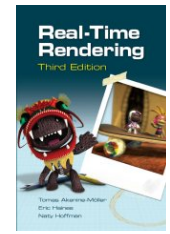
Vectors and Transforms

In

3D Graphics

Course Structure

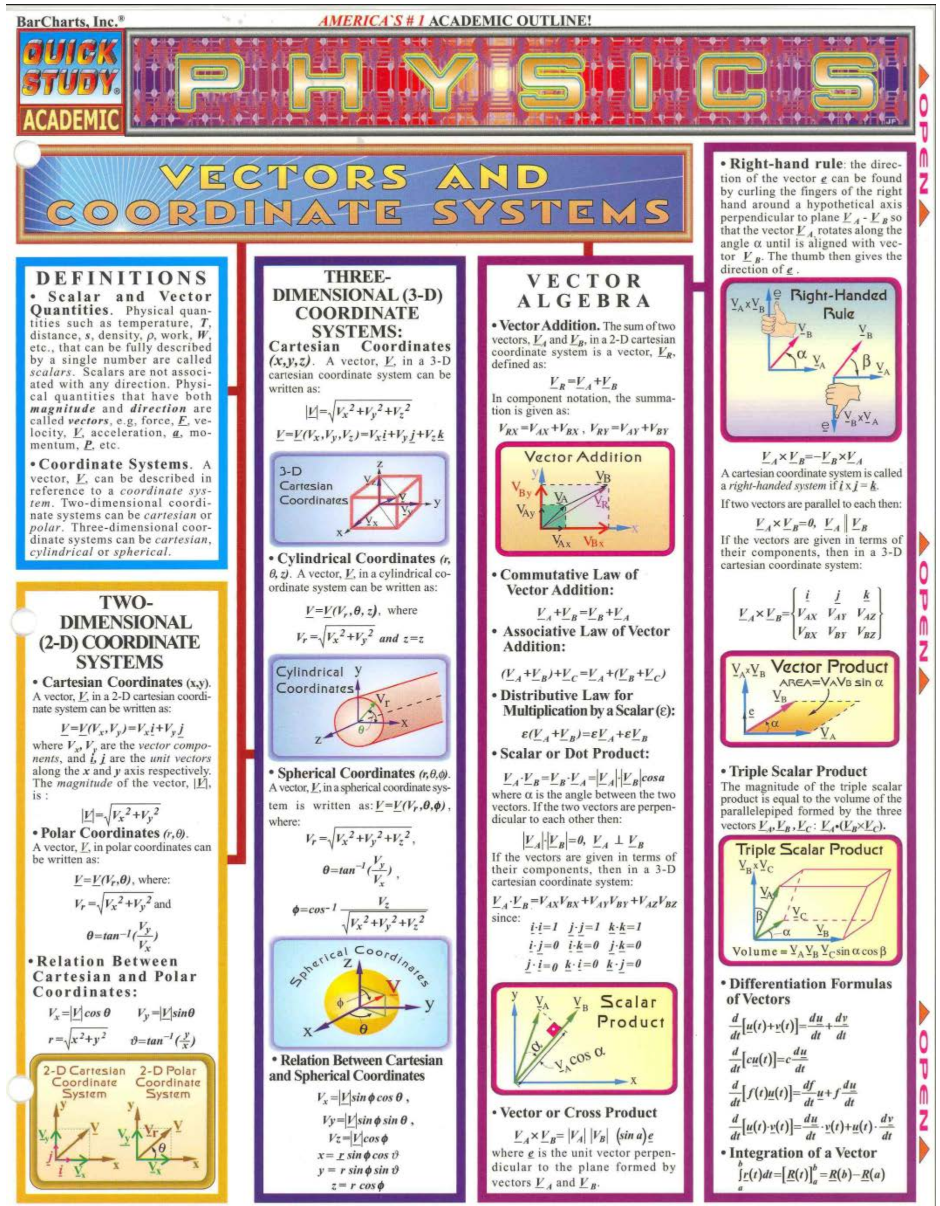


- 14 lectures
  - Book is the verbal format / more meticulous explanations
  - Lecture slides are only short summary
    - Perhaps not enough to fully understand
  - Exam (salstentamen):
    - I will only assume that you have studied the topics covered by the slides.
    - Reading instructions are pointers to more verbal descriptions in the book
    - May come a few “harder” questions, intended to force you to think beyond what’s in the slides (and that could of course accidentally be covered by the book).
- Tutorials – the practical experience
  - 1-6 “holds your hand”. Very fast. Intentionally lots of copy/paste. Do them in 2-3 weeks. No need to wait for their deadlines.
  - Project – Here, you apply the knowledge from tutorial 1-6, so you must have understood them.
    - You will need the 3-4 weeks for the project.

The Bonus Material

- Bonus material on home page
  - http://www.cse.chalmers.se/edu/course/TDA362/schedule.html
  - Purpose: **only** to be of help in case lectures and course book is not enough for you to understand. Sometimes, it helps having same topics explained in a second way.
  - Skip the bonus material if you are not very interested.
  - No exam questions on bonus material!

Quick Repetition of Vector Algebra


BarCharts, Inc.® AMERICA'S # 1 ACADEMIC OUTLINE!

QUICK STUDY ACADEMIC

PHYSICS

VECTORS AND COORDINATE SYSTEMS

DEFINITIONS

• **Scalar and Vector Quantities.** Physical quantities such as temperature, T, distance, s, density, ρ, work, W, etc., that can be fully described by a single number are called *scalars*. Scalars are not associated with any direction. Physical quantities that have both ***magnitude*** and ***direction*** are called ***vectors***, e.g. force, $\underline{F}$, velocity, $\underline{V}$, acceleration, $\underline{a}$, momentum, $\underline{P}$, etc.

• **Coordinate Systems.** A vector, $\underline{V}$, can be described in reference to a *coordinate system*. Two-dimensional coordinate systems can be *cartesian* or *polar*. Three-dimensional coordinate systems can be *cartesian*, *cylindrical* or *spherical*.

TWO-DIMENSIONAL (2-D) COORDINATE SYSTEMS

• **Cartesian Coordinates** (x,y). A vector, $\underline{V}$, in a 2-D cartesian coordinate system can be written as:

$$\underline{V}=\underline{V}(V_x,V_y)=V_x\underline{i}+V_y\underline{j}$$

where V_x, V_y are the *vector components*, and $\underline{i}$, $\underline{j}$ are the *unit vectors* along the x and y axis respectively. The *magnitude* of the vector, $|\underline{V}|$, is :

$$|\underline{V}|=\sqrt{V_x^2+V_y^2}$$

• **Polar Coordinates** (r,θ). A vector, $\underline{V}$, in polar coordinates can be written as:

$$\underline{V}=\underline{V}(V_r,\theta),\ \text{where:}$$

$$V_r=\sqrt{V_x^2+V_y^2}\ \text{and}$$

$$\theta=\tan^{-1}\left(\frac{V_y}{V_x}\right)$$

• **Relation Between Cartesian and Polar Coordinates:**

$$V_x=|\underline{V}|\cos\theta \qquad V_y=|\underline{V}|\sin\theta$$

$$r=\sqrt{x^2+y^2} \qquad \vartheta=\tan^{-1}\left(\frac{y}{x}\right)$$

2-D Cartesian Coordinate System — 2-D Polar Coordinate System

THREE-DIMENSIONAL (3-D) COORDINATE SYSTEMS:

Cartesian Coordinates (x,y,z). A vector, $\underline{V}$, in a 3-D cartesian coordinate system can be written as:

$$|\underline{V}|=\sqrt{V_x^2+V_y^2+V_z^2}$$

$$\underline{V}=\underline{V}(V_x,V_y,V_z)=V_x\underline{i}+V_y\underline{j}+V_z\underline{k}$$

3-D Cartesian Coordinates

• **Cylindrical Coordinates** (r,θ,z). A vector, $\underline{V}$, in a cylindrical coordinate system can be written as:

$$\underline{V}=\underline{V}(V_r,\theta,z),\ \text{where}$$

$$V_r=\sqrt{V_x^2+V_y^2}\ \text{and}\ z=z$$

Cylindrical Coordinates

• **Spherical Coordinates** (r,θ,ϕ). A vector, $\underline{V}$, in a spherical coordinate system is written as: $\underline{V}=\underline{V}(V_r,\theta,\phi)$, where:

$$V_r=\sqrt{V_x^2+V_y^2+V_z^2},$$

$$\theta=\tan^{-1}\left(\frac{V_y}{V_x}\right),$$

$$\phi=\cos^{-1}\frac{V_z}{\sqrt{V_x^2+V_y^2+V_z^2}}$$

Spherical Coordinates

• **Relation Between Cartesian and Spherical Coordinates**

$$V_x=|\underline{V}|\sin\phi\cos\theta,$$

$$V_y=|\underline{V}|\sin\phi\sin\theta,$$

$$V_z=|\underline{V}|\cos\phi$$

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

$$y=r\sin\phi\sin\vartheta$$

$$z=r\cos\phi$$

VECTOR ALGEBRA

• **Vector Addition.** The sum of two vectors, $\underline{V}_A$ and $\underline{V}_B$, in a 2-D cartesian coordinate system is a vector, $\underline{V}_R$, defined as:

$$\underline{V}_R=\underline{V}_A+\underline{V}_B$$

In component notation, the summation is given as:

$$V_{RX}=V_{AX}+V_{BX},\ V_{RY}=V_{AY}+V_{BY}$$

Vector Addition

• **Commutative Law of Vector Addition:**

$$\underline{V}_A+\underline{V}_B=\underline{V}_B+\underline{V}_A$$

• **Associative Law of Vector Addition:**

$$(\underline{V}_A+\underline{V}_B)+\underline{V}_C=\underline{V}_A+(\underline{V}_B+\underline{V}_C)$$

• **Distributive Law for Multiplication by a Scalar** (ε):

$$\varepsilon(\underline{V}_A+\underline{V}_B)=\varepsilon\underline{V}_A+\varepsilon\underline{V}_B$$

• **Scalar or Dot Product:**

$$\underline{V}_A\cdot\underline{V}_B=\underline{V}_B\cdot\underline{V}_A=|\underline{V}_A||\underline{V}_B|\cos\alpha$$

where α is the angle between the two vectors. If the two vectors are perpendicular to each other then:

$$|\underline{V}_A||\underline{V}_B|=0,\ \underline{V}_A\perp\underline{V}_B$$

If the vectors are given in terms of their components, then in a 3-D cartesian coordinate system:

$$\underline{V}_A\cdot\underline{V}_B=V_{AX}V_{BX}+V_{AY}V_{BY}+V_{AZ}V_{BZ}$$

since:

$$\underline{i}\cdot\underline{i}=1\quad \underline{j}\cdot\underline{j}=1\quad \underline{k}\cdot\underline{k}=1$$

$$\underline{i}\cdot\underline{j}=0\quad \underline{i}\cdot\underline{k}=0\quad \underline{j}\cdot\underline{k}=0$$

$$\underline{j}\cdot\underline{i}=0\quad \underline{k}\cdot\underline{i}=0\quad \underline{k}\cdot\underline{j}=0$$

Scalar Product

• **Vector or Cross Product**

$$\underline{V}_A\times\underline{V}_B=|V_A||V_B|(\sin\alpha)\underline{e}$$

where $\underline{e}$ is the unit vector perpendicular to the plane formed by vectors $\underline{V}_A$ and $\underline{V}_B$.

• **Right-hand rule**: the direction of the vector $\underline{e}$ can be found by curling the fingers of the right hand around a hypothetical axis perpendicular to plane $\underline{V}_A$ - $\underline{V}_B$ so that the vector $\underline{V}_A$ rotates along the angle α until is aligned with vector $\underline{V}_B$. The thumb then gives the direction of $\underline{e}$.

Right-Handed Rule

$$\underline{V}_A\times\underline{V}_B=-\underline{V}_B\times\underline{V}_A$$

A cartesian coordinate system is called a *right-handed system* if $\underline{i}\times\underline{j}=\underline{k}$.

If two vectors are parallel to each other then:

$$\underline{V}_A\times\underline{V}_B=0,\ \underline{V}_A\parallel\underline{V}_B$$

If the vectors are given in terms of their components, then in a 3-D cartesian coordinate system:

$$\underline{V}_A\times\underline{V}_B=\begin{vmatrix}\underline{i} & \underline{j} & \underline{k}\\ V_{AX} & V_{AY} & V_{AZ}\\ V_{BX} & V_{BY} & V_{BZ}\end{vmatrix}$$

Vector Product

AREA $=V_AV_B\sin\alpha$

• **Triple Scalar Product**

The magnitude of the triple scalar product is equal to the volume of the parallelepiped formed by the three vectors $\underline{V}_A,\underline{V}_B,\underline{V}_C$: $\underline{V}_A\cdot(\underline{V}_B\times\underline{V}_C)$.

Triple Scalar Product

Volume $=V_AV_BV_C\sin\alpha\cos\beta$

• **Differentiation Formulas of Vectors**

$$\frac{d}{dt}[\underline{u}(t)+\underline{v}(t)]=\frac{d\underline{u}}{dt}+\frac{d\underline{v}}{dt}$$

$$\frac{d}{dt}[c\underline{u}(t)]=c\frac{d\underline{u}}{dt}$$

$$\frac{d}{dt}[f(t)\underline{u}(t)]=\frac{df}{dt}\underline{u}+f\frac{d\underline{u}}{dt}$$

$$\frac{d}{dt}[\underline{u}(t)\cdot\underline{v}(t)]=\frac{d\underline{u}}{dt}\cdot\underline{v}(t)+\underline{u}(t)\cdot\frac{d\underline{v}}{dt}$$

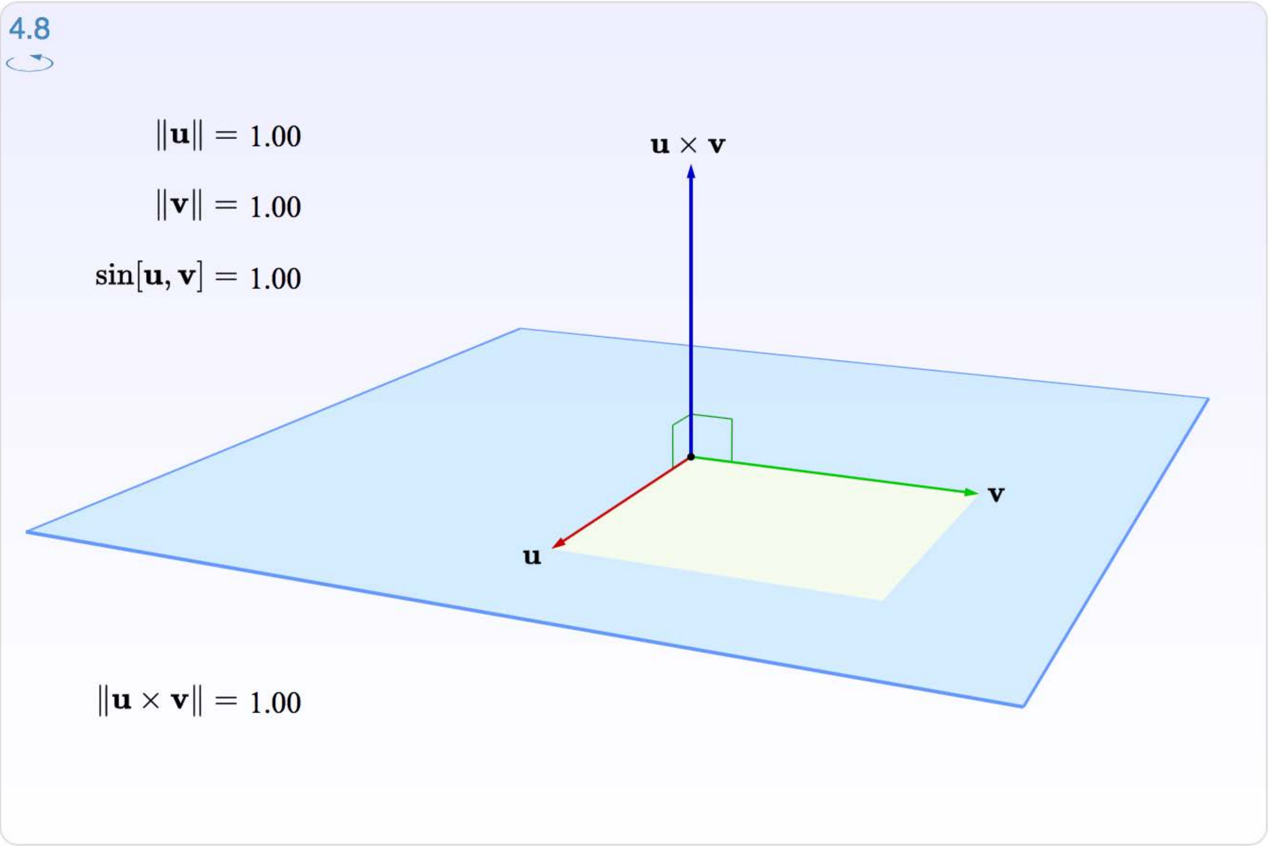
• **Integration of a Vector**

$$\int_a^b\underline{r}(t)dt=[\underline{R}(t)]_a^b=\underline{R}(b)-\underline{R}(a)$$

OPEN

Excellent interactive online linear algebra repetition:

- http://immersivemath.com/ila/index.html



Structure

- Matrices
  - Matrix mult.
  - Transformation Pipeline
  - Practical usage
  - Rotations
  - Translations
  - Homogeneous coordinates
  - Shear / scale / normal matrix
  - Euler matrices
  - Quaternions
  - Projections
- Bresenham’s line drawing algorithm

Why transforms?

- We want to be able to **animate** objects and the camera
  - Translations
  - Rotations
  - Shears
  - …
- We want to be able to use **projection** transforms

How implement transforms?

- Matrices!
- Can you really do everything with a matrix?
- Not everything, but a lot!
- We use 3x3 and 4x4 matrices

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \qquad \mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}\begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix}\begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

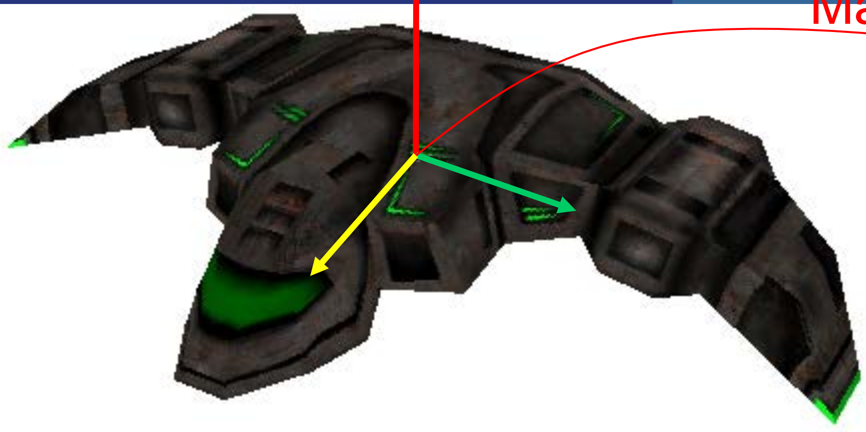
Matrix multiplication

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_{00}p_x + m_{01}p_y + m_{02}p_z \\ m_{10}p_x + m_{11}p_y + m_{12}p_z \\ m_{20}p_x + m_{21}p_y + m_{22}p_z \end{pmatrix}$$

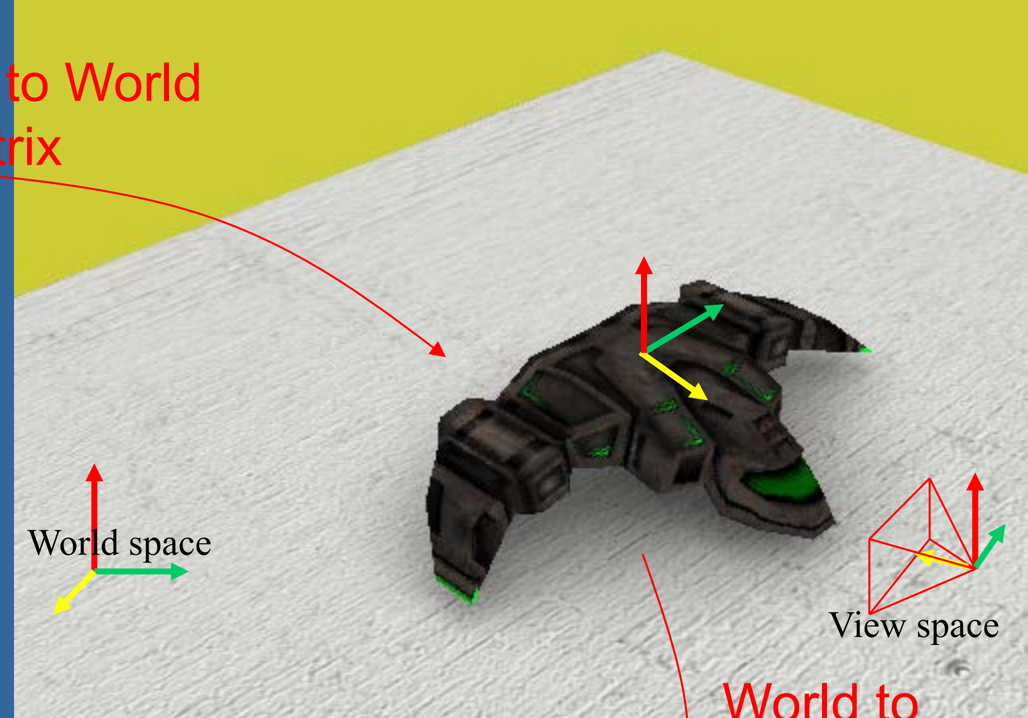
$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Model to World Matrix

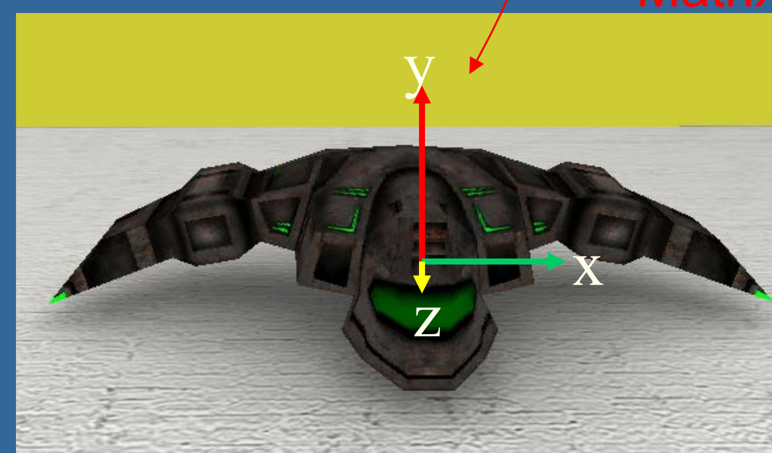


Model space



World space

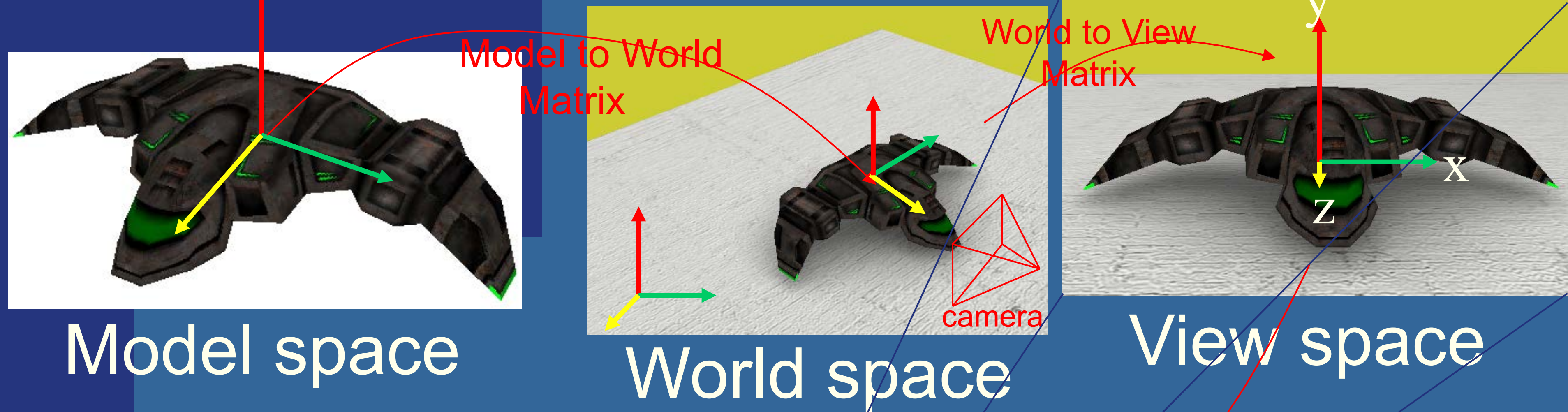
World to View Matrix



View space

ModelViewMtx = ”Model to View Matrix”

ModelViewMtx * v = $(M_{V \leftarrow W} * M_{W \leftarrow M}) * v$



ModelViewMtx = "Model to View Matrix"

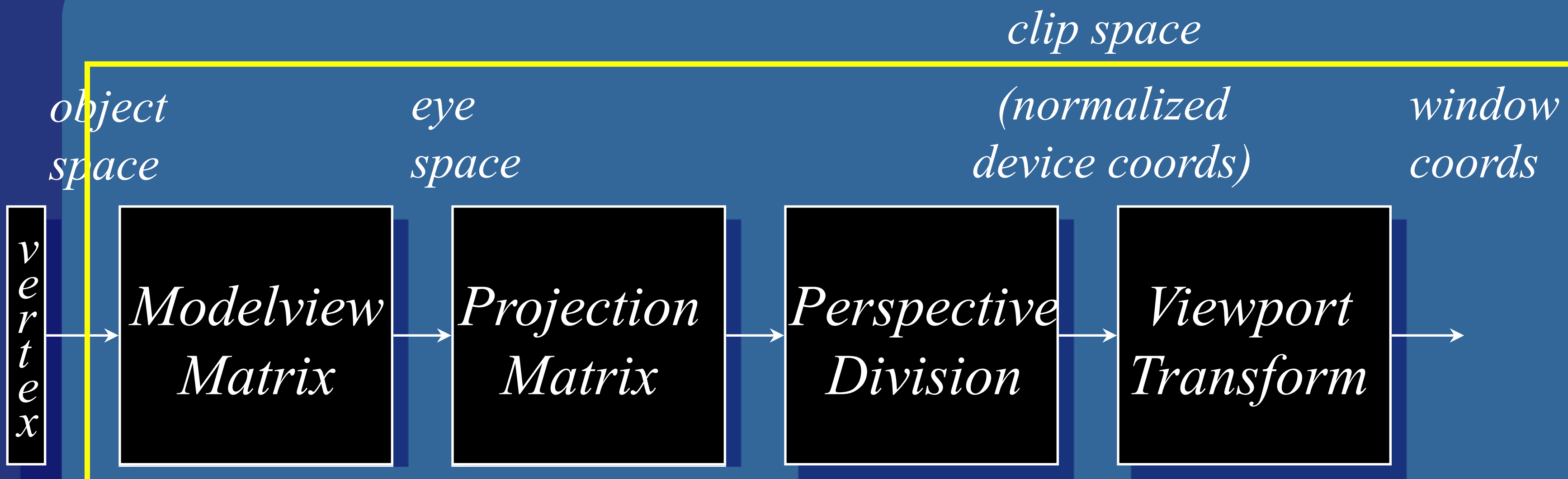
ModelViewMtx * v = $(M_{V \leftarrow W} * M_{W \leftarrow M}) * v$

Full projection:

V_{clip_space} = projectionMatrix * ModelViewMatrix * v_{model_space}

Or simply: $v_{clip\_space} = M_{MVP} * v$

Transformation Pipeline

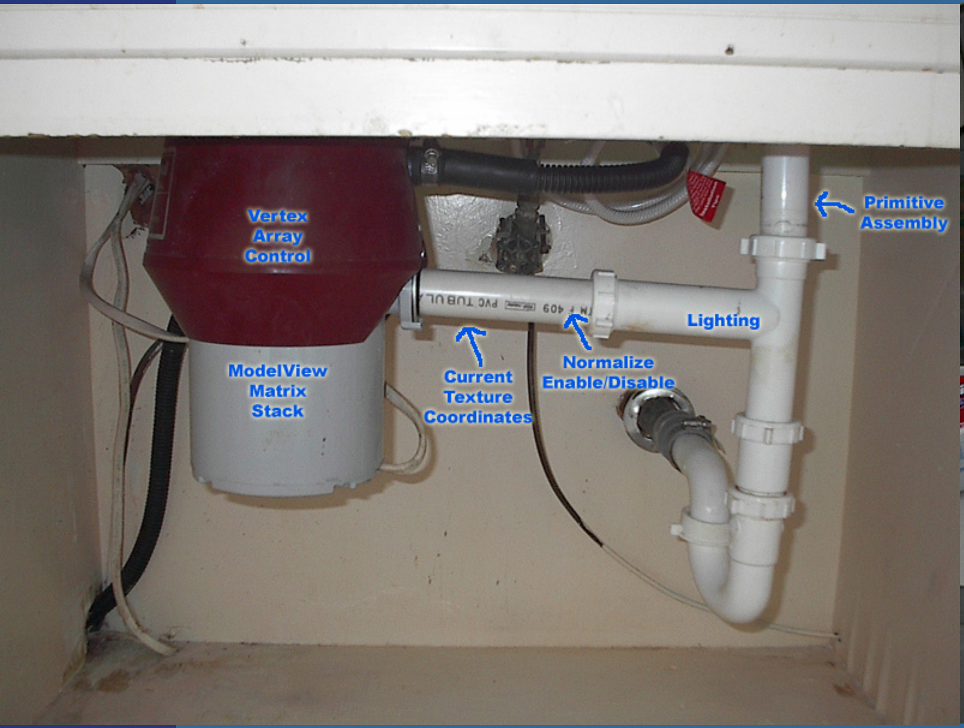


Done by the vertex shader:

```
gl_Position = modelViewProjectionMatrix*vec4(vertex,1);
```

OpenGL | Geometry stage | done on GPU

The OpenGL Pipeline



From http://deltronslair.com/glpipe.html

How do I use transforms practically?

- Say you have a circle with origin at (0,0,0) and with radius 1, i.e., a unit circle 

- `mat4 m = translate({8,0,0});` // create translation matrix
- `RenderCircle(m);` // Draw circle using m as
  // model-to-world matrix

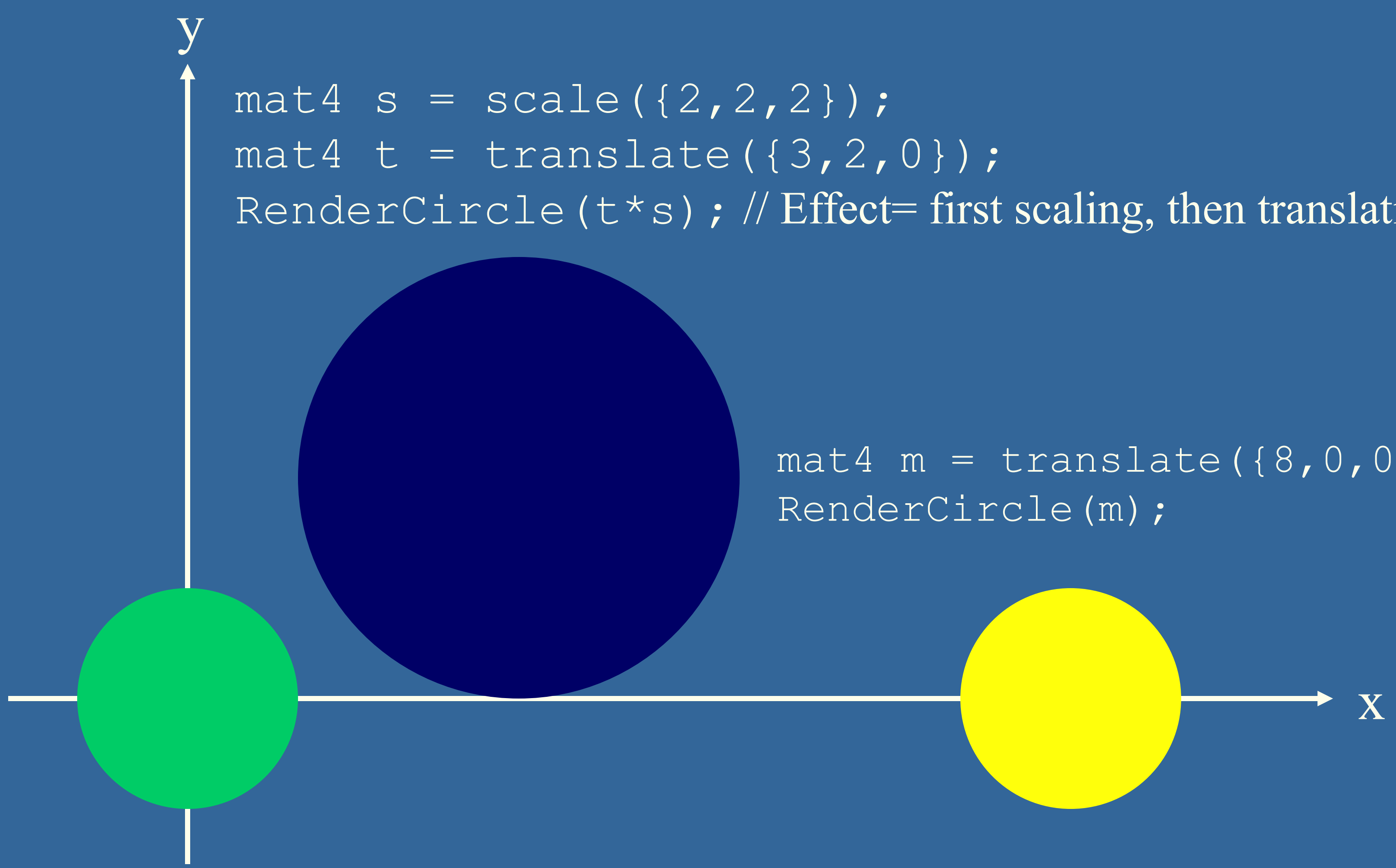
- `mat4 s = scale({2,2,2});` // create scaling matrix
- `mat4 t = translate({3,2,0});` // create translation matrix
- `RenderCircle(t*s);` // use matrix (t*s)

What happens?
See next slide...

Cont'd from previous slide
A simple 2D example

- A circle in **model space**

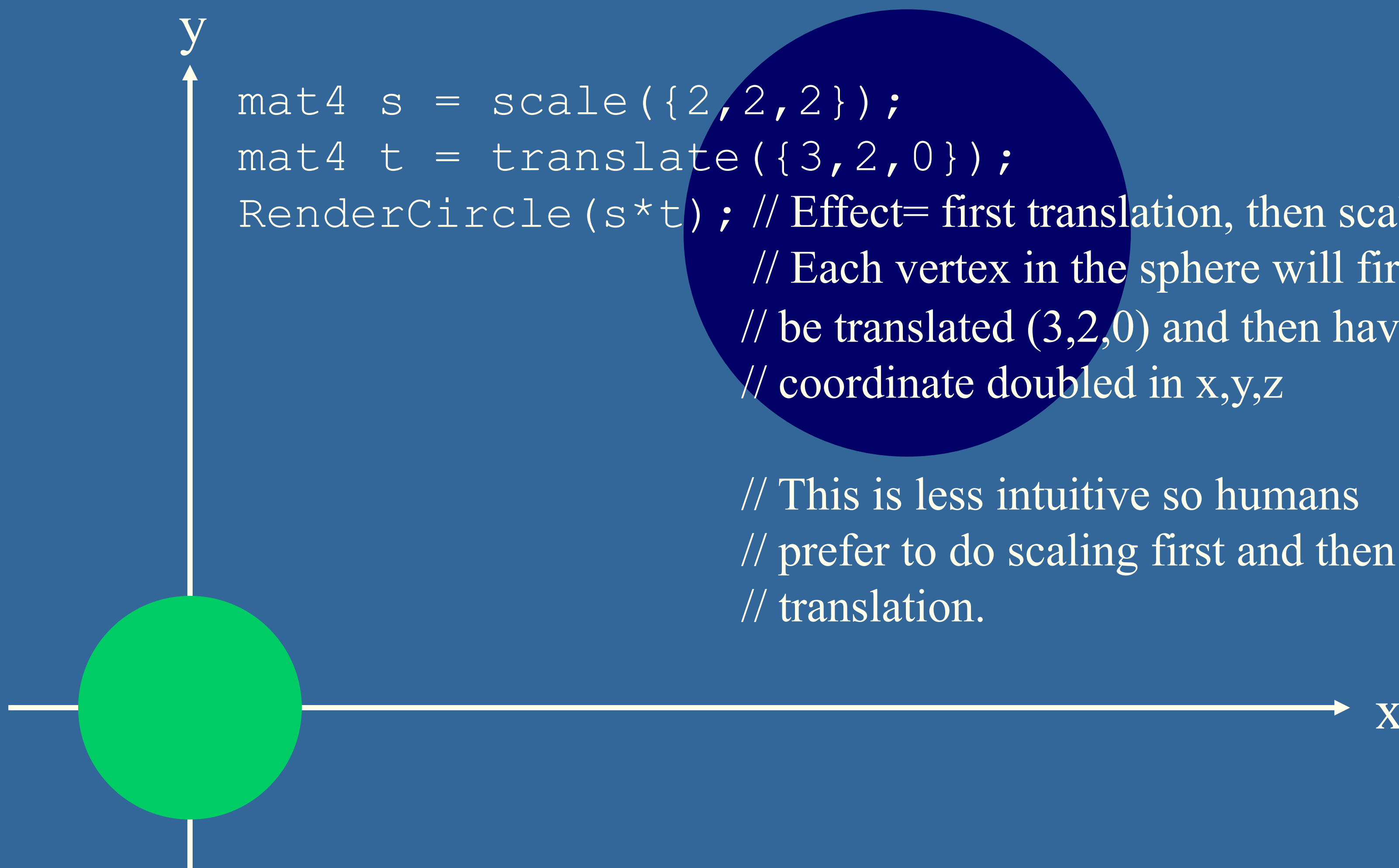
```
mat4 s = scale({2,2,2});
mat4 t = translate({3,2,0});
RenderCircle(t*s); // Effect= first scaling, then translation
```




```
mat4 m = translate({8,0,0});
RenderCircle(m);
```

Cont'd from previous slide
A simple 2D example

- A circle in **model space**




```
mat4 s = scale({2,2,2});
mat4 t = translate({3,2,0});
RenderCircle(s*t); // Effect= first translation, then scaling.
                   // Each vertex in the sphere will first
                   // be translated (3,2,0) and then have its
                   // coordinate doubled in x,y,z

                   // This is less intuitive so humans
                   // prefer to do scaling first and then
                   // translation.
```

Example of a simple GfxObject class

```
class GfxObject {
public:
   load("filename"); // Creates m_shaderProgram + m_vertexArrayObject
   render(mat4 projectionMatrix, mat4 viewMatrix)
   {
         …
         mat4 modelViewProjectionMatrix = projectionMatrix * viewMatrix *
                                          m_modelMatrix;
         int loc = glGetUniformLocation(shaderProgram, "modelViewProjectionMatrix");
         glUniformMatrix4fv(loc, 1, false, &modelViewProjectionMatrix[0].x);

         glEnableVertexAttribArray(0);
         glEnableVertexAttribArray(1);
         glUseProgram(m_shaderProgram);
         glBindVertexArray(m_vertexArrayObject);
         glDrawArrays( GL_TRIANGLES, 0,
   };
private:
   mat4   m_modelMatrix;
   uint   numVertices;
   Gluint m_shaderProgram;
   GLuint m_vertexArrayObject;
};
```

VERTEX SHADER

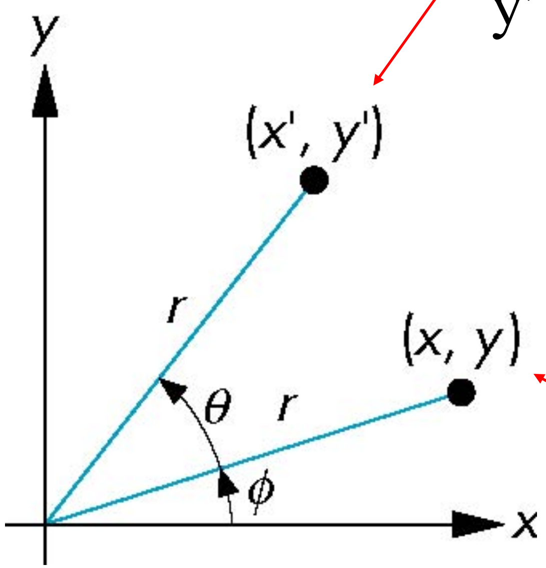
```
#version 420
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 color;

out vec4 outColor;
uniform mat4 modelViewProjectionMatrix;

void main()
{
   gl_Position = modelViewProjectionMatrix *
                 vec4(position, 1.0);
   outColor = vec4(color, 1.0);
}
```

Rotation (2D)

Consider rotation about the origin by θ degrees

– radius stays the same, angle increases by θ

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



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

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

Answer: $\boxed{\begin{array}{l} x' = x \cos\theta - y \sin\theta \\ y' = x \sin\theta + y \cos\theta \end{array}}$

Derivation of rotation matrix in 2D

$$\mathbf{n} = \mathbf{R}_z\mathbf{p}?$$

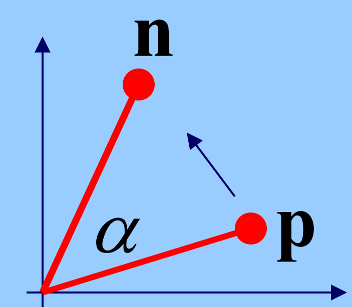
$$\mathbf{p} = re^{i\varphi} = r(\cos\varphi + i\sin\varphi)$$ [rotation is mult by $e^{i\alpha}$]

$$\mathbf{n} = e^{i\alpha}\mathbf{p} = re^{i\alpha}e^{i\varphi} =$$

$$= r[(\cos\alpha + i\sin\alpha)(\cos\varphi + i\sin\varphi)] =$$

$$= r(\cos\alpha\cos\varphi - \sin\alpha\sin\varphi) +$$

$$ir(\cos\alpha\sin\varphi + \sin\alpha\cos\varphi)$$



In vector form:

$$\mathbf{p} = (p_x, p_y)^T = (r\cos\varphi, r\sin\varphi)^T$$

$$\mathbf{n} = (n_x, n_y)^T = (r(\cos\alpha\cos\varphi - \sin\alpha\sin\varphi),$$

$$r(\cos\alpha\sin\varphi + \sin\alpha\cos\varphi))^T$$

Derivation 2D rotation, cont'd

In vector form:

$$\mathbf{p} = (p_x, p_y)^T = (\boxed{r\cos\phi}, \boxed{r\sin\phi})^T$$

$$\mathbf{n} = (n_x, n_y)^T = (\boxed{r}(\boxed{\cos\alpha}\boxed{\cos\phi} - \boxed{\sin\alpha}\boxed{\sin\phi}),$$

$$r(\boxed{\sin\alpha}\cos\phi + \boxed{\cos\alpha}\sin\phi))^T$$

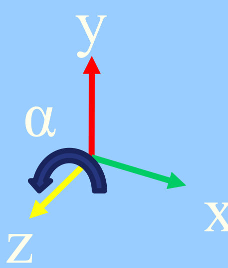
$$\mathbf{n} = \mathbf{R}_z\mathbf{p} \quad \text{what is } \mathbf{R}_z?$$

$$\begin{pmatrix} n_x \\ n_y \end{pmatrix} = \underbrace{\begin{pmatrix} \boxed{\cos\alpha} & -\boxed{\sin\alpha} \\ \boxed{\sin\alpha} & \boxed{\cos\alpha} \end{pmatrix}}_{\mathbf{R}_Z} \begin{pmatrix} \boxed{p_x} \\ \boxed{p_y} \end{pmatrix}$$

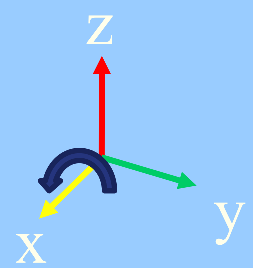
Rotations in 3D

- Same as in 2D for Z-rotations, but with a 3x3 matrix

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \Rightarrow \mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

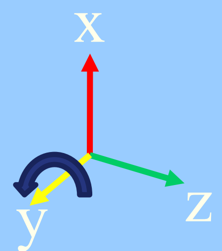


- For X



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

- For Y



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

Translations must be simple?

Translation | Rotation

$$\begin{pmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{pmatrix}\mathbf{p} = \mathbf{p} + \mathbf{t} \qquad \mathbf{n} = \mathbf{R}\mathbf{p}$$

- Rotation is matrix mult, translation is add
- Would be nice if we could only use matrix multiplications…
- Turn to **homogeneous coordinates**
- Add a new component to each vector

Homogeneous notation

- A point: $\mathbf{p} = \begin{pmatrix} p_x & p_y & p_z & 1 \end{pmatrix}^T$
- Translation becomes:

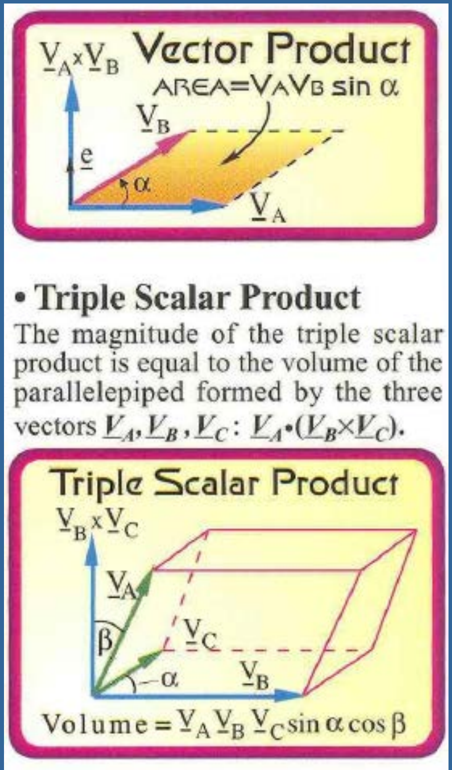
$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{T}(\mathbf{t})} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

- A vector (direction): $\mathbf{d} = \begin{pmatrix} d_x & d_y & d_z & 0 \end{pmatrix}^T$
- Translation of vector: $\mathbf{Td} = \mathbf{d}$
- Also allows for projections (later)

Rotations in 4x4 form

- Just add a row at the bottom, and a column at the right:

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



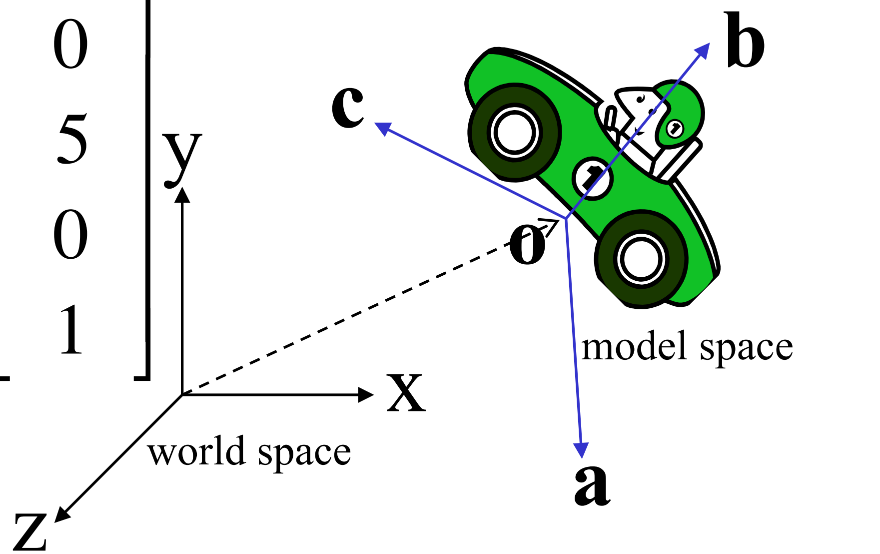
- Similarly for X and Y
- Determinant = volume change when the transform is applied to a unit cube
  - det(R) = 1 for all rot. matrices (=tripple scal. prod for 3x3 mtx)
- Trace(R) = 1+2cos(alpha) (for 3x3 rot-matrices)

Change of Frames

- How to get the $M_{\text{model-to-world}}$ matrix:

P= (0,5,0,1)

$$M_{\text{model-to-world}} = \begin{bmatrix} a_x & b_x & c_x & o_x \\ a_y & b_y & c_y & o_y \\ a_z & b_z & c_z & o_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 5 \\ 0 \\ 1 \end{bmatrix}$$



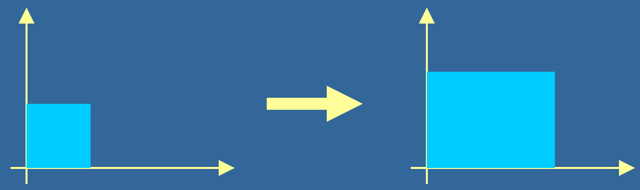
(Both coordinate systems are right-handed)

The basis vectors **a**,**b**,**c** are expressed in the world coordinate system

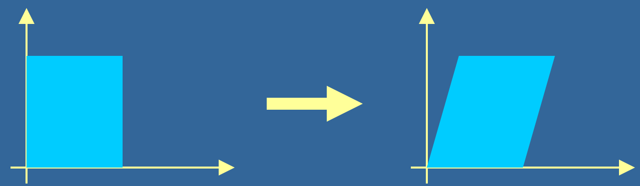
E.g.: $\mathbf{p}_{\text{world}} = M_{m\rightarrow w}\,\mathbf{p}_{\text{model}} = M_{m\rightarrow w}\,(0,5,0,1)^T = 5\,\mathbf{b} + \mathbf{o}$

More basic transforms

- Scaling



- Shear

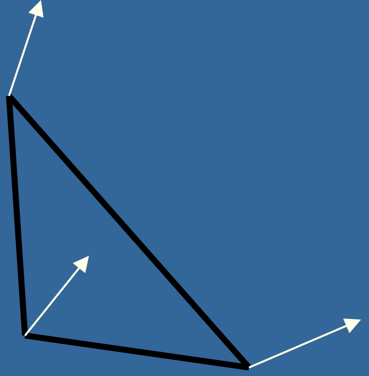


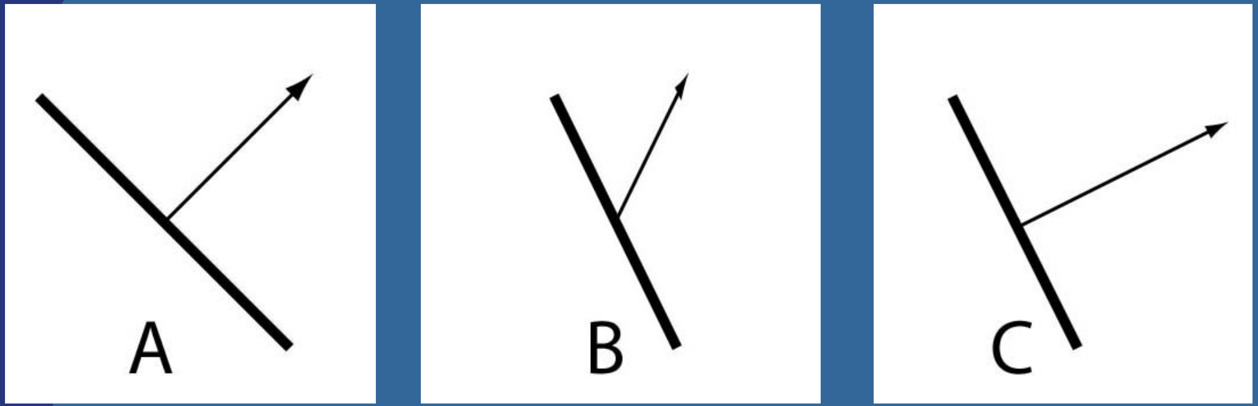
- Rigid-body: rotation and/or (then) translation

$$\mathbf{X} = \mathbf{TR}$$

- Concatenation of matrices
  - Not commutative, i.e., $\mathbf{RT} \neq \mathbf{TR}$
  - In $\mathbf{X} = \mathbf{TR}$, the rotation is done first
- Inverses and rotation about arbitrary axis:
  - Rigid body: $X^{-1} = X^T$ (for 3x3 matrices)

Normal transforms
Not so normal…



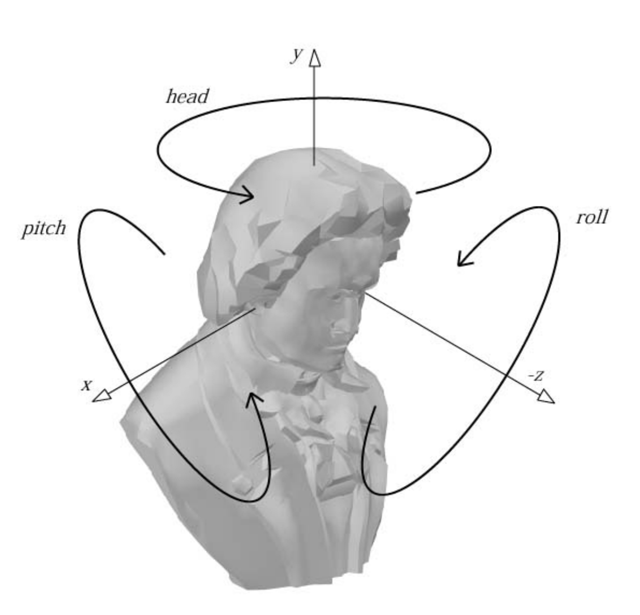


- Cannot use same matrix to transform normals

$$\text{Use}: \mathbf{N} = \left(\mathbf{M}^{-1}\right)^T \quad \text{instead of } \mathbf{M}$$

- **M** works for rotations and translations, though

The Euler Transform



- Assume the camera or object looks down the negative z-axis, with up in the y-direction, x to the right
- *h*=head
- *p*=pitch
- *r*=roll
- Optional
  - You may read about Gimbal lock in book, p: 67
  - See also
    - http://mathworld.wolfram.com/EulerAngles.html

Using Euler transforms

Head:

- Rotate around y-axis
- Recompute x- and z-axes
  - By rotating them as vectors

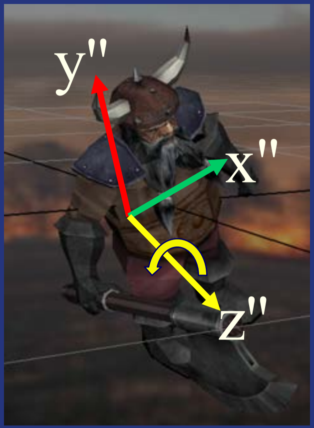
Pitch:

- Rotate around x'-axis
- Recompute y- and z'-axes

Roll:

- Rotate around z''-axis

How do we rotate vectors (axes)
and points around an **arbitrary** axis?



Quaternions

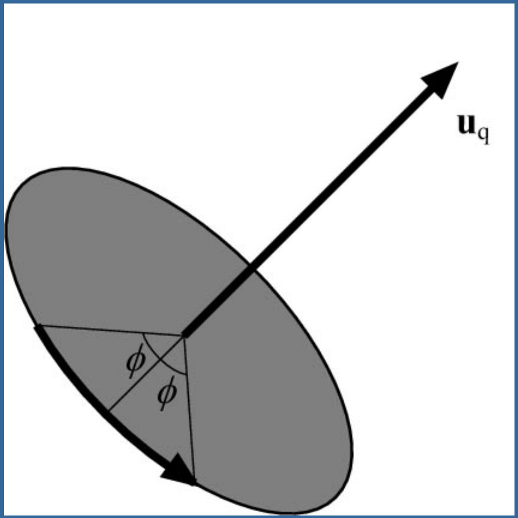
$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = (q_x, q_y, q_z, q_w)$$
$$= iq_x + jq_y + kq_z + q_w$$

- Extension of imaginary numbers
- Compact+fast representation of rotations
- Focus on unit quaternions:
  - Norm (or length): $n(\hat{\mathbf{q}}) = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2} = 1$
- A unit quaternion can be written as:

$$\hat{\mathbf{q}} = (\sin\phi\,\mathbf{u}_q, \cos\phi) \quad \text{where } \|\mathbf{u}_q\| = 1$$

Unit quaternions are perfect for rotations!

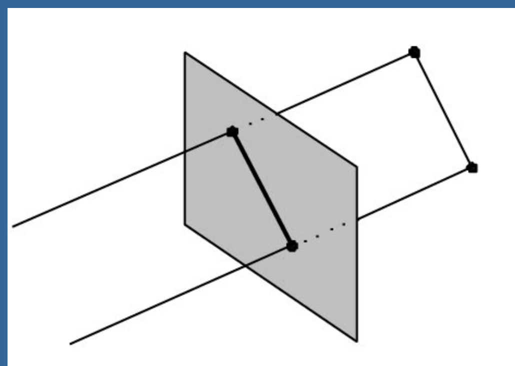
$$\hat{\mathbf{q}} = (\sin\phi\,\mathbf{u}_q, \cos\phi)$$

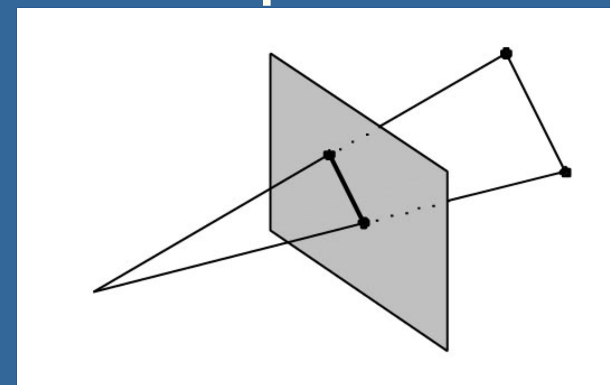


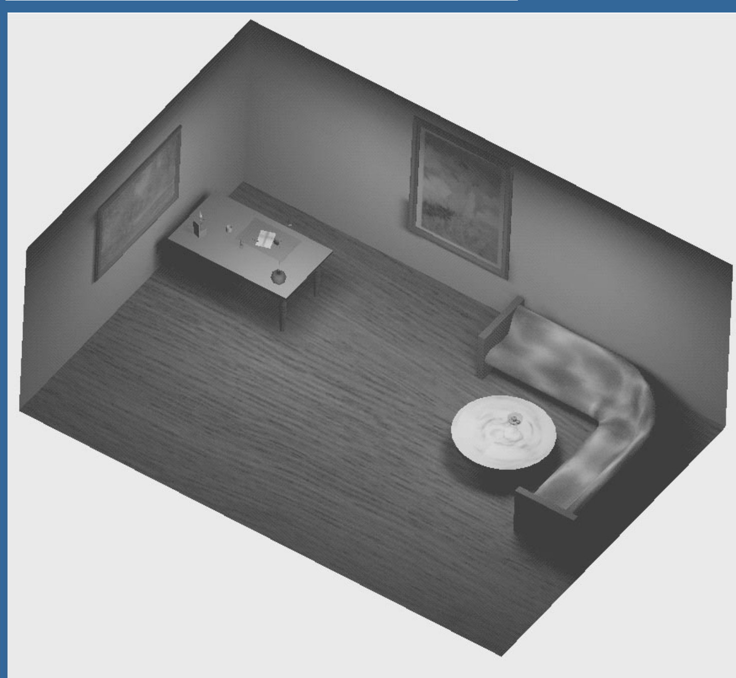
- Compact (4 components)
- Can show that $\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1}$
- …represents a rotation of 2ϕ radians around $\mathbf{u}_q$ of $\mathbf{p}$
- That is: a unit quaternion represents a rotation as a **rotation axis** and an **angle**
- `rotate(ux,uy,uz,angle);`
  - `See p:76 how to convert q to matrix.`
- Interpolation from one quaternion to another is much simpler, and gives optimal results

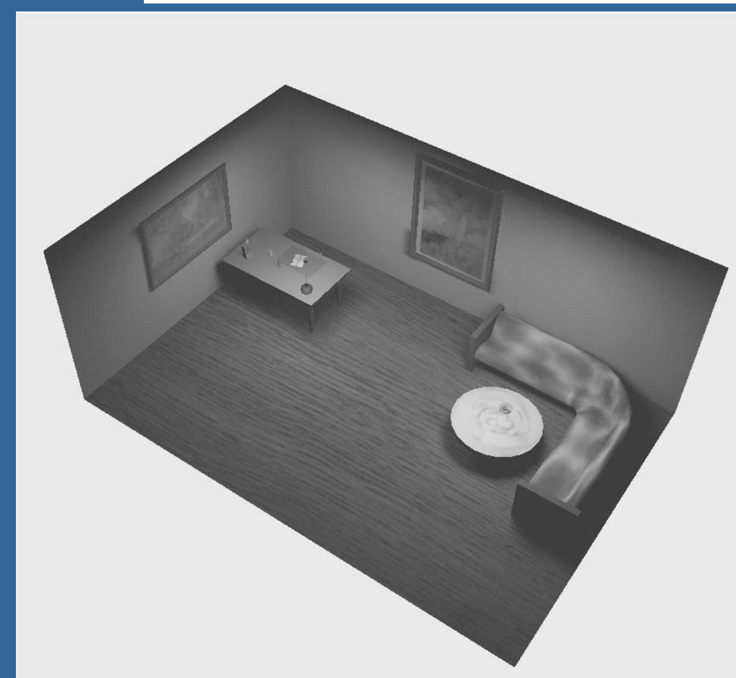
Projections

- Orthogonal (parallel) and Perspective

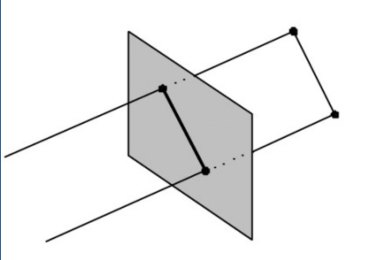






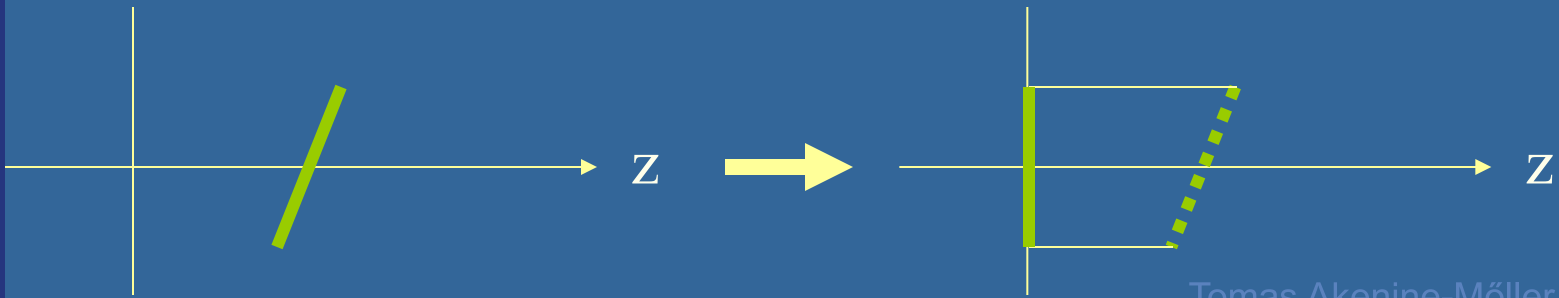


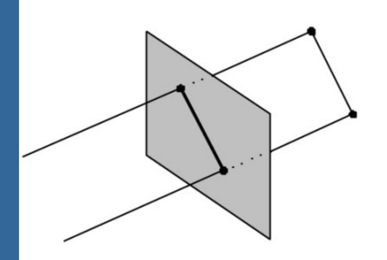
Orthogonal projection



- Simple, just skip one coordinate
  - Say, we're looking along the z-axis
  - Then drop z, and render

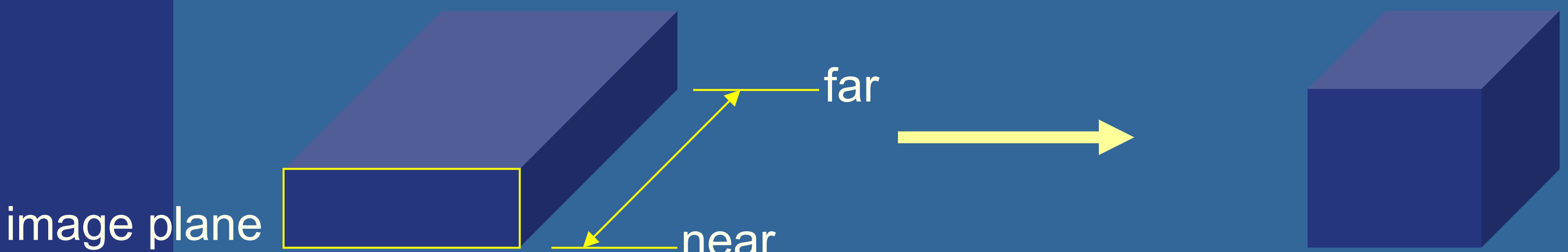
$$\mathbf{M}_{ortho} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \Rightarrow \quad \mathbf{M}_{ortho} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ 0 \\ 1 \end{pmatrix}$$



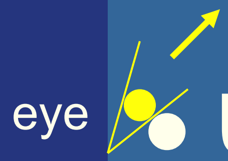


Orthogonal projection

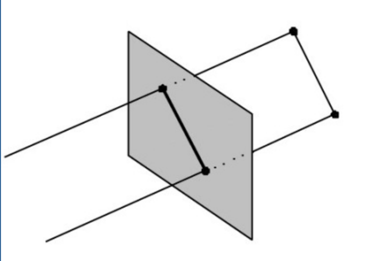
- Not invertible! (determinant is zero)
  - i.e., depth information is lost
- For Z-buffering
  - It is not sufficient to project to a plane
  - Rather, we need to ”project” to a box



Unit cube: [-1,-1,-1] to [1,1,1]

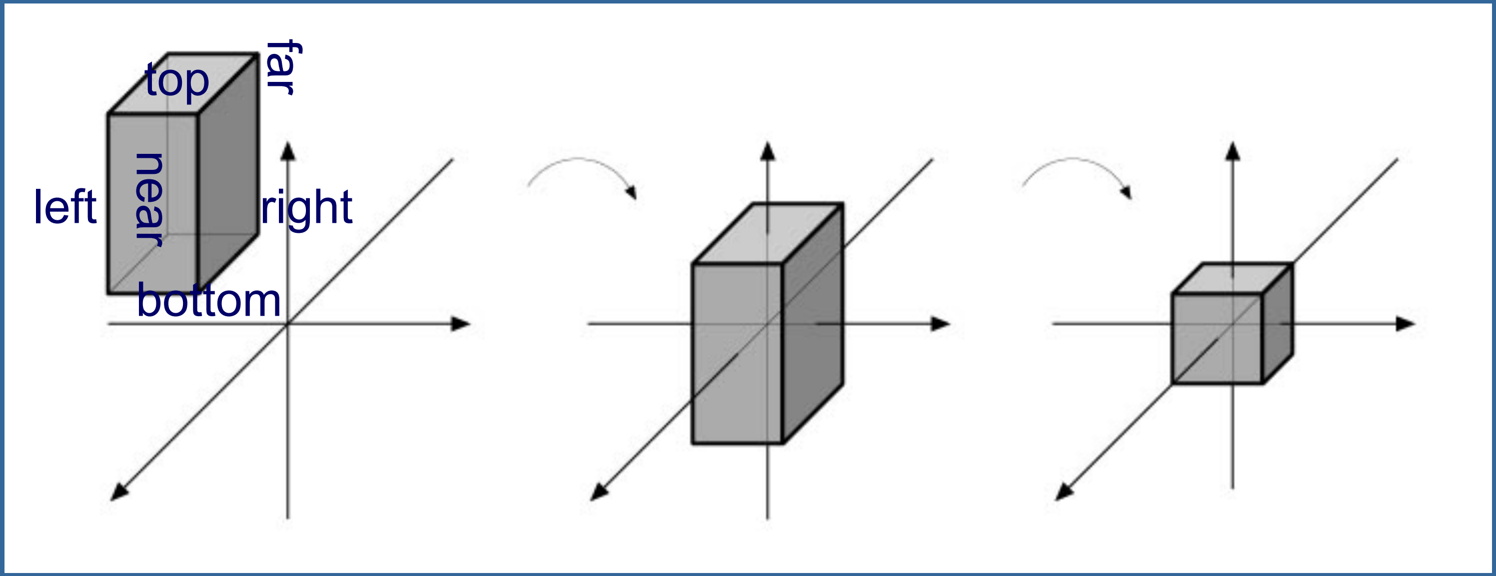


- Unit cube is also used for perspective proj.
- Simplifies clipping



Orthogonal projection

- The ”unitcube projection” is invertible
- Simple to derive
  - Just a translation and scale



What about those homogenenous coordinates?

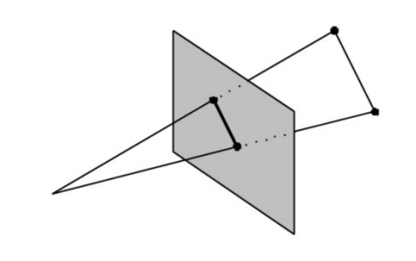
$$\mathbf{p} = \begin{pmatrix} p_x & p_y & p_z & p_w \end{pmatrix}^T$$

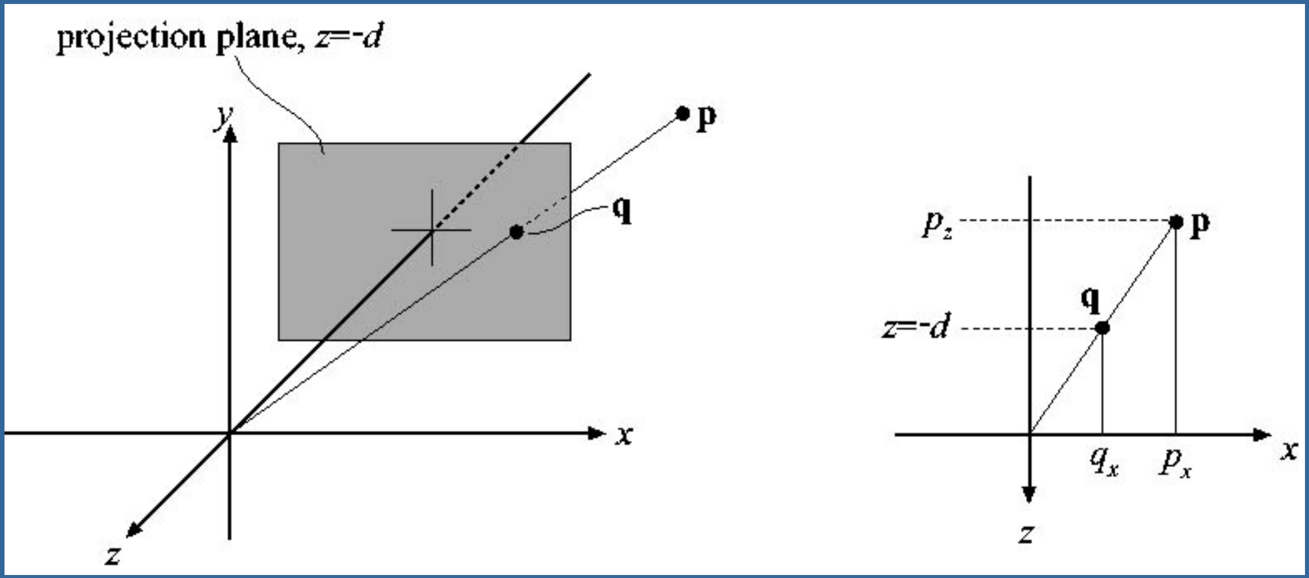
- $p_w=0$ for vectors, and $p_w=1$ for points
- What if p_w is **not** 1 or 0?
- Solution is to divide all components by p_w

$$\mathbf{p} = \begin{pmatrix} p_x / p_w & p_y / p_w & p_z / p_w & 1 \end{pmatrix}^T$$

- Gives a point again!
- Can be used for projections, as we will see

Perspective projection



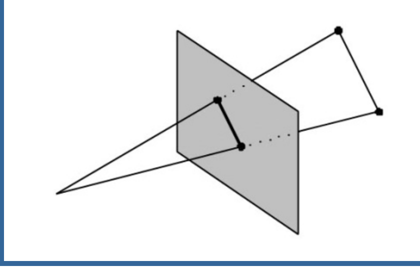


$$\frac{q_x}{p_x} = \frac{-d}{p_z} \Rightarrow q_x = -d\,\frac{p_x}{p_z}$$

For y: $q_y = -d\,\frac{p_y}{p_z}$

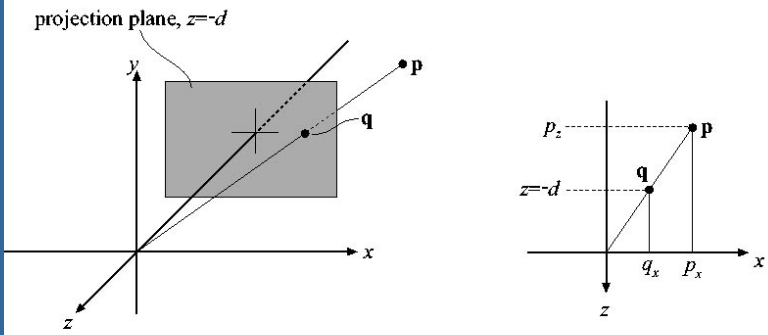
$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}$$

Perspective projection



$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}$$

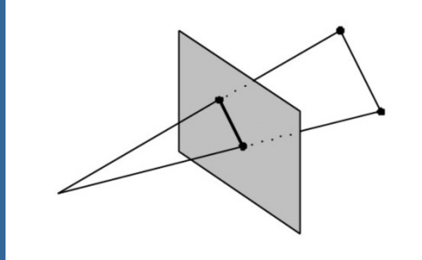
$$\mathbf{P}_p \mathbf{p} = \mathbf{q}$$



$$q_x = -d\frac{p_x}{p_z} \qquad q_y = -d\frac{p_y}{p_z} \qquad q_z = -d$$

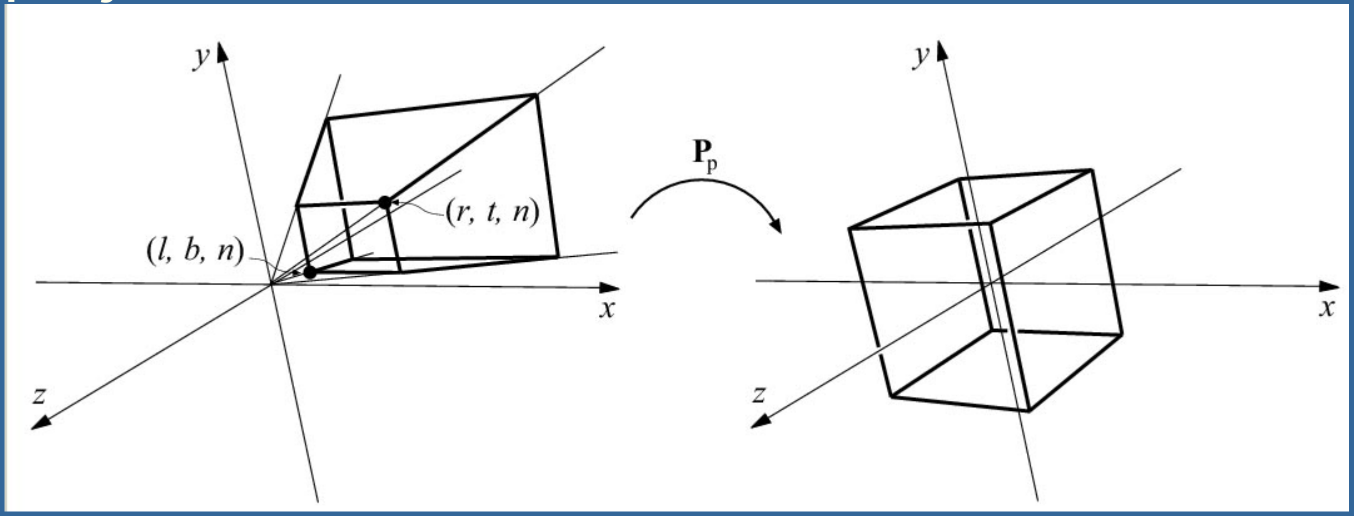
$$\mathbf{P}_p \mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}\begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \Rightarrow \mathbf{q} = \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -dp_z/p_z \\ 1 \end{pmatrix} = \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -d \\ 1 \end{pmatrix}$$

- The ”arrow” is the homogenization process



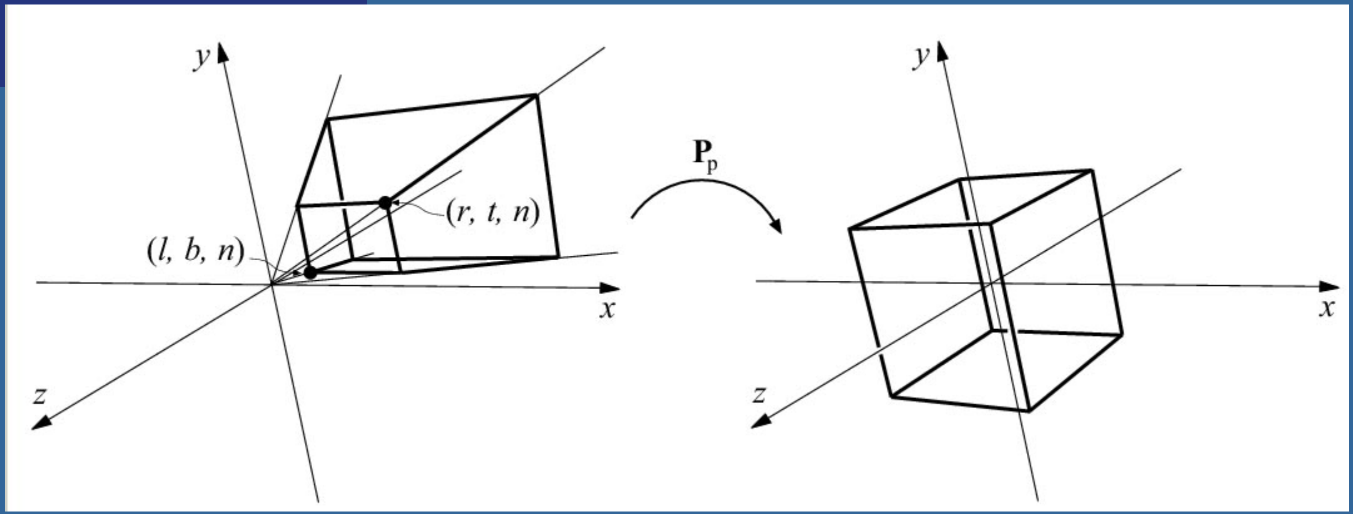
Perspective projection

- Again, the determinant is 0 (not invertible)
- To make the rest of the pipeline the same as for orhogonal projection:
  - project into unit-cube



- Not much different from $\mathbf{P}_p$
- Do not collapse z-coord to a plane

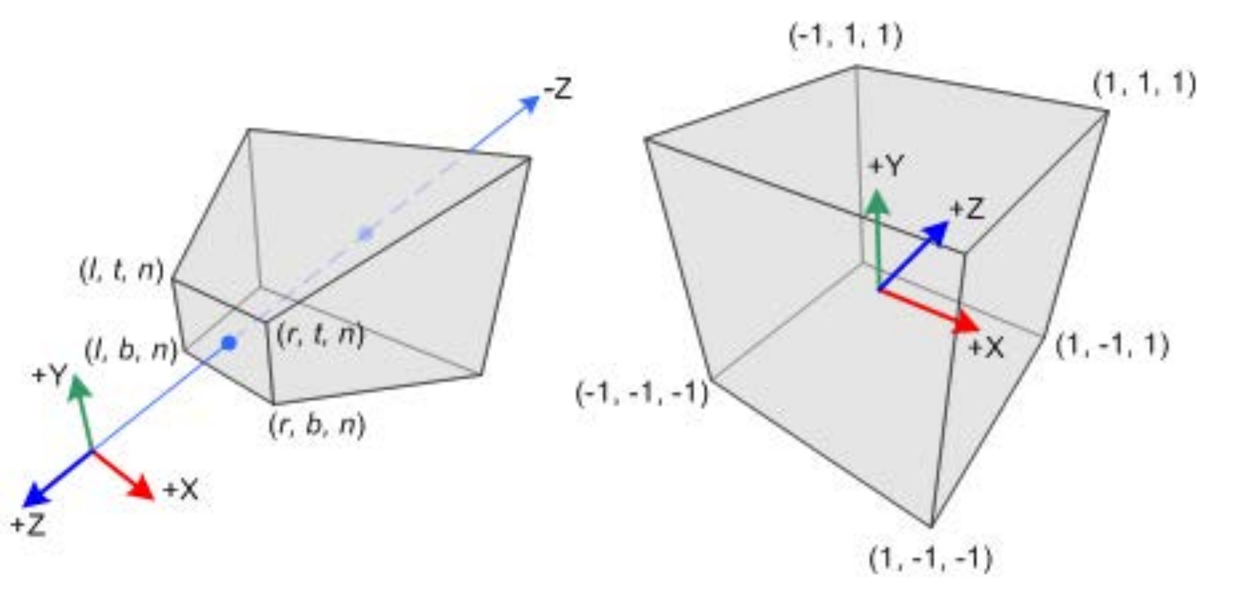
Understanding the projection matrix



$$\mathbf{P}_p\mathbf{p} = \begin{pmatrix} s_x & 0 & a & 0 \\ 0 & s_y & b & 0 \\ 0 & 0 & s_z & c \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x p_x + a p_z \\ s_y p_y + b p_z \\ s_z p_z + c \\ -p_z/d \end{pmatrix} \Rightarrow \mathbf{q} = \begin{pmatrix} -d(s_x p_x / p_z + a) \\ -d(s_y p_y / p_z + b) \\ -d(s_z p_z + c)/p_z \\ 1 \end{pmatrix}$$

- s_x, s_y, s_z –Scaling
- a, b – Due to homogenization, this controls asymmetry of the frustum
- c – Keep z-info
- -1/d – Perspective division based on p_z

OpenGL projection matrix



$$P_{OpenGL} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

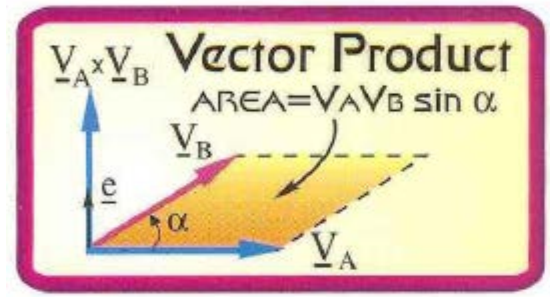
```
mat4 projectionMtx = perspective(fov, width / height, near, far);
```

Quick Repetition of Vector Algebra

Length of vector: $\|\mathbf{x}\| = \sqrt{\left(x^2 + y^2 + z^2\right)}$

$$\underline{V}_A \times \underline{V}_B = \begin{Bmatrix} \underline{i} & \underline{j} & \underline{k} \\ V_{AX} & V_{AY} & V_{AZ} \\ V_{BX} & V_{BY} & V_{BZ} \end{Bmatrix}$$

Normalizing a vector: $\hat{\mathbf{x}} = \dfrac{\mathbf{x}}{\sqrt{\left(x^2 + y^2 + z^2\right)}} = \dfrac{\mathbf{x}}{\|\mathbf{x}\|}$



Normal: $\mathbf{n} = (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)$

(usualy needs to be normalized as well)

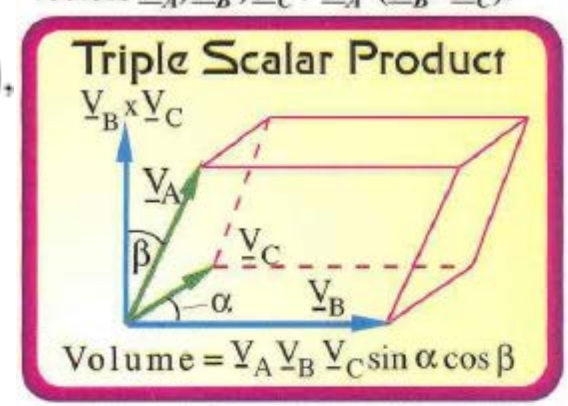
Cross Product:

- Perpendicular vector, Area
- sin α: $\sin\alpha = \dfrac{\mathbf{v}_a \times \mathbf{v}_b}{\|\mathbf{v}_a\|\|\mathbf{v}_b\|}\hat{\mathbf{e}}$, where $\hat{\mathbf{e}}$ is perp. to $\mathbf{v}_a$ and $\mathbf{v}_b$.

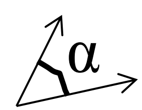
$$\mathbf{u} \times \mathbf{v} = \hat{\mathbf{x}}\left(u_y v_z - u_z v_y\right) + \hat{\mathbf{y}}\left(u_z v_x - u_x v_z\right) + \hat{\mathbf{z}}\left(u_x v_y - u_y v_x\right),$$

• Triple Scalar Product

The magnitude of the triple scalar product is equal to the volume of the parallelepiped formed by the three vectors $\underline{V}_A, \underline{V}_B, \underline{V}_C$: $\underline{V}_A \bullet (\underline{V}_B \times \underline{V}_C)$.



Dot product: $\cos\alpha = \dfrac{\mathbf{v}_a \bullet \mathbf{v}_b}{\|\mathbf{v}_a\|\|\mathbf{v}_b\|}$



$$\mathbf{a} \bullet \mathbf{b} = (a_x b_x + a_y b_y + a_z b_z)$$

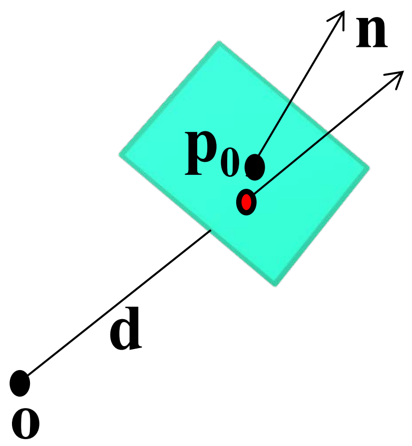
Ray/Plane Intersections

- Ray: $r(t)=\mathbf{o}+t\mathbf{d}$
- Plane: $\mathbf{n}\bullet\mathbf{x} + d = 0$; $(d=-\mathbf{n}\bullet\mathbf{p}_0)$
- Set $\mathbf{x}=r(t)$:

  $\mathbf{n}\bullet(\mathbf{o}+t\mathbf{d}) + d = 0$

  $\mathbf{n}\bullet\mathbf{o}+t(\mathbf{n}\bullet\mathbf{d}) + d = 0$

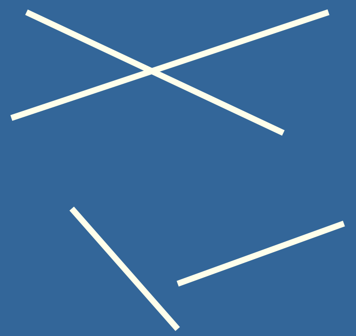
  $t = (-d - \mathbf{n}\bullet\mathbf{o}) / (\mathbf{n}\bullet\mathbf{d})$




```
Vec3f rayPlaneIntersect(vec3f o,dir, n, d)
{
        float t=(-d-n.dot(o)) / (n.dot(dir));
        return o + dir*t;
}
```

See book, p: 780

Line/Line intersection in 2D



- $r_1(s) = \mathbf{o}_1 + s\mathbf{d}_1$
- $r_2(t) = \mathbf{o}_2 + t\mathbf{d}_2$

- $r_1(s) = r_2(t)$ (1)
- $\mathbf{o}_1 + s\mathbf{d}_1 = \mathbf{o}_2 + t\mathbf{d}_2$ (2)

noting that $\mathbf{d} \cdot \mathbf{d}^{\perp} = 0$, $[\mathbf{d} = (a,b) \rightarrow \mathbf{d}^{\perp} = (b,-a)]$

$$s\mathbf{d}_1 \cdot \mathbf{d}_2^{\perp} = (\mathbf{o}_2 - \mathbf{o}_1) \cdot \mathbf{d}_2^{\perp}$$

$$t\mathbf{d}_2 \cdot \mathbf{d}_1^{\perp} = (\mathbf{o}_1 - \mathbf{o}_2) \cdot \mathbf{d}_1^{\perp}$$

$$s = \frac{(\mathbf{o}_2 - \mathbf{o}_1) \bullet \mathbf{d}_2^{\perp}}{(\mathbf{d}_1 \bullet \mathbf{d}_2^{\perp})}$$

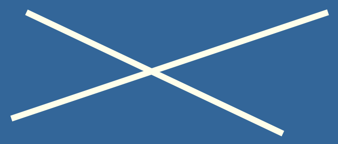
$$t = \frac{(\mathbf{o}_1 - \mathbf{o}_2) \bullet \mathbf{d}_1^{\perp}}{(\mathbf{d}_2 \bullet \mathbf{d}_1^{\perp})}$$

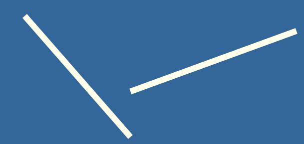
See book, p: 781-782

Line/Line intersection in 3D

- $\mathbf{r}_1(s) = \mathbf{o}_1 + s\mathbf{d}_1$
- $\mathbf{r}_2(t) = \mathbf{o}_2 + t\mathbf{d}_2$

s, t correspond to closest points





- $\mathbf{r}_1(s) = \mathbf{r}_2(t)$ (1)
- $\mathbf{o}_1 + s\mathbf{d}_1 = \mathbf{o}_2 + t\mathbf{d}_2$ (2)

noting that d x d=0

$\|(\mathbf{d}_1 \times \mathbf{d}_2)\|^2 = 0$ means parallel lines

$s\mathbf{d}_1 \times \mathbf{d}_2 = (\mathbf{o}_2 - \mathbf{o}_1) \times \mathbf{d}_2$ (i.e., cross mult. both sides with $\mathbf{d}_2$ to drop t)

$t\mathbf{d}_2 \times \mathbf{d}_1 = (\mathbf{o}_1 - \mathbf{o}_2) \times \mathbf{d}_1$ (i.e., cross mult. both sides with $\mathbf{d}_1$ to drop s)

=>

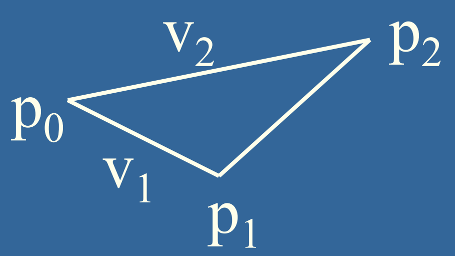
$s\,(\mathbf{d}_1 \times \mathbf{d}_2) \bullet (\mathbf{d}_1 \times \mathbf{d}_2) = ((\mathbf{o}_2 - \mathbf{o}_1) \times \mathbf{d}_2) \bullet (\mathbf{d}_1 \times \mathbf{d}_2)$

$t\,(\mathbf{d}_2 \times \mathbf{d}_1) \bullet (\mathbf{d}_2 \times \mathbf{d}_1) = ((\mathbf{o}_1 - \mathbf{o}_2) \times \mathbf{d}_1) \bullet (\mathbf{d}_2 \times \mathbf{d}_1)$

$$s = \frac{\det(\mathbf{o}_2 - \mathbf{o}_1, \mathbf{d}_2, \mathbf{d}_1 \times \mathbf{d}_2)}{\|(\mathbf{d}_1 \times \mathbf{d}_2)\|^2}$$

$$t = \frac{\det(\mathbf{o}_2 - \mathbf{o}_1, \mathbf{d}_1, \mathbf{d}_1 \times \mathbf{d}_2)}{\|(\mathbf{d}_1 \times \mathbf{d}_2)\|^2}$$

Area and Perimeter



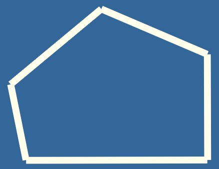
For polygon $p_0, p_1 \ldots p_n$

Perimeter = omkrets = sum of length of each edge in 2D and 3D:

$$O = \sum_{i=0}^{n-1} \left\| p_{i+1} - p_i \right\| = \sum_{i=0}^{n-1} \sqrt{\left(x_{i+1} - x_i\right)^2 + \left(y_{i+1} - y_i\right)^2 + \left(z_{i+1} - z_i\right)^2}$$

Area in 2D:

$$A = \frac{1}{2} \left| \sum_{i=1}^{n-1} \left\langle x_i y_{i+1} - x_{i+1} y_i \right\rangle \right|$$



We can understand the formula from using Greens theorem: integrating over border to get area

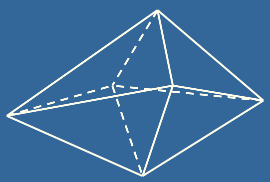
Choose arbitrary point to integrate from, e.g. Origin (0,0,0)

$$A_{triangle} = \frac{1}{2}\left(v_1 \times v_2\right)$$

Works for non-convex polygons as well

Volume in 3D

The same trick for computing area in 2D can be used to easily compute the volume in 3D for triangulated objects



Again, choose arbitrary point-of-integration, e.g. Origin (0,0,0)

With respect to point-of-integration

- For all backfacing triangles, add volume
- For all frontfacing triangles, subtract volume

Works for non-convex objects as well

$$V_{tetrahedron} = \frac{1}{3!}\left|\mathbf{a}\bullet(\mathbf{b}\times\mathbf{c})\right| = \frac{1}{3!}\left|\det(\mathbf{a},\mathbf{b},\mathbf{c})\right|$$

where
a = p_1 – origin
b = p_2 – origin
c = p_3 – origin

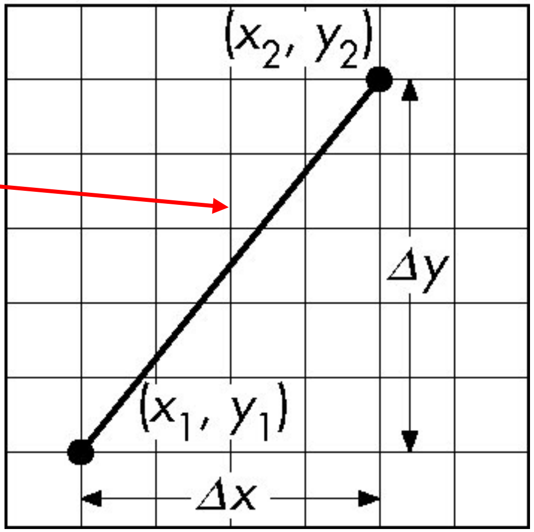
$$V_{object} = \frac{1}{3!}\sum_{i=1}^{n}\mathbf{a}\bullet(\mathbf{b}\times\mathbf{c})$$

The sign of the determinant will automatically handle positive and negative contribution

Scan Conversion of Line Segments

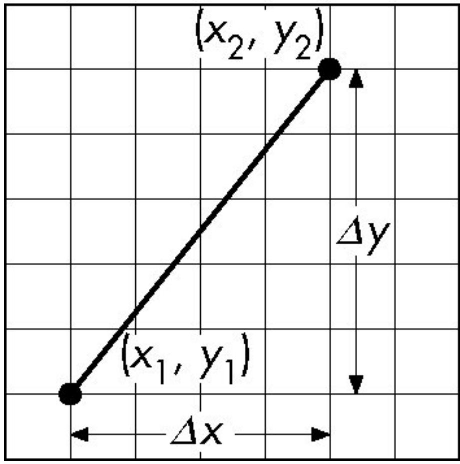
- Start with line segment in window coordinates with integer values for endpoints
- Assume implementation has a **`write_pixel`** function

y = kx + m



$$k = \frac{\Delta y}{\Delta x}$$

DDA Algorithm



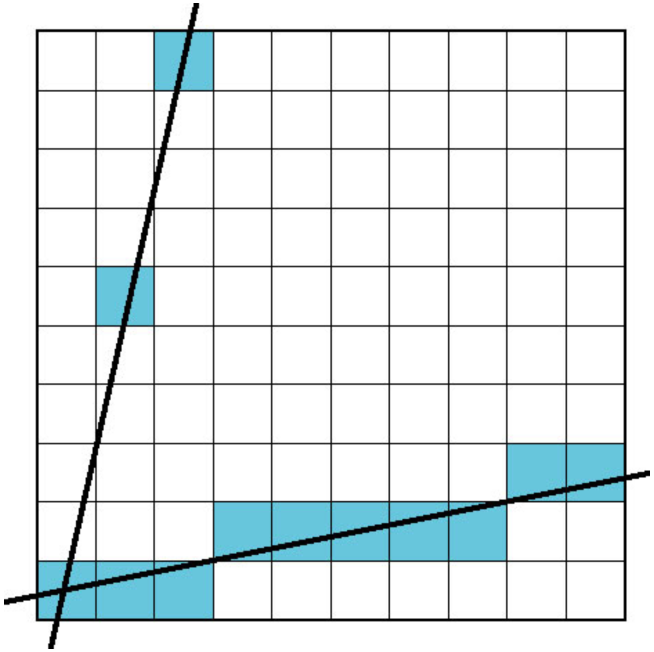
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line $y=kx+m$ satisfies differential equation

    $dy/dx = k = \Delta y/\Delta x = y_2\text{-}y_1/x_2\text{-}x_1$
- Along scan line $\Delta x = 1$

```
y=y1;
For(x=x1; x<=x2,ix++) {
  write_pixel(x, round(y), line_color)
  y+=k;
}
```

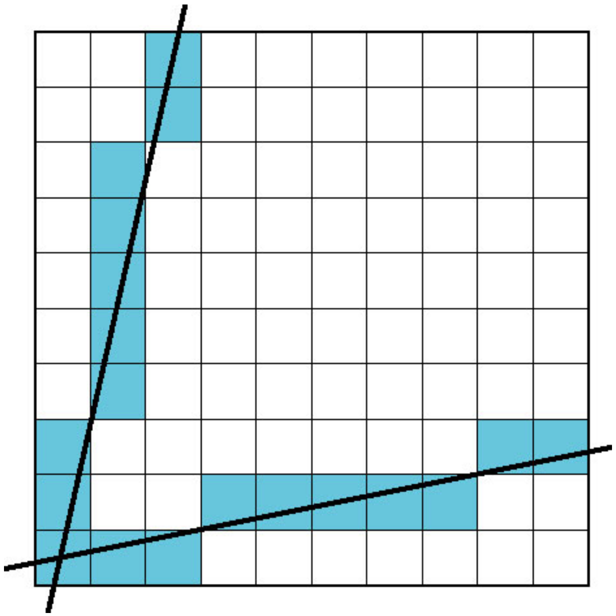
Problem

- DDA = for each x plot pixel at closest y
  - –Problems for steep lines



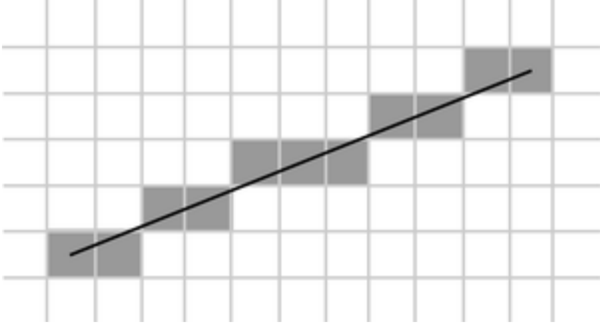
Using Symmetry

- Use for $1 \geq k \geq 0$
- For $k > 1$, swap role of x and y
  - For each y, plot closest x



- The problem with DDA is that it uses floats which was slow in the old days
- Bresenhams algorithm only uses integers

Bresenham's line drawing algorithm

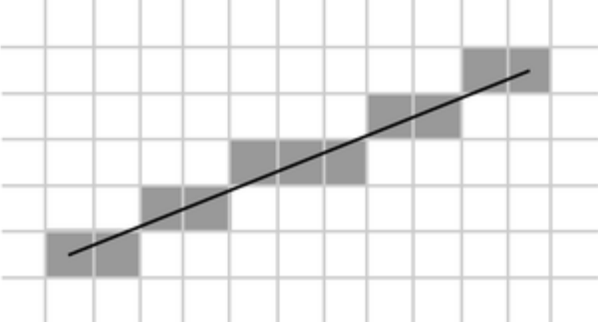


- The line is drawn between two points (x_0, y_0) and (x_1, y_1)
- Slope $k = \dfrac{(y_1 - y_0)}{(x_1 - x_0)}$ $(y = kx + m)$
- Each time we step 1 in x-direction, we should increment *y* with *k.* Otherwise the error in y increases with *k.*
- If the error surpasses 0.5, the line has become closer to the next *y*-value, so we add 1 to *y,* simultaneously decreasing the error by 1

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay / deltax
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if error ≥ 0.5
            y := y + 1
            error := error - 1.0
```

See also
http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

Bresenham's line drawing algorithm



- Now, convert algorithm to only using integer computations
- Trick: multiply the fractional number, *deltaerr*, by *deltax*
  - enables us to express *deltaerr* as an integer.
  - The comparison *if error>=0.5* is multiplied on both sides by *2*deltax*

Old float version:

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay / deltax
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if error ≥ 0.5
            y := y + 1
            error := error - 1.0
```

New integer version:

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay              Multiply by deltax
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if 2*error ≥ deltax              Multiply by 2 deltax
            y := y + 1
            error := error - deltax      Multiply by deltax
```

Complete Bresenham's line drawing algorithm

```
function line(x0, x1, y0, y1)
    boolean steep := abs(y1 - y0) > abs(x1 - x0)
    if steep then
        swap(x0, y0)
        swap(x1, y1)
    if x0 > x1 then
        swap(x0, x1)
        swap(y0, y1)
    int deltax := x1 - x0
    int deltay := abs(y1 - y0)
    int error := 0
    int ystep
    int y := y0
    if y0 < y1 then ystep := 1 else ystep := -1
    for x from x0 to x1
        if steep then plot(y,x) else plot(x,y)
        error := error + deltay
        if 2×error ≥ deltax
            y := y + ystep
            error := error - deltax
```

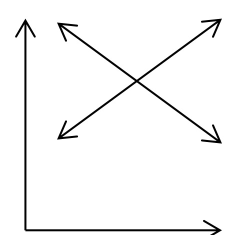
Swap loop axis

Swap start and end points

The first case is allowing us to draw lines that still slope downwards, but head in the opposite direction. I.e., swapping the initial points if x0 > x1.

To draw lines that go up, we check if y0 >= y1; if so, we step y by -1 instead of 1.

To be able to draw lines with a slope less than one, we take advantage of the fact that a steep line can be reflected across the line y=x to obtain a line with a small slope. The effect is to switch the x and y variables.



You need to know

- How to create a simple Scaling matrix, rotation matrix, translation matrix and orthogonal projection matrix
- Change of frames (creating model-to-view matrix)
- Understand how quaternions are used
- Understanding of Euler transforms
- DDA line drawing algorithm
- Understand what is good with Bresenhams line drawing algorithm, i.e., uses only integers.

The following slides are simply extra non-compulsory material that explains the content of the lecture in a different way.

Most of the following slides are from

Ed Angel

Professor of Computer Science, Electrical and Computer Engineering, and Media Arts

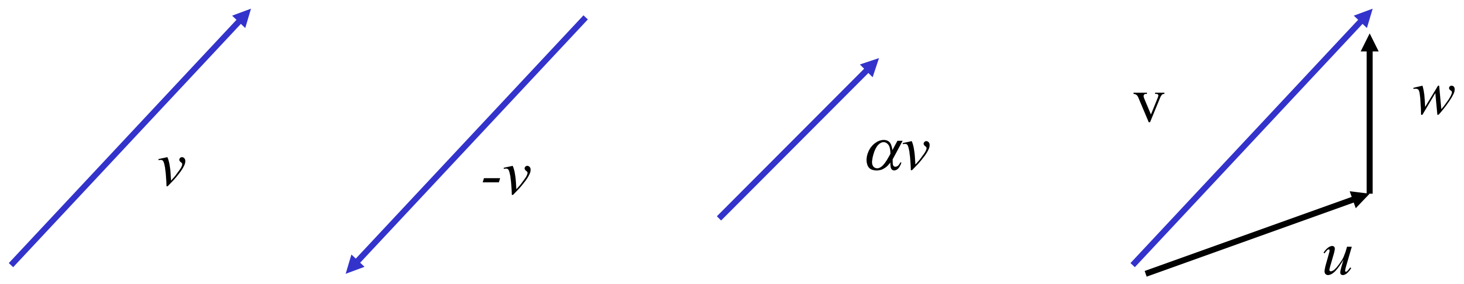
University of New Mexico

Scalars

- Need three basic elements in geometry
  - Scalars, Vectors, Points
- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutivity, inverses)
- Examples include the real and complex number systems under the ordinary rules with which we are familiar
- Scalars alone have no geometric properties

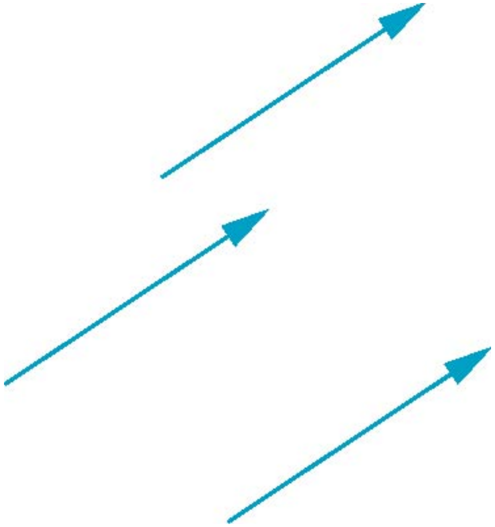
Vector Operations

- 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. 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



Vectors Lack Position

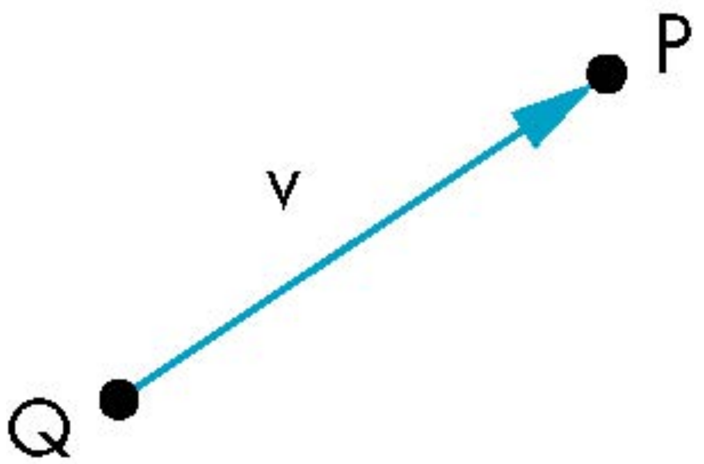
- These vectors are identical
  - Same length and magnitude



- Vectors insufficient for geometry
  - Need points

Points

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



v=P-Q

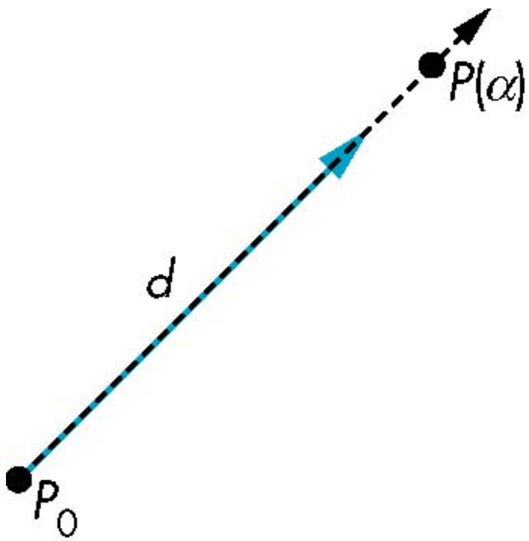
P=v+Q

Affine Spaces

- Point + a vector 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 **d**



Parametric Form

- This form is known as the parametric form of the line
  - –More robust and general than other forms
  - –Extends to curves and surfaces
- Two-dimensional forms
  - –Explicit: $y = kx + m$
  - –Implicit: $ax + by + c = 0$
  - –Parametric:

$$x(\alpha) = \alpha x_0 + (1-\alpha)x_1$$
$$y(\alpha) = \alpha y_0 + (1-\alpha)y_1$$

Rays and Line Segments

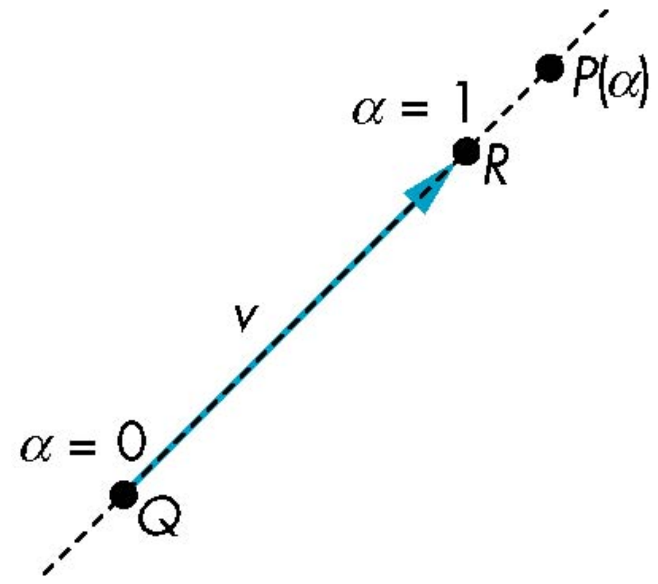
- If $\alpha >= 0$, then $P(\alpha)$ is the *ray* leaving P_0 in the direction **d**

If we use two points to define v, then

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

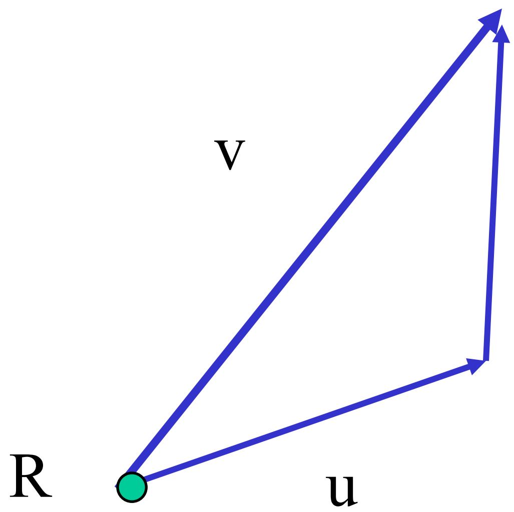
$= \alpha R + (1-\alpha)Q$

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

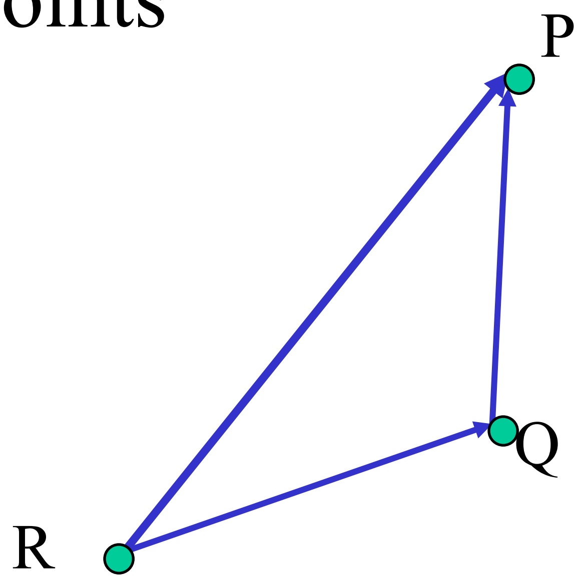


Planes

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

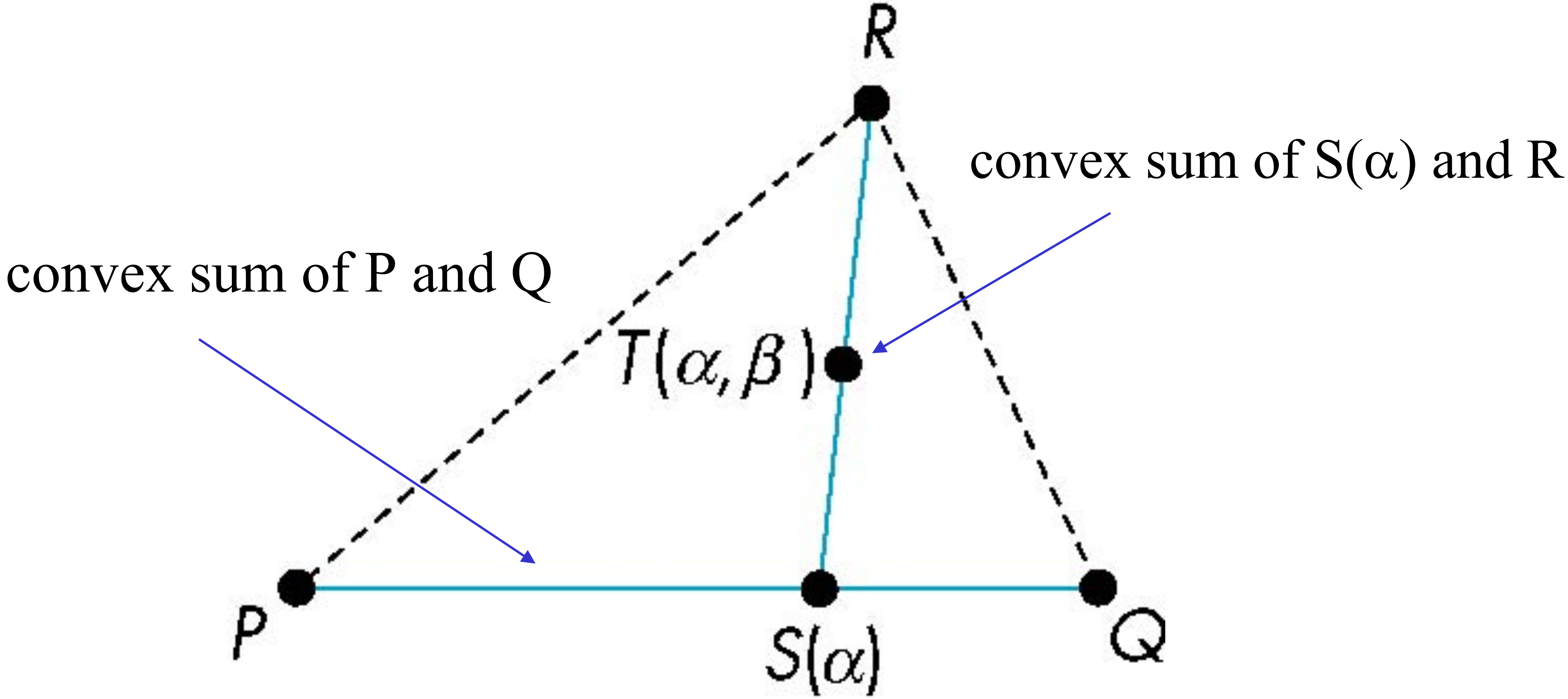


$P(\alpha,\beta)=R+\alpha u+\beta v$



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

Triangles

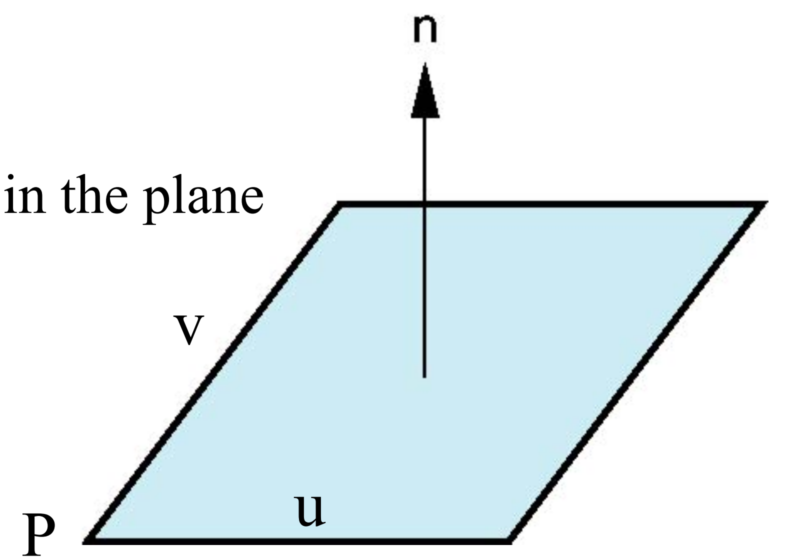


for 0<=α,β<=1, we get all points in triangle

Normals

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

  we know we can use the cross product to find
  - $n = u \times v$
- Plane equation:
  - $n \cdot \mathbf{x} - d = 0$,
  - where $d = -n \cdot p$ and p is any point in the plane

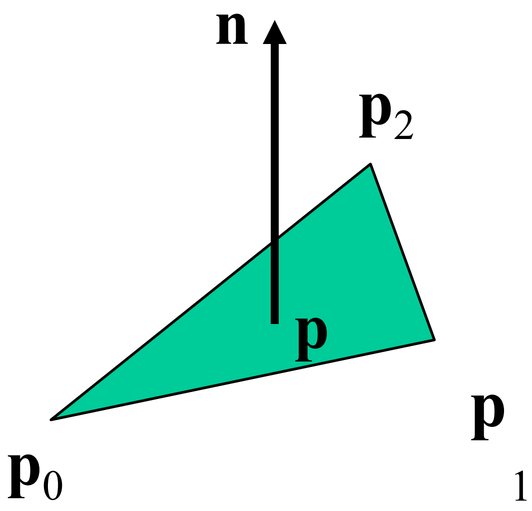


Normal for Triangle

plane $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$

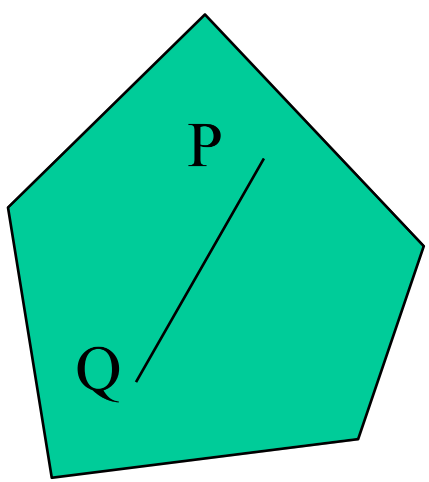
normalize $\mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$



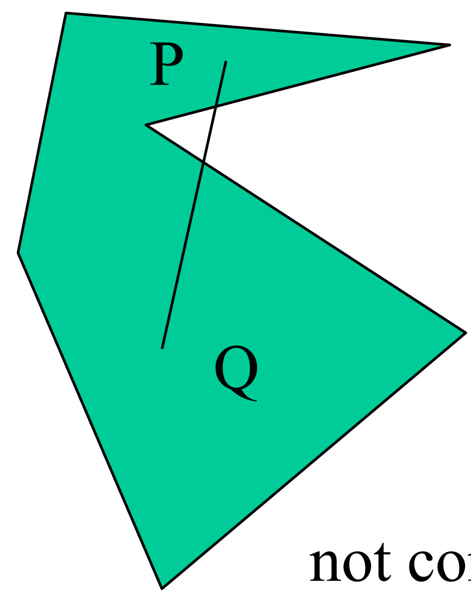
Note that right-hand rule determines outward face

Convexity

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



convex



not convex

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* of the points $P_1,P_2,\ldots\ldots P_n$

- If, in addition, $\alpha_i>=0$, we have the *convex hull* of $P_1,P_2,\ldots\ldots P_n$

Convex Hull

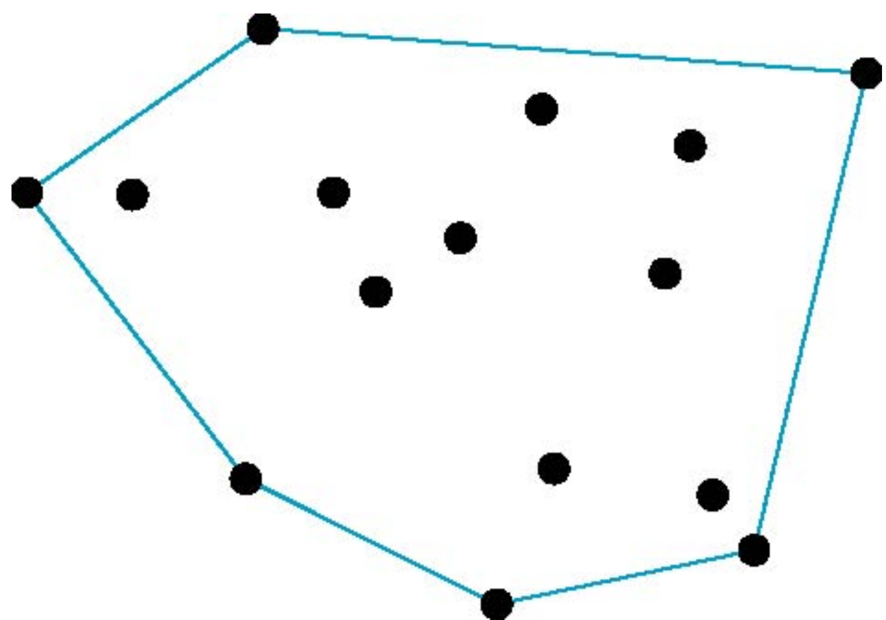
Consider the linear combination

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

- If $\alpha_1+\alpha_2+\ldots\ldots\alpha_n=1$
  - (in which case we have the *affine sum* of the points $P_1,P_2,\ldots\ldots P_n$)

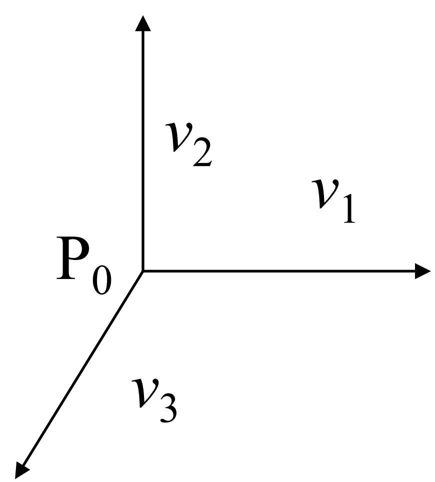
  and if $\alpha_i>=0$, we have the *convex hull* of $P_1,P_2,\ldots\ldots P_n$

- Smallest convex object containing $P_1,P_2,\ldots\ldots P_n$



Frames

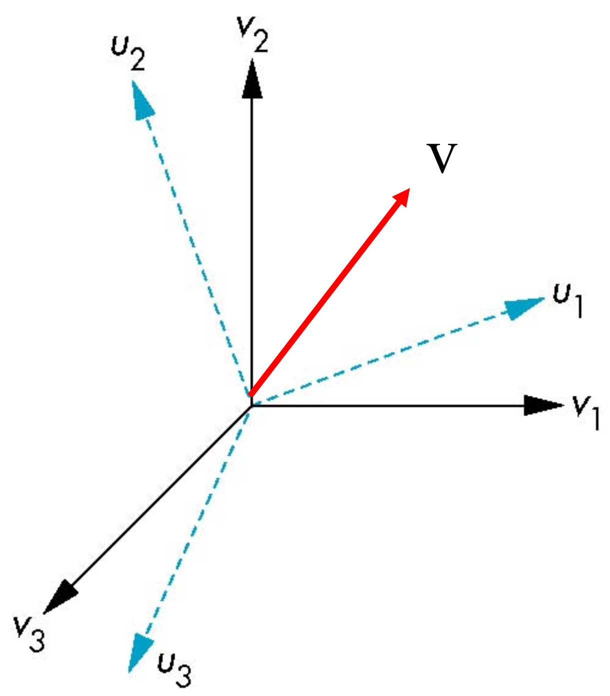
- A coordinate system is insufficient to represent points
- If we work in an affine space we can add a single point, the *origin*, to the basis vectors to form a *frame*



Representing one basis in terms of another

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

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



Matrix Form

The coefficients define a 3 x 3 matrix

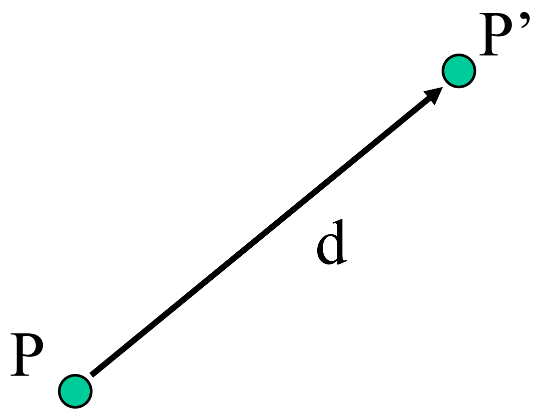
$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

and the bases can be related by

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

Translation

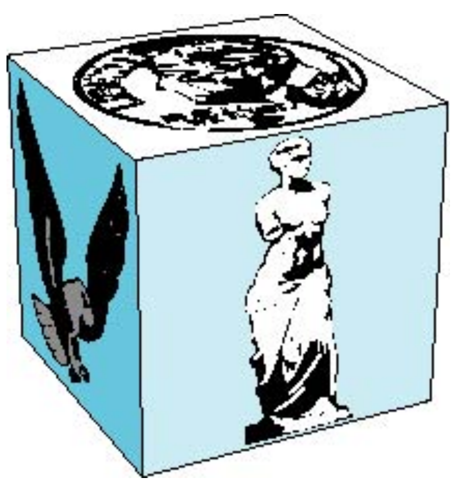
- Move (translate, displace) a point to a new location



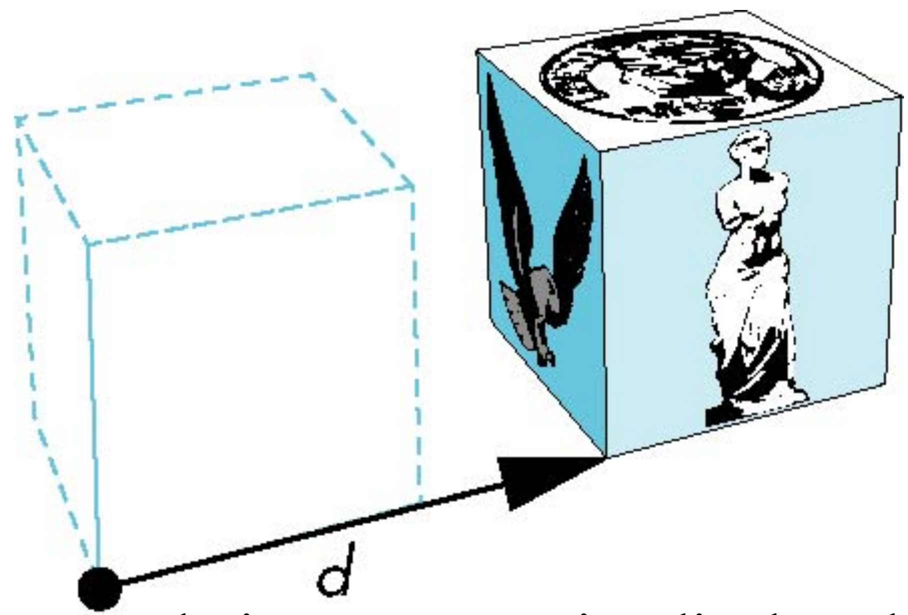
- Displacement determined by a vector d
  - Three degrees of freedom
  - P'=P+d

How many ways?

Although we can move a 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 y z 1]^T

$\mathbf{p}$'=[x' y' z' 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 also express translation using a
4 x 4 matrix **T** in homogeneous coordinates
p'=**Tp** where

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

This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together

Homogeneous Coordinates

The homogeneous coordinates form for 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 that homogeneous coordinates replaces points in three dimensions by lines through the origin in four 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*

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

$$\mathbf{p}' = \mathbf{R}_{\mathbf{z}}(\theta)\mathbf{p}$$

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)

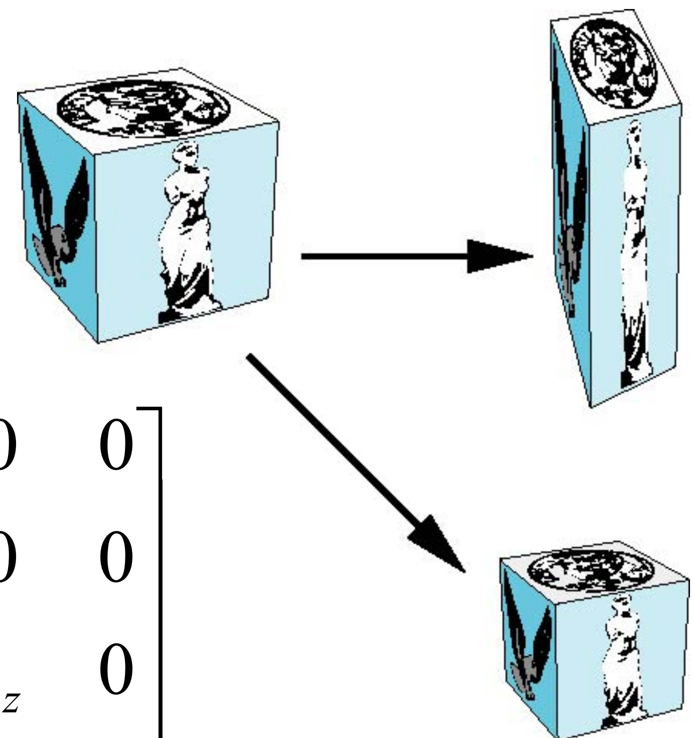
$x' = s_x x$

$y' = s_y x$

$z' = s_z x$

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

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

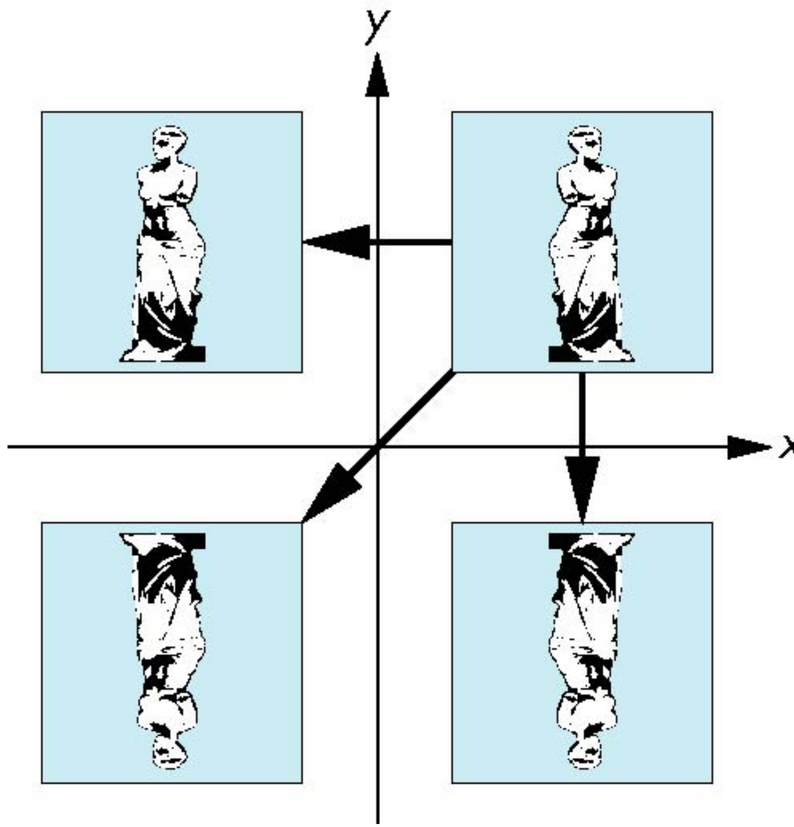


Reflection

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

    $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{T}(\theta)$
  - 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
- Because the same transformation is applied to many vertices, the cost of forming a matrix **M**=**ABCD** is not significant compared to the cost of computing **Mp** for many vertices **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 many references use column matrices to represent points. In terms of column matrices

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

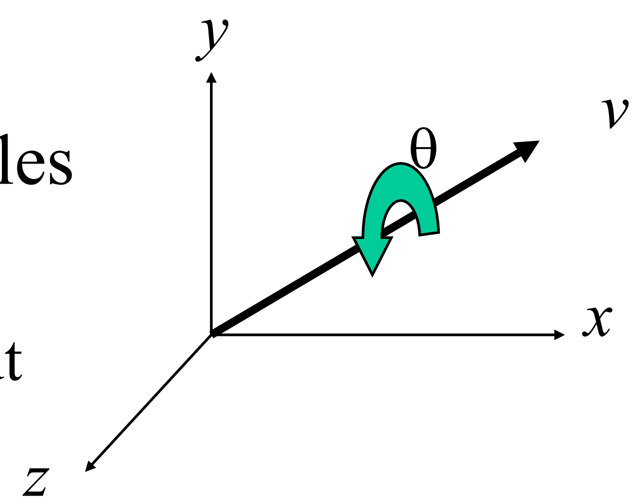
General Rotation About the Origin

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 that rotations do not commute
We can use rotations in another order but with different angles



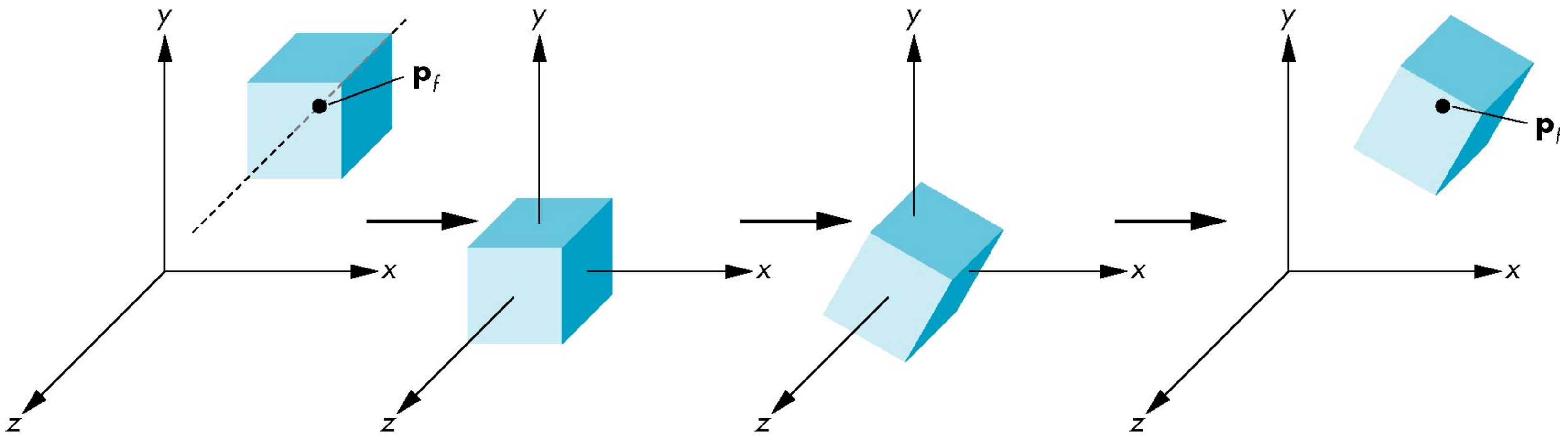
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

Move fixed point back

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



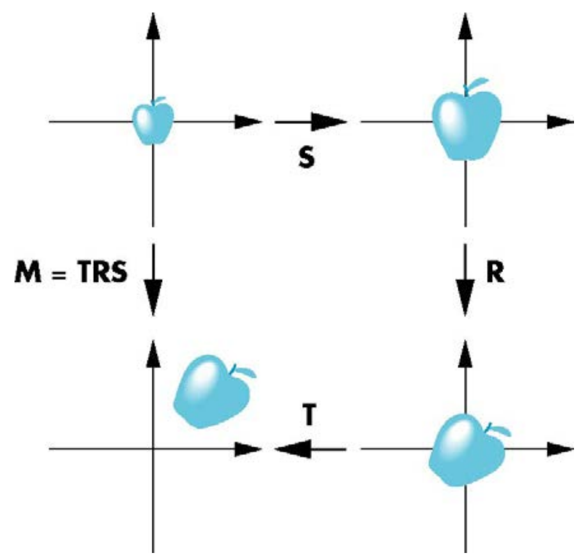
Instancing

- 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

    Scale

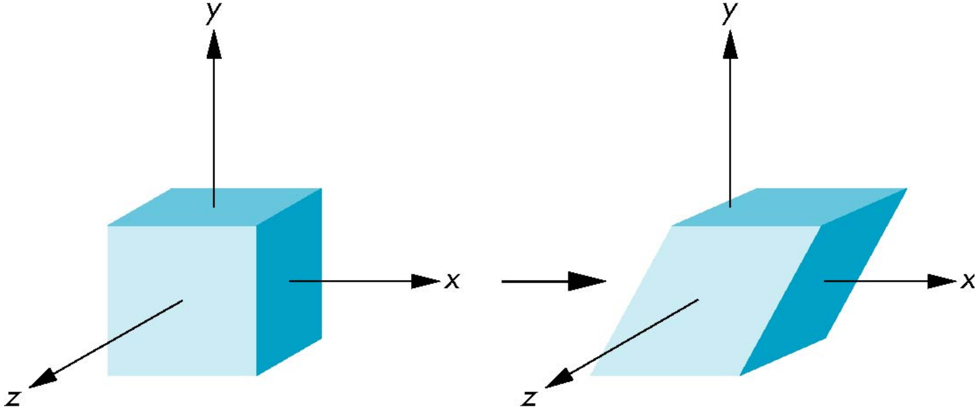
    Orient

    Locate



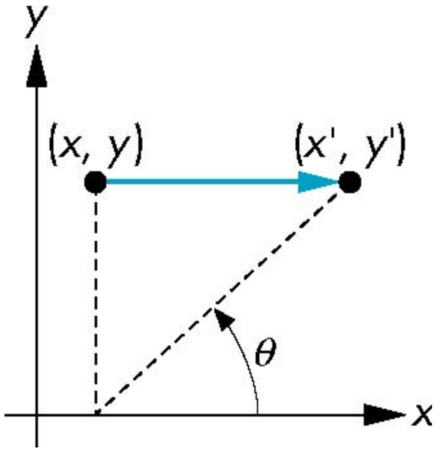
Shear

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



Shear Matrix

Consider simple shear along *x* axis



x' = x + y cot θ
y' = y
z' = z

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

Computer Viewing

Ed Angel
Professor of Computer Science, Electrical and Computer Engineering, and Media Arts
University of New Mexico

Objectives

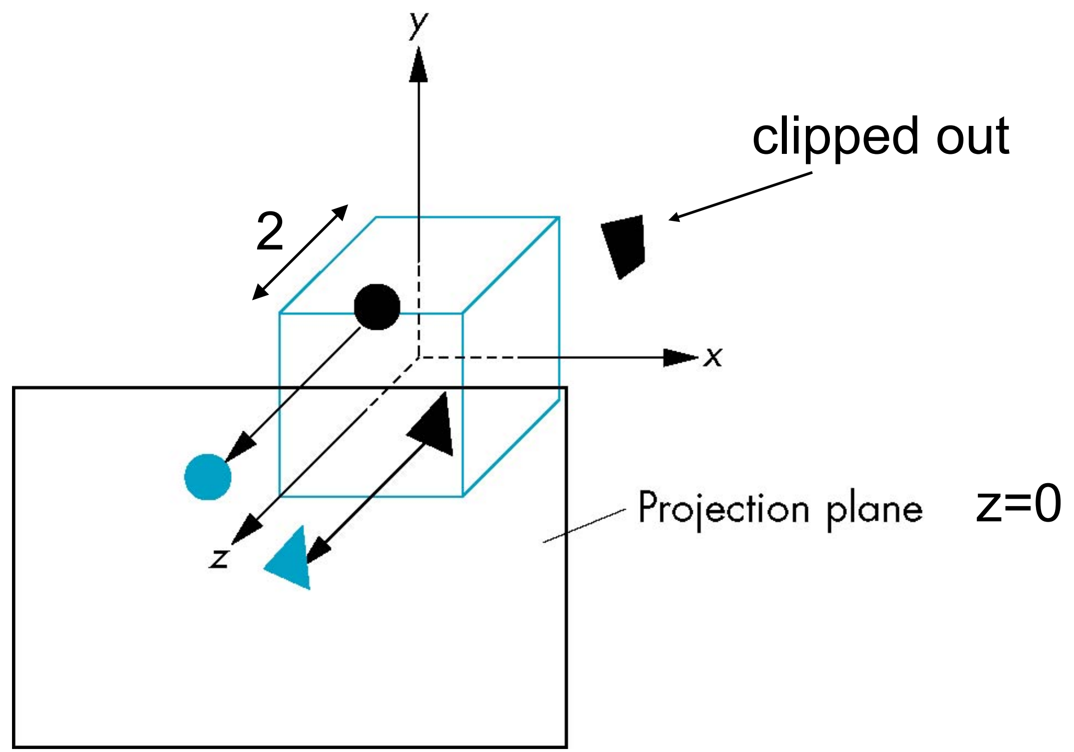
- Introduce the mathematics of projection

Computer Viewing

- There are three aspects of the viewing process, all of which are implemented in the pipeline,
  - Positioning the camera
    - Setting the model-view matrix
  - Selecting a lens
    - Setting the projection matrix
  - Clipping
    - Setting the view volume
    - (default is unit cube, R^3, [-1,1])

Default Projection

Default projection is orthogonal

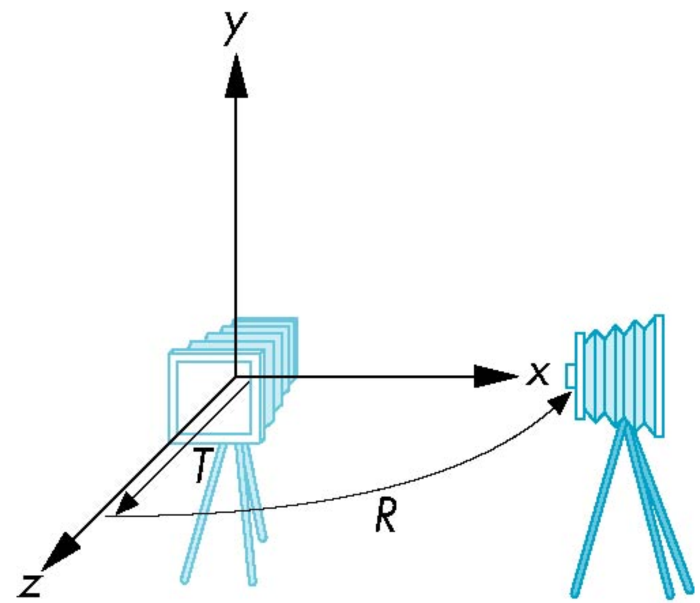


Moving the Camera Frame

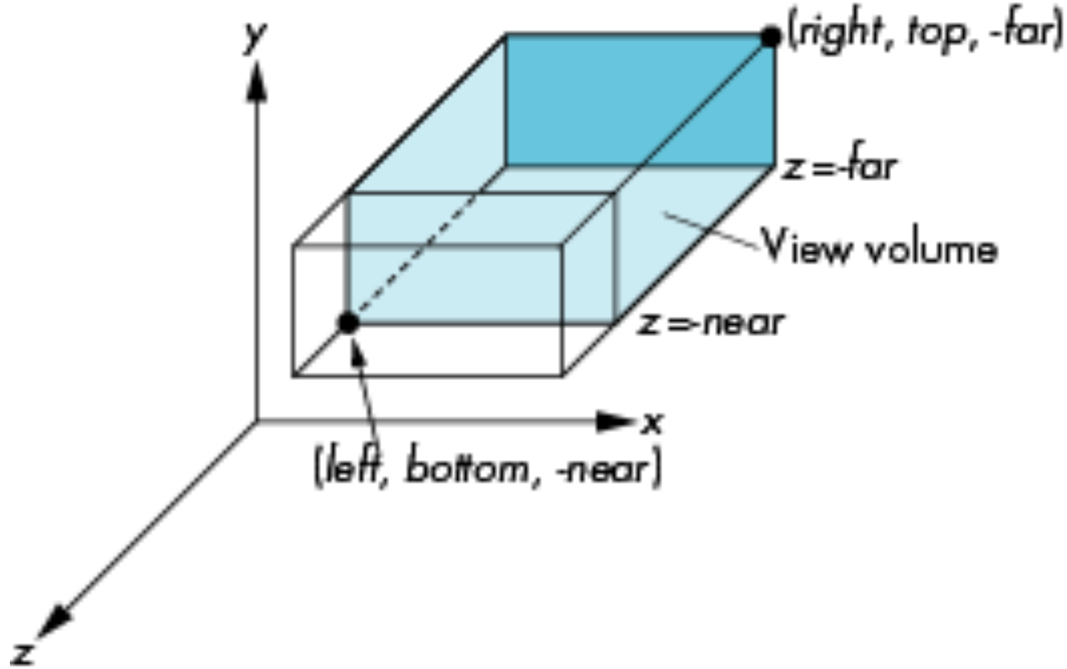
- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in the positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix

Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
  - –Rotate the camera
  - –Move it away from origin
  - –Model-view matrix C = TR

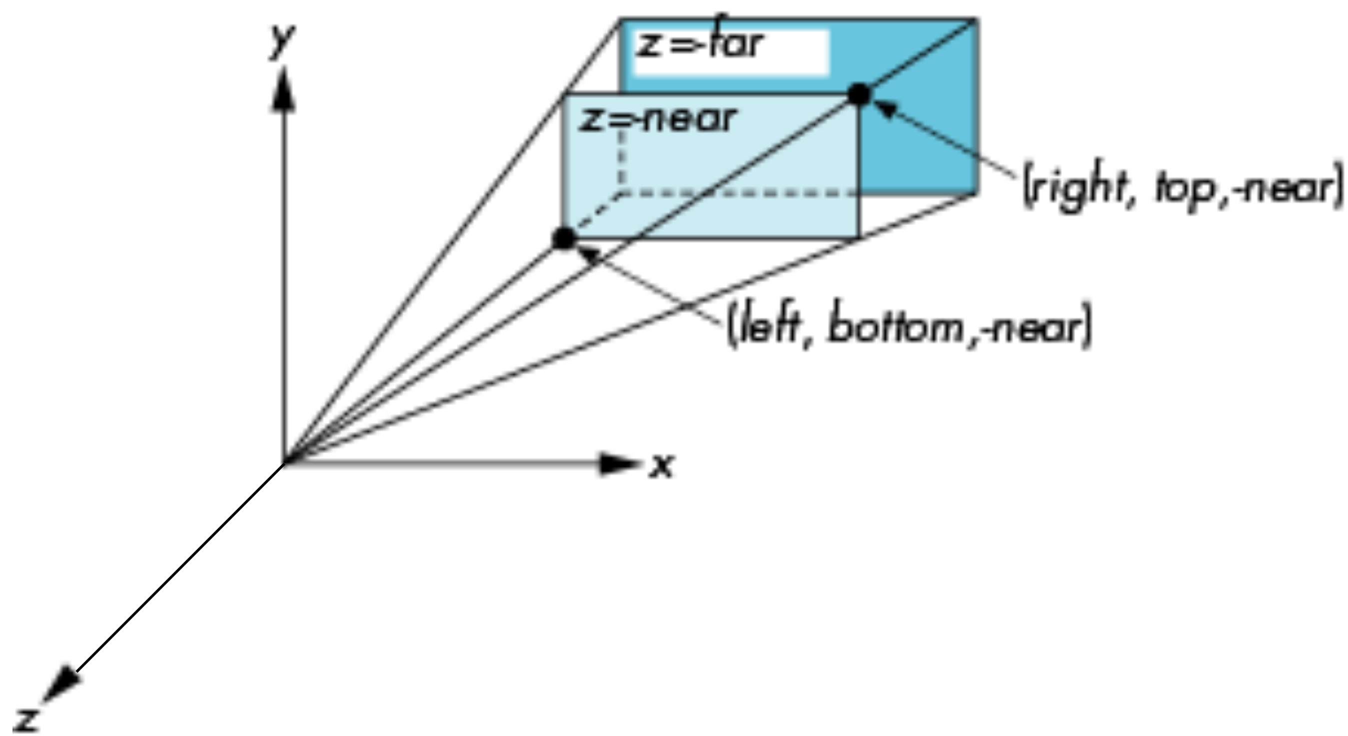


OpenGL Orthogonal Viewing



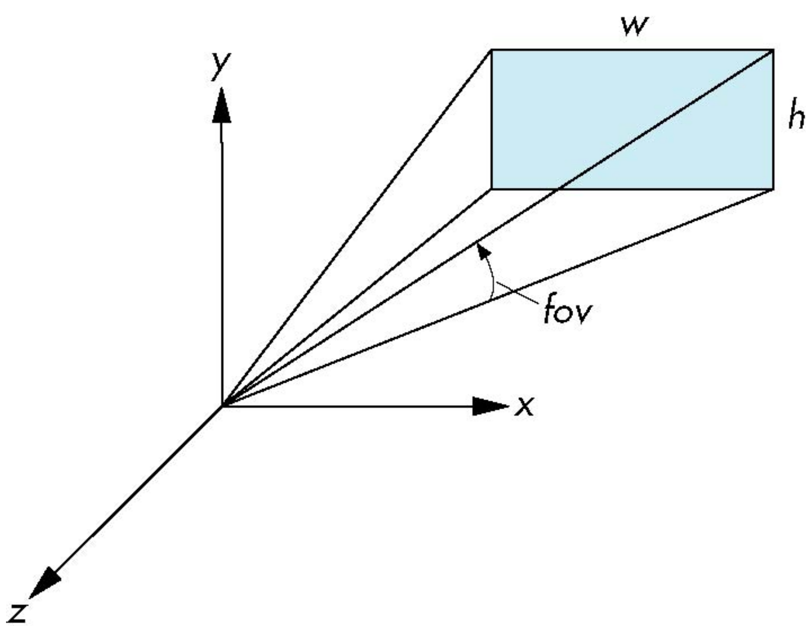
`near` and **`far`** measured <u>from</u> camera

OpenGL Perspective



Using Field of View

- Parameters **`fovy, aspect, near, far`** often provides a better interface



front plane

`aspect = w/h`

Projections explained differently

- Read the following slides about orthogonal and perspective projections by your selves
- They present the same thing, but explained differently

Projections and Normalization

- The default projection in the eye (camera) frame is orthogonal
- For points within the default view volume

$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$

- Most graphics systems use *view normalization*
  - All other views are converted to the default view by transformations that determine the projection matrix
  - Allows use of the same pipeline for all views

Homogeneous Coordinate Representation

default orthographic projection

$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

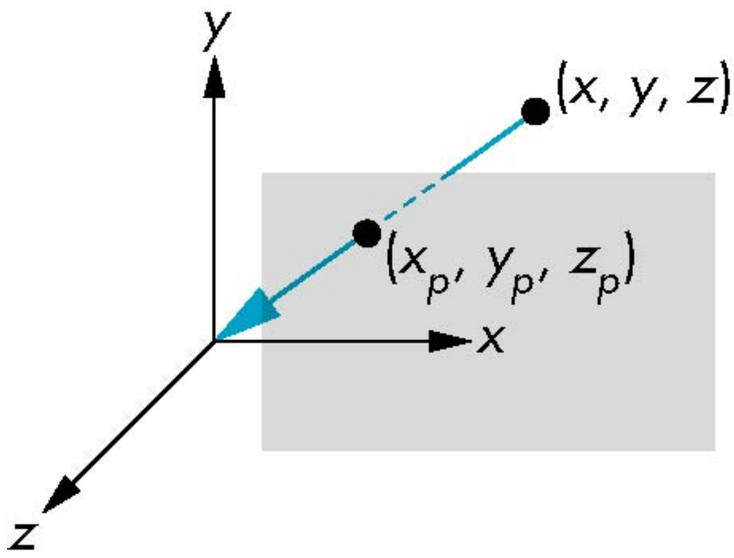
$x_p = x$
$y_p = y$
$z_p = 0$
$w_p = 1$

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

In practice, we can let $\mathbf{M} = \mathbf{I}$ and set the z term to zero later

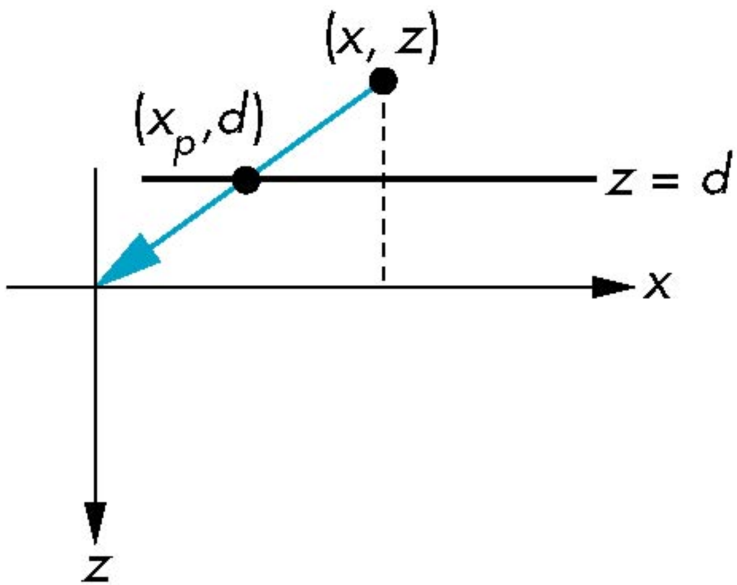
Simple Perspective

- Center of projection at the origin
- Projection plane $z = d$, $d < 0$

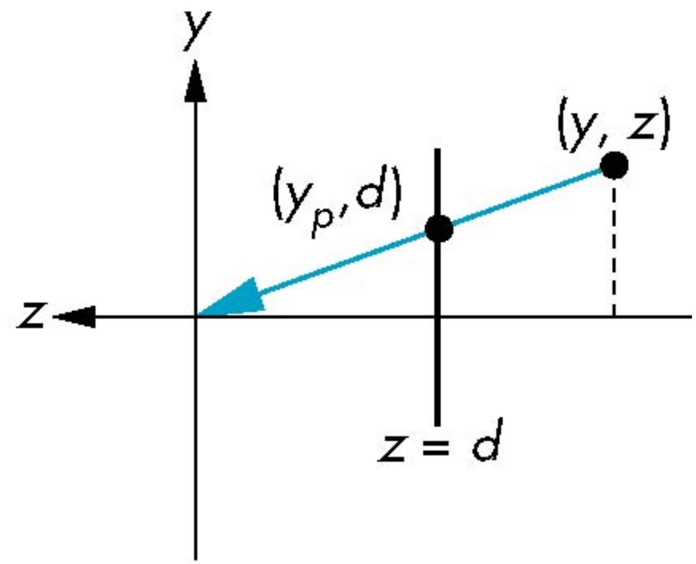


Perspective Equations

Consider top and side views



$$\frac{x}{z/d}$$



$$x_{\mathrm{p}} = \frac{x}{z/d} \qquad y_{\mathrm{p}} = \frac{y}{z/d} \qquad z_{\mathrm{p}} = d$$

Homogeneous Coordinate Form

consider $\mathbf{q} = \mathbf{M}\mathbf{p}$ where $\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Perspective Division

- However $w \neq 1$, so we must divide by w to return from homogeneous coordinates
- This *perspective division* yields

$$x_\mathrm{p} = \frac{x}{z/d} \qquad y_\mathrm{p} = \frac{y}{z/d} \qquad z_\mathrm{p} = d$$

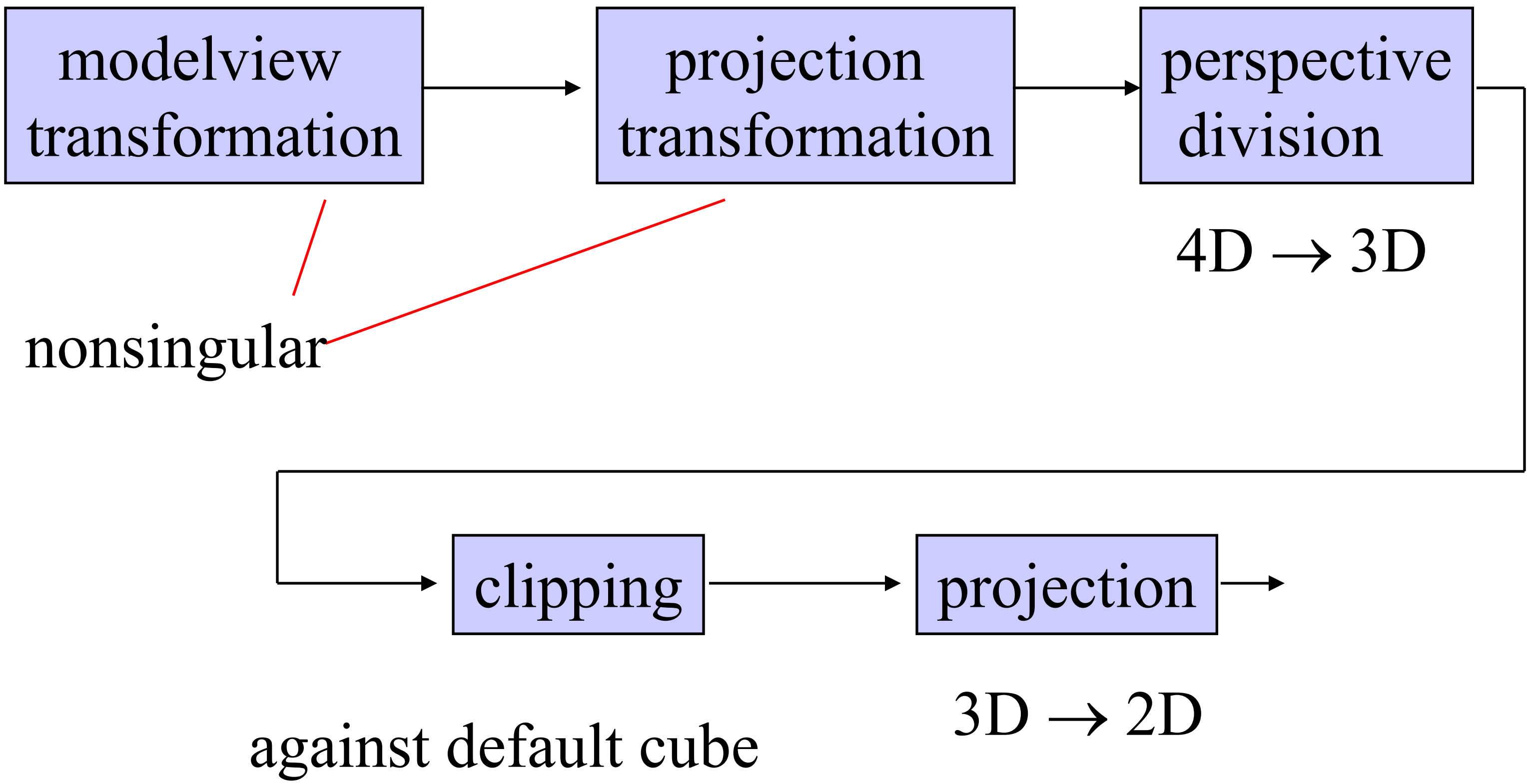
the desired perspective equations

- We will consider the corresponding clipping volume with the OpenGL functions

Normalization

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping

Pipeline View

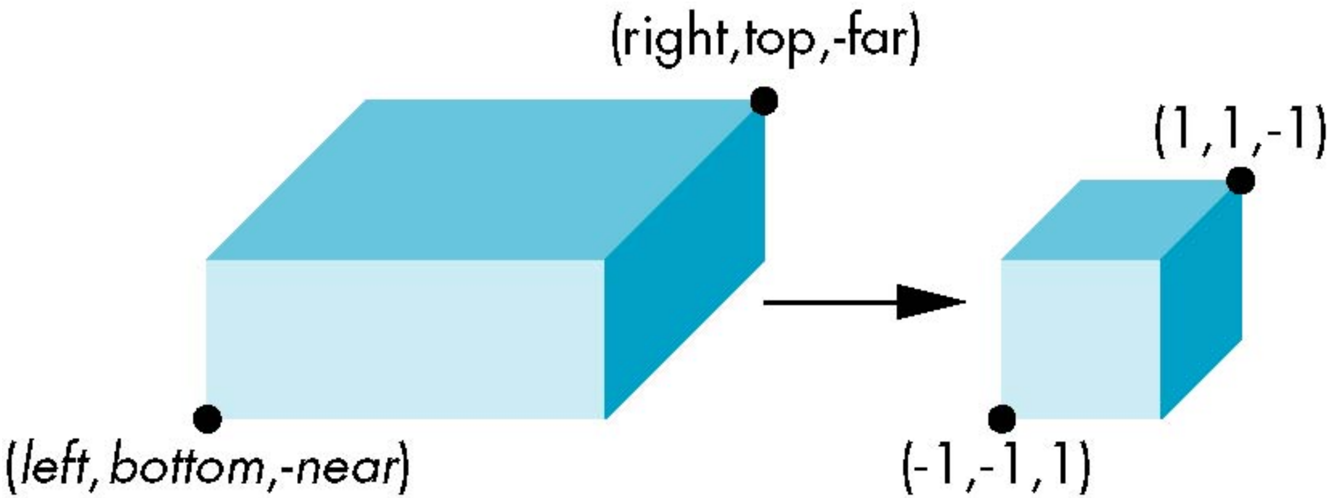


Notes

- We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  - Both these transformations are nonsingular
  - Default to identity matrices (orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection
- Delay final projection until end
  - Important for hidden-surface removal to retain depth information as long as possible

Orthogonal Normalization

normalization ⇒ find transformation to convert specified clipping volume to default



Orthogonal Matrix

- Two steps
  - –Move center to origin

    T(-(left+right)/2, -(bottom+top)/2,(near+far)/2))
  - –Scale to have sides of length 2

    S(2/(left-right),2/(top-bottom),2/(near-far))

$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right-left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Final Projection

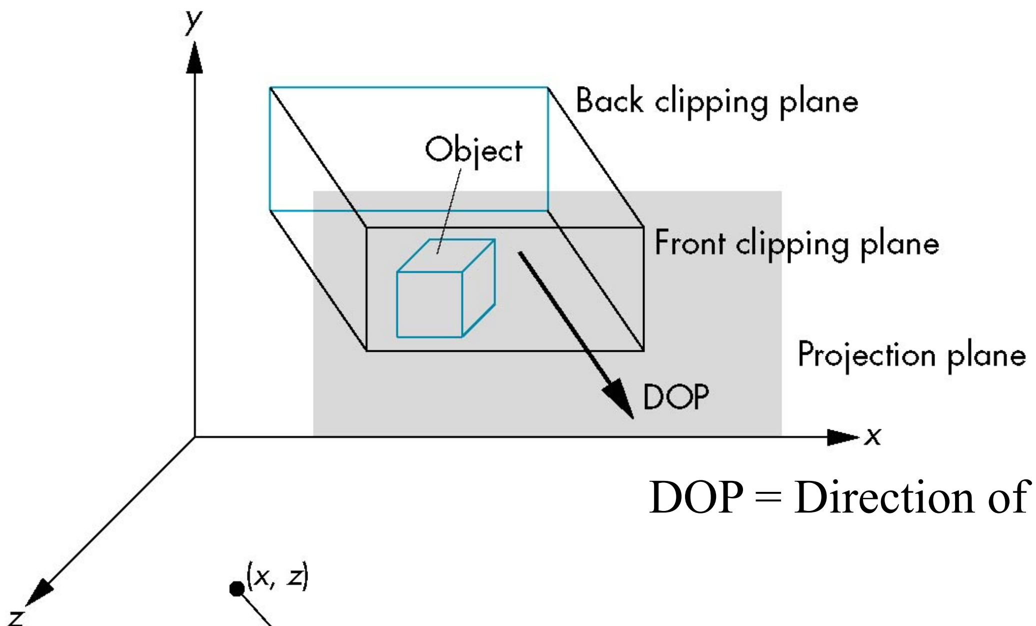
- Set z =0
- Equivalent to the homogeneous coordinate transformation

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

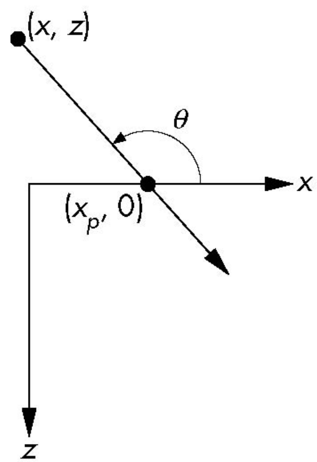
- Hence, general orthogonal projection in 4D is

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$$

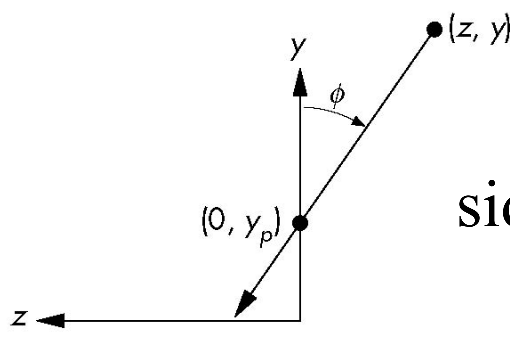
General Shear



DOP = Direction of Projection



top view



side view

Shear Matrix

xy shear (*z* values unchanged)

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

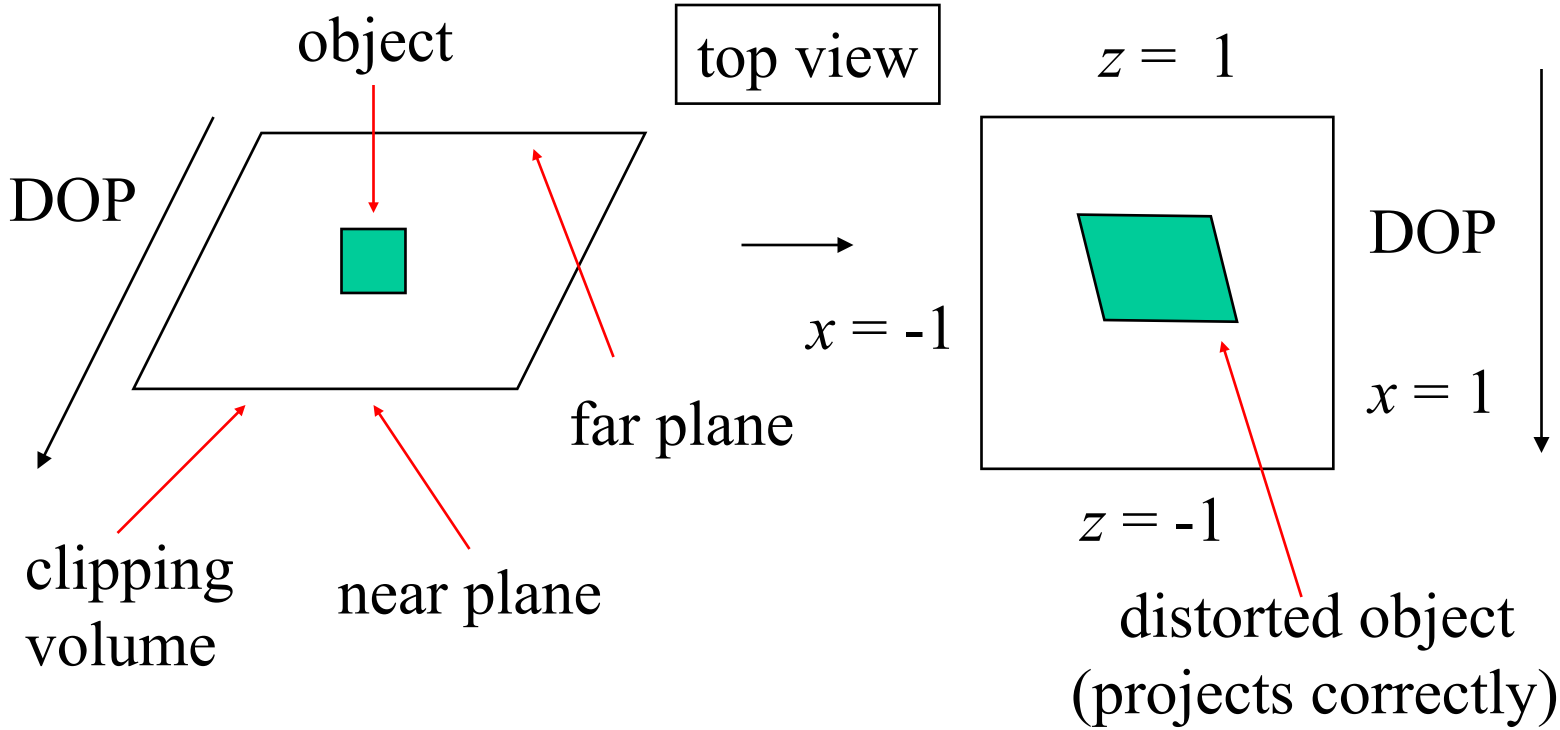
Projection matrix

$$\mathbf{P} = \mathbf{M}_{\mathrm{orth}}\,\mathbf{H}(\theta,\phi)$$

General case: $\mathbf{P} = \mathbf{M}_{\mathrm{orth}}\,\mathbf{STH}(\theta,\phi)$

Effect on Clipping

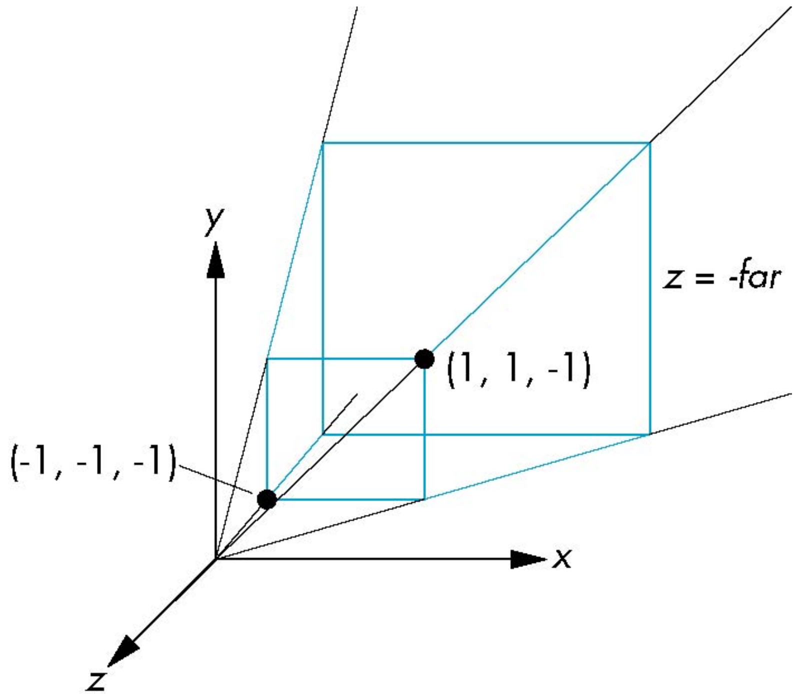
- The projection matrix **P** = **STH** transforms the original clipping volume to the default clipping volume



Simple Perspective

Consider a simple perspective with the COP (=center of projection) at the origin, the near clipping plane at $z = -1$, and a 90 degree field of view determined by the planes

$x = \pm z, y = \pm z$



Perspective Matrices

Simple projection matrix in homogeneous coordinates

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

Note that this matrix is independent of the far clipping plane

Generalization

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

after perspective division, the point $(x, y, z, 1)$ goes to

$x'' = x/z$
$y'' = y/z$
$Z'' = -(\alpha+\beta/z)$

which projects orthogonally to the desired point regardless of α and β

Picking α and β

If we pick

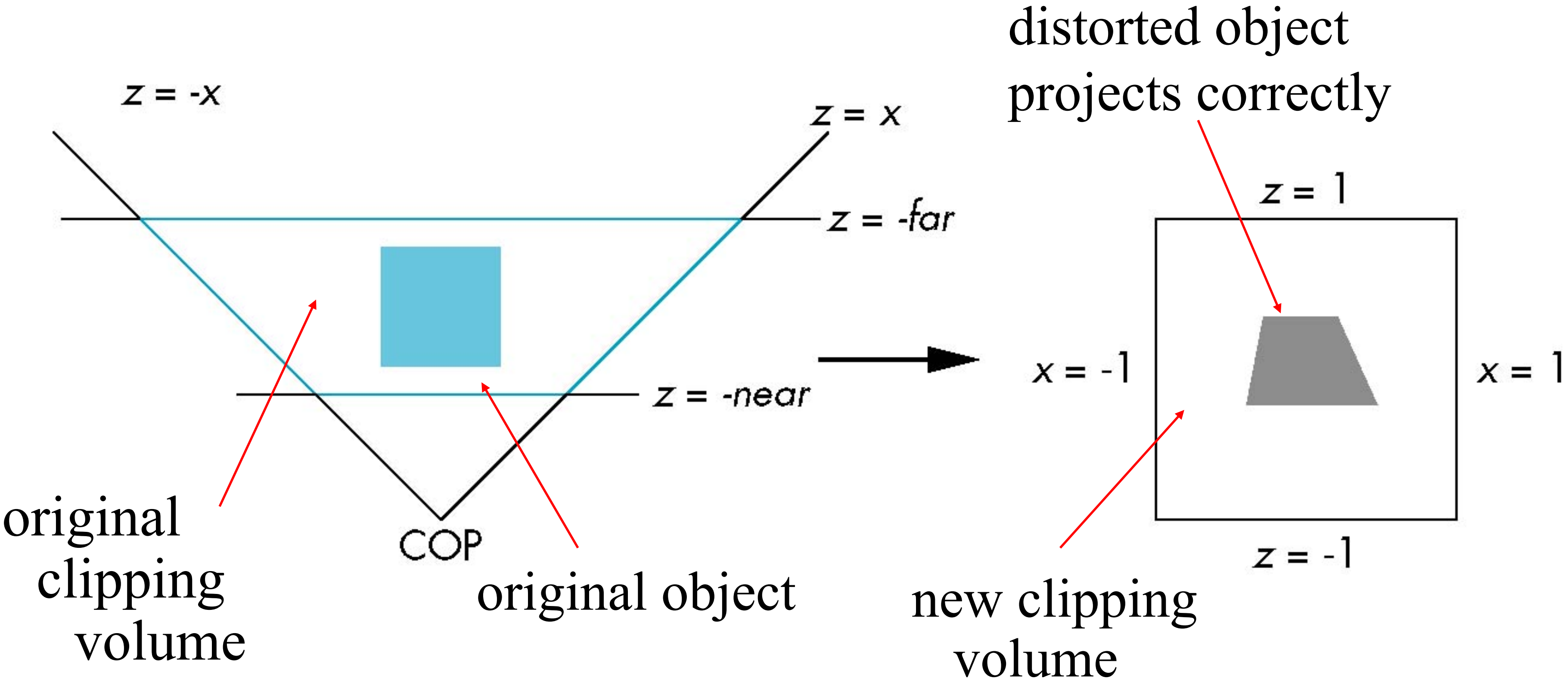
$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

$$\beta = \frac{2\text{near} * \text{far}}{\text{near} - \text{far}}$$

the near plane is mapped to $z = -1$
the far plane is mapped to $z = 1$
and the sides are mapped to $x = \pm 1, y = \pm 1$

Hence the new clipping volume is the default clipping volume

Normalization Transformation

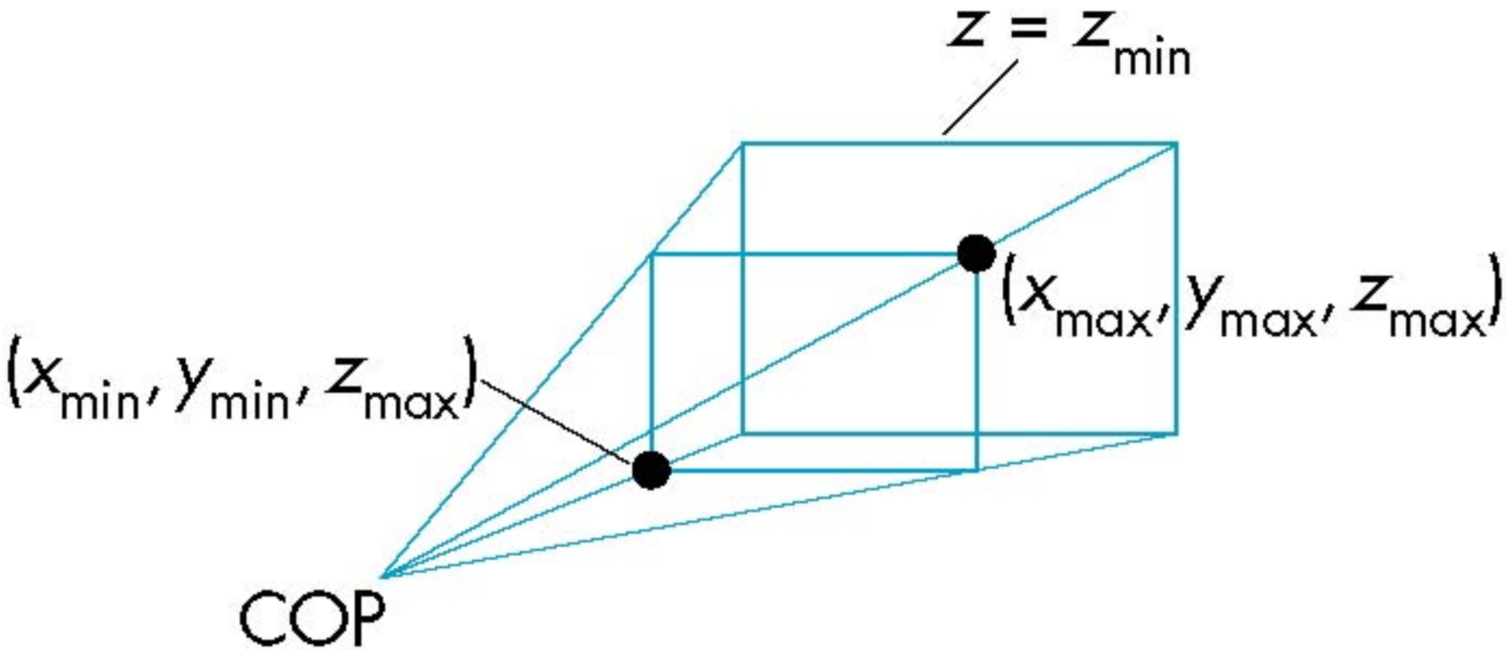


Normalization and Hidden-Surface Removal

- Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if $z_1 > z_2$ in the original clipping volume then the for the transformed points $z_1' > z_2'$
- Thus hidden surface removal works if we first apply the normalization transformation
- However, the formula $z'' = -(\alpha+\beta/z)$ implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small

OpenGL Perspective

- Unsymmetric viewing frustum possible:



OpenGL Perspective Matrix

- The normalization by a perspective projection requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthogonal transformation

$$\mathbf{P} = \mathbf{NSH}$$

our previously defined perspective matrix

shear and scale

Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping