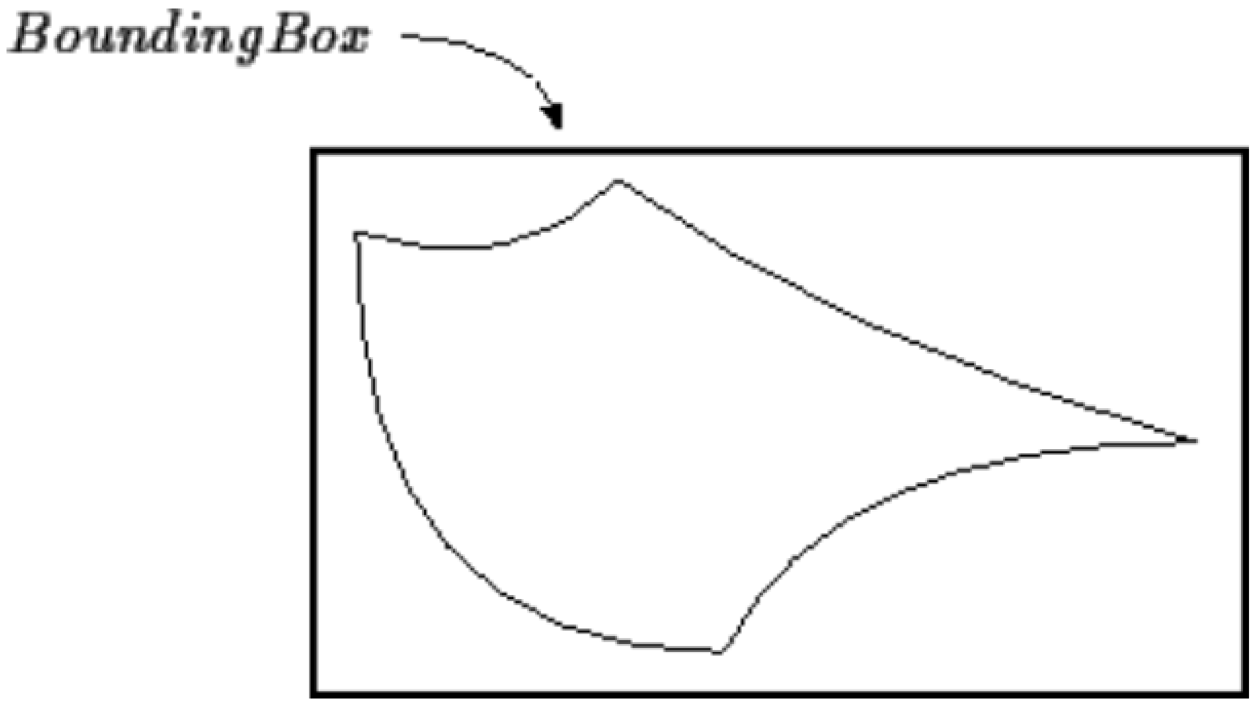


Convex Hull

- Convex combinations are an extremely important concept in computer graphics and geometric modeling.
- The convex-hull concept will allow us to take a set of points, put a **bounding box** about the set of points, and since the bounding box is convex, we are insured that the convex-hull of the set of points is also contained in the bounding box.
- These bounding boxes are the method that we can use to “keep track of” objects without having to continually reference the object's complex mathematical definition.
  - In many cases, a bounding box can be placed about the object and the algorithms can refer to the box when necessary, rather than the object.

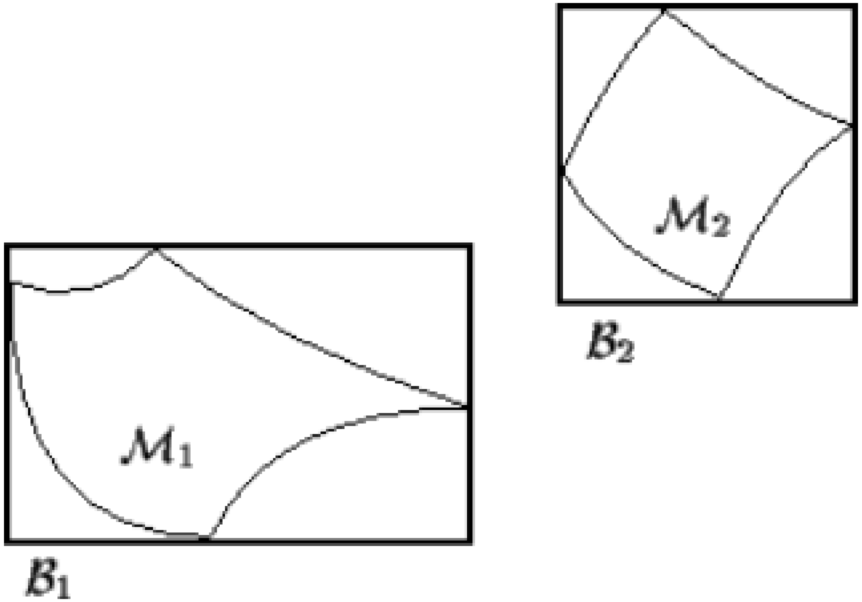
Bounding Box

- A bounding box for an object is just a rectangular box in three-dimensional space, with sides parallel to the coordinate planes, that contains (or surrounds) the object. This illustration below shows a two-dimensional box surrounding a curved object.



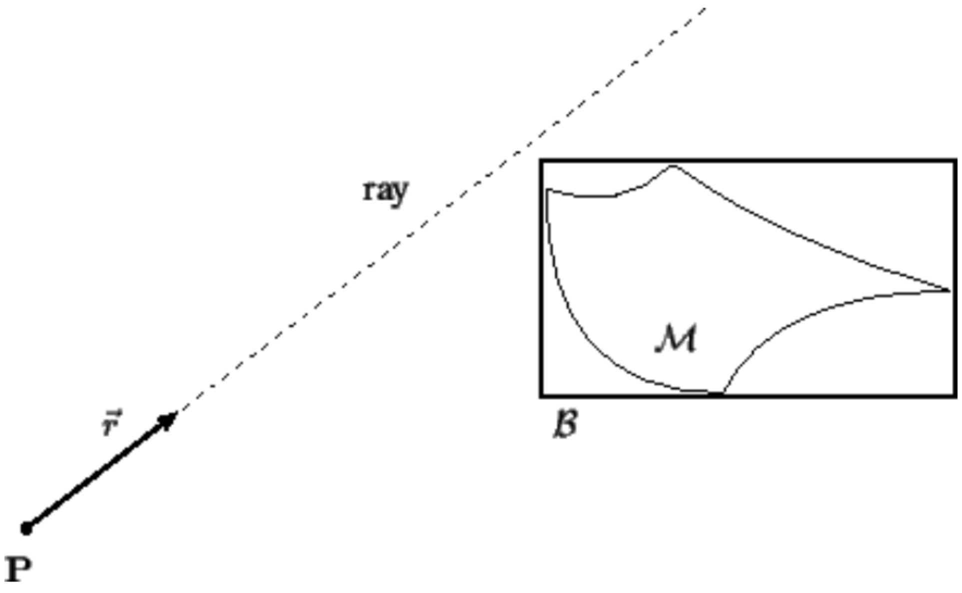
A Simple Intersection Test

- If we have two complex models M1 and M2 and we wish to see if these models do not intersect, we can use a “bounding-box test” to give a quick initial answer.
- If B1 and B2 are bounding boxes containing M1 and M2 respectively, it is easily seen that M1 and M2 cannot intersect if the two bounding boxes do not intersect.



A Ray/Object Intersection Test

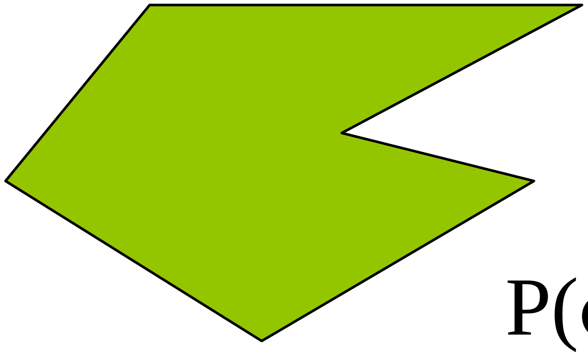
- Want to see if a ray intersects a model M. This is normally a complex operation, and we can simplify it somewhat by using a simple "bounding-box test" to see if the ray misses M.
- By placing a bounding box B around M, we first see if the ray hits B, and if not, we know that the ray does not hit the model M.
- Of course, if the ray hits the bounding box, we then must test it against M for intersection which may be expensive. But by testing first against the bounding box, we can eliminate a number of complex expensive calculations.



Curves and Surfaces

- Curves are one parameter entities of the form $P(\alpha)$ where the function is nonlinear
- Surfaces are formed from two-parameter functions $P(\alpha, \beta)$
  - Linear functions give planes and polygons

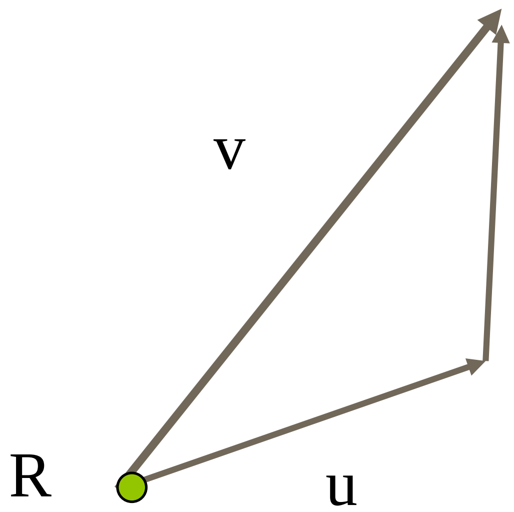
$P(\alpha)$

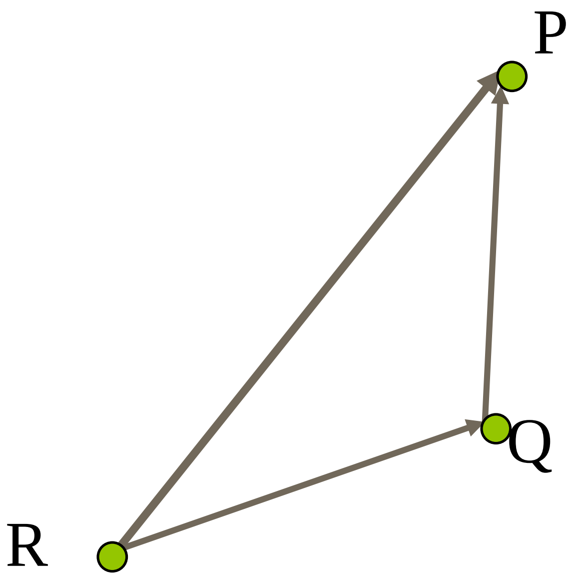


$P(\alpha, \beta)$

Planes

- A plane can be defined by
  - a point and two vectors
  - or by three points





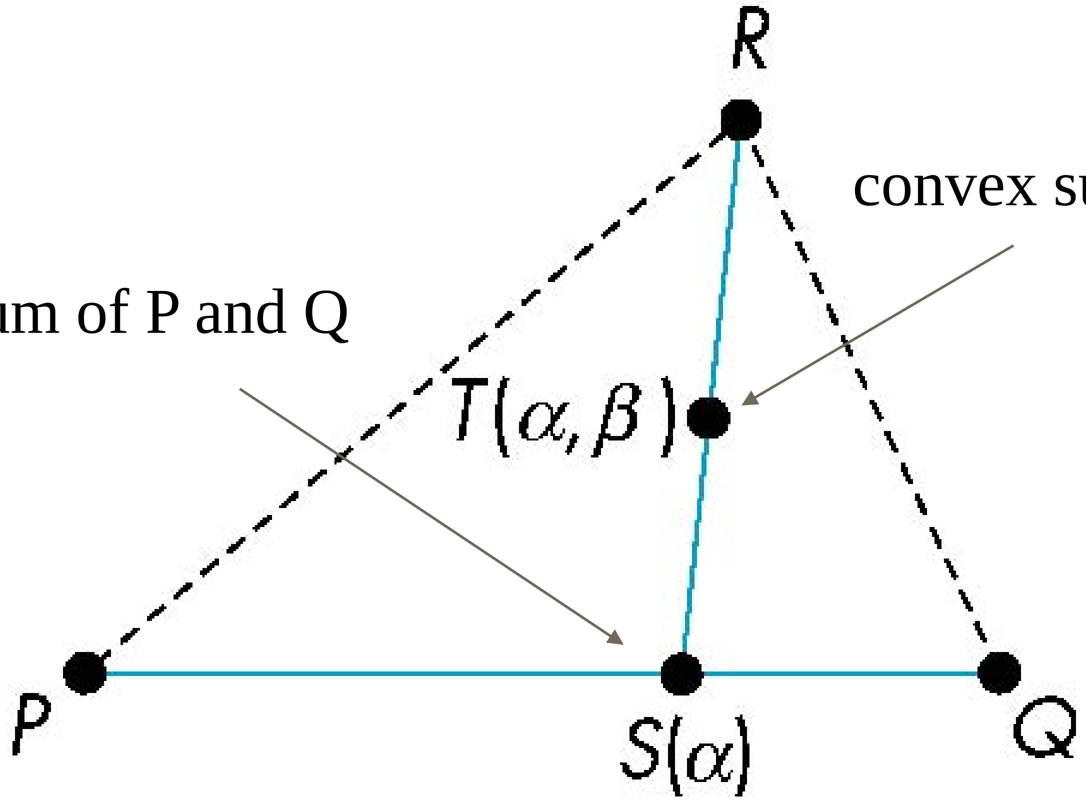
$P(\alpha,\beta)=R+\alpha u+\beta v$

$P(\alpha,\beta)=R+\alpha(Q-R)+\beta(P-R)$

Triangles

convex sum of S(α) and R

convex sum of P and Q



for $0 <= \alpha, \beta <= 1$, we get all points in triangle

Barycentric Coordinates

Triangle is convex so any point inside can be represented as an affine combination

$P(\alpha_{1,}\ \alpha_{2,}\ \alpha_3)=\alpha_1 P+\alpha_2 Q+\alpha_3 R$

where

$\alpha_1 + \alpha_2 + \alpha_3 = 1$

$\alpha_i >= 0$

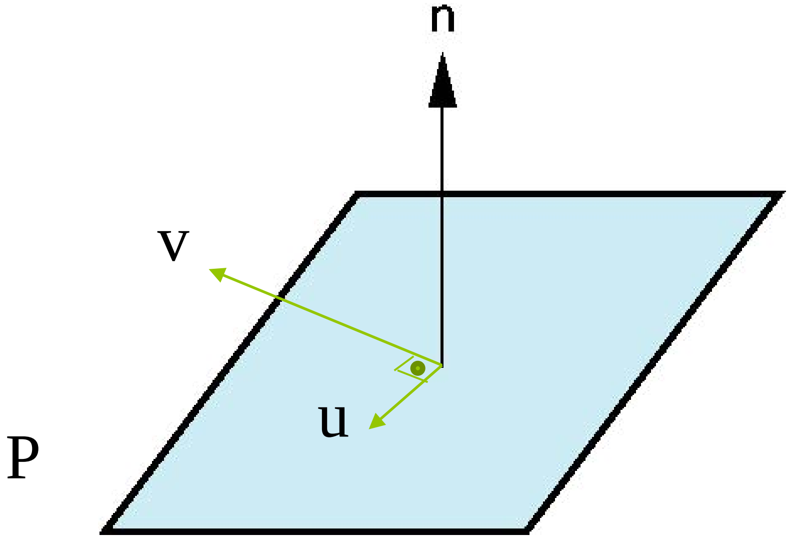
The representation is called the **barycentric coordinate** representation of P

Normals

- Every plane has a vector n **normal** (perpendicular, orthogonal) to it
- From point & two vector form: $P(\alpha,\beta) = R+\alpha u+\beta v$,

we know we can use the cross product to find $n = u \times v$ and the equivalent form

$$(P(\alpha) - P) \cdot n = 0$$



REPRESENTATION

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

Linear Independence

- A set of vectors $v_1, v_2, \ldots, v_n$ is *linearly independent* if

  $\alpha_1 v_1 + \alpha_2 v_2 + .. \alpha_n v_n = 0$ iff $\alpha_1 = \alpha_2 = \ldots = 0$

- If a set of vectors is linearly **independent**, we cannot represent one in terms of the others

- If a set of vectors is linearly **dependent**, at least one of them can be written in terms of the others

Dimension

- In a vector space, the maximum number of linearly independent vectors is fixed and is called the ***dimension*** of the space
- In an *n*-dimensional space, any set of n linearly independent vectors form a ***basis*** for the space
- Given a basis $v_1, v_2,\ldots, v_n$, any vector v can be written as

$$v=\alpha_1 v_1+ \alpha_2 v_2 +\ldots+\alpha_n v_n$$

where the $\{ \alpha_i \}$ are <u>unique</u>

Representation

- Until now we have been able to work with geometric entities without using any frame of reference, such as a coordinate system

- Now, need a frame of reference to relate points and objects to our physical world.
  - For example, where is a point exactly? Can’t answer without a reference system
    - World coordinates
    - Camera coordinates

Coordinate Systems

- Consider a basis $v_1, v_2, \ldots., v_n$
- A vector is written as $v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots. + \alpha_n v_n$
- The list of scalars $\{\alpha_1, \alpha_2, \ldots.\ \alpha_n\}$ is the ***representation*** of v with respect to the given basis
- We can write the representation as a row or a column array of scalars

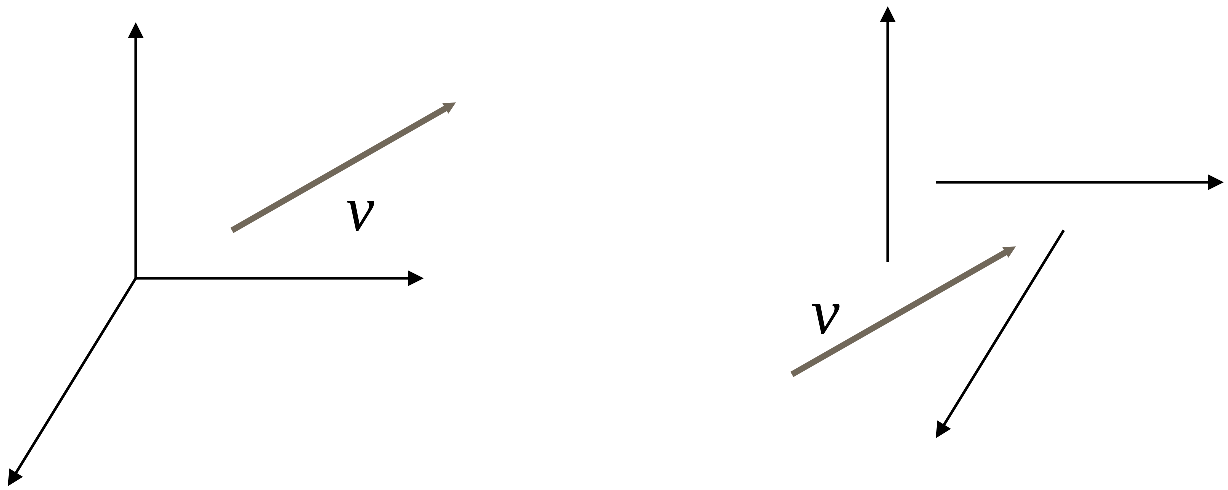
$$\mathbf{a} = [\alpha_1 \;\; \alpha_2 \;\; \ldots. \;\; \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdot \\ \alpha_n \end{bmatrix}$$

Example

- $v = 2v_1 + 3v_2 - 4v_3$
- **a=?**
- **a**$=[2\ 3\ {-4}]^T$
- Note that this representation is with respect to a particular basis.
- For example, in OpenGL
  - we start by representing vectors using the object basis
  - but later the system needs a representation in terms of the camera/eye basis

Coordinate Systems

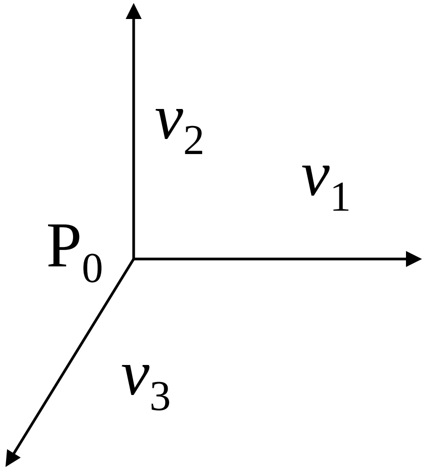
- Which is correct?



- Both are correct, because vectors have no fixed location

Frames

- A coordinate system by itself is insufficient to represent points
- If we work in an affine space, we can add a single point, the ***origin***, to the basis vectors to form a ***frame***



Representation in a Frame

- Frame determined by (P_0, v_1, v_2, v_3)

- Within this frame, every vector can be written as

  $v=\alpha_1 v_1 + \alpha_2 v_2 + \ldots. + \alpha_n v_n$

- Every point can be written as

  $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots. + \beta_n v_n$

Confusing Points and Vectors

Consider the point and the vector

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots. + \beta_n v_n$$

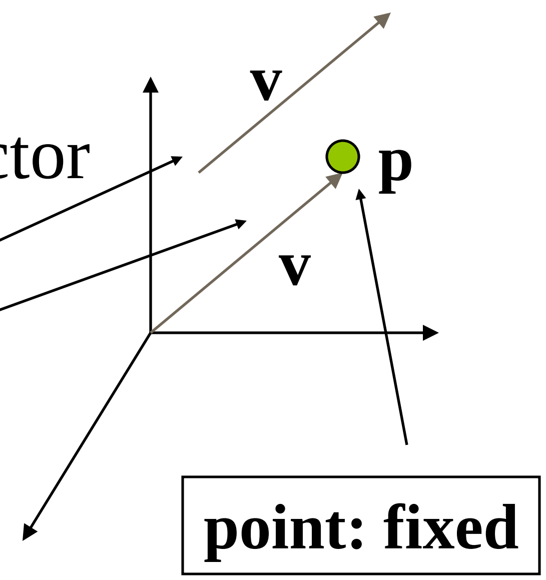
$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots. + \alpha_n v_n$$

They appear to have the similar representations

p$=[\beta_1\ \beta_2\ \beta_3]$ **v**$=[\alpha_1\ \alpha_2\ \alpha_3]$

which confuses the point with the vector

A vector has no position



Vector can be placed anywhere

A Single Representation

>> If we define $0 \cdot P = \mathbf{0}$ and $1 \cdot P = P$ then we can write

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 0]\ [v_1\ v_2\ v_3\ P_0]^T$$

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1\ \beta_2\ \beta_3\ 1]\ [v_1\ v_2\ v_3\ P_0]^T$$

Thus we obtain the four-dimensional ***homogeneous coordinate*** representation

$$\mathbf{v} = [\alpha_1\ \alpha_2\ \alpha_3\ 0]^T$$

$$\mathbf{p} = [\beta_1\ \beta_2\ \beta_3\ 1]^T$$

Homogeneous Coordinates

- The homogeneous coordinates form of a three dimensional point [x y z] is given as

$$\mathbf{p} = [x'\ y'\ z'\ w]^T = [wx\ \ wy\ \ wz\ \ w]^T$$

- We return to a three dimensional point (for $w \neq 0$) by

$x \leftarrow x'/w$

$y \leftarrow y'/w$

$z \leftarrow z'/w$

- If $w=0$, the representation is that of a vector

- Note**: homogeneous coordinates replaces points in 3-dimensions by lines through the origin in 4-dimensions**

- For w=1, the representation of a point is [x y z 1]

Homogeneous Coordinates and Computer Graphics

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices
  - **Hardware pipeline works with 4 dimensional representations**
  - For orthographic viewing, we can maintain $w=0$ for vectors and $w=1$ for points
  - For perspective we need a *perspective division*

Change of Coordinate Systems

- Consider two representations of the same vector with respect to two different bases. The representations are

$$\mathbf{a} = [\alpha_1 \; \alpha_2 \; \alpha_3]$$
$$\mathbf{b} = [\beta_1 \; \beta_2 \; \beta_3]$$

where

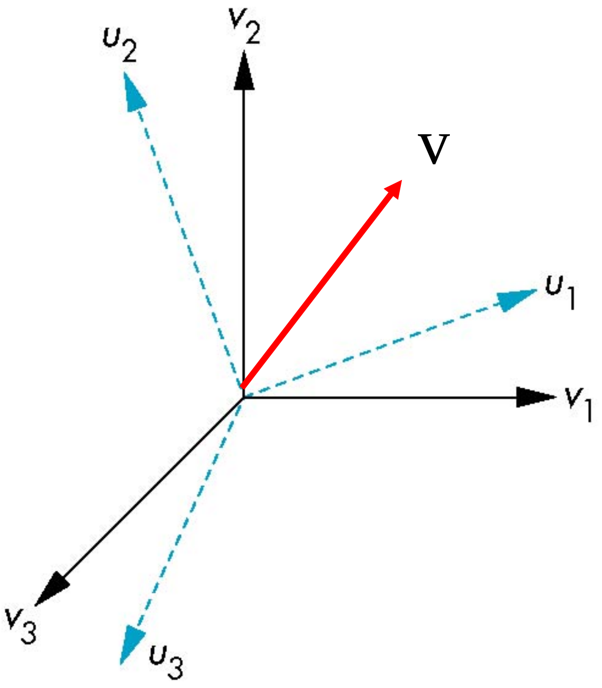
$$\mathrm{v} = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1 \; \alpha_2 \; \alpha_3] \, [v_1 \; v_2 \; v_3]^T$$

$$= \beta_1 u_1 + \beta_2 u_2 + \beta_3 u_3 = [\beta_1 \; \beta_2 \; \beta_3] \, [u_1 \; u_2 \; u_3]^T$$

Representing second basis in terms of first

Each of the basis vectors, u1,u2, u3, are vectors that can be represented in terms of the first basis

$$u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$$
$$u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$$
$$u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$$



Matrix Form

The coefficients define a 3 x 3 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

and the bases can be related by

$$\mathbf{a} = \mathbf{M}^{\mathrm{T}}\mathbf{b}$$

see the textbook for numerical examples

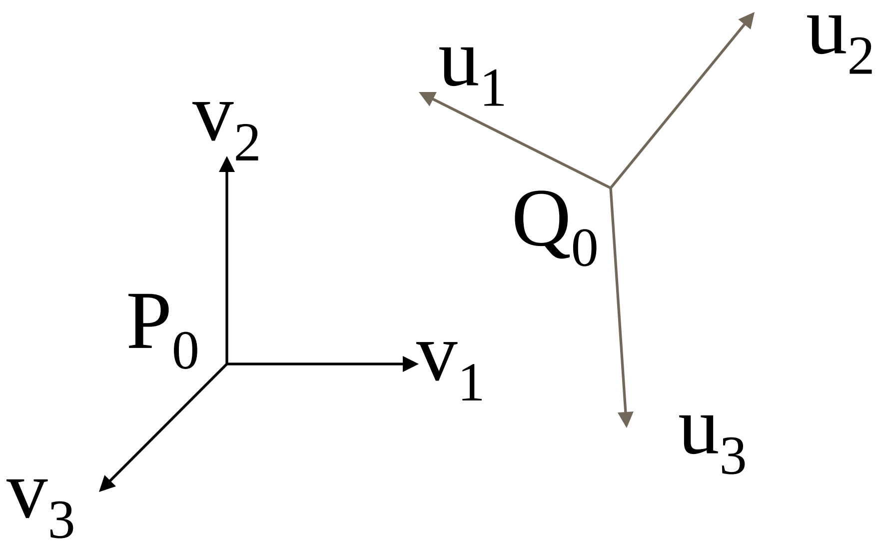
Change of Frames

- We can apply a similar process in homogeneous coordinates to the representations of both points and vectors

Consider two frames:

(P_0, v_1, v_2, v_3)

(Q_0, u_1, u_2, u_3)



- Any point or vector can be represented in either frame
- We can represent Q_0, u_1, u_2, u_3 in terms of P_0, v_1, v_2, v_3

Representing One Frame in Terms of the Other

Extending what we did with change of bases

$$
\begin{aligned}
u_1 &= \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3 \\
u_2 &= \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3 \\
u_3 &= \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3 \\
Q_0 &= \gamma_{41}v_1+\gamma_{42}v_2+\gamma_{43}v_3 +\gamma_{44}P_0
\end{aligned}
$$

defining a 4 x 4 matrix

$$
\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & \gamma_{44} \end{bmatrix}
$$

Working with Representations

Within the two frames, any point or vector has a representation of the same form

$\mathbf{a}=[\alpha_1\ \alpha_2\ \ \alpha_3\ \alpha_4\ ]$ in the first frame
$\mathbf{b}=[\beta_1\ \beta_2\ \ \beta_3\ \beta_4\ ]$ in the second frame

where $\alpha_4 = \beta_4 = 1$ for points and $\alpha_4 = \beta_4 = 0$ for vectors and

$$\mathbf{a}=\mathbf{M}^{T}\mathbf{b}$$

The matrix $\mathbf{M}$ is 4 x 4 and specifies an affine transformation in homogeneous coordinates

Affine Transformations

- Every linear transformation is equivalent to a change in frames
- **Every affine transformation preserves lines**
  - preserves collinearity: so Affine Transformations
    - transform **parallel lines into parallel lines**
    - and **preserve ratios of distances along parallel lines**.

- However, an affine transformation has **only 12 *degrees of freedom*** because 4 of the elements in the matrix are fixed and are a subset of all possible 4 x 4 linear transformations

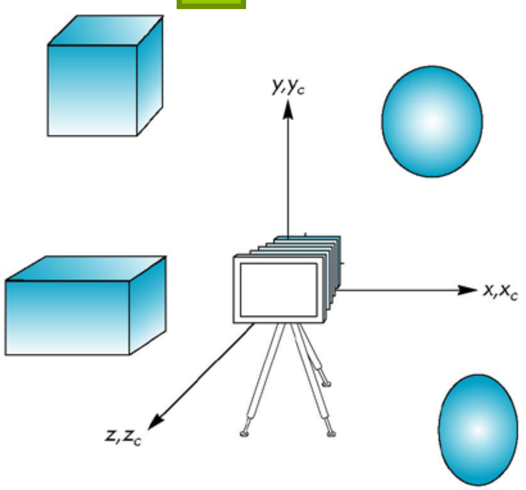
The World and Camera Frames

- When we work with representations, we work with **n-tuples** (arrays of n scalars)
- Changes in frame are then defined by 4 x 4 matrices
- **In OpenGL, the base frame that we start with is the world frame**
- **Eventually we represent entities in the camera frame by changing the world representation using the model-view matrix**
- Initially these frames are the same ($\mathbf{M}=\mathbf{I}$) until we change them using the model-view matrix

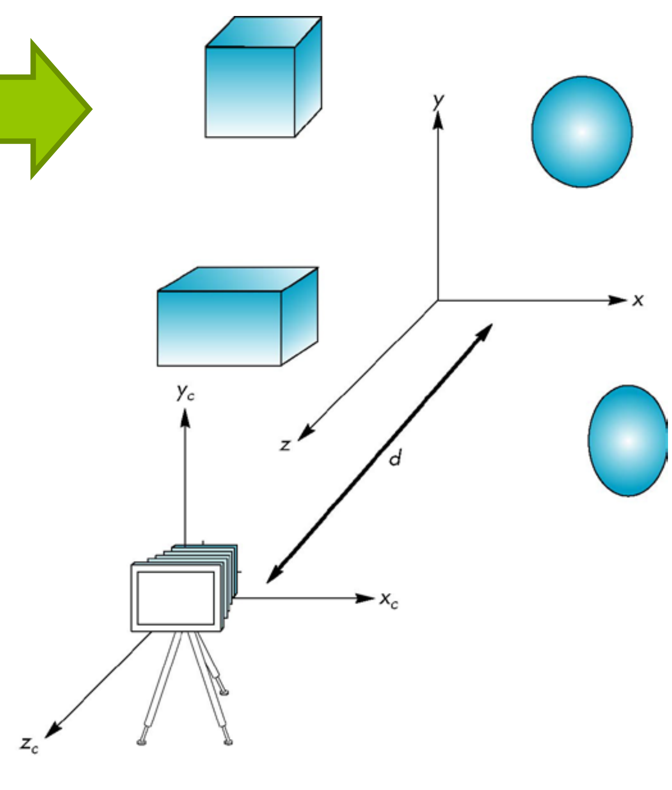
Moving the Camera

If objects are on both sides of z=0, we must move camera frame

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



(a)



(b)

TRANSFORMATIONS

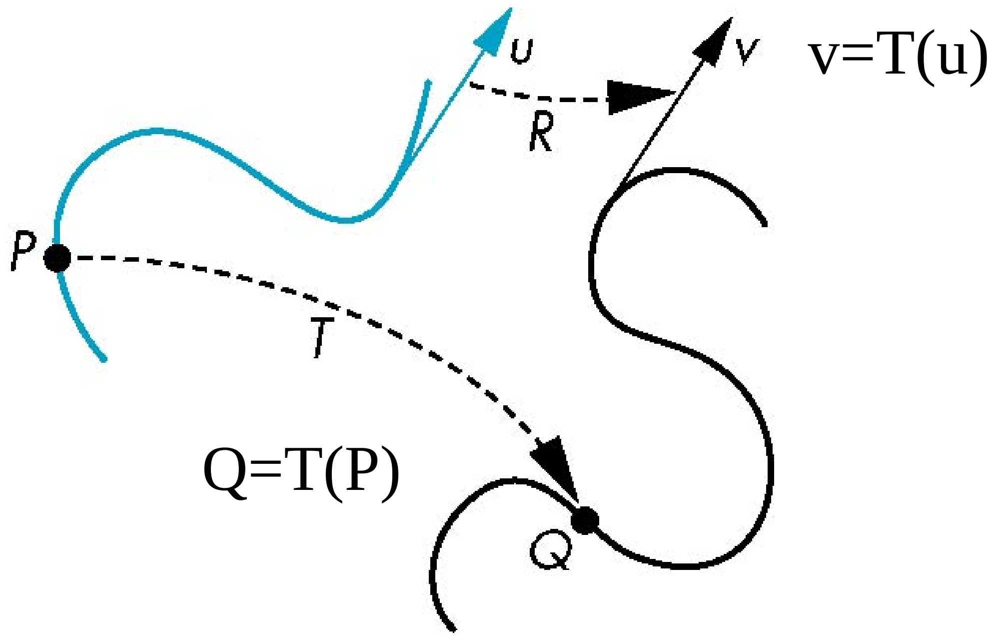
Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

General Transformations

A transformation

- maps points to other points
- and/or maps vectors to other vectors



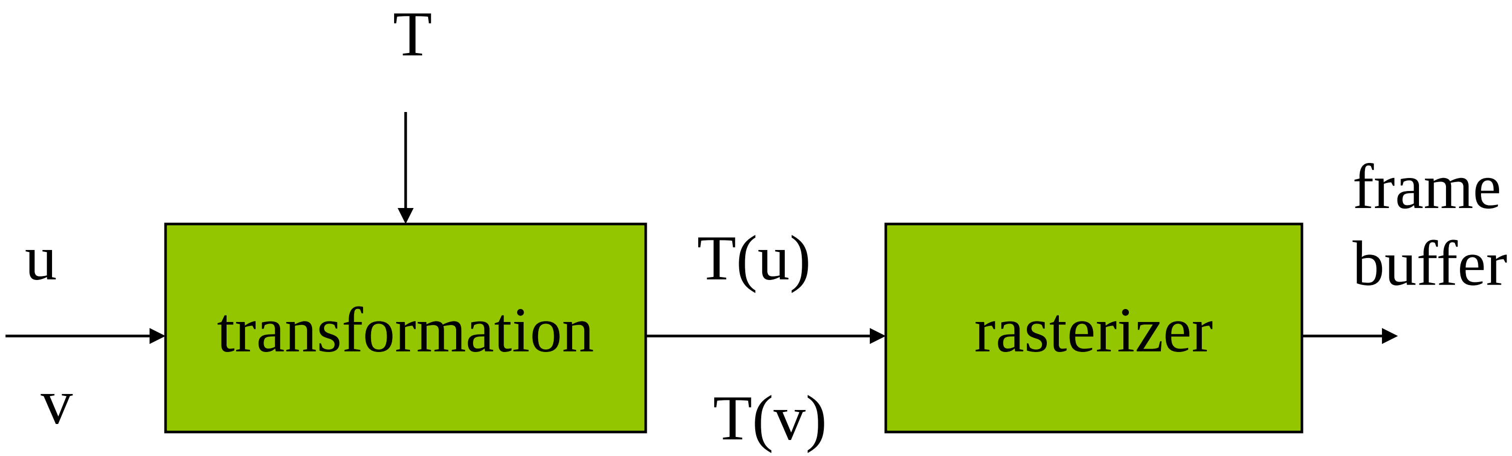
Affine Transformations

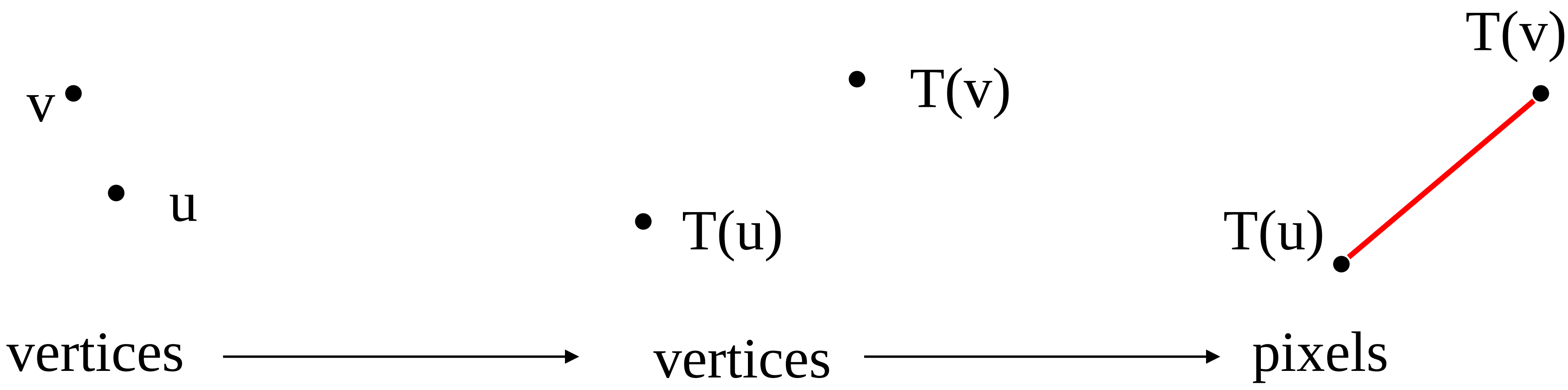
- Line preserving

- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear

- **Importance in computer graphics is that:**
  - we need to transform only endpoints of line segments
  - and let implementation draw line segment between the transformed endpoints

Pipeline Implementation





Notation

We will be working with both: coordinate-free representations of transformations and representations within a particular frame

Our choice of notation:

- P,Q, R: points in an affine space
- u, v, w: vectors in an affine space
- α, β, γ: scalars

- $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$: representations of points

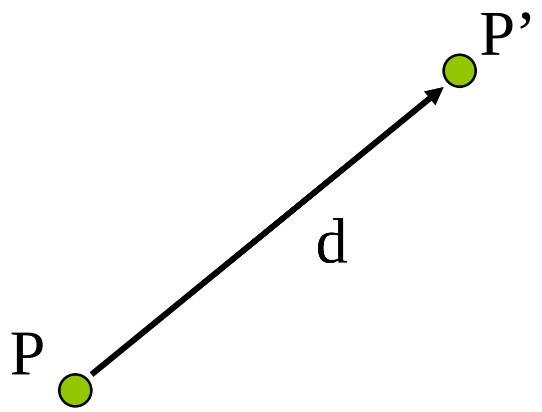
  -array of 4 scalars in homogeneous coordinates

- $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$: representations of vectors

  -array of 4 scalars in homogeneous coordinates

Translation

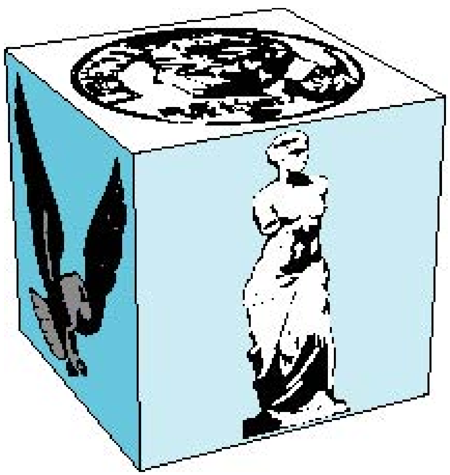
- Move (translate, displace) a point to a new location



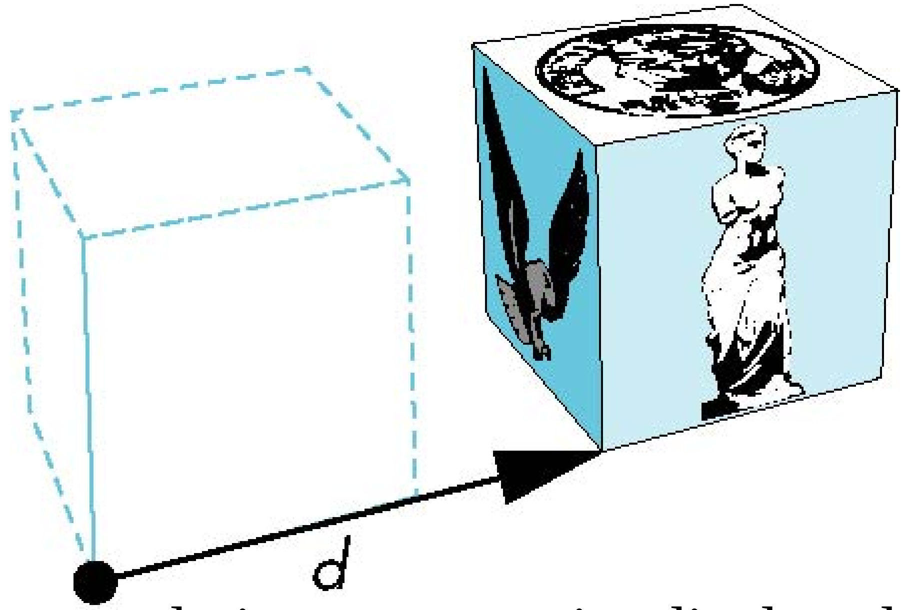
- Displacement determined by a vector d
  - Three degrees of freedom
  - P'=P+d

How many ways?

Although we can move a single point to a new location in infinite ways, when we move many points there is usually only one way



object



translation: every point displaced by same vector

Translation Using Representations

Using the homogeneous coordinate representation in some frame

$\mathbf{p} = [x \quad y \quad z \quad 1]^T$

$\mathbf{p}' = [x' \quad y' \quad z' \quad 1]^T$

$\mathbf{d} = [dx \; dy \; dz \; 0]^T$

Hence $\mathbf{p}' = \mathbf{p} + \mathbf{d}$ or

$x' = x + d_x$

$y' = y + d_y$

$z' = z + d_z$

note that this expression is in four dimensions and expresses point = vector + point

Translation Matrix

We can express translation using a
4 x 4 matrix **T** in homogeneous coordinates
p'=Tp where

$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

<u>This form is better for implementation because</u>

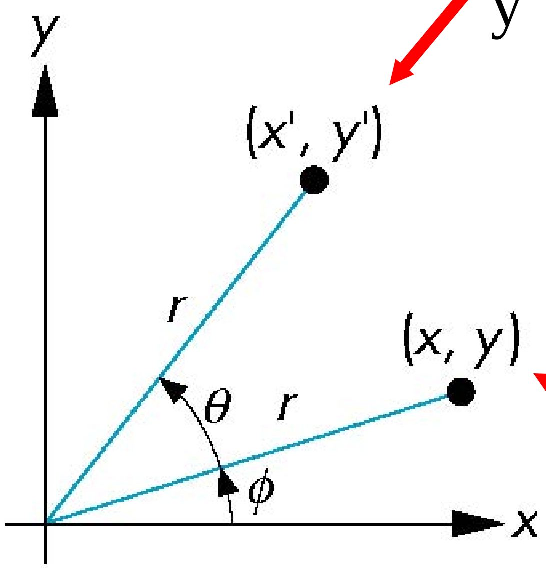
- all affine transformations can be expressed this way
- and multiple transformations can be concatenated together

Rotation (2D)

Consider rotation about the origin by θ degrees

- radius stays the same, angle increases by θ

$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$



remember

$$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$$

$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$$

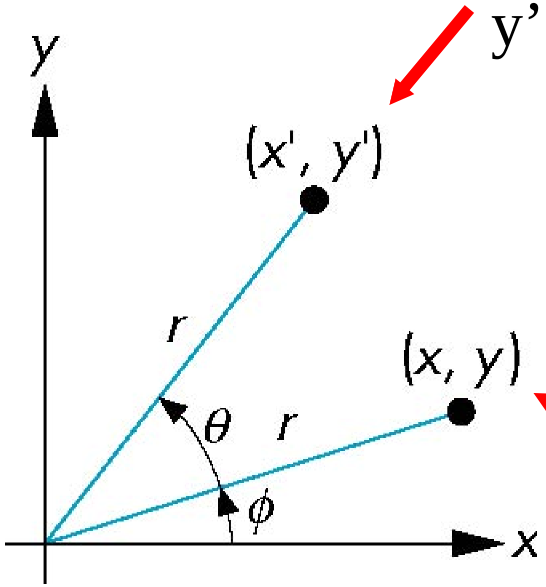
$$x = r\cos\phi$$
$$y = r\sin\phi$$

Rotation (2D)

Consider rotation about the origin by θ degrees

- radius stays the same, angle increases by θ

$$x' = r \cos(\phi + \theta)$$
$$y' = r \sin(\phi + \theta)$$



$$\boxed{\begin{aligned} x' &= x \cos\theta - y \sin\theta \\ y' &= x \sin\theta + y \cos\theta \end{aligned}}$$

$$x = r \cos\phi$$
$$y = r \sin\phi$$

Rotation about the z-axis

- Rotation about z-axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

$$x' = x \cos\theta - y \sin\theta$$
$$y' = x \sin\theta + y \cos\theta$$
$$z' = z$$

  - or in homogeneous coordinates

$$\mathbf{p}' = \mathbf{R}_z(\theta)\, \mathbf{p}$$

Rotation Matrix

$$\mathbf{R} = \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about x and y axes

- Same argument as for rotation about *z*-axis
  - For rotation about x-axis >> x is unchanged
  - For rotation about y-axis >> y is unchanged

$$\mathbf{R} = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

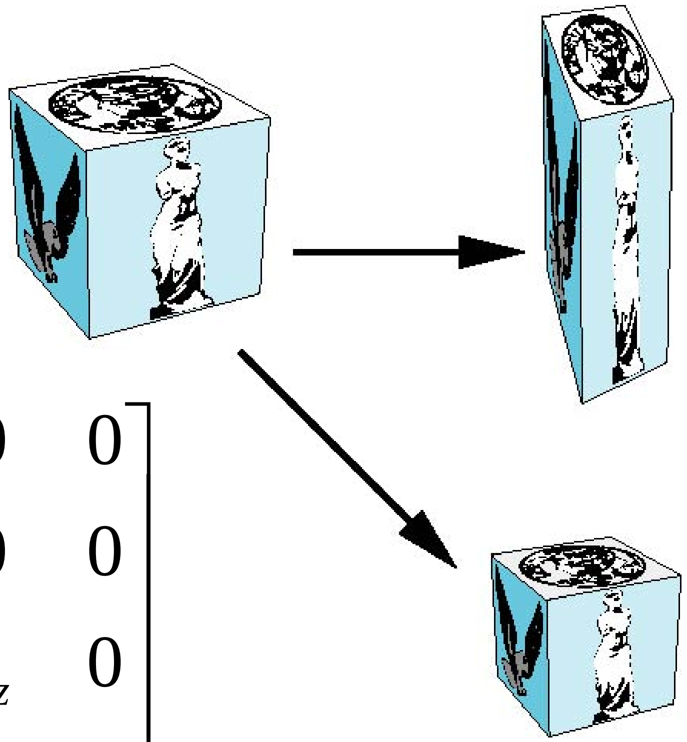
Scaling

Expand or contract along each axis (fixed point of origin)

$$x' = s_x x$$
$$y' = s_y x$$
$$z' = s_z x$$

$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

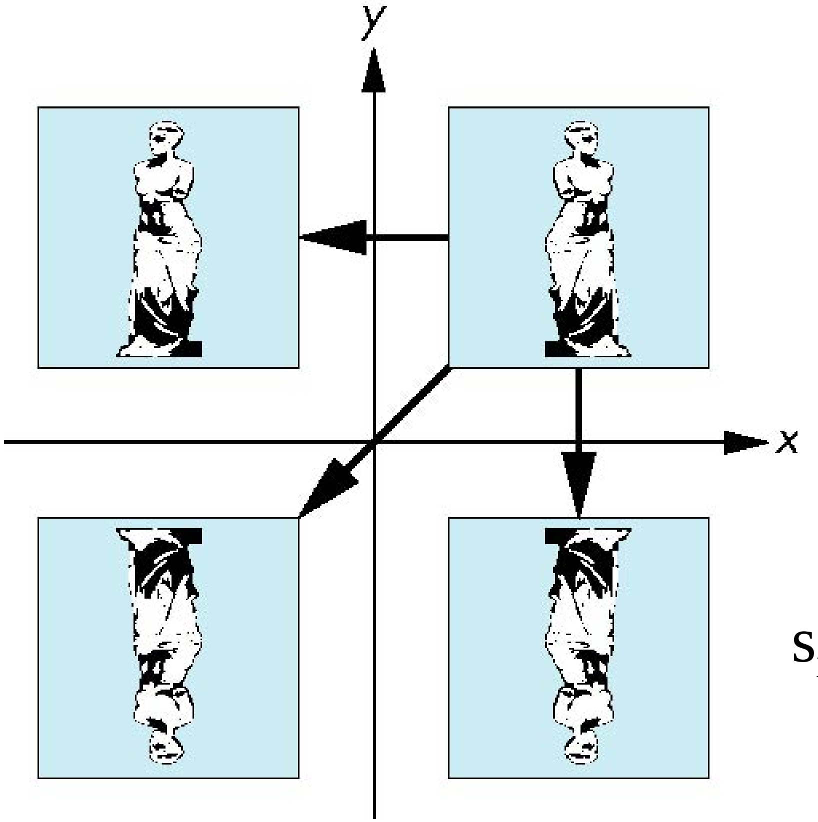


Reflection

corresponds to negative scale factors

$s_x = -1 \; s_y = 1$

original



$s_x = -1 \; s_y = -1$

$s_x = 1 \; s_y = -1$

Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
  - Rotation: $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$

      $>> \mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta) = \mathbf{R}^T(\theta)$
  - Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices in any order
- Because the same transformation is applied to many vertices, the cost of forming a matrix $\mathbf{M}=\mathbf{ABCD}$ is not significant compared to the cost of computing $\mathbf{Mp}$ for many vertices $\mathbf{p}$
- The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

- Note that matrix on the right is the first applied

- Mathematically, the following are equivalent

$$\mathbf{p}' = \mathbf{ABCp} = \mathbf{A}(\mathbf{B}(\mathbf{Cp}))$$

- Note that many references use column matrices to represent points. In terms of column matrices

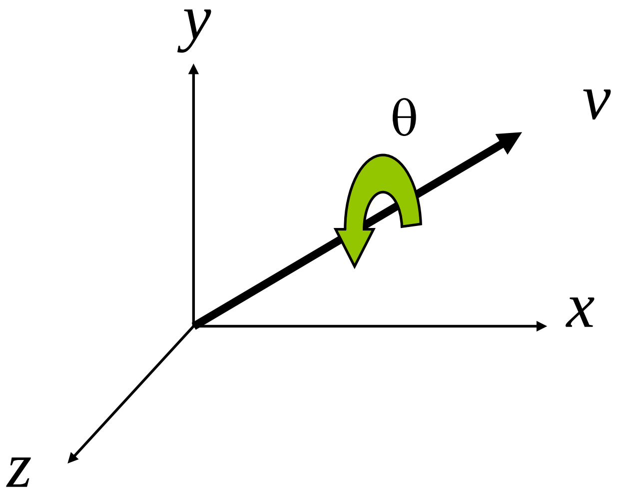
$$\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$$

General Rotation About the Origin

A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the *x*, *y*, and *z* axes

$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\,\mathbf{R}_y(\theta_y)\,\mathbf{R}_x(\theta_x)$$

$\theta_x\ \theta_y\ \theta_z$ are called the Euler angles

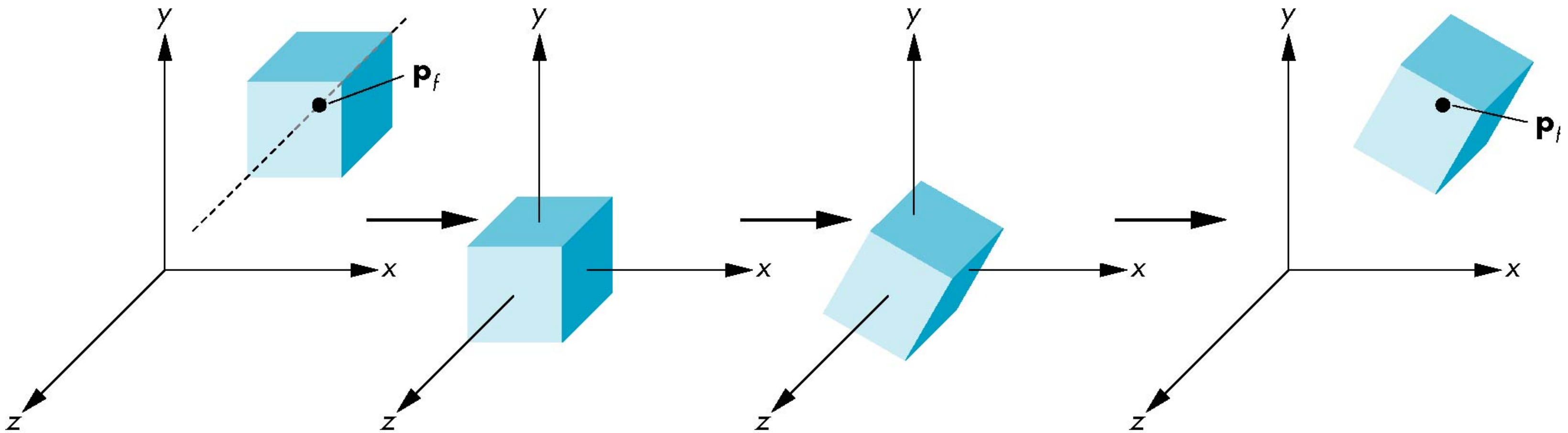


Note: rotations do not commute.
>> We can use rotations in another order but with different angles.

Rotation About a Fixed Point other than the Origin

1. Move fixed point to origin
2. Rotate
3. Move fixed point back

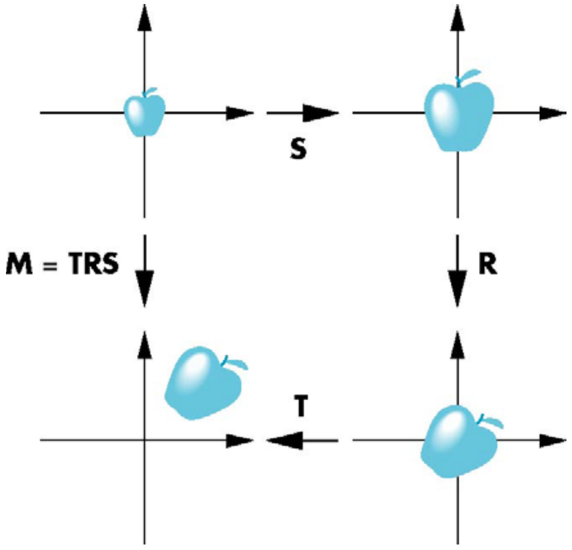
$$\mathbf{M} = \mathbf{T}(p_f)\, \mathbf{R}(\theta)\, \mathbf{T}(-p_f)$$



Instancing

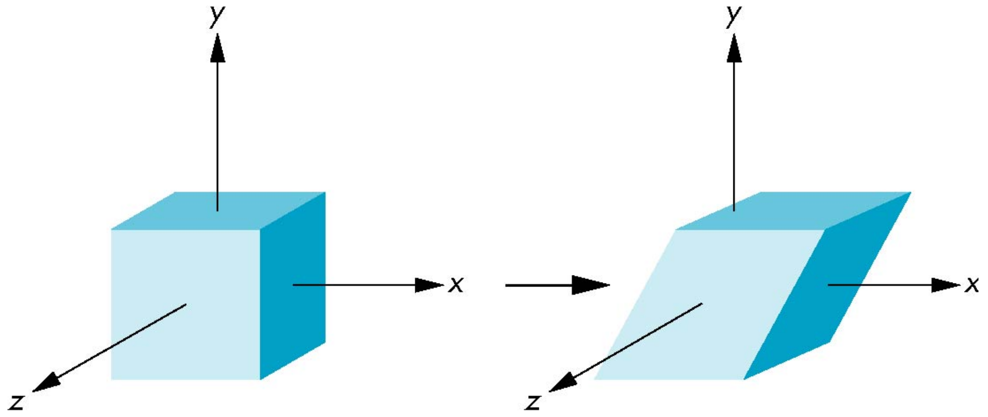
- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an ***instance transformation*** to its vertices to

1. Scale
2. Orient
3. Locate



Shear

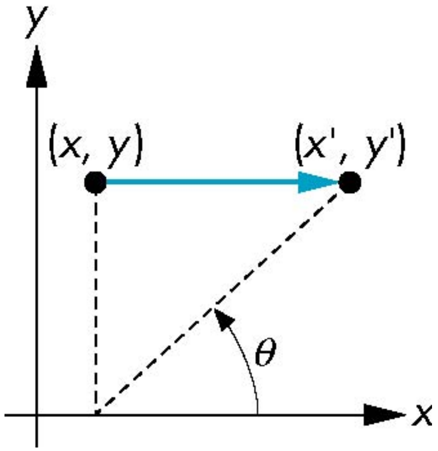
- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



Shear Matrix

Consider a simple shear along *x*-axis

$$x' = x + y \cot\theta$$
$$y' = y$$
$$z' = z$$



$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OPENGL TRANSFORMATIONS

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

Objectives

- Learn how to carry out transformations in OpenGL
  - Rotation
  - Translation
  - Scaling
- Introduce mat.h and vec.h transformations
  - Model-view
  - Projection

3D Transformations

- A vertex is transformed by 4×4 matrices
  - all affine operations are matrix multiplications
- Perspective projections and translations require the 4th row and column.
  - Otherwise, these operations would require a vector-addition operation, in addition to the matrix multiplication.
  - **For operations other than perspective projection, the fourth row is always (0, 0, 0, 1) which leaves the w-coordinate unchanged..**

- All matrices are stored column-major in OpenGL
  - this is opposite of what "C" programmers expect

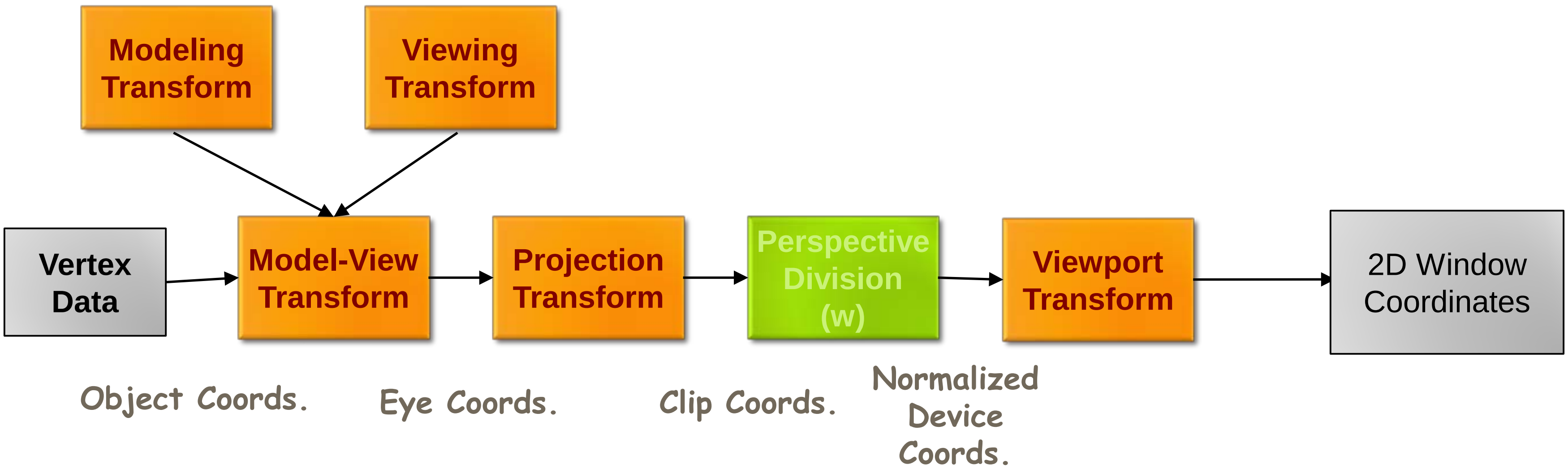
- **Matrices are always post-multiplied**
  - product of matrix and vector is $\mathbf{M}\vec{v}$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Transformations

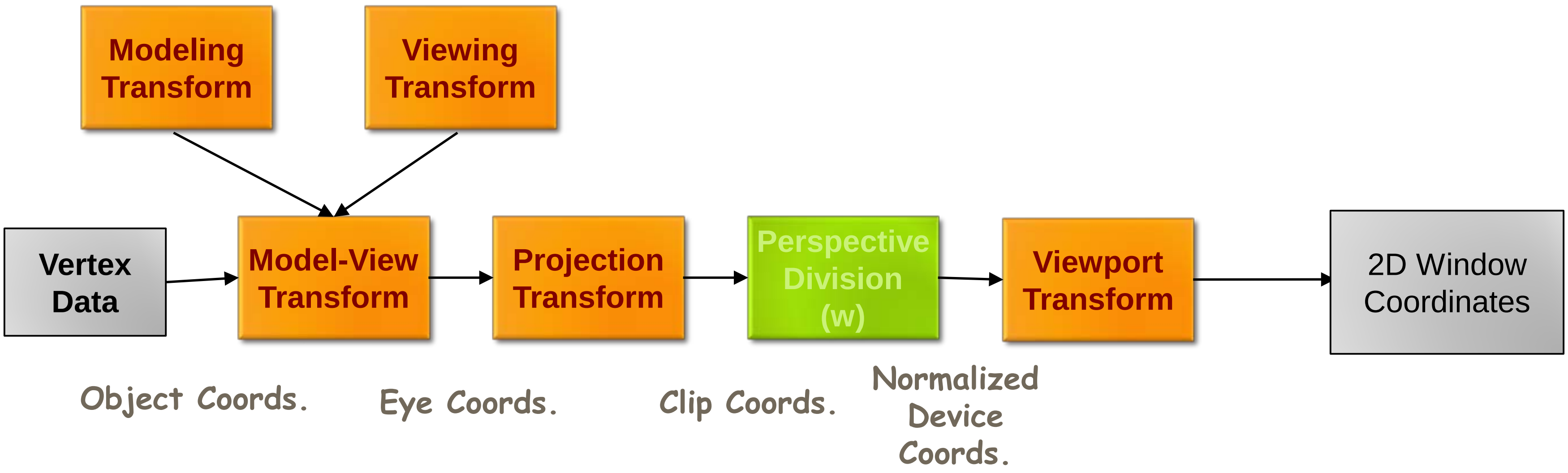
- The processing required for converting a vertex from 3D or 4D space into a 2D window coordinate is done by the transform stage of the graphics pipeline. The operations in that stage are illustrated below.
  - <u>The orange boxes represent a matrix multiplication operation.</u>

- Transformations take us from one “space” to another
  - All of our transforms are 4×4 matrices



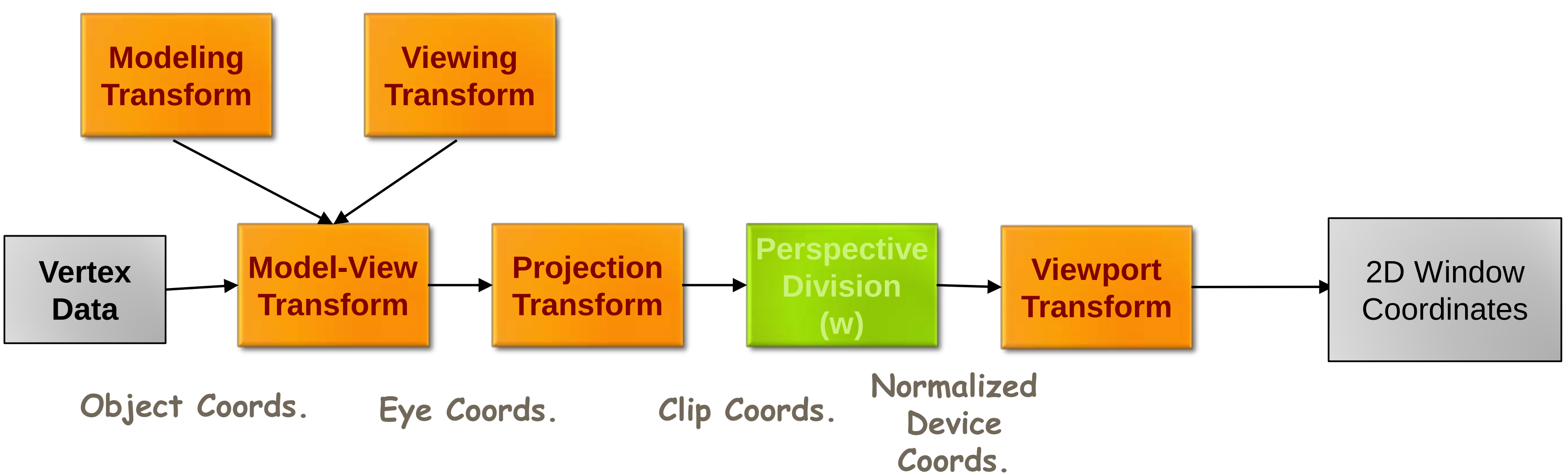
Transformations

- When we want to draw a geometric object, like a chair for instance, we first determine all of the vertices that we want to associate with the chair.
- Next, we determine how those vertices should be grouped to form geometric primitives, and the order we're going to send them to the graphics subsystem. This process is called ***modeling***. Quite often, we'll model an object in its own 3D coordinate system (called **object coordinates,** also called as **model coordinates**).
- When we want to add that object into the scene we're developing, we need to determine its ***world coordinates.***
- We do this by specifying a ***modeling transformation***, which tells the system how to move from one coordinate system to another. i.e., **modeling transforms bring the object into world space.**



Transformations

- ***Viewing transformations*** dictate where the viewing frustum is in world coordinates.
- Modeling transforms, in combination with viewing transforms, are the first transformation that a vertex goes through.
- After model-view transformations, vertices are at the **eye coordinates (camera coordinates)** where the camera [=eye] works in.
- Next, the ***projection transform*** is applied which maps the vertex into ***clip coordinates***, which is where clipping occurs.
- After clipping, we divide by the ***w*** value of the vertex (**perpective division**), which is modified by projection. This division operation is what allows the farther-objects-being-smaller activity.
- The transformed, clipped coordinates are then mapped into the window (**viewport transform**).



Camera Analogy and Transformations

- Modeling transformations >> world coordinates
  - *moving the model* in the scene
- Viewing transformations >> eye coordinates
  - *tripod–define position and orientation of the camera in the world*
  - Can be done by rotations and translations but is often easier to use **`LookAt()`**
- Projection transformations >> clip coordinates
  - *adjust the lens of the camera*
  - The **projection matrix** is used to define the view volume and to select a camera lens
- Viewport transformations >> screen coordinates
  - *enlarge or reduce the physical photograph*

Model-view and Projection Matrices

- Although both are manipulated by the same functions, we have to be careful because incremental changes are always made by postmultiplication
  - For example, rotating model-view and projection matrices by the same matrix are not equivalent operations. Postmultiplication of the model-view matrix is equivalent to premultiplication of the projection matrix

Smooth Rotation

- From a practical standpoint, we often want to use transformations to move and reorient an object smoothly
- Problem: find a sequence of model-view matrices $\mathbf{M_0}$, $\mathbf{M_1}$,….., $\mathbf{M_n}$ so that when they are applied successively to one or more objects we see a smooth transition
- >> For orientating an object, we can use the fact that every rotation corresponds to part of a great circle on a sphere
  - Find the axis of rotation and angle
  - Virtual trackball (see text)

Incremental Rotation

- Consider the two approaches

a) For a sequence of rotation matrices $\mathbf{R_0}, \mathbf{R_1}, \ldots.., \mathbf{R_n}$, find the Euler angles for each and use $\mathbf{R_i} = \mathbf{R_{iz}}\, \mathbf{R_{iy}}\, \mathbf{R_{ix}}$

- Not very efficient

b) **instead: Use the final positions to determine the axis and angle of rotation, then increment only the angle**

- Quaternions can be more efficient than either
  - But we keep those for advanced computer graphics class

Interfaces

- One of the major problems in interactive computer graphics is how to use two-dimensional devices such as a mouse to interface with three dimensional objects
- Example: how to form an instance matrix?
- Some alternatives
  - Virtual trackball
  - 3D input devices such as the spaceball
  - Use areas of the screen
    - Distance from center controls angle, position, scale depending on mouse button depressed

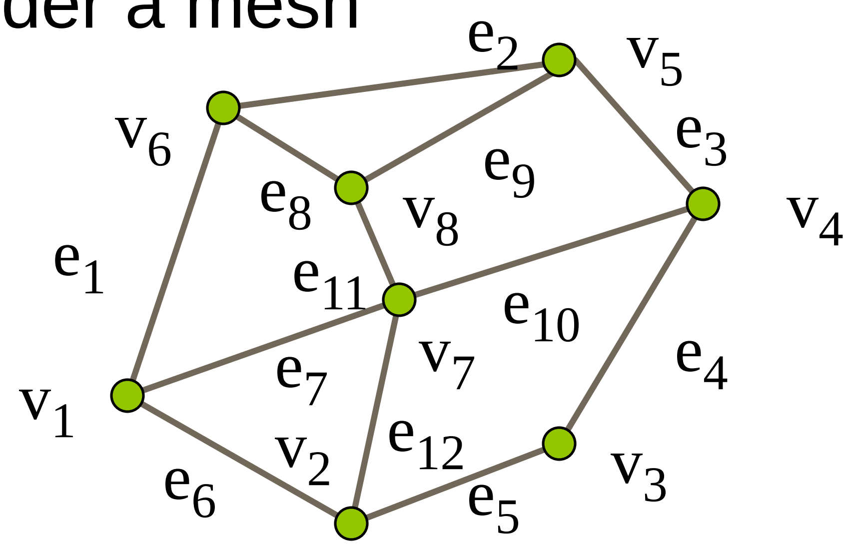
BUILDING MODELS

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

Representing a Mesh

- Consider a mesh



- There are 8 nodes and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges
- Each vertex has a location $v_i = (\, x_i \; y_i \; z_i \,)$

Simple Representation

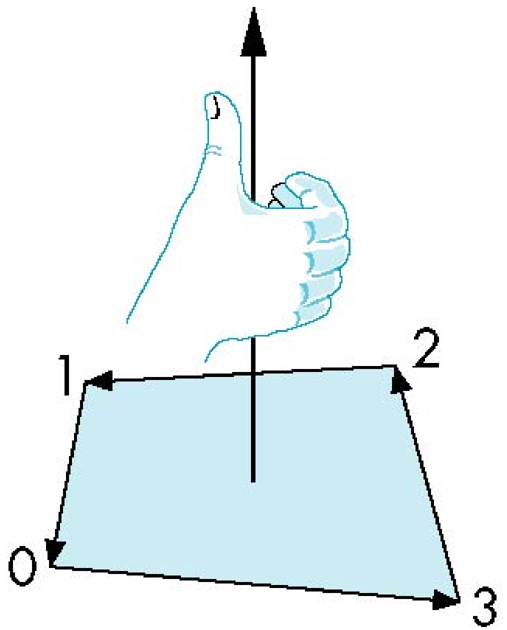
- Define each polygon by the geometric locations of its vertices
- Leads to OpenGL code such as

```
vertex[i] = vec3(x1, x1, x1);
vertex[i+1] = vec3(x6, x6, x6);
vertex[i+2] = vec3(x7, x7, x7);
i+=3;
```

- Inefficient and unstructured
  - For example: Consider moving a vertex to a new location
    - Must search for all occurrences

Inward and Outward Facing Polygons

- The order $\{v_1, v_6, v_7\}$ and $\{v_6, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_6\}$ is different
- The first two describe *outwardly facing* polygons
- Use the ***right-hand rule*** = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and outward facing polygons differently

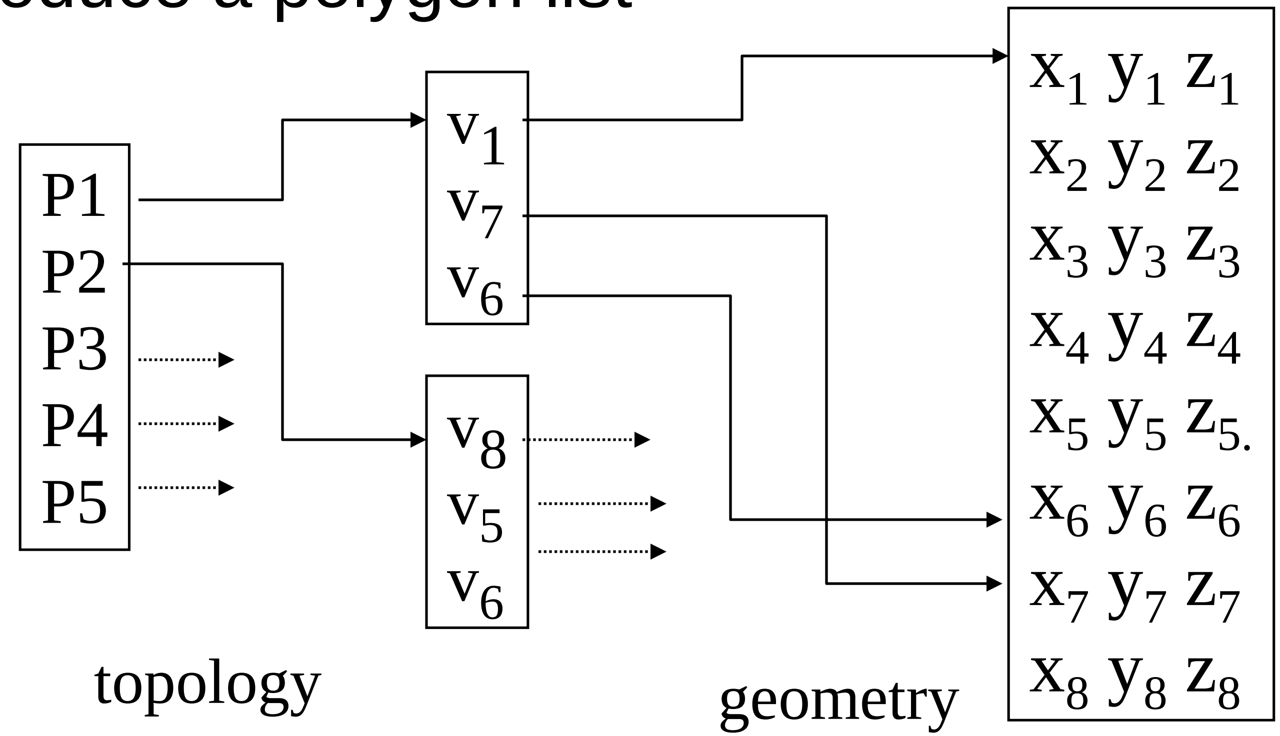


Geometry vs Topology

- Generally it is a good idea to look for data structures that **separate the geometry from the topology**
  - **Geometry**: locations of the vertices
  - **Topology**: organization of the vertices and edges
    - Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
  - **Topology holds even if geometry changes**

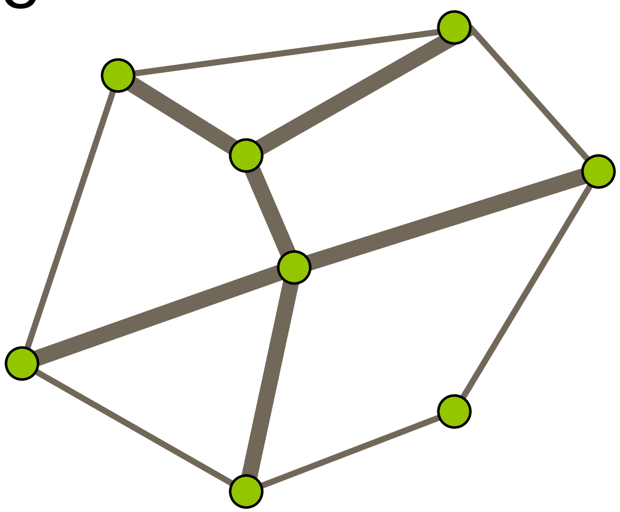
Vertex Lists

- Put the geometry in an array
- Use pointers from the vertices into this array
- Introduce a polygon list



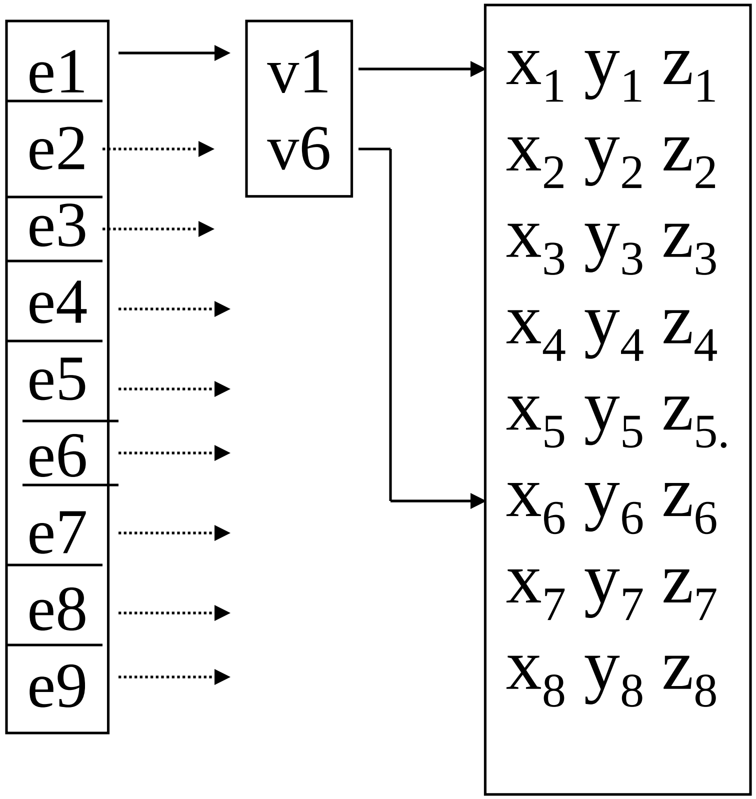
Shared Edges

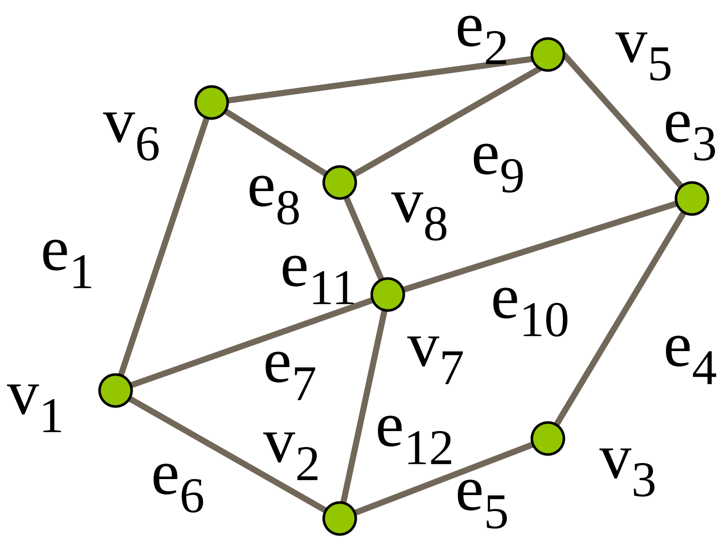
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Better Alternative: Can store mesh by *edge list*

Edge List





Note: polygons are not represented

Modeling a Cube

Model a color cube for the rotating cube program

Define global arrays for vertices and colors

```
GLfloat vertices[][3] =
     {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},{1.0,1.0,-1.0},
      {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},{1.0,-1.0,1.0},
      {1.0,1.0,1.0}, {-1.0,1.0,1.0}};


GLfloat colors[][3] =
     {{0.0,0.0,0.0},{1.0,0.0,0.0},{1.0,1.0,0.0},
      {0.0,1.0,0.0}, {0.0,0.0,1.0},{1.0,0.0,1.0},
      {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

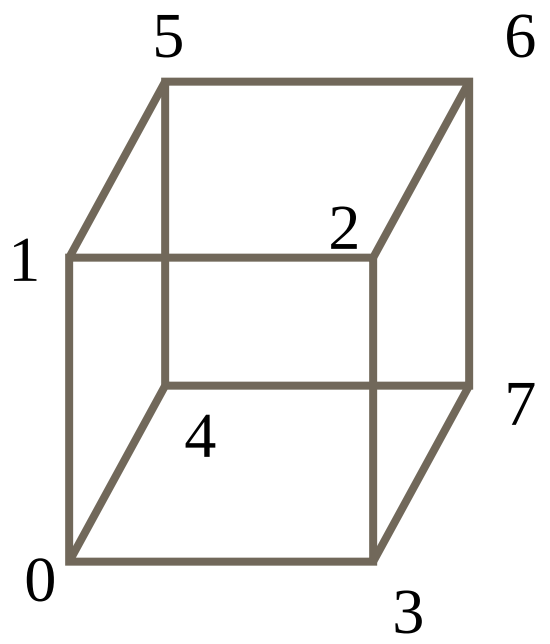
Drawing a triangle from a list of indices

Draw a triangle from a list of indices into the array **`vertices`** and assign a color to each index

```
void triangle(int a, int b, int c, int d)
{
   vcolors[i] = colors[d];
   position[i] = vertices[a];
   vcolors[i+1] = colors[d]);
   position[i+1] = vertices[b];
   vcolors[i+2] = colors[d];
   position[i+2] = vertices[c];
   i+=3;
 }
```

Draw cube from faces

```
void colorcube( )
{
    quad(0,3,2,1);
    quad(2,3,7,6);
    quad(0,4,7,3);
    quad(1,2,6,5);
    quad(4,5,6,7);
    quad(0,1,5,4);
}
```



Note that vertices are ordered so that
we obtain correct outward facing normals

Efficiency

- The weakness of this approach is that we are building the model in the application and must do many function calls to draw the cube

Vertex Arrays

- OpenGL provides a facility called *vertex arrays* that allows us to store array data in the implementation
- Vertex arrays can be used for any attributes including
  - Vertices
  - Colors
  - Color indices
  - Normals
  - Texture coordinates
  - Edge flags

Mapping indices to faces

- So instead, we can form an array of face indices

```
GLubyte cubeIndices[24] = {0,3,2,1,2,3,7,6
     0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};
```

- Each successive four indices describe a face of the cube
  - But we will not pursue efficiency in our example

Rotating Cube

- **Full example**
  - Model Colored Cube
  - Use 3 button mouse to change direction of rotation
  - Use idle function to increment angle of rotation

Cube Vertices

```
// Vertices of a unit cube centered at
   origin, sides aligned with axes
point4 vertices[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```

Colors

```
// RGBA colors
color4 vertex_colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Quad Function

```
// quad generates two triangles for each face
// and assigns colors to the vertices
int Index = 0;
void quad( int a, int b, int c, int d )
{
      colors[Index] = vertex_colors[a];
      points[Index] = vertices[a]; Index++;
      colors[Index] = vertex_colors[b];
      points[Index] = vertices[b]; Index++;
      colors[Index] = vertex_colors[c];
      points[Index] = vertices[c]; Index++;
      colors[Index] = vertex_colors[a];
      points[Index] = vertices[a]; Index++;
      colors[Index] = vertex_colors[c];
      points[Index] = vertices[c]; Index++;
      colors[Index] = vertex_colors[d];
      points[Index] = vertices[d]; Index++;
}
```

Color Cube

```
// generate 12 triangles: 36 vertices
// and 36 colors
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

```
// Array of rotation angles (in degrees) for each
// coordinate axis
enum { Xaxis = 0, Yaxis = 1, Zaxis = 2, NumAxes = 3 };
int Axis = Xaxis;
GLfloat  Theta[NumAxes] = { 0.0, 0.0, 0.0 };

GLuint  theta;
// The location of the "theta" shader uniform variable
```

Initialization I

```
void
init()
{
    colorcube();

    // Create a vertex array object

    GLuint vao;
    glGenVertexArrays ( 1, &vao );
    glBindVertexArray ( vao );
```

Initialization II

```
// Create and initialize a buffer object
   GLuint buffer;
   glGenBuffers( 1, &buffer );
   glBindBuffer( GL_ARRAY_BUFFER, buffer );
   glBufferData( GL_ARRAY_BUFFER, sizeof(points) +
      sizeof(colors), NULL, GL_STATIC_DRAW );
   glBufferSubData( GL_ARRAY_BUFFER, 0,
       sizeof(points), points );
   glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
       sizeof(colors), colors );
// Load shaders and use the resulting shader program
   GLuint program = InitShader( "vshader36.glsl",
       "fshader36.glsl" );
   glUseProgram( program );
```

Initialization III

```
// set up vertex arrays
    GLuint vPosition = glGetAttribLocation( program,
        "vPosition" );
    glEnableVertexAttribArray( vPosition );
    glVertexAttribPointer( vPosition, 4, GL_FLOAT,
        GL_FALSE, 0, BUFFER_OFFSET(0) );

    GLuint vColor = glGetAttribLocation( program,
        "vColor" );
    glEnableVertexAttribArray( vColor );
    glVertexAttribPointer( vColor, 4, GL_FLOAT,
        GL_FALSE, 0, BUFFER_OFFSET(sizeof(points)) );

    theta = glGetUniformLocation( program, "theta" );

    glEnable( GL_DEPTH_TEST );
    glClearColor( 1.0, 1.0, 1.0, 1.0 );
}
```

Display Callback

```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT |
                            GL_DEPTH_BUFFER_BIT );

    glUniform3fv( theta, 1, Theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );

    glutSwapBuffers();
}
```

Mouse Callback

```
void mouse( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN ) {
        switch( button ) {
         case GLUT_LEFT_BUTTON:   Axis = Xaxis; break;
         case GLUT_MIDDLE_BUTTON: Axis = Yaxis; break;
         case GLUT_RIGHT_BUTTON:  Axis = Zaxis; break;
        }
    }
}
```

Idle Callback

```
void
idle( void )
{
    Theta[Axis] += 0.01;

    if ( Theta[Axis] > 360.0 )
    {
        Theta[Axis] -= 360.0;
    }

    glutPostRedisplay();
}
```

Vertex Shader

```
#version 150

in  vec4 vPosition;
in  vec4 vColor;
out vec4 color;

uniform vec3 theta;

void main()
{
   // Convert degrees to radians and compute the sines and cosines of theta for each of
   //   the three axes in one computation.
   vec3 angles = radians( theta );
   vec3 c = cos( angles );
   vec3 s = sin( angles );
   //these matrices are column-major, and the rotation is with -theta
   mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                   0.0,  c.x,  s.x, 0.0,
                   0.0, -s.x,  c.x, 0.0,
                   0.0,  0.0,  0.0, 1.0 );
```

$\sin(-\theta) = -\sin(\theta)$

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Vertex Shader

```
mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );

// Workaround for bug in ATI driver
ry[1][0] = 0.0; ry[1][1] = 1.0;

mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                s.z,  c.z, 0.0, 0.0,
                0.0,  0.0, 1.0, 0.0,
                0.0,  0.0, 0.0, 1.0 );

// Workaround for bug in ATI driver
rz[2][2] = 1.0;

color = vColor;
gl_Position = rz * ry * rx * vPosition;
}
```

$\sin(-\theta) = - \sin(\theta)$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\sin(-\theta) = - \sin(\theta)$

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fragment Shader

```
#version 150

in  vec4 color;
out vec4 fColor;

void main()
{
    fColor = color;
}
```