

## CMPS 415/515, Fall 2018

Notes:

- 1) Do not redistribute without permission except according to "fair use" standards. Assume content is copyrighted, and that distributing more than small excerpts exceeds fair use. (See policies in the syllabus, UL's Copyright Handbook, and the publishers' licenses for instructors).
- 2) The lectures are mostly whiteboard-based, rather than slide-based, so content will not match exactly.

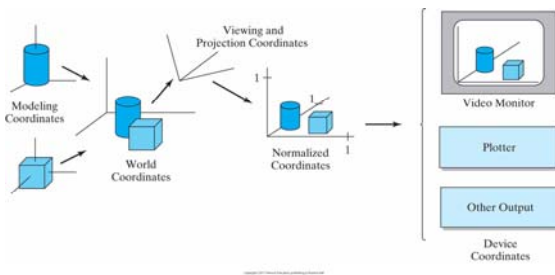
## Note

[HB] identifies figures from *Computer Graphics with OpenGL: Third Edition*, by Hearn and Baker, ©2004 Pearson Education

[FVD] identifies figures from *Computer Graphics: Principles and Practice: 2nd Edition*, by Foley, van Dam, Feiner, and Hughes, ©1996 Addison-Wesley

[Red] identifies figures from *OpenGL Programming Guide*, by Shreiner et al., ©1994-2013, Addison-Wesley ("Redbook")

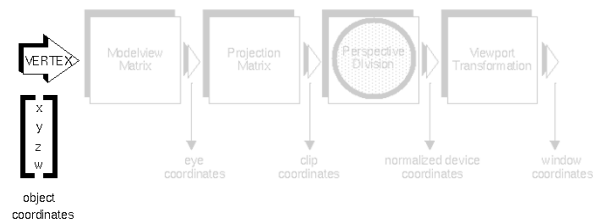
**Figure 3-1** The transformation sequence from modeling coordinates to device coordinates for a three-dimensional scene. Object shapes can be individually defined in modeling-coordinate reference systems. Then the shapes are positioned within the world-coordinate scene. Next, world-coordinate specifications are transformed through the viewing pipeline to viewing and projection coordinates and then to normalized coordinates. At the final step, individual device drivers transfer the normalized-coordinate representation of the scene to the output devices for display.



PEARSON Computer Graphics with OpenGL<sup>®</sup>, Fourth Edition  
Donald Hearn • M. Pauline Baker • Warren R. Carithers Copyright ©2011, ©2004 by Pearson Education, Inc.  
All rights reserved.

## Overview

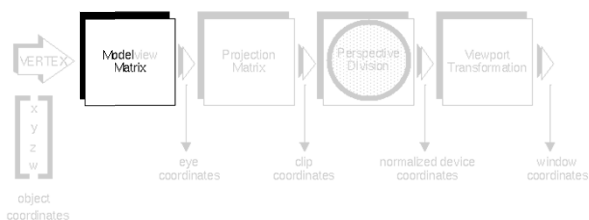
Object representation: geometry, topology, attributes.



[Red]

## Overview

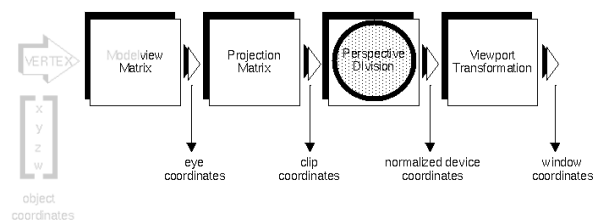
Modeling transforms: place objects to compose scene (e.g., rotate, scale)



[Red]

## Overview

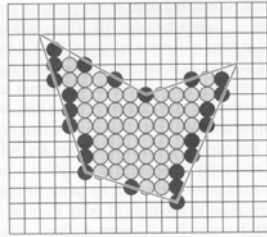
Other transforms: place camera, project 3D to 2D, convert to window coordinates



[Red]

## Overview

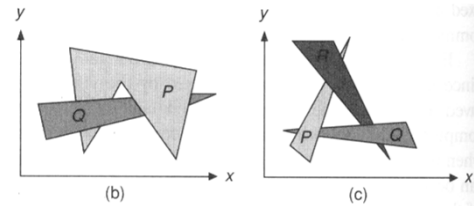
2D scan conversion: converts 2D object descriptions to pixel values



[FVD]

## Overview

Visible surface determination: prevents occluded regions from being seen



[FVD]

## Overview

Direct illumination: compute pixel color using a light description, textures, shading

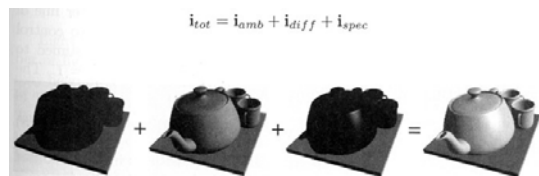
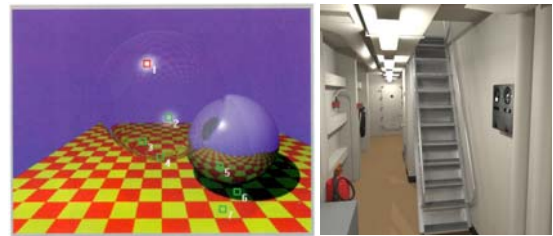


Figure 4.13. The basic lighting equation is illustrated for a teapot by adding (from the left) the ambient, the diffuse, and the specular components. The resulting lighting is shown at the right. (Tea cup model is reused courtesy of Joachim Hellenkaken.) [FVD]

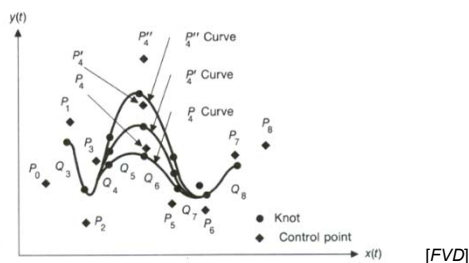
## Overview

Global illumination: methods for handling indirect light (interreflections)



## Overview

Curves and surfaces: a smoother alternative to polygon representations of objects



[FVD]

## Overview

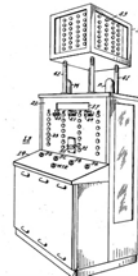
Other topics: probably particle dynamics



**Introduction:** brief intro to graphics; vector vs. raster displays; frame buffer.

**Scan Conversion:** lines; circles; filled polygons; other topics such as anti-aliasing and thick primitives; clipping of lines and polygons; brief summary of other topics

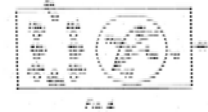
## Early electronic games



Nimatron  
Patent Diagram  
(1940)



Nimatron  
*American Mathematical  
Monthly* (1942)



CRT Amusement Device  
Patent Diagram (1948)

Maybe first "video" game  
(uses an oscilloscope)

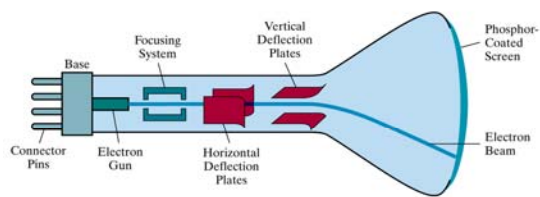


Figure 2-4  
Electrostatic deflection of the electron beam in a CRT.

[HB; 2.3]

## Early "video" games



Alexander S. Douglas  
programmed tic-tac-toe  
on EDSAC, 1952.

Oscilloscope showed  
memory content as a  
32 X 16 dot grid.  
Rotary phone dial input

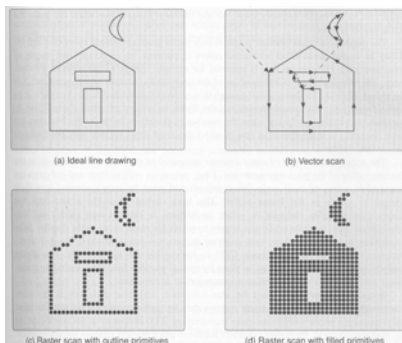


Tennis for Two (1958) at  
Brookhaven National Lab.  
Controlled by an analog  
computer

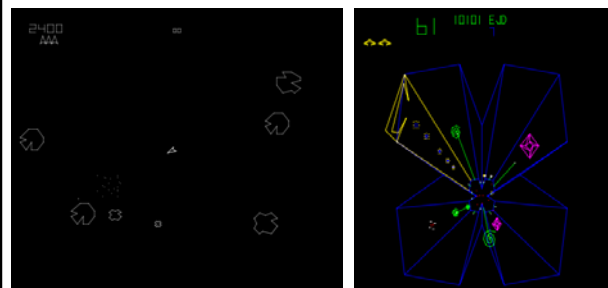


Spacewar! (1962) ran  
on a DEC PDP-1 at MIT

## Display comparison: vector vs. raster

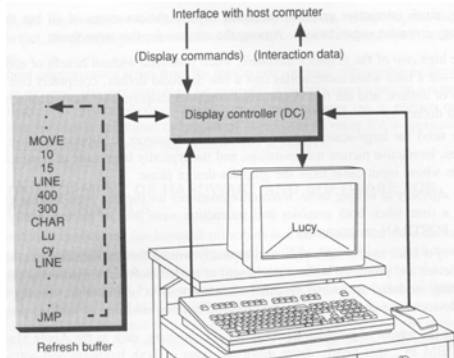


[FVD]



**Asteroids and Tempest** - games using vector displays.  
(however, here we see them as raster images)

## Vector (stroke, line drawing, calligraphic) Display



[FVD]

## Raster Displays

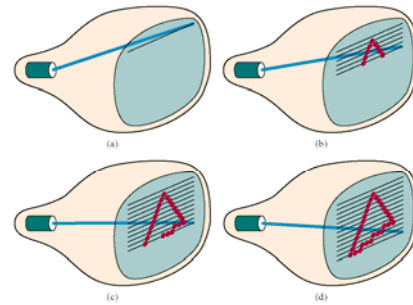


Figure 2-7

[HB] – compare to Fig. 2-9

## Raster Displays

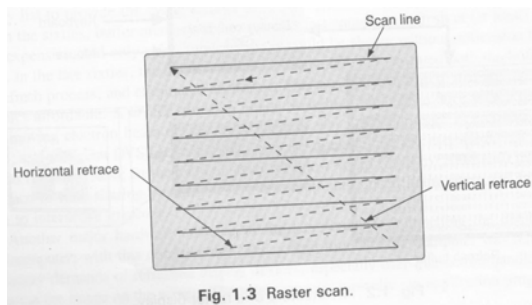


Fig. 1.3 Raster scan.

[FVD]

## Raster Displays

- A pixel (“picture element”) is a point on the screen
- A raster line is a row of pixels
- A raster is a set of horizontal raster lines
- A bitmap represents an image with one bit per pixel (black/white)
- A pixmap uses more bits (for color)
- Pixmap are now often also called bitmaps, but “pixmap” is more precise

## An example raster system architecture

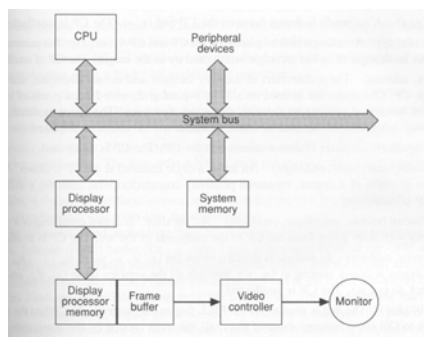
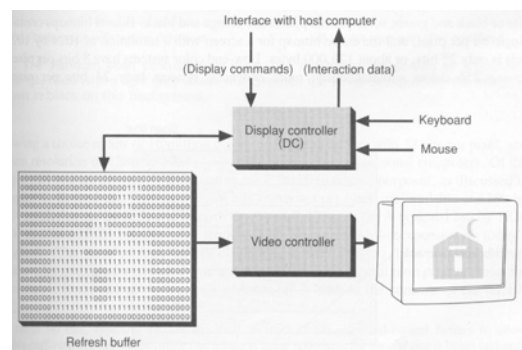


Fig. 4.22 Raster system architecture with a peripheral display processor.

[FVD] – same as Fig. 2-20 in [HB] (3-38 in older edition)

## Raster Displays



[FVD]

## Raster (Compared to Vector)

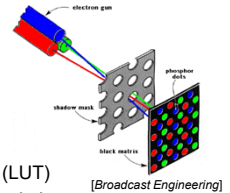
Advantages: lower cost scan-out logic, supports area fills, video refreshing independent of scene

Disadvantages: discrete approximation results in aliasing artifacts ("jaggies," "staircasing"), computational expense of scan conversion

What about buffer memory?

## Color pixels in a raster: two examples

- 24-bit RGB:
  - Each color pixel is represented by three bytes to control levels of three electron guns illuminating red, green, and blue phosphors
  - 16,777,216 million colors



- 8-bit indexed color:
  - One-byte index into a look-up table (LUT)
  - LUT typically stores 24-bit color descriptions
  - Only 256 colors can be displayed at once

## Frame buffers can include more:

- Color buffers
  - front/back for double buffering
  - left/right for stereoscopic rendering
- Depth buffer (z-buffer)
  - visible surface determination
- Stencil buffer
  - restricts drawing region
- Accumulation buffer (multiple exposures)
  - Used for motion blur, depth of field, etc.
  - A color buffer with high dynamic range
- Multisample buffer (for anti-aliasing)

## 2D scan conversion (rasterization)

- Generates pixel values from descriptions of primitives (e.g., lines, polygons)
- 3D polygon rendering uses 2D scan conversion after projection of primitives to a plane
- Should be as fast as possible
- Commonly done by a display processor

## Scan converting lines

- Compute coordinates for pixels close to line
  - For now, assume:
    - line specified by integer endpoints
    - 1-pixel-thick line
    - black/white display
    - line slope between -1 and 1 (inclusive)
- Basic approach: color 1 pixel per column between endpoints

## A naïve algorithm

- Compute  $m$  and  $B$  for the line equation
$$y = mx + B \quad (m = \Delta y / \Delta x)$$
- For each integer x-coordinate,  $x_i$ , compute:
$$y_i = mx_i + B,$$
and color the pixel at coordinate
$$(x_i, \text{floor}(0.5 + y_i))$$
- The equation requires an fp multiply, an addition, and a rounding operation

### A better algorithm: Basic Incremental, like Digital Differential Analyzer (DDA)

*Incremental* evaluation: computes a new value by adding an increment to a previous value

The x increment is 1 (move right one pixel at a time)

The y increment is  $y_{i+1} - y_i$

$$\begin{aligned} &= (mx_{i+1} + B) - (mx_i + B) \\ &= mx_{i+1} - mx_i \\ &= m(x_i + 1) - mx_i \\ &= m \end{aligned}$$

So,  $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i + m)$

### DDA algorithm (cont'd)

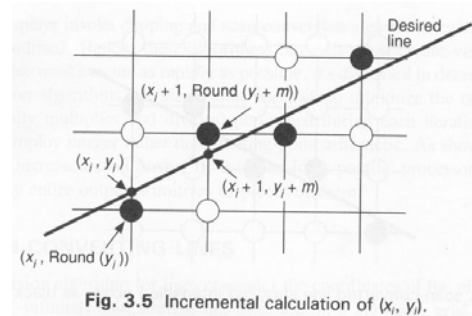


Fig. 3.5 Incremental calculation of  $(x_i, y_i)$ .

[FVD]

### Abbreviated pseudocode

{ to draw a line from  $(x_0, y_0)$  to  $(x_1, y_1)$  }

$m = (y_1 - y_0) / (x_1 - x_0);$  // floating point divide

$y = y_0;$

for  $(x = x_0; x \leq x_1; x++)$

{

Color the pixel at  $(x, \text{round}(y))$

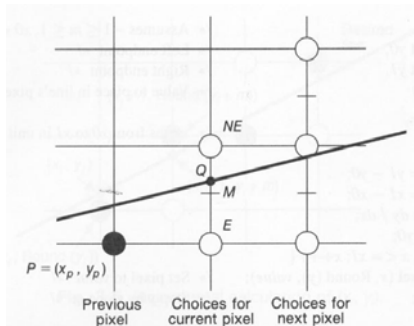
$y += m;$

}

### Even better: Midpoint Line Algorithm

- Similar to Bresenham's Algorithm (1965) that controlled an incremental plotter (and calculated an iteration in 1.5 millisecs).
- Eliminates fp operations and rounding
- Note: now assume slope between 0 and 1.
- Main idea: take steps in E or NE direction depending on value of a "decision variable"

### Midpoint Line Algorithm



[FVD]

### Midpoint Line Algorithm

(caution: M is midpoint,  $m$  is slope)

The vertical distance between Q and M is:

$$d_i = y_Q - y_M$$

A pixel is chosen (E or NE) based on the sign of  $d_i$ , and then  $d_{i+1}$  is computed:

When the E pixel is chosen,  $d_{i+1} = d_i + m$

When NE is chosen,  $d_{i+1} = d_i + m - 1$

Initialization:  $d_i = m - 0.5$

## Midpoint Line Algorithm

Since the decision is based only on the sign of  $d_i$ , scaling by a constant results in the same decisions. So, scale by  $2\Delta x$  to eliminate floating point in main loop:

init:  $d_1 = 2\Delta y - \Delta x$   
 move E:  $d_{i+1} = d_i + 2\Delta y$   
 move NE:  $d_{i+1} = d_i + 2\Delta y - 2\Delta x$

(precompute  $\Delta_E = 2\Delta y$  and  $\Delta_{NE} = 2\Delta y - 2\Delta x$ )

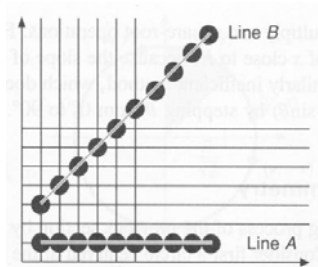
## Abbreviated pseudocode

```
Initialize integers  $\Delta_E$ ,  $\Delta_{NE}$ ,  $d$ ,  $x$ , and  $y$ 
Set pixel at  $x$ ,  $y$  (first pixel)
while  $x < \text{last column (given by an endpoint)}$ 
{
    increment  $x$  by 1
    if ( $d < 0$ )
        add  $\Delta_E$  to  $d$ 
    else
    {
        increment  $y$  by 1
        add  $\Delta_{NE}$  to  $d$ 
    }
    set pixel at  $x$ ,  $y$ 
}
```

## Other notes

- When  $d_i$  is 0, use some consistent convention for choice between E and NE
- Handle other slopes using symmetry
- Make sure right-to-left order results in same choice of pixels
- Perceived line intensity depends on slope

## Slope and line intensity



[FVD]

## Differences between [HB] and lecture ([FVD])

In [HB]:

- For DDA, the subscript is  $k$  instead of  $i$ , and the algorithm is generalized
- For Midpoint approach (Bresenham),  $k$  instead of  $i$  and  $p$  instead of  $d$
- They choose grids with pixels in grid cells instead of at crossings
- *Learn from the source you find most clear; you don't need to know both forms*

## Thick Primitives: Replicating Pixels

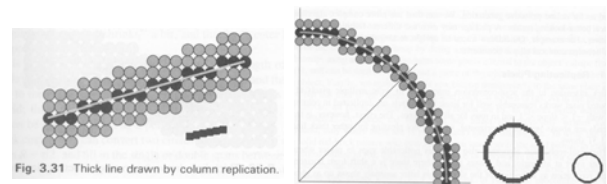
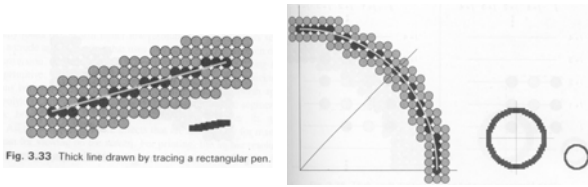


Fig. 3.31 Thick line drawn by column replication.

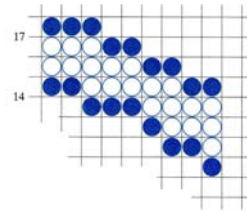
[FVD]

## Thick Primitives: Moving Pen



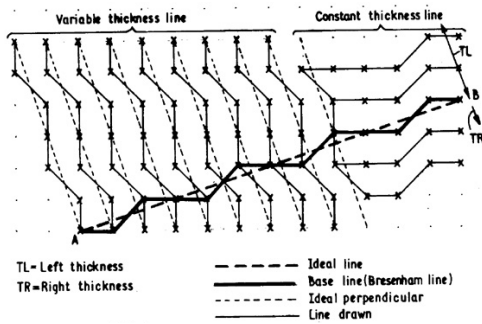
[FVD]

## Thick Primitives: Filling Areas



[HB, FVD]

## Thick Lines: using perpendicular

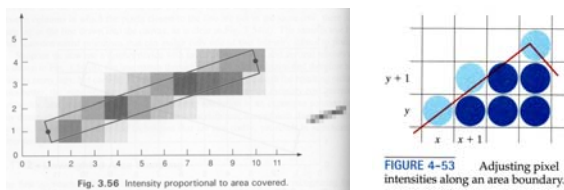


(see paper by A. S. Murphy if interested)

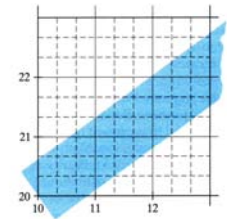
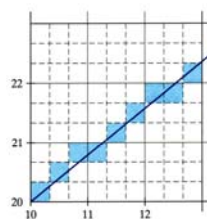
## Antialiasing



## Antialiasing: Unweighted Area Sampling

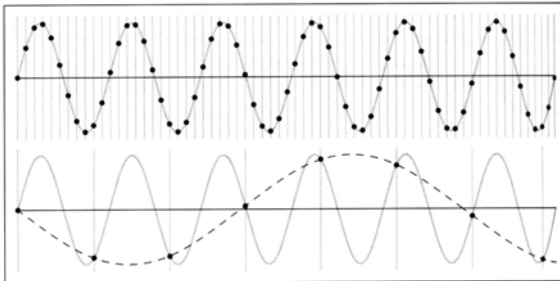


[FVD, HB]



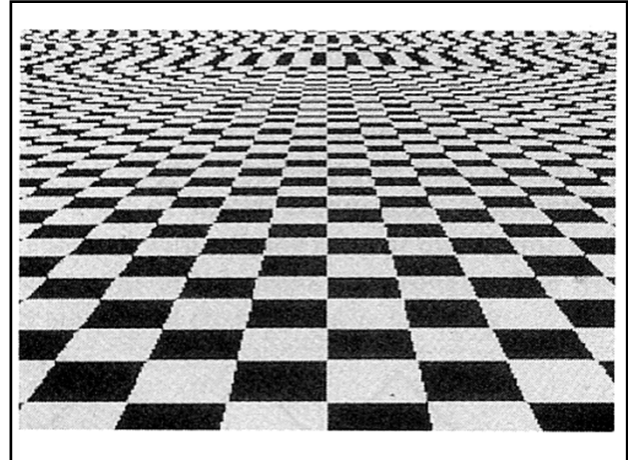
[HB]



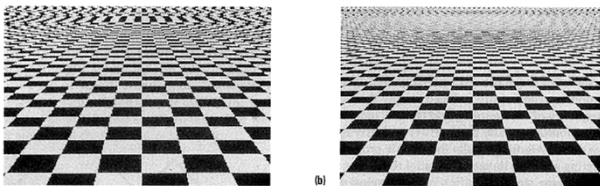


**Figure 9.2.** A sine wave (gray curve) sampled at two different rates. Top: at a high sample rate, the resulting samples (black dots) represent the signal well. Bottom: a lower sample rate produces an ambiguous result: the samples are exactly the same as would result from sampling a wave of much lower frequency (dashed curve).

[Fundamentals of Computer Graphics, 3<sup>rd</sup> Ed., Shirley and Marschner, 2009]



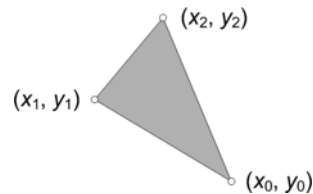
Assuming a regular grid of samples, no amount of supersampling can avoid aliasing artifacts, for example, for regions arbitrarily close to the vanishing point of a checker pattern



The pattern in (b) is a supersampled version of that in (a). Aliases still occur but appear at a higher spatial frequency.

### Filled Single Triangle Scan Conversion

- Given three vertices  $(x_0, y_0)$ ,  $(x_1, y_1)$ ,  $(x_2, y_2)$



- Let  $L_{i,j}$  denote edge from  $(x_i, y_i)$  to  $(x_j, y_j)$
- Assume  $y_0 \leq y_1 \leq y_2$  (can sort if needed)

For each scan line from  $y_0$  to  $y_1-1$ :

Compute the two x-coords where  $L_{0,1}$  and  $L_{0,2}$  intersect this scan line, and color pixels in this range

For each scan line from  $y_1$  to  $y_2$ :

Compute the two x-coords where  $L_{1,2}$  and  $L_{0,2}$  intersect this scan line, and color pixels in this range

Notes:

- Intersections can be computed by adding an increment to a previous value (like DDA, but add inverse slope, i.e., rate of change in x per change in y)
- If  $y_0 = y_1$  or  $y_1 = y_2$ , there is a horizontal edge; make sure not to invert a 0 slope

### Rethinking these “classic” approaches

- As scenes grow complex, with many tiny triangles, setup steps no longer save as much time later
- Consider special hardware features:
  - Parallelism
  - Frame memory may be tiled into pages and it's best to finish one before moving to another
  - Special devices. For example: a mobile phone, where power consumption is important (and memory access is expensive)

## Clipping to a rectangular window

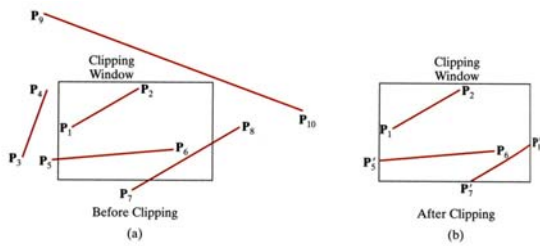


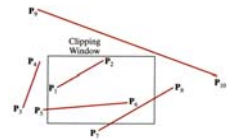
FIGURE 6-11 Clipping straight-line segments using a standard rectangular clipping window.

[HB]

## Cohen-Sutherland line-clipping algorithm

### • Overview

- Window bounded by 4 lines, each dividing the 2D space into two halfplanes (inner and outer)
- *Accept* line if both endpoints are in rectangle
- *Reject* if any outer halfplane contains both endpoints
- Otherwise, cut the line into two segments: one can be rejected, and the other can be processed recursively



## Region codes for Cohen-Sutherland

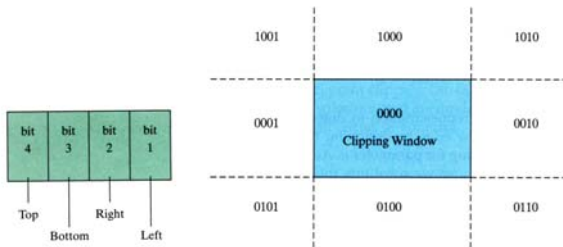


FIGURE 6-12 A possible ordering for the clipping-window boundaries corresponding to the bit positions in the Cohen-Sutherland endpoint region code.

FIGURE 6-13 The nine binary region codes for identifying the position of a line endpoint, relative to the clipping-window boundaries.

[HB]

## Use of region codes

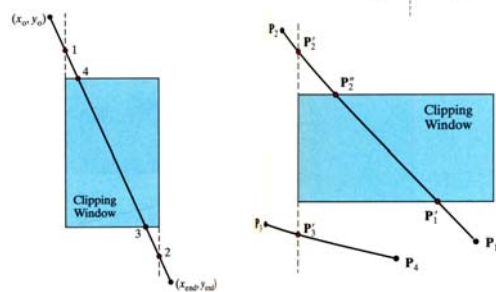
To clip a line:

- Assign codes to line endpoints (using coordinate tests)
- If both codes are zero, accept
- If logical **and** is nonzero, reject
- Otherwise: the line crosses a halfplane boundary (and no outer halfplane contains both endpoints)
  - select an endpoint with nonzero code
  - cut at the boundary represented by the first nonzero bit
  - discard the outer segment
  - clip the new line segment using C-S algorithm



## Line clipping example

[Textbook error in third edition for  $P_3$ - $P_4$  processing, lower right figure.]



[HB]

## Clipping Polygons

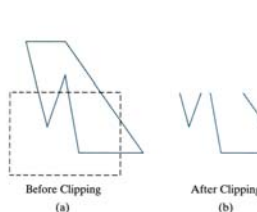


FIGURE 6-21 A line-clipping algorithm applied to the line segments of the polygon boundary in (a) generates the unconnected set of lines in (b).

FIGURE 6-22 Display of a correctly clipped polygon fill area.

[HB]

## Sutherland-Hodgman polygon clipping

- Consider one clipping edge at a time
- For each, compute a new polygon description by adding/deleting vertices

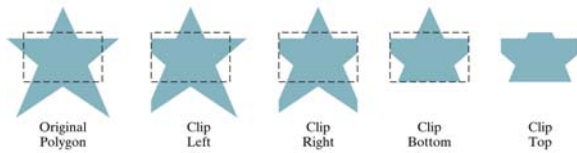


Figure 6-23

Processing a polygon fill area against successive clipping-window boundaries. [HB]

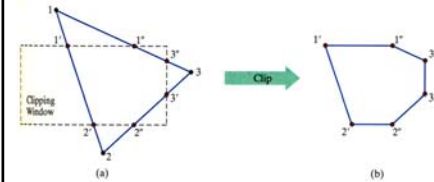


FIGURE 6-25 A convex-polygon fill area (a), defined with the vertex list (1, 2, 3), is clipped to produce the fill-area shape shown in (b), which is defined with the output vertex list (1', 2', 2'', 3', 1').

[HB]

## Sutherland-Hodgman (cont'd)

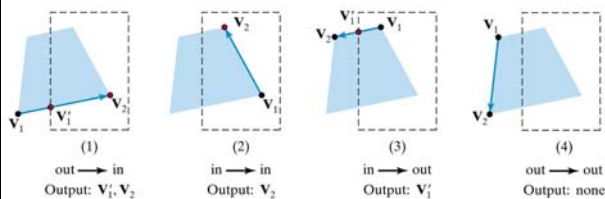


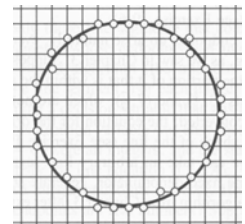
Figure 6-26

The four possible outputs generated by the left clipper, depending on the position of a pair of endpoints relative to the left boundary of the clipping window.

[HB]

## Other topics

- More robust algorithms for filled primitives (to solve problems with slivers, repeated pixels)
- Other primitives, e.g. ellipses, arcs, ...
- Floating point endpoints, radius, etc.



## Other topics (cont'd)

- Line styles or pattern/textured fills

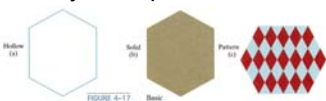


FIGURE 4-17

- Clipping to multiple windows or nonrectangular regions
- Line joints

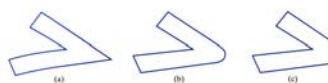
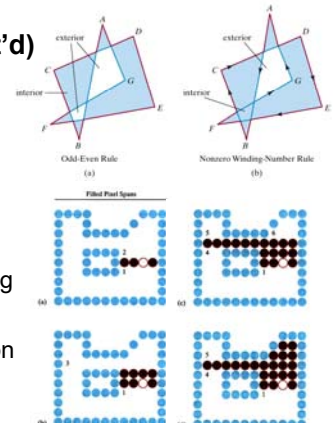


FIGURE 4-5 Thick line segments connected with a miter joint (a), a round joint (b), and a bevel joint (c).

[HB]

## Other topics (cont'd)

- Self-intersecting primitives
- Seed-fill algorithms
- Raster image scaling
- Character generation



[HB]

## Generating Characters

- Bitmaps in font cache, or
- Outline



Fig. 3.50 Portion of an example of a font cache.

[FVD]

## Book notes:

- strange comment: “improved line-drawing capabilities of raster systems”
- nyquist limit (discussion of aliasing): sampling interval should be strictly smaller than one-half the cycle interval (book just says “no larger”). It is not sufficient to sample at exactly one-half the cycle interval.
- one of the diagrams illustrating C-S line clipper : p3 to p4 are below bottom border and should be rejected entirely by the clipper

## Colored light

Perceptual terms:

- *Hue*: distinguishes between colors, relates to “dominant wavelength”
- *Saturation*: how far a color is from gray, relates to “excitation purity”
- *Lightness*: perceived intensity reflected, relates to “luminance”  
(*Brightness* used instead of lightness for light-emitting objects)

## Example spectra

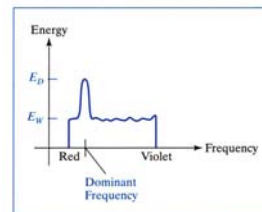


FIGURE 12-4 Energy distribution for a light source with a dominant frequency near the red end of the frequency range.

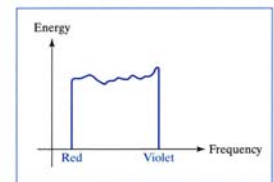


FIGURE 12-3 Energy distribution for a white light source.

[HB]

## Response of human color receptors

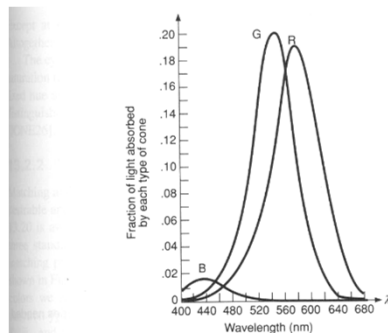


Fig. 13.18 Spectral-response functions of each of the three types of cones on the human retina.

[FVD]

## Use of red, green, blue primaries

A wide range of colors is matched by R,G,B sum

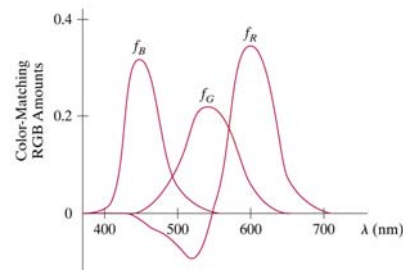
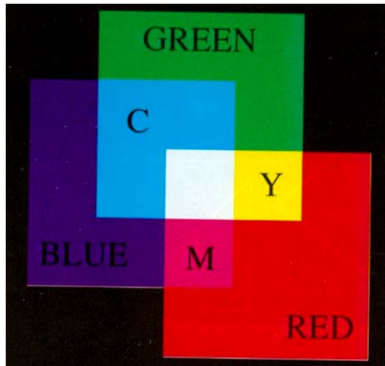


FIGURE 12-5 Three color-matching functions for displaying spectral frequencies within the approximate range from 400 nm to 700 nm.

(but note a range of negative levels for the red primary!)

[HB]



[FVD]

TABLE 4-1

THE EIGHT RGB COLOR CODES FOR A THREE-BIT PER PIXEL FRAME BUFFER

Color Code	Stored Color Values in Frame Buffer			Displayed Color
	RED	GREEN	BLUE	
0	0	0	0	Black
1	0	0	1	Blue
2	0	1	0	Green
3	0	1	1	Cyan
4	1	0	0	Red
5	1	0	1	Magenta
6	1	1	0	Yellow
7	1	1	1	White

(Intensity ranges from 0 to 1. For 24-bit color, this would scale to 0 to 255.) [HB]

## RGB Cube

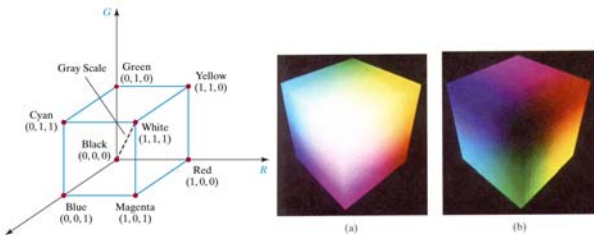


Figure 12-11

The RGB color model. Any color within the unit cube can be described as an additive combination of the three primary colors.

[HB]

## CMY color model (cyan, magenta, yellow)

- Subtractive (paint vs. light)
- Used in printing

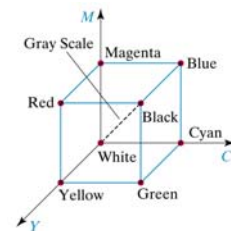


Figure 12-14

The CMY color model. Positions within the unit cube are described by subtracting the specified amounts of the primary colors from white.

[FVD, HB]

## CMY

- C, M, Y, are complements of R, G, B
- RGB to CMY conversion:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- CMYK – add black primary K using undercolor removal equation below. Same color in theory, but primaries aren't really perfect:

$$\begin{aligned} K &\leftarrow \min(C, M, Y); \\ C &\leftarrow C - K; \\ M &\leftarrow M - K; \\ Y &\leftarrow Y - K; \end{aligned}$$

Some printers and displays use additional primaries to improve color

## HSV model (hue, saturation, value)

- “User-oriented” rather than “hardware-oriented”

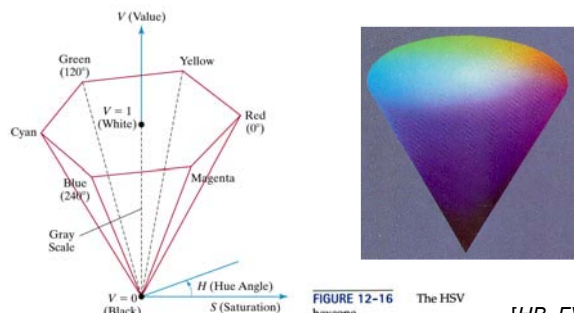
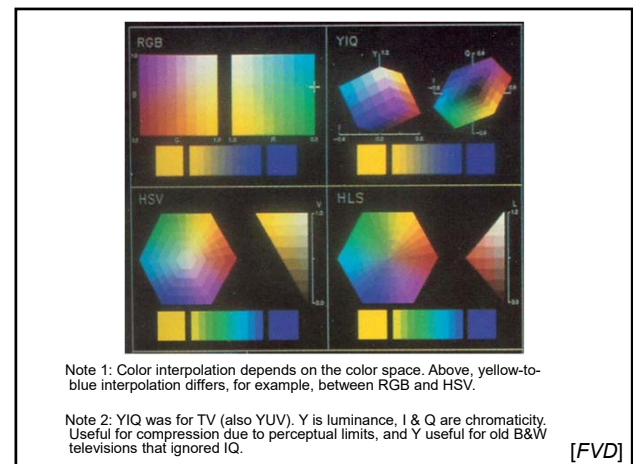
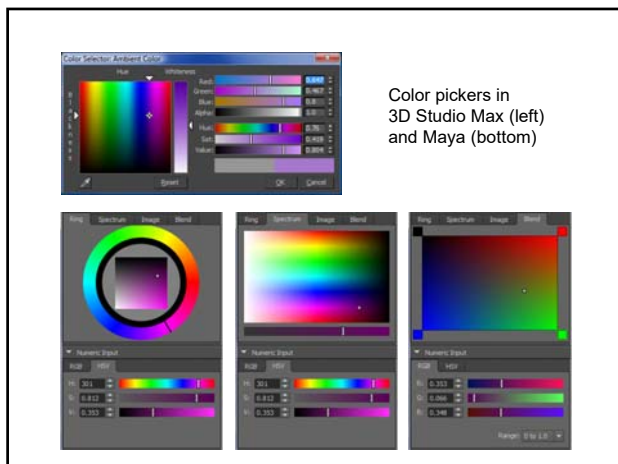
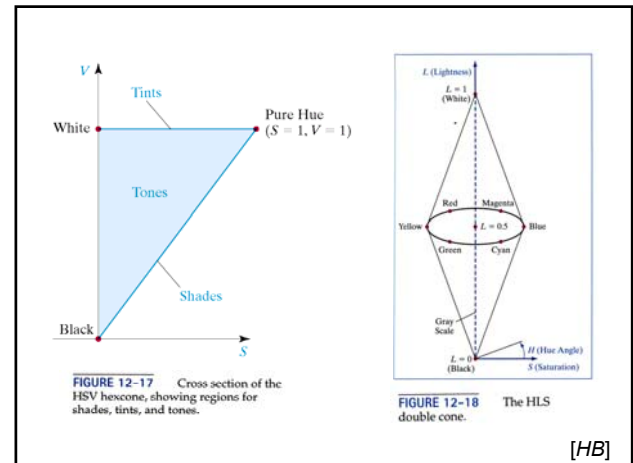
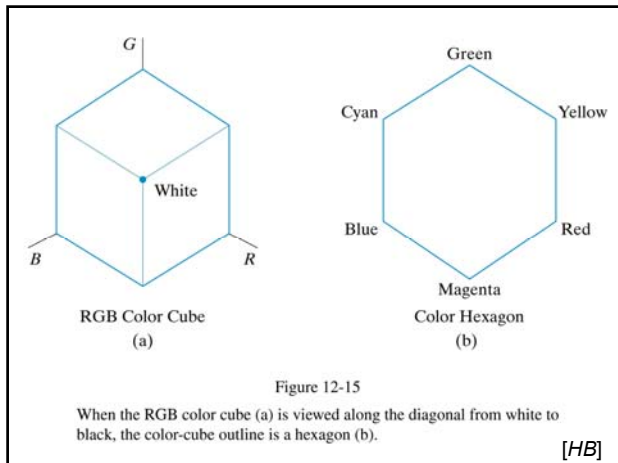


FIGURE 12-16

The HSV

[HB, FVD]



## Modeling the world

- Models describe shape, color, etc. of objects, and spatial relation between objects
  - Surface Modeling
  - Material properties (described later, for lighting)
  - Scene graph organization (described later)
- Some surface representation techniques:
  - Polygon meshes
  - Continuous functions: e.g. parametric or quadric (discussed later)
  - Subdivision surfaces

## Polygon Rendering

For now, we'll stick to polygons.

Polygon rendering:

Consider each polygon in a 3D scene, project it onto a 2D viewing plane, and use a scan conversion routine to draw it



## Basic Surface Description Components

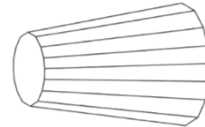
Geometry: specification of position of points (vertices) used

Topology: specification of connectivity of the points to form edges, polygons, meshes

Attributes: supplemental information, e.g., a color at each point

## Polygon meshes

- A collection of vertices, edges, and polygons with each edge shared by at most two polygons
- Piecewise linear approximations when used to model smooth surfaces
- Often using only triangles or quadrilaterals



[FVD]

## Polygon meshes

- How do we create a mesh?
  - Using a modeling application, or
  - by sampling points on a real object, or
  - procedurally (code-generated), or
  - manually specifying points and connectivity
- Next we'll look at basic representations for storing meshes in memory or passing them to the graphics system

## Lightwave Modeler application



(User edits control meshes that are not necessarily polygon meshes, but that can generate polygon meshes for rendering)

## Object scanning

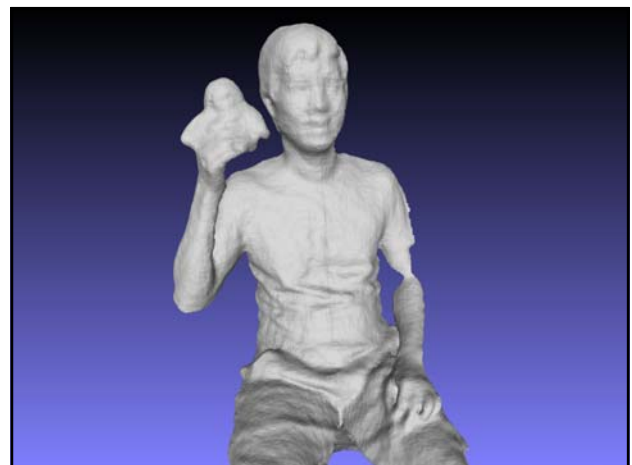


Top left: magnetically tracked wand

Left: free-hand laser scanner with magnetically tracked base

Top: Statue scanner (laser)

Scanners sample points on a surface; Points can be connected into a mesh

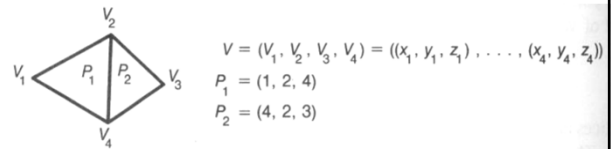


### Explicit representation of mesh (2018 note: not covered in lecture)

- List all polygons, each as a list of vertex coordinates  
 $P = ((x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_n, y_n, z_n))$
- Ok for a single polygon
- Inefficient for a mesh: duplicated vertices
  - Wastes memory or bandwidth
  - Extra time needed for duplicate transformations (e.g., when we want to move a point shared by multiple polygons)

### Pointers to a vertex list

- Store each vertex once in a *vertex list*  
 $V = ((x_1, y_1, z_1), \dots, (x_n, y_n, z_n))$
- Each polygon is then described by a list of indices referencing vertices



[FVD]

### Pointers to a vertex list (cont'd)

- Saves space (no duplicate vertices; an index is cheap relative to a 3D coord)
- Simpler to change coords of a vertex that is shared by multiple polygons
- Used with other lists that represent vertex properties: color, normals, texture coords, etc.

### Pointers to an edge list

Pointers to an edge list:

- Use vertex list plus an edge list that lists each edge as two vertex indices and can be augmented to reference the polygons using the edge
- Each polygon is now described by a list of indices referencing edges

### Pointers to an edge list (cont'd)

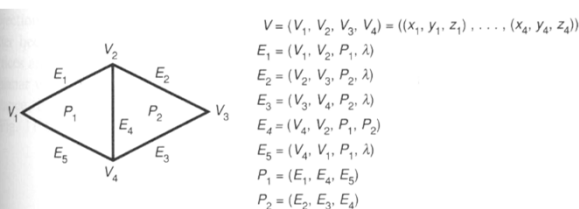
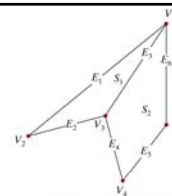


Fig. 11.4 Polygon mesh defined with edge lists for each polygon ( $\lambda$  represents null).

[FVD]



VERTEX TABLE	EDGE TABLE	SURFACE FACET TABLE
$V_1: x_1, y_1, z_1$ $V_2: x_2, y_2, z_2$ $V_3: x_3, y_3, z_3$ $V_4: x_4, y_4, z_4$ $V_5: x_5, y_5, z_5$	$E_1: V_1, V_2$ $E_2: V_2, V_3$ $E_3: V_3, V_4$ $E_4: V_4, V_5$ $E_5: V_5, V_1$ $E_6: V_5, V_1$	$S_1: E_1, E_2, E_3$ $S_2: E_3, E_4, E_5, E_6$

$E_1: V_1, V_2, S_1$ $E_2: V_2, V_3, S_1$ $E_3: V_3, V_4, S_1, S_2$ $E_4: V_4, V_5, S_2$ $E_5: V_5, V_1, S_2$ $E_6: V_5, V_1, S_2$
---

FIGURE 3-51 Edge table for the surfaces of Fig. 3-50 expanded to include pointers into the surface-facet table.

FIGURE 3-50 Geometric data-table representation for two adjacent polygon surface facets, formed with six edges and five vertices.

[HB]



### Pointers to an edge list (cont'd)

- Adding the edge list increases storage requirements but speeds up certain operations (give an example)
- Other representations (e.g., winged-edge) store even more information about connectivity to support search queries

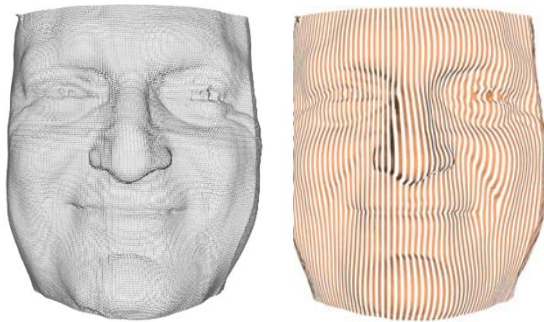
### Some notes about meshes and OpenGL

- It is often beneficial to use strip primitives such as triangle strips, which reduce the amount of information needed to describe a mesh structure.



[Red]

Stripification of a triangle mesh, producing 134 strips of 390 triangles each. The right image is rendered to show half the strips.  
From *The Visualization Toolkit*, 4th Edition, Schroeder, Martin, Lorensen, ©2006



[Aside: long strips that minimize vertex references may not produce optimal performance; performance-optimal stripification is nontrivial]

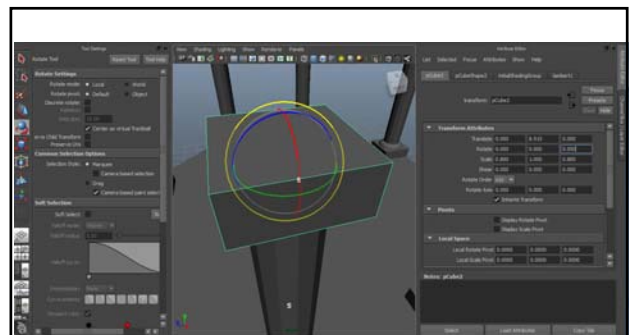
### Some notes about meshes and OpenGL

- Client-side Vertex arrays (deprecated) reduce function calls, but data must still be transferred to graphics system per draw
- Buffer Object: store vertex array in the graphics subsystem (server, graphics card) to reduce transfers.  
Homework: Vertex Array Object.
- Display List (deprecated): store a sequence of calls as a compiled module on the graphics system

CMPS 415/515

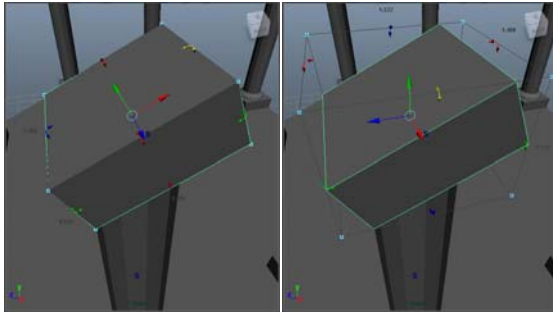
### Basic transformation arithmetic:

overview; translation, scaling, rotation, shear; homogeneous coordinates and transforms; composition (e.g. rotating about an arbitrary point)

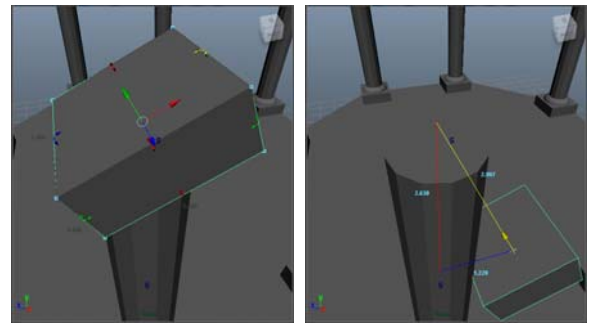


An object being transformed in a Modeler (Maya).

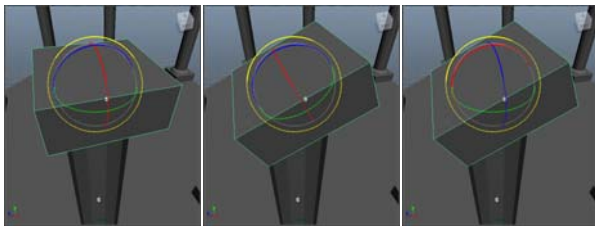
Left: Manipulator icons and options (rotation tool selected).  
Middle: Manipulator on object (interactive rings for rotation).  
Right: Object's transform in text form.



"Universal manipulator" in *local* (left) and *world* (right) modes.  
Transformations can act in different coordinate frames (e.g., move along object's own axis vs. move along a world axis direction).  
NOTE: Maya distinguishes between "local" and "object" frames, but we will not make this distinction (consider them the same for now).

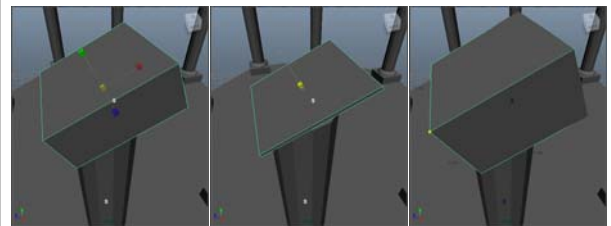


*Translation* changes *position*. Here, a box translated in a local Y direction (green manipulator axis, turns yellow when grabbed).  
At right, yellow shows translation path and amount, while red and blue show components along another frame's axes.



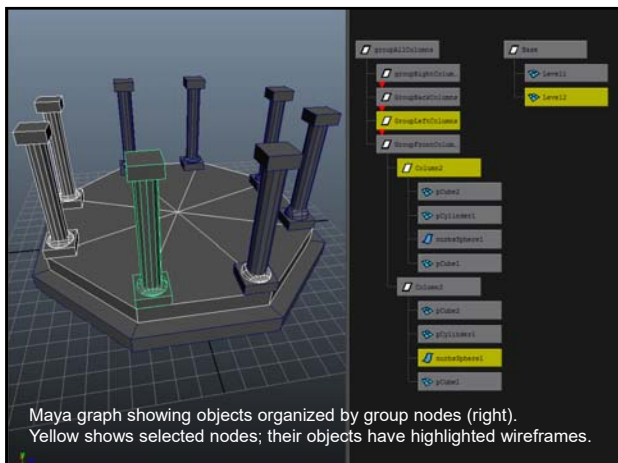
*Rotation* changes *orientation*.

Left: Rotation manipulator before rotation, in local mode.  
Middle: after local Z rotation (by grabbing blue ring).  
Right: manipulator changed from local to world mode, which aligns it to different axes.  
Note local and world Y axes (green) were the same before the rotation, but not after. Local axes rotate with the object.



*Scale* changes *size*.

Left: Scale manipulator in local mode.  
Middle: Object being scaled along local Y dimension.  
Right: Scale by grabbing corner in universal manipulator (local mode).



Maya graph showing objects organized by group nodes (right).  
Yellow shows selected nodes; their objects have highlighted wireframes.

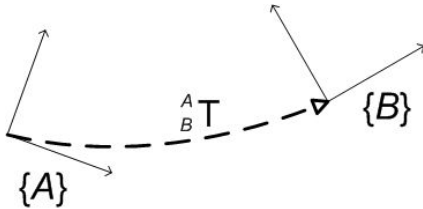
## Modeling Transforms

Determine position, orientation, scale, etc. of objects in the scene

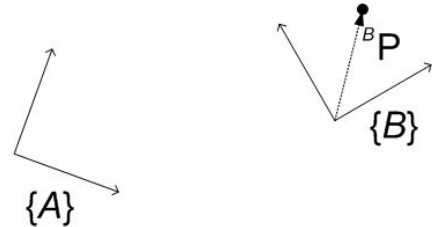
A transform can be viewed as:

- A description of a coordinate system with respect to another
- A mapping of coordinates from one coordinate system to another
- An operator that modifies objects

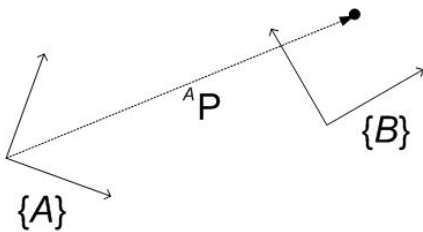
For two coordinate frames  $\{A\}$  and  $\{B\}$ , let  ${}^A_B T$  denote the transform describing frame  $\{B\}$  with respect to frame  $\{A\}$ . Commonly, it is a matrix describing position and orientation of  $\{B\}$  wrt  $\{A\}$ .



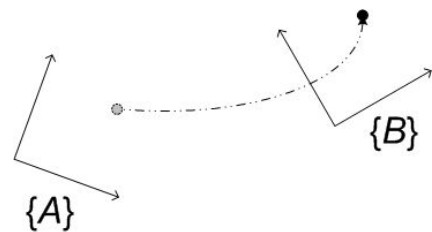
If  ${}^B P$  is a point having coordinates given with respect to  $\{B\}$ ...



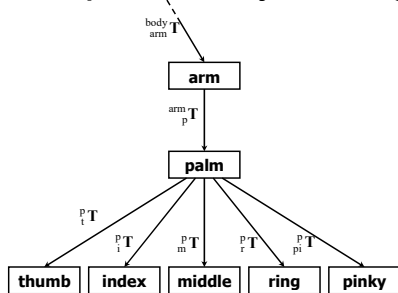
We compute the point's coordinates with respect to  $\{A\}$  as:  ${}^A P = {}^A_B T {}^B P$



We can also view the transform as an operator that modifies (moves, rotates, etc.) an object, with an object-attached coordinate system at  $\{A\}$  before the operation, and at  $\{B\}$  after

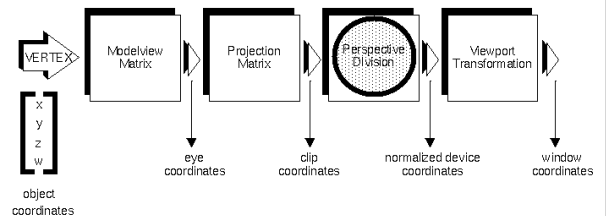


## Relationship between objects as a graph



Objects arranged hierarchically with transforms between them. Coordinates can be converted between any two nodes' frames using a sequence of transforms on a path between them (may need inverses). During rendering, a camera-to-object path gives a *modelview* matrix that converts from object coordinates to eye coordinates.

## Coordinate systems and transforms



[Red]

### Note about conventions used

- Coordinates written as column vectors
- Right-handed coordinate systems
- Warning: not all books and tools use these conventions. Other conventions result in transforms appearing transposed, multiplication order appearing reversed, and/or elements being negated.

### 2D Translation as vector addition

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}, \text{ or } P' = P + T$$

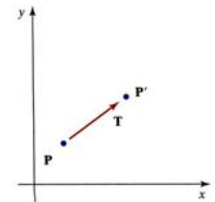


FIGURE 5-1 Translating a point from position  $P$  to position  $P'$  using a translation vector  $T$ .

[HB]

### 2D Translation as vector addition

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}, \text{ or } P' = P + T$$

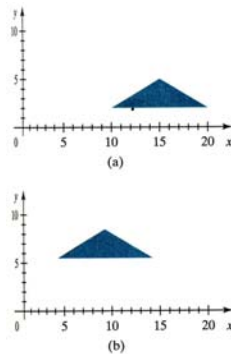


FIGURE 5-2 Moving a polygon from position (a) to position (b) with the translation vector  $(-5.50, 3.75)$ .

[HB]

### 2D Scaling

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}, \text{ or } P' = S \cdot P$$

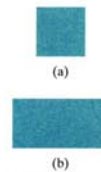


FIGURE 5-6 Turning a square (a) into a rectangle (b) with scaling factors  $s_x = 2$  and  $s_y = 1$ .

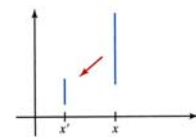


FIGURE 5-7 A line scaled with Eq. 5-12 using  $s_x = s_y = 0.5$  is reduced in size and moved closer to the coordinate origin.

[HB]

### 2D Scaling

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}, \text{ or } P' = S \cdot P$$

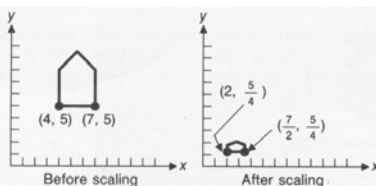


Fig. 5.2 Scaling of a house. The scaling is nonuniform, and the house changes position.

[FVD]

### Rotation of a point about the origin

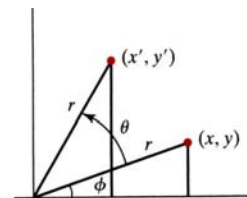


FIGURE 5-4 Rotation of a point from position  $(x, y)$  to position  $(x', y')$  through an angle  $\theta$  relative to the coordinate origin. The original angular displacement of the point from the  $x$  axis is  $\phi$ .

$$\begin{aligned} x &= r \cdot \cos \phi, & y &= r \cdot \sin \phi, \\ x' &= r \cdot \cos(\theta + \phi) = r \cdot \cos \phi \cdot \cos \theta - r \cdot \sin \phi \cdot \sin \theta, \\ y' &= r \cdot \sin(\theta + \phi) = r \cdot \cos \phi \cdot \sin \theta + r \cdot \sin \phi \cdot \cos \theta \\ \Rightarrow x' &= x \cdot \cos \theta - y \cdot \sin \theta, & y' &= x \cdot \sin \theta + y \cdot \cos \theta \end{aligned}$$

[HB]

## 2D Rotation (about origin)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}, \text{ or } P' = R \cdot P$$

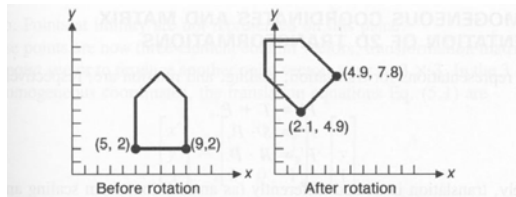


Fig. 5.3 Rotation of a house. The house also changes position.

Note: positive angles are counterclockwise

[FVD]

## Shear

$P' = SH \cdot P$ , where  $SH$  is one of :

$$SH_x = \begin{bmatrix} 1 & sh_x \\ 0 & 1 \end{bmatrix}, \text{ or } SH_y = \begin{bmatrix} 1 & 0 \\ sh_y & 1 \end{bmatrix}$$

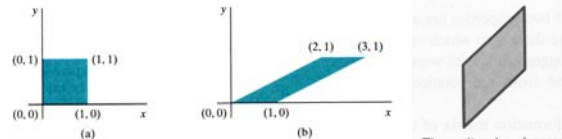


FIGURE 5-23 A unit square (a) is converted to a parallelogram (b) using the x-direction shear matrix 5-57 with  $sh_x = 2$ .

The unit cube sheared in the y direction

[HB, FVD]

## Homogeneous coordinates

- Add an extra coordinate:  $(x, y, W)$
- $(x, y, W)$  represents the point with 2D Cartesian coordinates  $(x/W, y/W)$
- So, all triples  $(tx, ty, tW)$  represent the same 2D point (nonzero  $t$ )
- Can *homogenize* the point by computing  $(x/W, y/W, 1)$
- For most of this class,  $W$  is 1. Exceptions:
  - points after perspective projection
  - $W = 0$  used for free vectors (directions)

## Note

In scientific or academic language:

- Homogeneous means uniform
- Homogenous means homologous : of common ancestry (especially biologically)
- Homogenize means to make homogeneous

## Homogeneous transforms

- Homogeneous form supports all common modeling transformations as matrix multiplication

$$P' = M \cdot P$$

- So, transformation sequences can be composed into a single transformation matrix using matrix multiplication

$$P' = (M_3 \cdot M_2 \cdot M_1) \cdot P$$

## Homogeneous transforms for 2D

Translation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

## Homogeneous transforms for 2D (cont'd)

Shear:

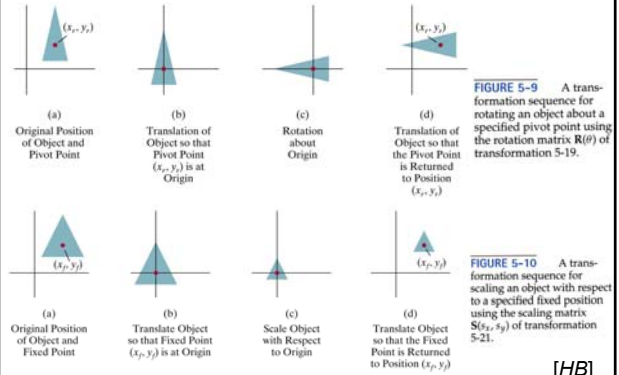
in x direction:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

in y direction:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

## Composite transformations



## Composing (combining) transforms

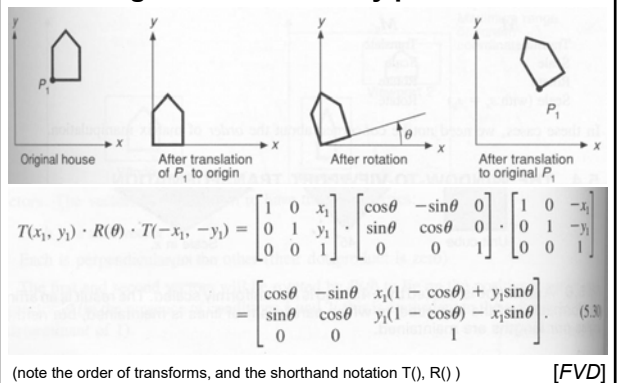
- Example: rotating about an arbitrary point

- 1) Translate P to origin ( $T$ )
- 2) Rotate about origin ( $R$ )
- 3) Translate back ( $T^{-1}$ )

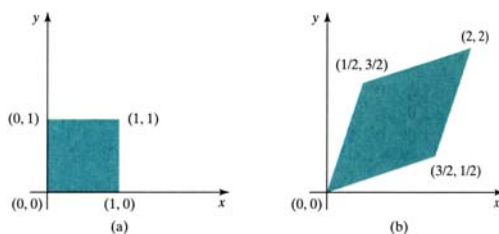
$$P' = M \cdot P, \quad M = T^{-1} \cdot R \cdot T$$

- Objects often have many vertices. Transforming them with one composite matrix is faster than applying multiple transformations to each.

## Rotating about an arbitrary point

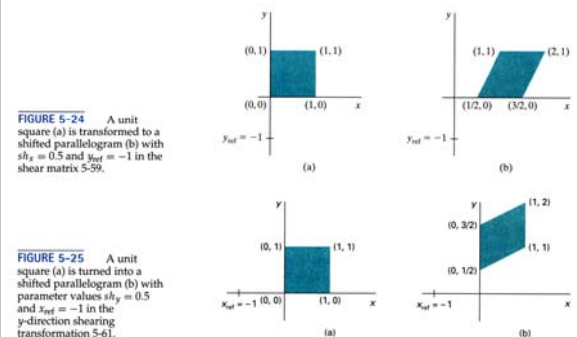


Give a composite transform for (a) to (b):



[HB]

Give a composite transform for (a) to (b):



## Composition (cont'd)

- Transformations are not commutative in general

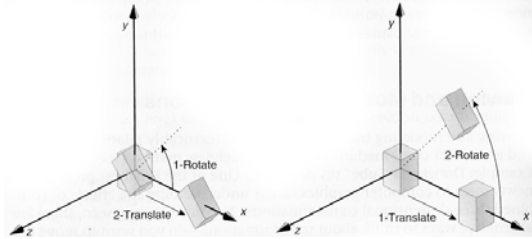


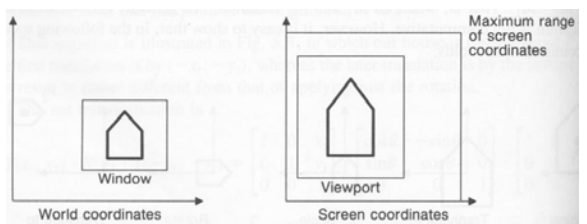
Figure 3-4 from "red book" 4<sup>th</sup> edition, rotating first vs. translating first [Red]

## Composition (cont'd)

- In special cases, transformations are commutative;  $M_1 \cdot M_2 = M_2 \cdot M_1$  when
  - both are translations
  - both are scaling transforms
  - both are rotations *about the same axis*
  - one is a rotation and one is a *uniform* scaling
  - one is a rotation and the other is a translation parallel to the rotation axis
  - for some other more elaborate cases and trivial cases (e.g., one is identity)

## Window-to-Viewport transformation

Map a world-coordinate *window* to a *viewport* on the output device



[FVD]

## Window-to-Viewport transformation

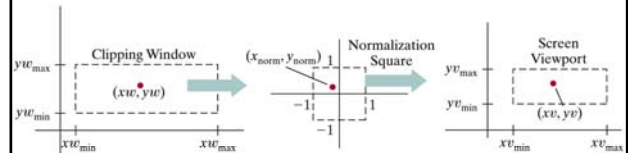


Figure 6-8

A point  $(x_w, y_w)$  in a clipping window is mapped to a normalized coordinate position  $(x_{norm}, y_{norm})$ , then to a screen-coordinate position  $(x_v, y_v)$  in a viewport. Objects are clipped against the normalization square before the transformation to viewport coordinates.

What are the matrices?

[HB]

## 3D transforms

- Assume a right-handed coordinate system
- Positive rotations are counterclockwise (when rotation axis points "at you")
- $(x, y, z, W)$  represents the point  $(x/W, y/W, z/W)$  using homogeneous coordinates

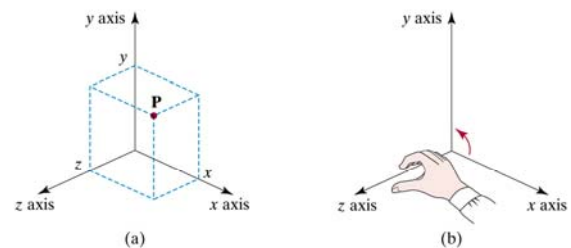
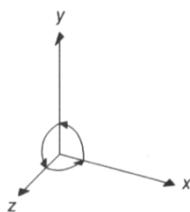
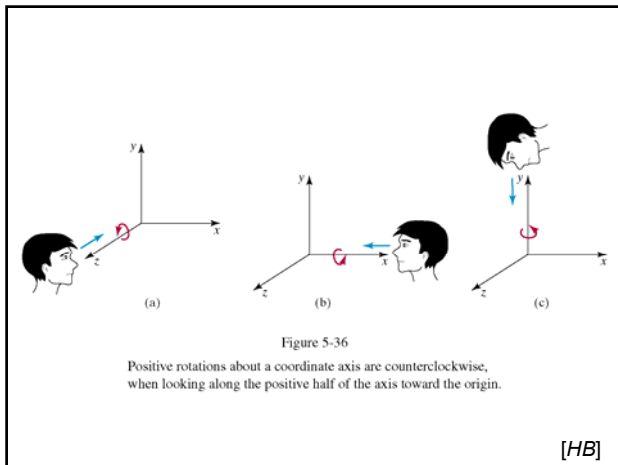


Figure A-6

Coordinate representation for a point  $P$  at position  $(x, y, z)$  in a standard right-handed Cartesian reference system.

[HB]



### Translation

2D:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3D:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

[HB]

### Scale

2D:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3D:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

### Shear

2D, x:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3D, (x, y):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

### Rotation

2D:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3D, about z axis:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

### Rotation (cont'd)

about x axis:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

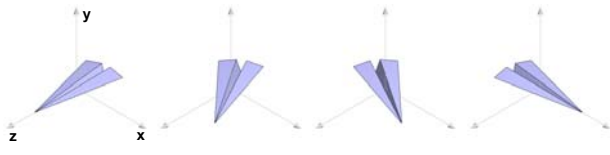


## Rotation (cont'd)

about y axis:

$$\begin{bmatrix} x' \\ y \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

(We will consider an arbitrary axis later)



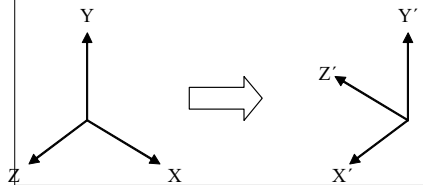
## Properties of a rotation matrix

(ignoring the last row and column in the case of a homogeneous transform)

- It is “orthonormal”
  - Each row (column) is a unit vector
  - The rows (columns) are mutually orthogonal
  - Consequently, the inverse is the transpose
- Its columns are the local x, y, and z axes of the rotated object described w.r.t. the unrotated frame

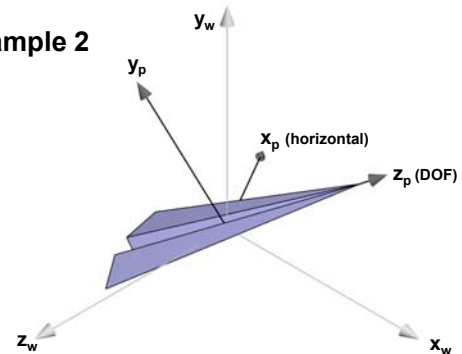
## Example 1

Rotate  $-90^\circ$  about y axis



(write the transform using the property that its columns are the rotated axes described w.r.t. the original frame)

## Example 2



Call this airplane's local frame  $\{p\}$  and the unrotated world frame  $\{w\}$ . Given direction of flight DOF (expressed in  $\{w\}$ ), and assuming axis  $x_p$  stays horizontal (perpendicular to axis  $y_w$ ), how can we build a rotation matrix describing plane orientation, i.e.,  $\{p\}$  w.r.t.  $\{w\}$ ?

## Resulting rotation matrix

${}^w_p R$  = Rotation matrix describing  $\{p\}$  with respect to  $\{w\}$

$$= \begin{bmatrix} | & | & | & 0 \\ x_p & y_p & z_p & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} | & | & | & 0 \\ y_w \times DOF & DOF \times (y_w \times DOF) & DOF & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where  $v/|v|$  normalizes vector  $v$ , and all vectors are w.r.t.  $\{w\}$

## Rigid body transform properties

Any composed sequence of translations and rotations results in the form:

$$\begin{bmatrix} R & T \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which is a rigid body transform (preserves angles and lengths) and corresponds to a single rotation by  $R$  followed by a single translation by  $T$

### Transform properties (cont'd)

Its inverse is

$$\begin{bmatrix} R^T & -R^T T \\ 0 & 1 \end{bmatrix}$$

Since  $R^T = R^{-1}$  for rotation,  
and  $-T$  w.r.t. the rotated system is  $(R^T)(-T)$

### Rigid body transformation of free vectors

So far, this has been about transforming points (positions)

If transforming a direction (e.g., velocity vectors, surface normals for shading), then apply only the rotation  $R$ , no translation. (Why?)

To represent vectors in the homogeneous system, we set  $W = 0$  for vectors so the translation won't be applied.

### Adding uniform scale

Any composed sequence of translations, rotations, and *uniform nonzero* scale results in form:

$$\begin{bmatrix} R \cdot S & T \\ 0 & 1 \end{bmatrix}$$

which is a type of affine transform (preserves ratios of lengths along a line) and corresponds to a uniform scale  $S$  followed by rotation  $R$  followed by a translation  $T$

### Adding uniform scale (cont'd)

And its inverse is

$$\begin{bmatrix} S^{-1} R^T & -S^{-1} R^T T \\ 0 & 1 \end{bmatrix}$$

Note: Each diagonal of  $S^{-1}$  is the reciprocal of scale factor in  $S$ , which is the length of any one of the first three columns of the previous slide's matrix. So, each row of  $(S^{-1} R^T)$  is a column of  $R$  multiplied by this reciprocal.

### 3-Angle Set representation of orientation

- Any orientation can be achieved by composing 3 rotations about principal axes
- There are 24 different 3-angle set conventions: 12 Euler angle sets and 12 fixed-axis angle sets
  - Euler angles: the three axes rotate with the object
  - Fixed-axis angles: the three axes do not rotate

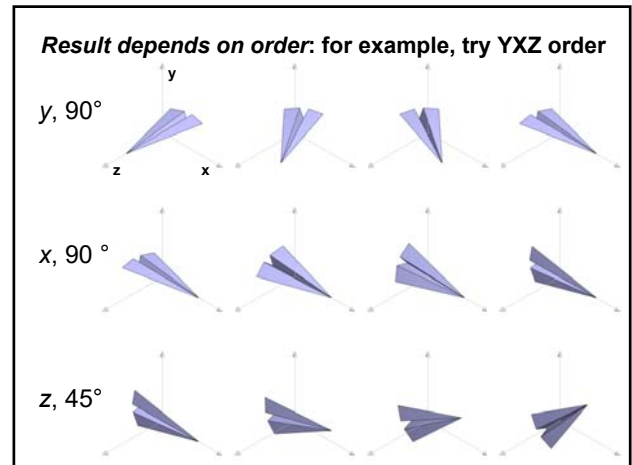
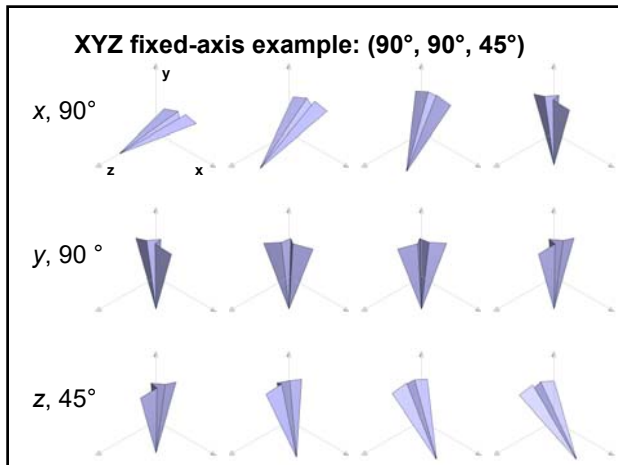
### Fixed-axis angle example

X-Y-Z fixed-axis angles: Rotations are performed in X, Y, Z order using axes fixed with respect to the initial orientation

- 1) Rotate about x axis by angle  $\theta$
- 2) Rotate about y axis by angle  $\phi$
- 3) Rotate about z axis by angle  $\alpha$

$$P' = [R_z(\alpha) \cdot R_y(\phi) \cdot R_x(\theta)] \cdot P$$

(note the order of transforms, and the shorthand notation for matrices)



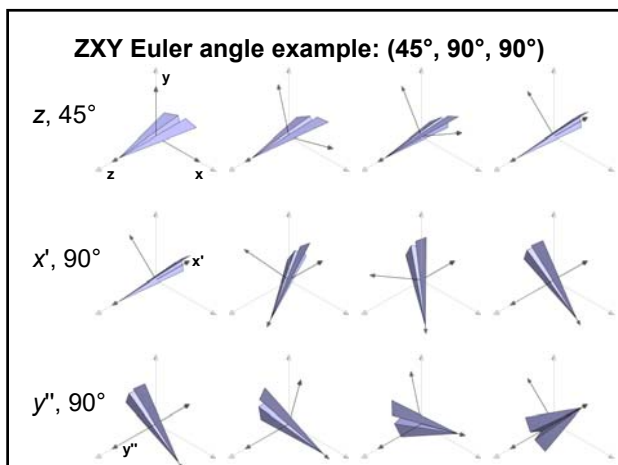
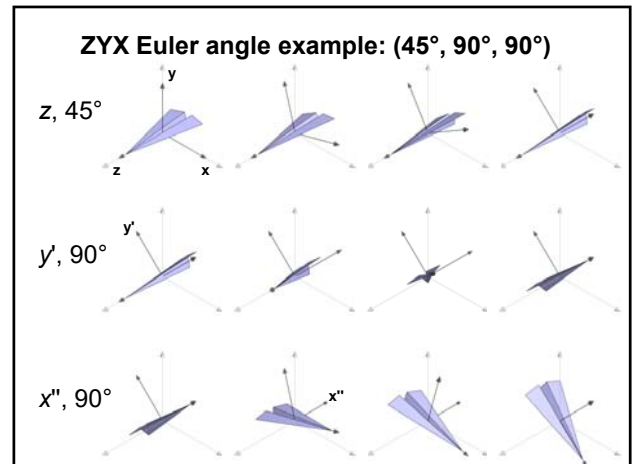
**Euler angle example**

Z-Y-X Euler angles: Rotations are performed in Z, Y, X order using rotating axes

- 1) Rotate about z axis by angle  $\alpha$
- 2) Rotate about y' axis by angle  $\phi$
- 3) Rotate about x'' axis by angle  $\theta$

$$P' = [R_z(\alpha) \cdot R_y(\phi) \cdot R_x(\theta)] \cdot P$$

(Note: An Euler angle set gives the same result as the fixed-axis set with opposite order of axes and angles. So, the above matches X-Y-Z fixed-axis angles.)

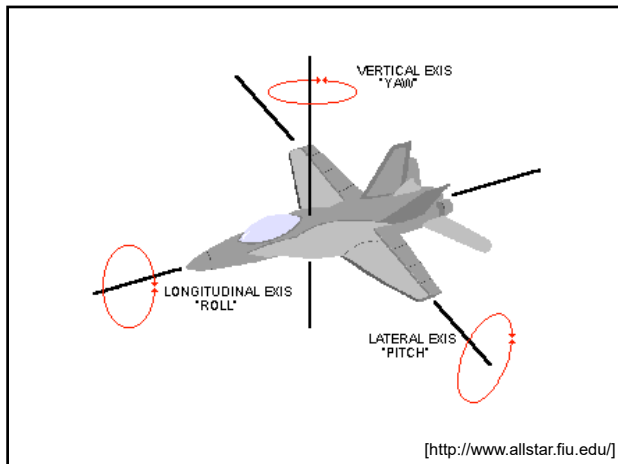
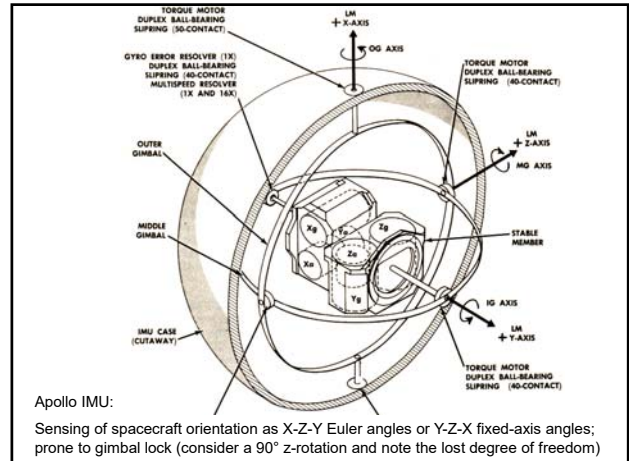


**Notes**

- The 12 choices for angle order are:  
X-Y-Z, X-Y-X, Y-Z-X, Y-Z-Y, Z-X-Y, Z-X-Z,  
X-Z-Y, X-Z-X, Y-X-Z, Y-X-Y, Z-Y-X, Z-Y-Z
- Sometimes, the distinction between “Euler angles” and “fixed-axis angles” is not made or is not clearly stated.  
(example: Unity 4 manual states ZXY order for Euler set, but it acts like ZXY fixed axis angles)

### 3-angle sets (Euler and fixed-axis angles)

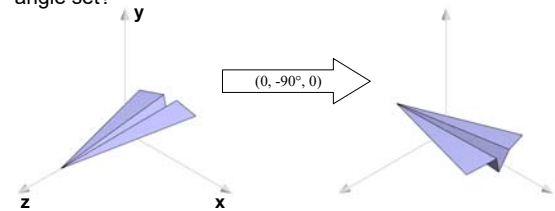
- 3-angle sets seem conceptually simple, until we consider certain problems
- Problems because the 3 rotations aren't independent
  - One axis can be rotated into another, causing a lost degree of freedom ("gimbal lock").
  - Multiple solutions with very different angles e.g.  $(180^\circ, 0, 0) = (0, 180^\circ, 180^\circ)$  for X-Y-Z
  - Interpolations of orientation can look strange and depend on choice of angle set convention



### A Question

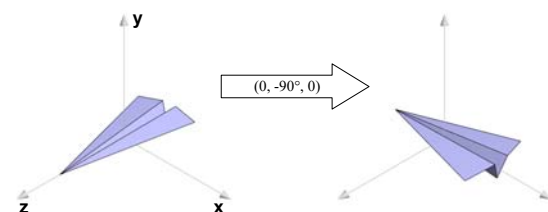
If we choose X-Y-Z fixed-axis angles to store orientation of the plane below, and angles become  $(0, -90^\circ, 0)$ ,

Then how can we roll the already-rotated plane with this 3-angle set?

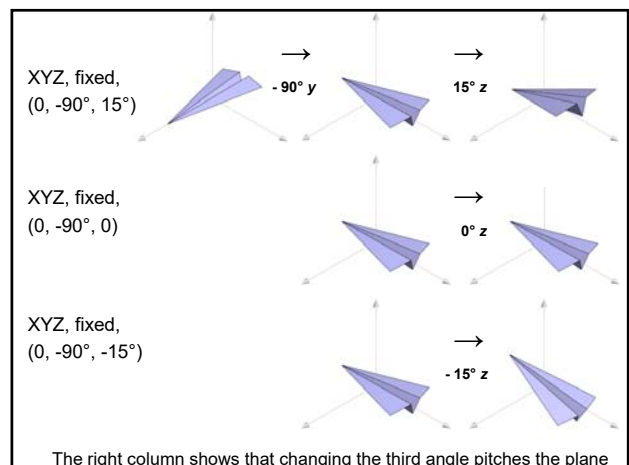


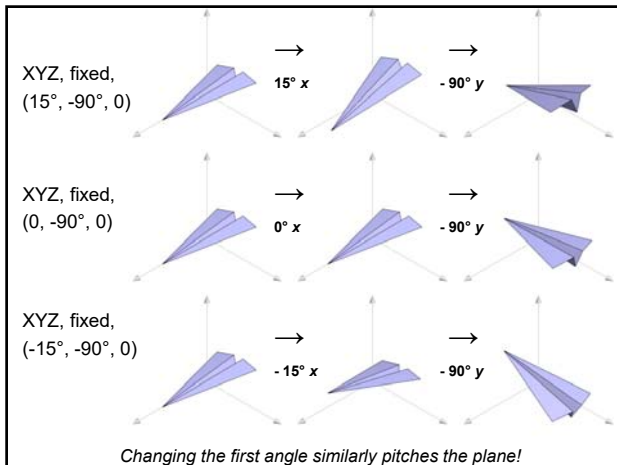
### Gimbal lock example

If we use X-Y-Z fixed-axis angles to store orientation of the plane below, and angles become  $(0, -90^\circ, 0)$ , then no simple change in an angle is a roll of the rotated plane!  
(Such singularities exist for all 3-angle sets)



The second angle yaws; what is the effect of a change in the other angles?

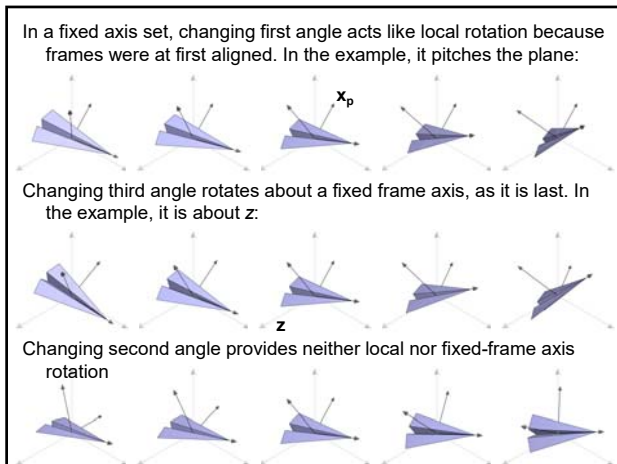
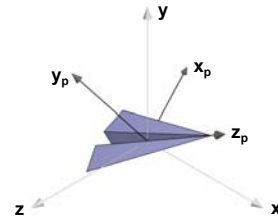




### Another problem

(still assuming X-Y-Z fixed-axis set:)

For arbitrary plane orientation ( $\theta, \phi, \alpha$ ), you can rotate about  $x_p$  axis by changing  $\theta$ , or rotate about fixed  $z$  by changing  $\alpha$ . But, what change rotates about other principal axes? It's not straightforward.

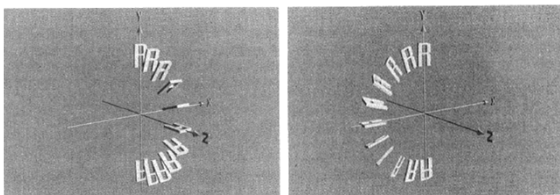


For a sequence of composed rotation matrices, the rightmost one acts in the local frame, and the leftmost one acts in the world (or parent) frame.

(for our notational conventions)

### 3-angle interpolation

Interpolation results vary depending on the choice of angle set convention and on which of multiple values representing the same orientation is used



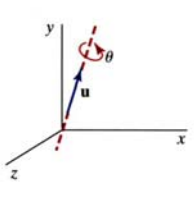
[Interpolation results from two different 3-angle sets or two different solutions with the same 3-angle set, *Advanced Animation and Rendering Techniques*, Watt and Watt, ©1992]

### 3-angle sets (cont'd)

- 3-angle sets are not simple after all
- They are good in some applications, e.g., when duplicating a physical mechanism that actually behaves this way.
- Otherwise, use something else (often stick to matrix form as main representation)

### Angle-axis representation

- Any orientation can be achieved by a single rotation about some axis  $\mathbf{u}$



$$R_{\mathbf{u}}(\theta) = \begin{bmatrix} u_x u_x v + c & u_x u_y v - u_z s & u_x u_z v + u_y s & 0 \\ u_x u_y v + u_z s & u_y u_y v + c & u_y u_z v - u_x s & 0 \\ u_x u_z v - u_y s & u_y u_z v + u_x s & u_z u_z v + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

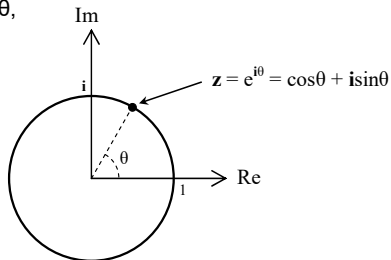
where:  
 $c = \cos \theta$ ,  $s = \sin \theta$ ,  $v = 1 - \cos \theta$ , and  
 $u_x, u_y, u_z$  are the components of unit vector  $\mathbf{u}$

### $\theta \mathbf{u}$ : “rotation vector” or “exponential map”

- Given the unit axis description  $\mathbf{u}$  and the rotation amount  $\theta$ , we can scale  $\mathbf{u}$  by  $\theta$  to encode both direction and angle into a single 3-element vector
- We will not study this further, as it is the least common of the mentioned methods

### Reminder: complex numbers

- complex number:  $\mathbf{z} = a + ib$ , where  $i^2 = -1$
- for unit complex numbers,  $a^2 + b^2 = 1$ 
  - so for some  $\theta$ ,  
 $a = \cos \theta$ ,  
 $b = \sin \theta$



- For 2D, Suppose a rotation by  $\theta$  is represented by the number  $\mathbf{z} = a + ib$ , where  $a = \cos \theta$  and  $b = \sin \theta$ , then
  - any unit complex number represents a rotation
  - multiplication composes rotations: rotation  $\mathbf{z}_1$  followed by rotation  $\mathbf{z}_2$  is the same as the single rotation  $\mathbf{z}_3 = \mathbf{z}_2 \cdot \mathbf{z}_1$
  - the inverse of rotation  $\mathbf{z}$  is represented by  $\mathbf{z}$ 's complex conjugate (negate  $b$ )

- How do we use related concepts for 3D?

### Quaternions: definition

Quaternion  $\mathbf{q} = s + i\mathbf{a} + j\mathbf{b} + k\mathbf{c}$ ,  
 where  $i^2 = j^2 = k^2 = -1$ ,  $ij = k$ ,  $ji = -k$

Notes:

$i, j, k$  are not vectors, they are imaginary units defined above.

Do not be discouraged if you cannot derive or explain this from any number system you know: it is a definition of a different system.

### Unit quaternions (a.k.a. Euler parameters)

Rotation by  $\theta$  about unit axis  $\mathbf{u}$  is represented by the unit quaternion:

$$\mathbf{q} = \cos(\theta/2) + i u_x \sin(\theta/2) + j u_y \sin(\theta/2) + k u_z \sin(\theta/2)$$

(unit means  $s^2 + a^2 + b^2 + c^2 = 1$ )

Graphics/game APIs often use names  $(x, y, z, w)$  instead of  $(a, b, c, s)$ . Also note element order.

## Basic Operations

- Sometimes written  $\mathbf{q} = (s, \mathbf{v})$  :  $s$  is the “real” part (scalar) and  $\mathbf{v}$  “imaginary” part ( $[a \ b \ c]^t$ )
- Quaternion algebra derived from definition

for  $\mathbf{q}_1 = (s_1, \mathbf{v}_1)$ ,  $\mathbf{q}_2 = (s_2, \mathbf{v}_2)$ ,

addition :  $\mathbf{q}_1 + \mathbf{q}_2 = (s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2)$

multiplication :  $\mathbf{q}_1 \cdot \mathbf{q}_2 = (s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2)$

conjugate :  $\mathbf{q}^* = (s, -\mathbf{v})$  for unit quaternion  $\mathbf{q}$

## Quaternions (cont'd)

- multiplication composes rotations
- conjugate represents inverse rotation
- $\mathbf{q}$  and  $-\mathbf{q}$  represent same orientation
- point  $\mathbf{p}$  can be rotated by  $\mathbf{q}$  using:  
 $(0, \mathbf{p}') = \mathbf{q} \cdot (0, \mathbf{p}) \cdot \mathbf{q}^*$  ; shorthand notation:  $\mathbf{q} \cdot \mathbf{p}$   
 or by first converting  $\mathbf{q}$  to a rotation matrix:

$$R = \begin{bmatrix} 1 - 2(b^2 + c^2) & 2(ab - sc) & 2(ac + sb) & 0 \\ 2(ab + sc) & 1 - 2(a^2 + c^2) & 2(bc - sa) & 0 \\ 2(ac - sb) & 2(bc + sa) & 1 - 2(a^2 + b^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Quaternion / angle-axis conversion

- Easy to convert to/from a user-friendly form:
  - Conversion from angle-axis to quaternion: just plug angle and unit axis into

$$\mathbf{q} = \cos(\theta/2) + i u_x \sin(\theta/2) + j u_y \sin(\theta/2) + k u_z \sin(\theta/2)$$

- Conversion to angle-axis from quaternion:

Let  $t = \sqrt{1 - s^2}$ , or 1 when  $s = 1$ ,

then  $u_x = a / t$

$u_y = b / t$

$u_z = c / t$

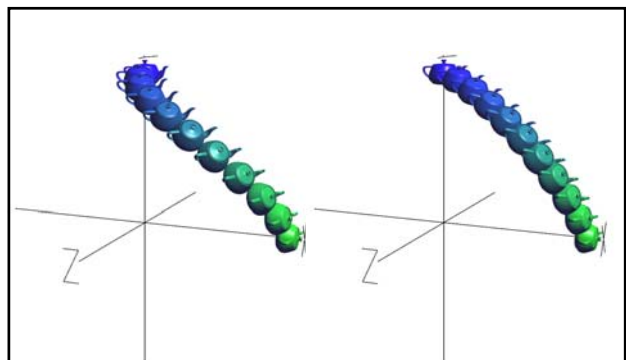
$\theta = 2 \arccos(s)$

## 3-angle set to quaternion conversion

- Converting a 3-angle set to a quaternion:
  - for example, consider X-Y-Z fixed axis set
  - Each of the 3 rotations can be represented as a quaternion by plugging the axis and angle into the formula given above
  - Combine these 3 quaternions using multiplication, so the final quaternion representation is  $\mathbf{q} = \mathbf{q}_{z,\alpha} \mathbf{q}_{y,\phi} \mathbf{q}_{x,\theta}$

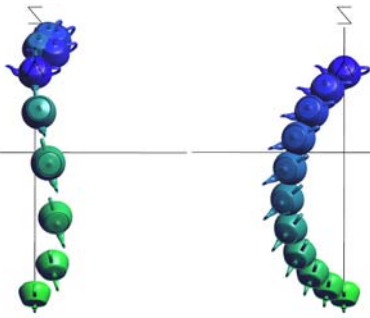
## Why are quaternions useful?

- Overcome shortcomings of 3-angle sets
- Unlike 3-angle sets, angle-axis, or exponential maps, a basic algebraic operation provides composition and rotation (multiplication)
- Sometimes more efficient than matrices:
  - Smaller (saves memory or bandwidth)
  - Faster composition for rotation sequences
  - Quaternion renormalization is simpler than matrix reorthogonalization (used to prevent problems of error accumulation in animation or physics tools)
- Good for interpolation



## 3-angle interpolation vs. quaternion interpolation

A teapot initially centered at coordinate (0,1,0) was rotated about the origin by X-Y-Z fixed-axis angles  $90^\circ, 90^\circ, 0^\circ$ , with intermediate configurations generated by interpolation in 3-angle set space (left) and quaternion space (right).



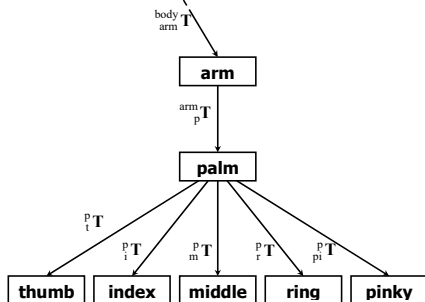
### 3-angle interpolation vs. quaternion interpolation

Here, the X-Y-Z fixed-axis angles were 90°, 180°, 90°.  
(images from a CMPS 515 Spring 2003 assignment)

### Object Hierarchy

- Construct complex objects in hierarchical fashion, often using repeated building blocks
- Object hierarchy represented as a DAG
- Objects may inherit properties of parents
- Instancing: use of transformed “master” objects
- Update propagation of building block changes

### Relationship between objects as a graph



(an example with modeling transforms associated with graph edges)

### Traversing the tree during rendering

Suppose each node stores the “incoming” (parenting) transform and object geometry

#### NodeRender:

1. Add node’s transform to modeling matrix (postmultiply)
2. Draw objects stored in node (if any)
3. Recurse (call NodeRender for each child node)
4. Restore previous modeling matrix

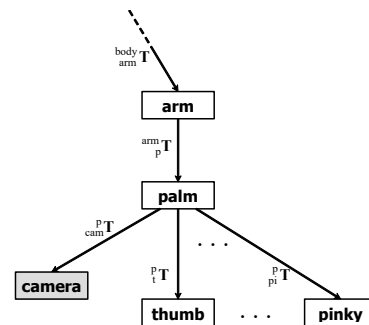
In effect, step 4 adds the inverse transform of step 1.  
With a matrix stack, step 4 can be done by popping if a push is added before step 1.

```

...
PushMatrix()
add  $body_{arm} T$  to modelview      ( $M = \dots body_{arm} T$ )
draw arm
PushMatrix()
add  $arm_p T$  to modelview          ( $M = \dots body_{arm} T \cdot arm_p T$ )
draw palm
PushMatrix()
add  $p_t T$  to modelview            ( $M = \dots body_{arm} T \cdot arm_p T \cdot p_t T$ )
draw thumb
PopMatrix()                      ( $M = \dots body_{arm} T \cdot arm_p T$ )
... (repeat these 4 for index, middle, ring, pinky)
PopMatrix()                      ( $M = \dots body_{arm} T$ )
PopMatrix()                      ( $M = \dots$ )
...

```

### How do we handle a camera node?



(here, a camera is attached to the palm)



## Addition of a camera (eye) node

- *View matrix* combines transforms along the path from camera to world node, giving world w.r.t. eye.

- Can move “up” (against an arrow) using a transform’s inverse. View matrix is inverse of camera w.r.t. world.

Intuition: the impression of moving (transforming) a camera is created by applying the inverse transform to the rest of the scene.

Example: moving a camera right 1 unit gives the same image as moving the rest of the world left 1 unit.

- For rendering, vertices are transformed by *modelview*, i.e., object w.r.t. eye, or path from camera to object. Order is:

$$\text{eye} \xrightarrow{\mathbf{T}} \text{world} \xrightarrow{\mathbf{T}} \text{object}$$

- (In our conventions: eye coordinate system’s x-axis points to viewer’s right, y-axis up, z-axis towards viewer)

## A more explicit description of modelview use

$${}^e P = {}^e T {}^w P = [({}^w C)^{-1} {}^w M] {}^w P, \text{ where}$$

${}^e P$ : transformed point (w.r.t. eye coordinate system  $\{e\}$ )

${}^w P$ : untransformed point (w.r.t. its local coordinate system  $\{l\}$ )

${}^e T$ : the modelview matrix, which can be seen as a description of  $\{l\}$  w.r.t.  $\{e\}$

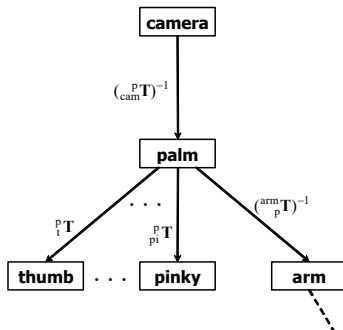
${}^w C$ : a camera description w.r.t. the world ( $\{w\}$ ); note its inverse  $({}^w C)^{-1}$  is view matrix  ${}^e C$  and describes  $\{w\}$  w.r.t.  $\{e\}$

${}^w M$ : model matrix: a description of  $\{l\}$  w.r.t.  $\{w\}$  that may be built by combining multiple modeling transforms

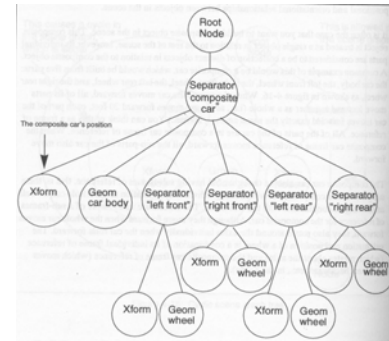
Warning: Books may use  $\mathbf{M}$  for either modelview or model only.

$\mathbf{V}$  is usually used for view matrix  ${}^e C$ .

This could be viewed as a restructuring: camera becomes root, transforms invert



## Example of a WorldToolKit scene graph



(an example from a scene graph tool with a special node type for transforms)

## Updating transforms in scene graph

For clarity, suppose we only need rigid body transforms.

Per object in the scene, the graph can store pose (position and orientation) with respect to a parent object or coord. system as a homogeneous transform.

## Letting the transform be called $\mathbf{A}$ :

Translation by  $\mathbf{T}$  interpreted w.r.t. to parent frame:

$$\mathbf{A} \leftarrow \mathbf{T} \cdot \mathbf{A}$$

Translation by  $\mathbf{T}$  interpreted w.r.t. local frame:

$$\mathbf{A} \leftarrow \mathbf{A} \cdot \mathbf{T}$$

Rotation by  $\mathbf{R}$  interpreted w.r.t. parent frame:

$$\mathbf{A} \leftarrow \mathbf{R} \cdot \mathbf{A}$$

Rotation by  $\mathbf{R}$  interpreted w.r.t. local frame:

$$\mathbf{A} \leftarrow \mathbf{A} \cdot \mathbf{R}$$

[‘←’ is assignment operator, ‘·’ is multiplication]

- A transform added on the right acts w.r.t. the local frame, while one added on the left transforms w.r.t. a parent frame
- So, read a transform sequence right-to-left if you think of transforms acting in a fixed (parent) frame, or left-to-right if you think of them acting w.r.t. a moving (local) frame

(in our notational convention that writes points as column vectors)

We can speed up the matrix multiplication:

$\mathbf{A} \leftarrow \mathbf{T} \cdot \mathbf{A}$  : we know orientation won't change, so modify only  $\mathbf{A}$ 's fourth column, by adding translation amount

$\mathbf{A} \leftarrow \mathbf{A} \cdot \mathbf{T}$  : update fourth column by adding  $\mathbf{R} \cdot \mathbf{t}$ , where  $\mathbf{t}$  is the translation vector and  $\mathbf{R}$  is the upper left  $3 \times 3$  part of  $\mathbf{A}$

$\mathbf{A} \leftarrow \mathbf{R} \cdot \mathbf{A}$  : not much speedup... known last row

$\mathbf{A} \leftarrow \mathbf{A} \cdot \mathbf{R}$  : we know the frame position doesn't change and we know the last row, so process only the upper left  $3 \times 3$  components of the transforms

Variations of the third case:

Rotation about an axis having direction defined in the parent frame, but about local origin: don't update fourth column (position maintained as seen in some modeling programs' manipulators)

Rotation of local origin about a parent-defined axis, but maintaining relative orientation of coordinate systems: update only the fourth column (think of Ferris wheel pods)

[Optional] The transform could instead be stored as position-quaternion pair  $(\mathbf{p}, \mathbf{q})$  meaning rotation  $\mathbf{q}$  followed by translation  $\mathbf{p}$ :

Translation by  $\mathbf{v}$  interpreted w.r.t. to parent frame:

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}$$

Translation by  $\mathbf{v}$  interpreted w.r.t. to local frame:

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{q} \cdot \mathbf{v}, \text{ note } \mathbf{q} \cdot \mathbf{v} \text{ is shorthand for } \mathbf{q} \cdot (0, \mathbf{v}) \cdot \mathbf{q}^* \text{ and only the vector part of this quaternion multiplication result gets added to } \mathbf{p}$$

Rotation by  $\theta$  about axis  $u$  interpreted in parent frame:

$$(\mathbf{p}, \mathbf{q}) \leftarrow (\mathbf{q}_{u, \theta} \cdot \mathbf{p}, \mathbf{q}_{u, \theta} \cdot \mathbf{q}), \text{ where } \mathbf{q}_{u, \theta} \text{ is the rotation as a quaternion, and recalling shorthand notation}$$

Rotation by  $\theta$  about  $u$  interpreted in local frame:

$$\mathbf{q} \leftarrow \mathbf{q} \cdot \mathbf{q}_{u, \theta}$$

By modifying the third case:

Rotation about an axis having direction defined in the parent frame, but about local origin:

$$\text{use only } \mathbf{q} \leftarrow \mathbf{q}_{u, \theta} \cdot \mathbf{q}$$

Rotation of local origin about a parent-defined axis, but maintaining object orientation:

$$\text{use only } \mathbf{p} \leftarrow \mathbf{q}_{u, \theta} \cdot \mathbf{p}$$

**More notes about  $(\mathbf{p}, \mathbf{q})$  representation**

Inverse of transform  $(\mathbf{p}, \mathbf{q})$ :

$$(-\mathbf{q}^* \cdot \mathbf{p}, \mathbf{q}^*)$$

Composition:

$(\mathbf{p}_1, \mathbf{q}_1)$  followed by  $(\mathbf{p}_2, \mathbf{q}_2)$  in fixed frame:

$$(\mathbf{p}_2 + \mathbf{q}_2 \cdot \mathbf{p}_1, \mathbf{q}_2 \cdot \mathbf{q}_1)$$