

# CIS 781

## 3D Raster Graphics

Roger Crawfis  
Ohio State University



## Realism Through Synthesis



Real



Fake



## Play Games ...



NFL Fever screen shot



Stalker screen shot



## Design



Subdivision planning



Golf course LPGA

3D Nature Construction images

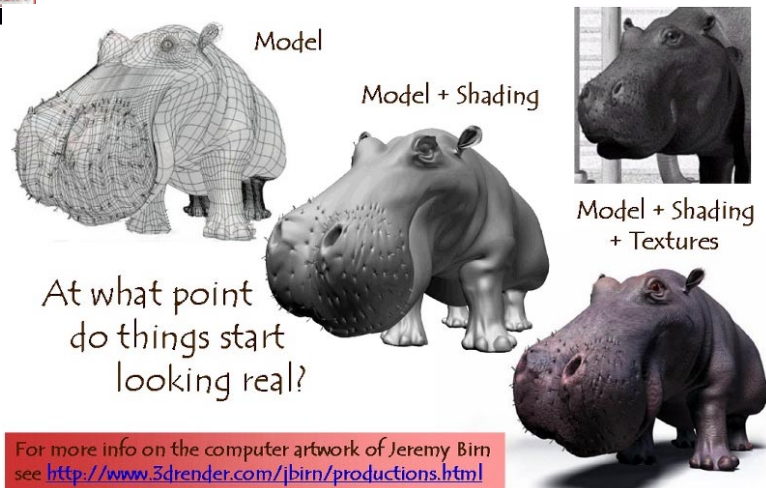
## Goals of Computer Graphics

- Generate synthetic images that look real !
- Do it in a practical way and scientifically sound.
- In real time, obviously. And make it look easy...

## Major Topics

- **Modeling:** representing objects; *building* those representations.
- **Rendering:** how to simulate the image-forming process.
- **Interaction:** change / manipulate objects, immersion
- **Real-Time:** render quickly (30 frames/sec)

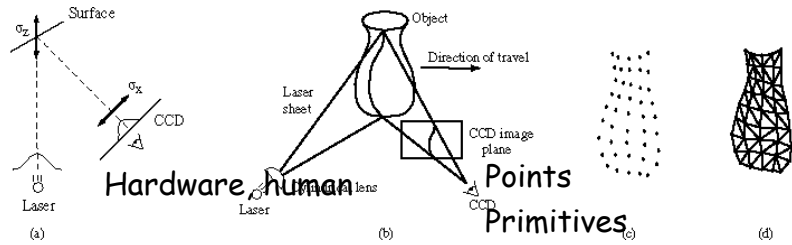
## The Quest for Visual Realism



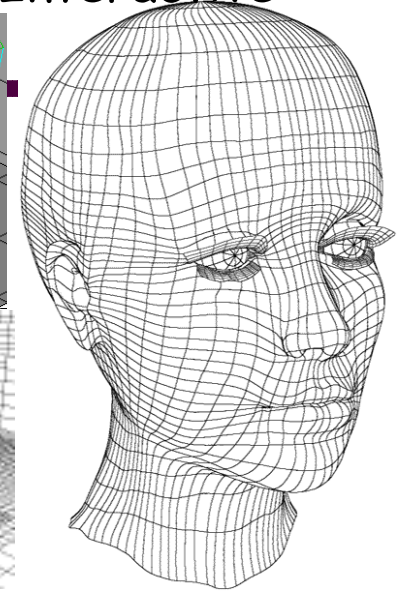
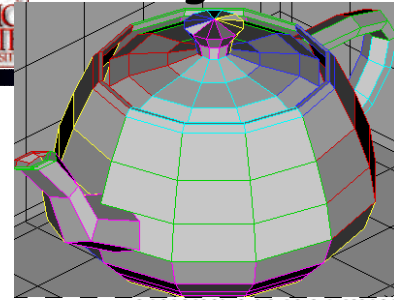
## Modeling

- How to represent real environments
  - geometry: curves, surfaces, volumes
  - photometry: light, color, reflectance
- How to *build* these representations
  - declaratively: write it down
  - interactively: sculpt it
  - programmatically: let it grow (fractals, algebraic/geometric Methods, extraction)
  - via 3D sensing: scan it in
- Get Primitives -lines, triangles, quads, patches !

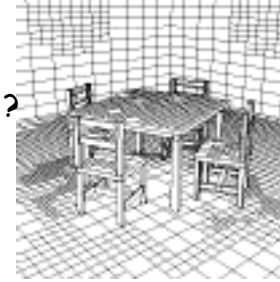
# Modeling - Declarative, Scanning



# Algebraic, Interactive



Primitives ?



# Modeling - Procedural



Crawfis, 2001



3D Nature Construction

Mountains

# Modeling - Procedural



3D Nature Construction



Bryce, 2002

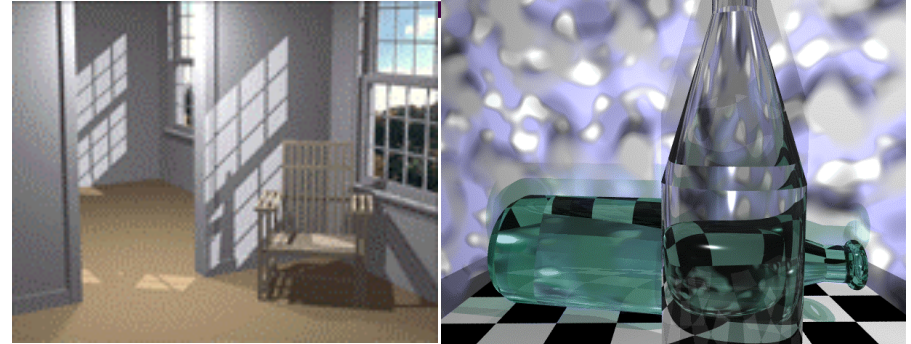
Plants



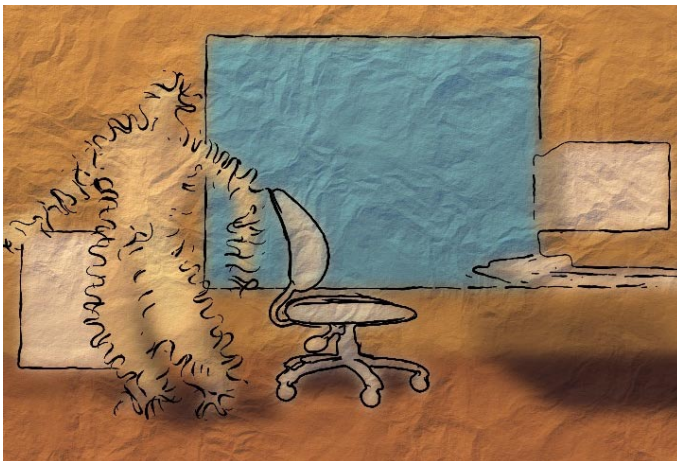
## Rendering

- What's an image?
  - distribution of light energy on 2D "film"
- How do we represent and store images
  - sampled array of "pixels":  $p[x,y]$
- How to generate images from scenes
  - input: 3D description of scene, camera
  - project to camera's viewpoint
  - illumination

## Examples



## Other Examples



## Outline

- Review
  - Transformations
  - OpenGL
- Polygonal models and model construction
- Viewing
  - Projections
  - Clipping





## Outline

- 3D polygonal rendering
  - Rasterization
  - Clipping
  - Hidden surface determination
- Shadows
- Texture Mapping



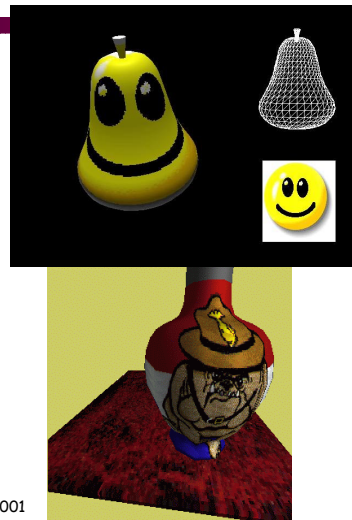
## Other Courses @ OSU

- cis581 - Intro to 3D Graphics, OpenGL
- Cis681 - Ray Tracing, Local Illumination, Anti-aliasing
- Cis782 - Global Illumination, Special Topics
- Cis 784 - Geometric Modeling
- Cis 694? - Scientific Visualization (Crawfis/Shen)
- Cis 694R - Animation (Parent)



## Course Topics

- Texture Mapping
  - Texture Parameterization:
    - Mapping an image to a model
  - Determining the pixel value during scan-conversion
  - Avoiding Aliasing in Texture Mapping



Coleman 2001



## Quote (CIS 681 and 782)

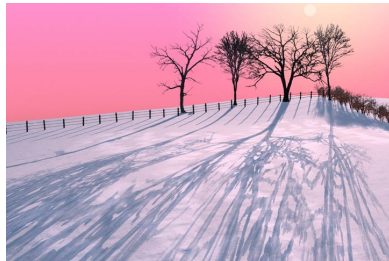
“Now when I paint, I am able to see the bits and the whole at the same time, and colors and shapes pop out at me more readily. For example, now, instead of seeing just an apple inside a bowl, I see an apple catching the reflection from the bowl and reciprocally the color of the apple transferring onto the ceramic surface of the bowl. The bowl must then have a reflective surface capturing other parts of the still life and its shadow on the white cloth below is not gray but is actually a bluish tinge with purple edges, and so forth.”

- Owen Demers  
[digital] Texturing & Painting, 2002

## Lights and Shadows



Wreckless screen shot, 2001



3D Nature Construction

## Quote

“I am interested in the effects on an object that speak of human intervention. This is another factor that you must take into consideration. How many times has the object been painted? Written on? Treated? Bumped into? Scraped? This is when things get exciting. I am curious about: the wearing away of paint on steps from continual use; scrapes made by a moving dolly along the baseboard of a wall; acrylic paint peeling away from a previous coat of an oil base paint; cigarette burns on tile or wood floors; chewing gum – the black spots on city sidewalks; lover’s names and initials scratched onto park benches...”

- Owen Demers

[digital] Texturing & Painting, 2002

## Adding Detail



Medal of Honor screen snapshot

Prestene



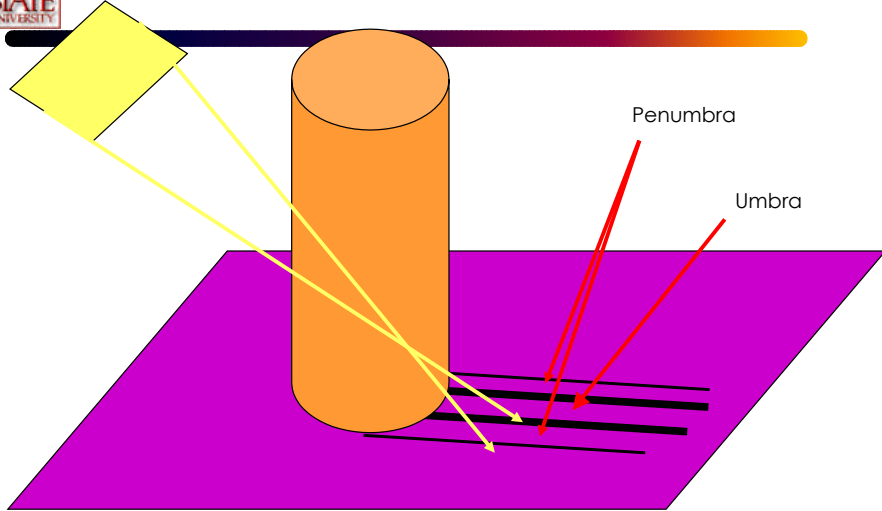
Stalker screen snapshot

Worn and tattered

## Shadows



## Essential Process



## Texture Mapping

- Why use textures?

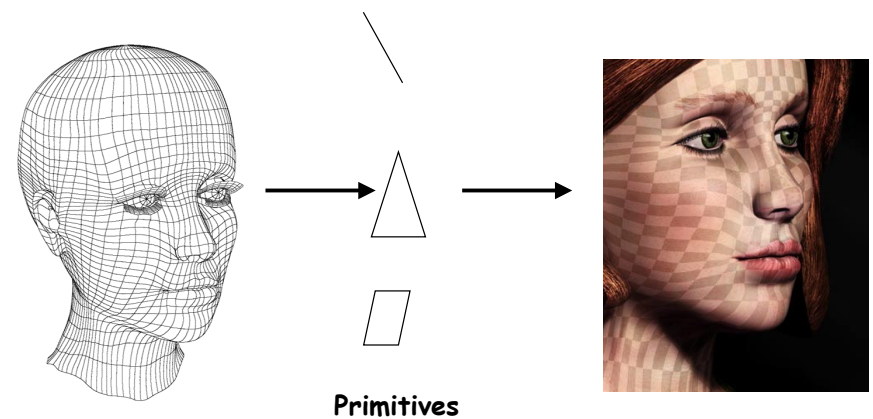


## Texture Mapping

- Modeling complexity

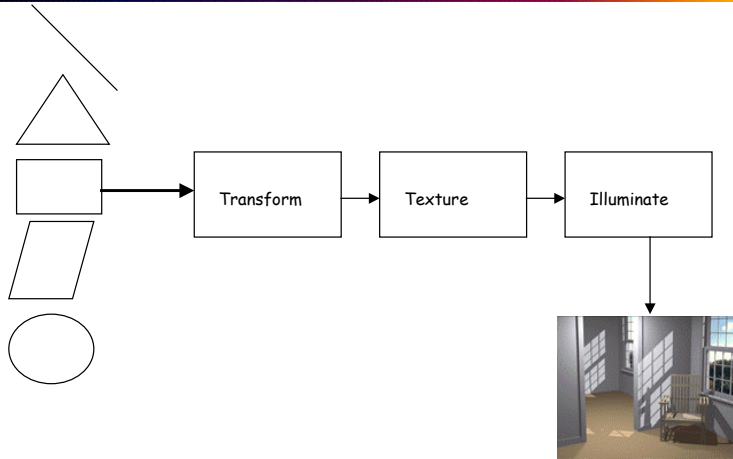


## Essential Process

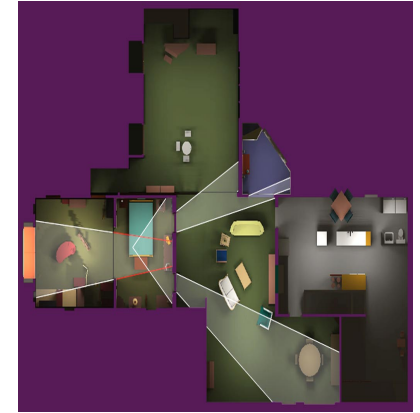




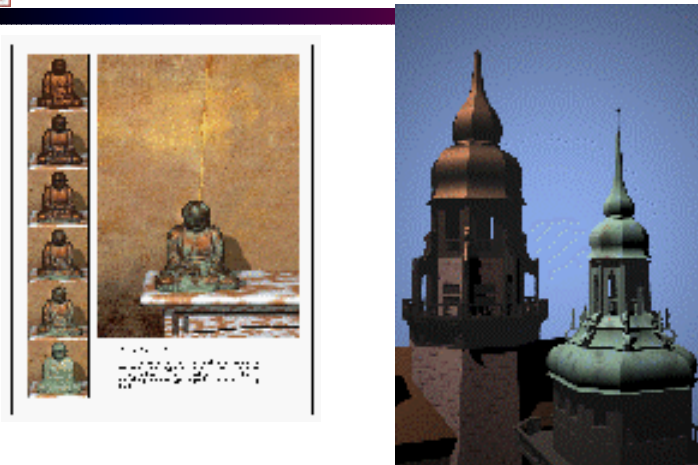
## Graphics Pipeline (OpenGL)



## The Problem of Visibility



## Light Material Interaction - ?

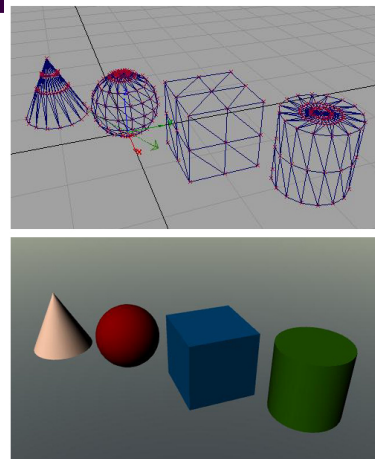


## Modeling

- Types:
  - Polygon surfaces
  - Curved surfaces
- Generating models:
  - Interactive
  - Procedural

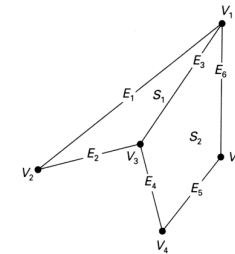
## Polygon Mesh

- Set of surface polygons that enclose an object interior, polygon mesh
- De facto: **triangles, triangle mesh.**



## Representing Polygon Mesh

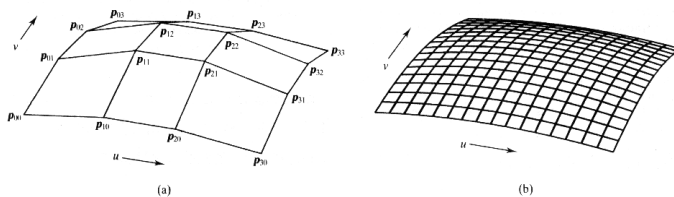
- Vertex coordinates list, polygon table and (maybe) edge table
- Auxiliary:
  - Per vertex normal
  - Neighborhood information, arranged with regard to vertices and edges



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

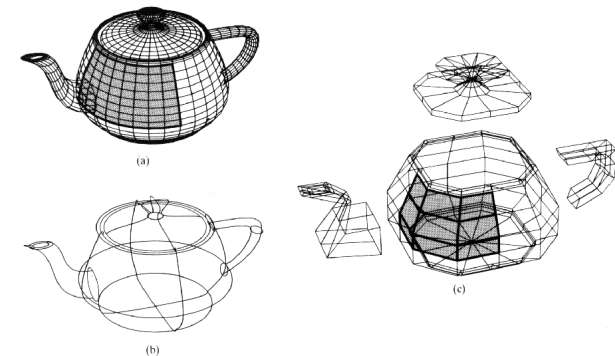
## Arriving at a Mesh

- Use patches model as implicit or parametric surfaces
- Beziér Patches : control polyhedron with 16 points and the resulting bicubic patch:



## Example: The Utah Teapot

- 32 patches





## Patch Representation vs. Polygon Mesh

- Polygons are simple and flexible building blocks.
- But, a parametric representation has advantages:
  - Conciseness
    - A parametric representation is exact and analytical.
  - Deformation and shape change
    - Deformations appear smooth, which is not generally the case with a polygonal object.

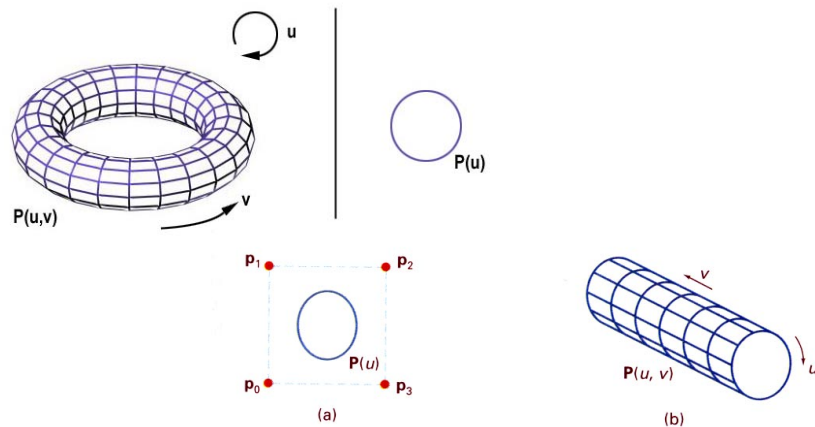


## Shape Construction Operations

- **Extrude:** add a height to a flat polygon
- **Revolve:** Rotate a polygon around a certain axis
- **Sweep:** sweep a shape along a certain curve (a generalization of the above two)
- **Loft:** shape from contours (usually in parallel slices)
- Set operations (intersection, union, difference), **CSG** (constructive solid geometry)

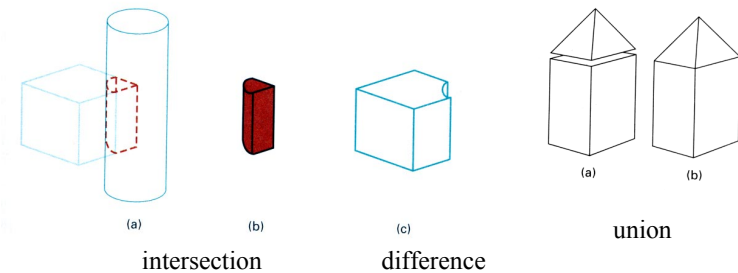


## Sweep (Revolve and Extrude)



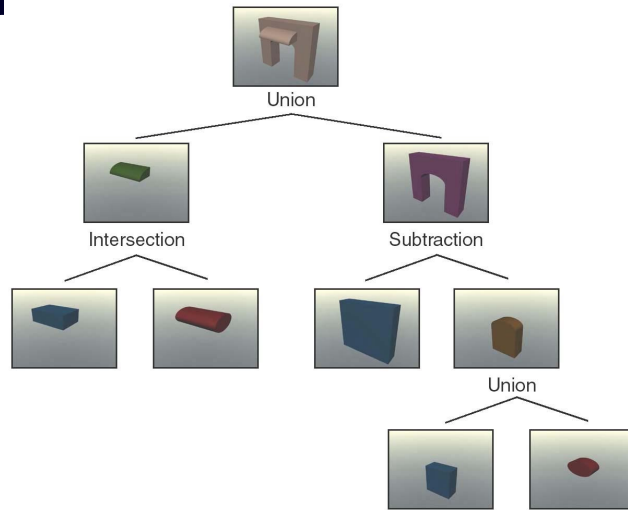
## Constructive Solid Geometry (CSG)

- To combine the volumes occupied by overlapping 3D shapes using set operations.

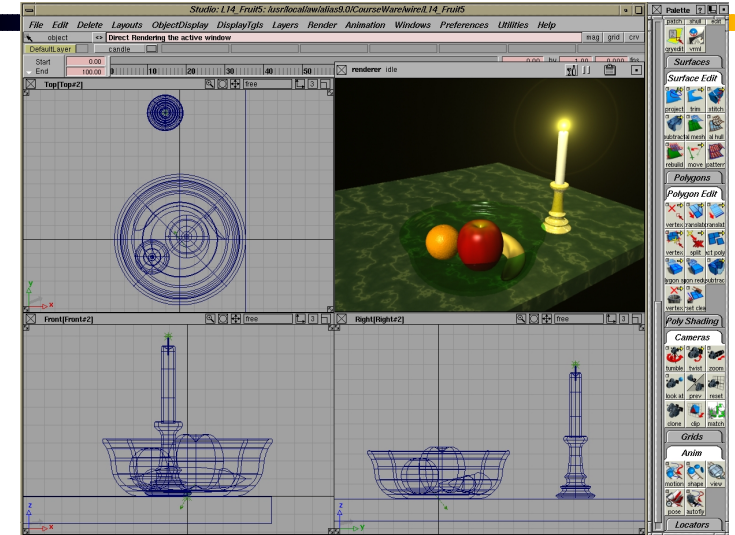




## A CSG Tree

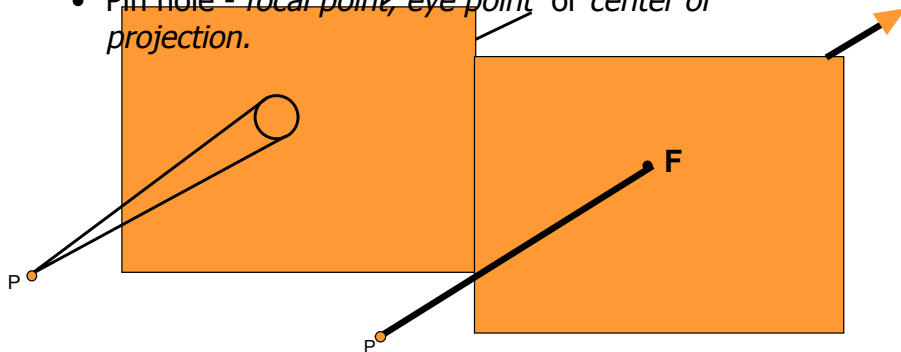


## Example Modeling Package: Alias Studio



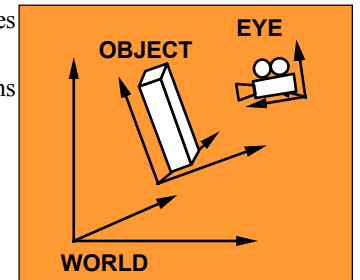
## Pin Hole Model

- Visibility Cone with apex at observer
- Reduce hole to a point - the cone becomes a ray
- Pin hole - focal point, eye point or center of projection.



## Transformations

- **Modeling transformations**
  - build complex models by positioning simple components
- **Viewing transformations**
  - placing virtual camera in the world
  - transformation from world coordinates to eye coordinates
- Side note: animation: vary transformations over time to create motion



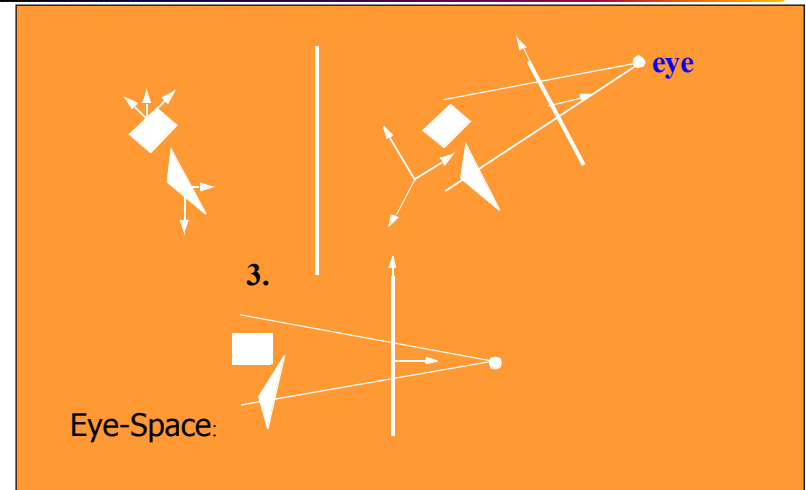
## Viewing Pipeline

Object Space	World Space	Eye Space	Clipping Space	Canonical view volume	Screen Space
--------------	-------------	-----------	----------------	-----------------------	--------------

- **Object space:** coordinate space where each component is defined
- **World space:** all components put together into the same 3D scene via affine transformation. (camera, lighting defined in this space)
- **Eye space:** camera at the origin, view direction coincides with the z axis. Hither and Yon planes perpendicular to the z axis
- **Clipping space:** do clipping here. All points are in homogeneous coordinates, i.e., each point is represented by  $(x,y,z,w)$
- **3D image space (Canonical view volume):** a parallelepiped shape defined by  $(-1:1,-1:1,0,1)$ . Objects in this space are distorted
- **Screen space:** x and y screen pixel coordinates

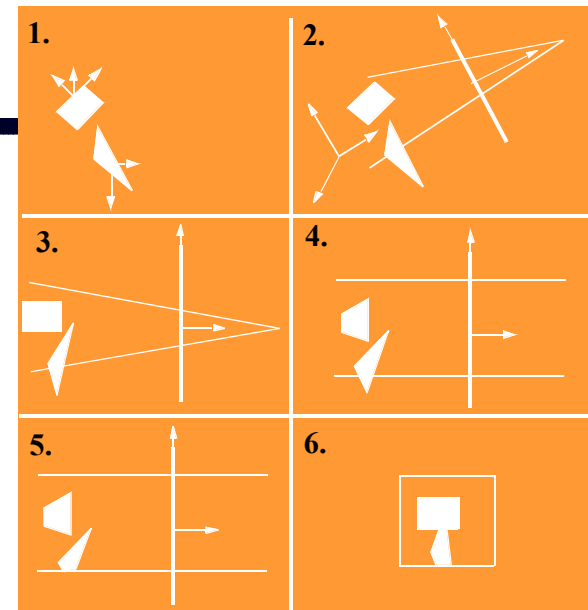
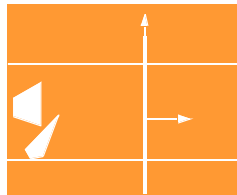
## Model->Eye Space

Object Space and World Space:



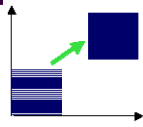
## Clip and Image Spaces

- Clip Space
- Image Space



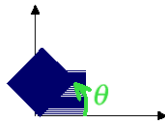
## 2D Transformation

- Translation



$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

- Rotation



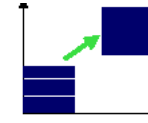
$$\begin{cases} x' = x \cdot \cos\theta - y \cdot \sin\theta \\ y' = x \cdot \sin\theta + y \cdot \cos\theta \end{cases}$$

**Matrix and Vector format:**

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

## Homogeneous Coordinates

- Matrix/Vector format for translation:



$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

**Matrix format?**

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

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

**Homogenous coordinates!**

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

## Translation in Homogenous Coordinates

- There exists an inverse mapping for each function
- There exists an identity mapping

$$M^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$M \Big|_{\substack{t_x=0 \\ t_y=0}} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \text{Identity}(I)$$

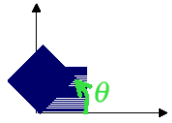
## Why these properties are important

- when these conditions are shown for any class of functions it can be proven that such a class is closed under composition
- i. e. any series of translations can be composed to a single translation.

$$x' = \underbrace{T_1 T_2 \cdots T_n}_{T'} x$$



## Rotation in Homogeneous Space



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

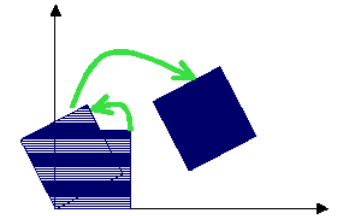
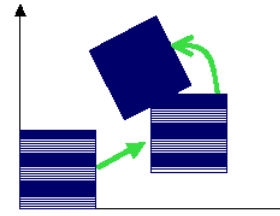
$$M_R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M_R^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M_R|_{\theta=0} = \text{Identity}$$

## Putting Translation and Rotation Together

- Order matters !!



## Affine Transformation

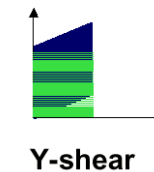
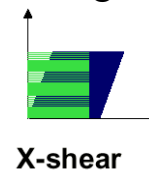
- Property: preserving parallel lines
- The coordinates of three corresponding points uniquely determine any **Affine Transform**!!



## Affine Transformations

- Translation
- Rotation
- Scaling
- Shearing

$$M = \begin{bmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{bmatrix}^T$$



## How to determine an Affine 2D Transformation?



- We set up 6 linear equations in terms of our 6 unknowns. In this case, we know the 2D coordinates before and after the mapping, and we wish to solve for the 6 entries in the affine transform matrix

$$\underbrace{\begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \end{bmatrix}}_{x'} = \underbrace{\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix}}_X \underbrace{\begin{bmatrix} m_{00} \\ m_{01} \\ m_{10} \\ m_{11} \\ m_{20} \\ m_{21} \end{bmatrix}}_m$$



## Affine Transformation in 3D

- Translation

$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Rotate

$$\begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Scale

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Shear

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



## More Rotation

- Which axis of rotation?

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

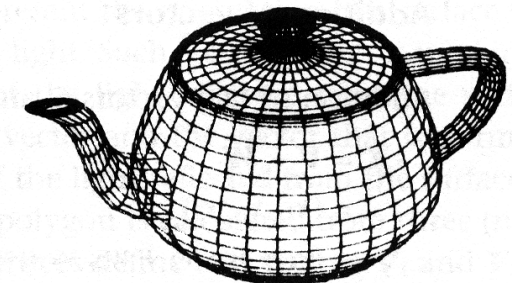
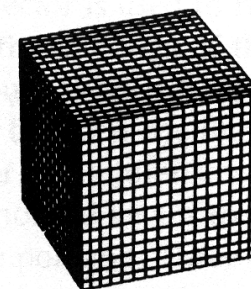
$$R_y = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



## Global Deformations

- Taper
- Twist
- Bend



## Global Deformations

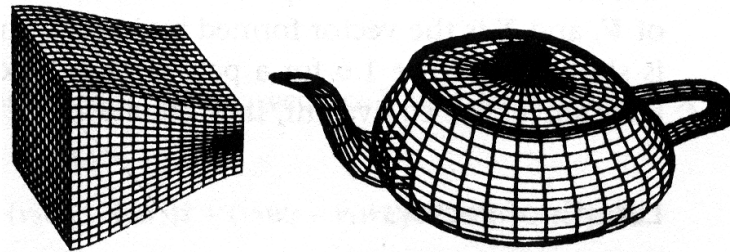
- Tapering:

$$r = f(z)$$

$$x = r * x$$

$$y = r * y$$

$$z = z$$



## Global Deformations

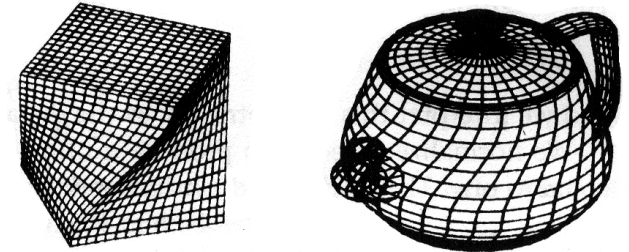
- Twisting:

$$\theta = f(z)$$

$$x = x * \cos \theta - y * \sin \theta$$

$$y = x * \sin \theta + y * \cos \theta$$

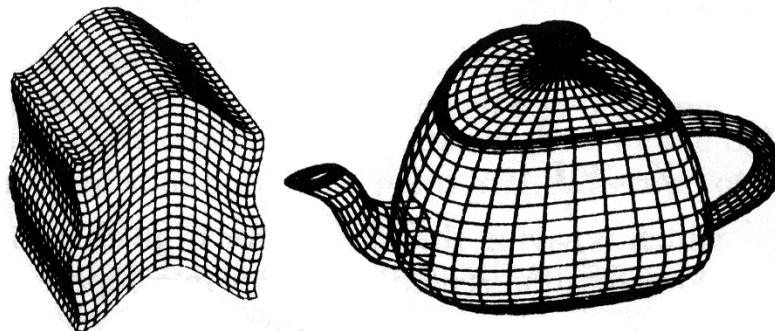
$$z = z$$



## Global Deformations

- Bending:

- More general, bend about some axis.



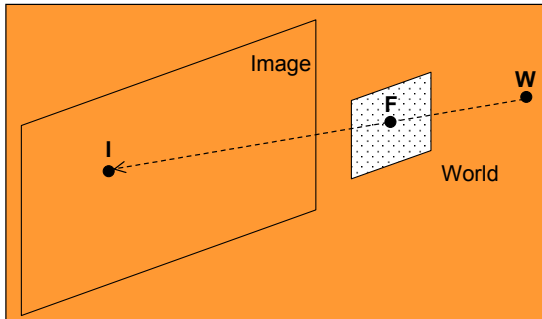
## Viewing

- Placing objects in World space: affine transformations
- World space to Eye space: ???
- Eye space to Clipping space: involves projection and viewing frustum

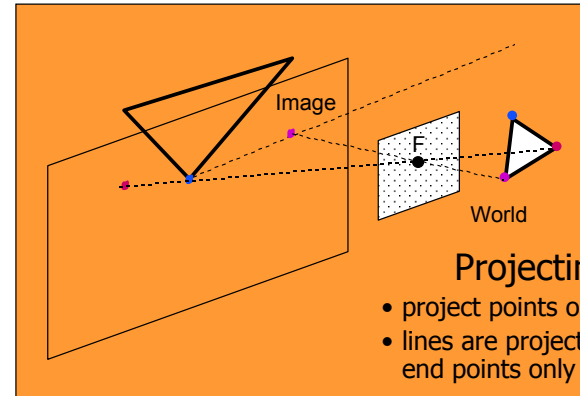


# Perspective Projection and Pin Hole Camera

- Projection point sees anything on ray through pinhole  $F$
- Point  $W$  projects along the ray through  $F$  to appear at  $I$  (intersection of  $WF$  with image plane)

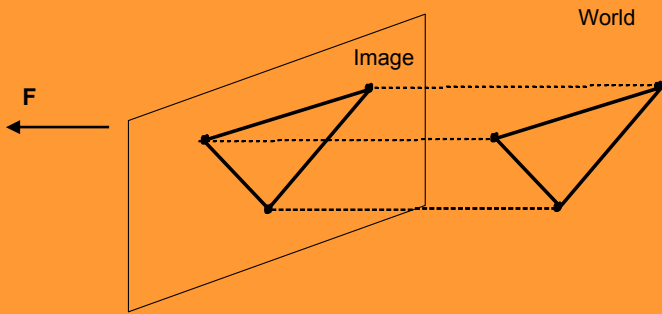


# Image Formation

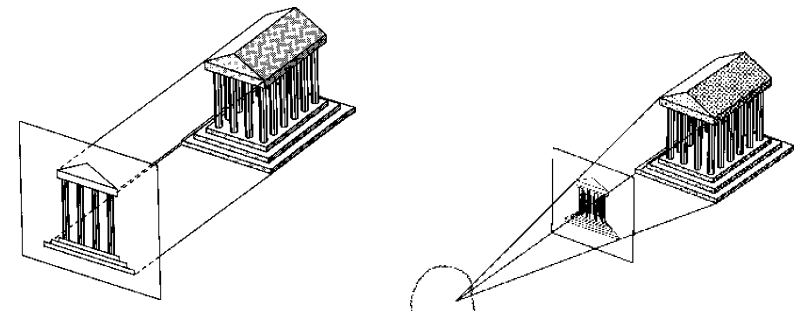


# Orthographic Projection

- focal point at infinity
- rays are parallel and orthogonal to the image plane

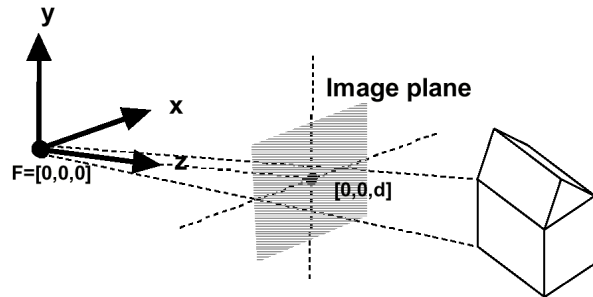


# Comparison



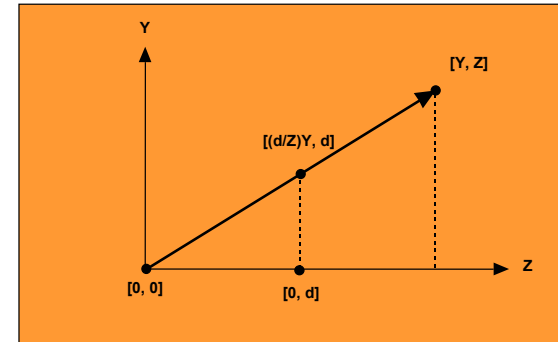
## Simple Perspective Camera

- camera looks along z-axis
- focal point is the origin
- image plane is parallel to xy-plane at distance  $d$



## Similar Triangles

- Similar situation with x-coordinate
- Similar Triangles:  
point  $[x,y,z]$  projects to  $[(d/z)x, (d/z)y, d]$



## Projection Matrix

### Projection using homogeneous coordinates:

- transform  $[x, y, z]$  to  $[(d/z)x, (d/z)y, d]$

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = [dx \quad dy \quad dz \quad z] \Rightarrow \begin{bmatrix} d & & & \\ & d & & \\ & & d & \\ & & & z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

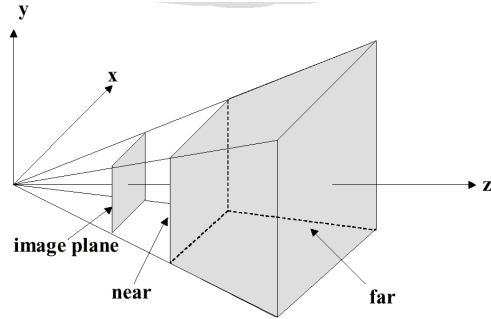
Divide by 4th coordinate  
(the "w" coordinate)

## Image Space

- 2-D image point:
  - discard third coordinate
  - apply viewport transformation to obtain physical pixel coordinates

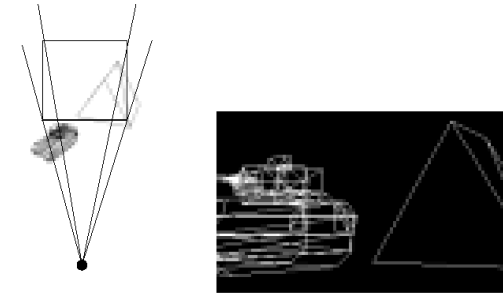
## View Volume

- Defines visible region of space, pyramid edges are clipping planes
- *Frustum* :truncated pyramid with near and far clipping planes
  - Near (Hither) plane ? Don't care about behind the camera
  - Far (Yon) plane, define field of interest, allows  $z$  to be scaled to a limited fixed-point value for  $z$ -buffering.



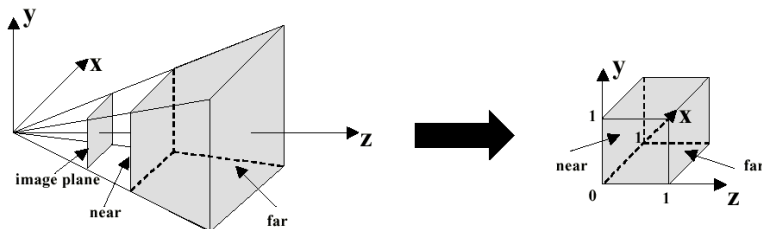
## Difficulty

- It is difficult to do clipping directly in the viewing frustum



## Canonical View Volume

- Normalize the viewing frustum to a cube, canonical view volume
- Converts perspective frustum to orthographic frustum – perspective transformation



## Perspective Transform

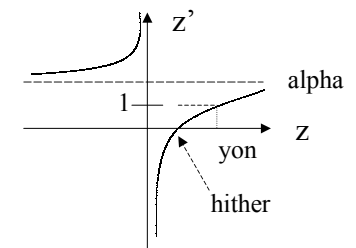
- The equations

$$\begin{cases} x \leftarrow \frac{x d}{z s} \\ y \leftarrow \frac{y d}{z s} \\ z \leftarrow \alpha + \frac{\beta}{z} \end{cases}$$

alpha = yon/(yon-hither)

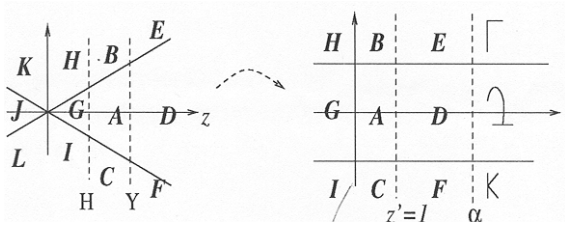
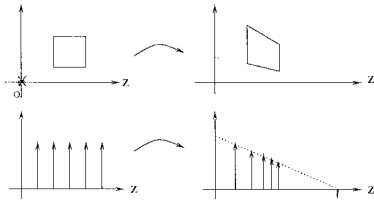
beta = yon\*hither/(hither - yon)

s: size of window on the image plane



## About Perspective Transform

- Some properties



## About Perspective Transform

- Clipping can be performed against the rectilinear box
- Planarity and linearity are preserved
- Angles and distances are not preserved
- Side effects: objects behind the observer are mapped to the front.

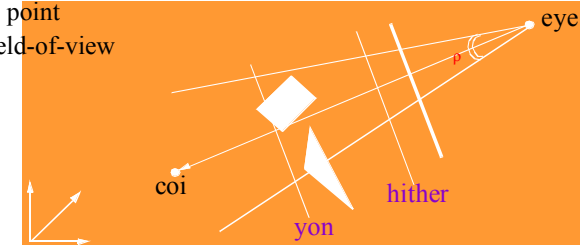
## Perspective + Projection Matrix

- AR: aspect ratio correction, ResX/ResY
- s= ResX,
- Theta: half view angle,  $\tan(\theta) = s/d$

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & AR & 0 & 0 \\ 0 & 0 & \alpha \tan \theta & \tan \theta \\ 0 & 0 & \beta \tan \theta & 0 \end{pmatrix}$$

## Camera Control and Viewing

- Focal length (d), image size/shape and clipping planes included in perspective transformation
  - $\rho$  - Angle or Field of view (FOV)
  - AR - Aspect Ratio of view-port
  - Hither, Yon - Nearest and farthest vision limits (WS).
  - Lookat - COI
  - Lookfrom - Eye point
  - View angle - Field-of-view







## Complete Perspective

- Specify near and far clipping planes - transform  $z$  between  $z_{near}$  and  $z_{far}$  on to a fixed range
- Specify field-of-view (fov) angle
- OpenGL's **glFrustum** and **gluPerspective** do these



## More Viewing Parameters

Camera, Eye or Observer:

*lookfrom*: location of focal point or camera

*lookat*: point to be centered in image

Camera orientation about the *lookat-lookfrom* axis

*vup*: a vector that is pointing straight up in the image. This is like an orientation.



## Implementation ... Full Blown

- Translate by *-lookfrom*, bring focal point to origin
- Rotate *lookat-lookfrom* to the  $z$ -axis with matrix R:
  - $\mathbf{v} = (\text{lookat-lookfrom})$  (normalized) and  $\mathbf{z} = [0,0,1]$
  - rotation axis:  $\mathbf{a} = (\mathbf{v} \times \mathbf{z}) / |\mathbf{v} \times \mathbf{z}|$
  - rotation angle:  $\cos\theta = \mathbf{a} \cdot \mathbf{z}$  and  $\sin\theta = |\mathbf{r} \times \mathbf{z}|$
- OpenGL: `glRotate( $\theta$ ,  $a_x$ ,  $a_y$ ,  $a_z$ )`
- Rotate about  $z$ -axis to get *vup* parallel to the  $y$ -axis



## Viewport mapping

- Change from the image coordinate system  $(x,y,z)$  to the screen coordinate system  $(X,Y)$ .
- Screen coordinates are always non-negative integers.
- Let  $(v_r, v_l)$  be the upper-right corner and  $(v_b, v_b)$  be the lower-left corner.
- $X = x * (v_r - v_l) / 2 + (v_r + v_l) / 2$
- $Y = y * (v_t - v_b) / 2 + (v_t + v_b) / 2$



## True Or False

- In perspective transformation parallelism is not preserved.
  - Parallel lines converge
  - Object size is reduced by increasing distance from center of projection
  - Non-uniform foreshortening of lines in the object as a function of orientation and distance from center of projection
  - Aid the depth perception of human vision, but shape is not preserved



## True Or False

- Affine transformation is a combination of linear transformations
- The last column/row in the general 4x4 affine transformation matrix is  $[0\ 0\ 0\ 1]^T$ .
- After affine transform, the homogeneous coordinate  $w$  maintains unity.

## Introduction to OpenGL

Roger Crawfis

This set of slides are from Jian Huang and are based upon the slides from the Interactive OpenGL Programming course given by Dave Shreine, Ed Angel and Vicki Shreiner on SIGGRAPH 2001.



## OpenGL an GLUT Overview

- What is OpenGL & what can it do for me?
- OpenGL in windowing systems
- Why GLUT
- GLUT program template

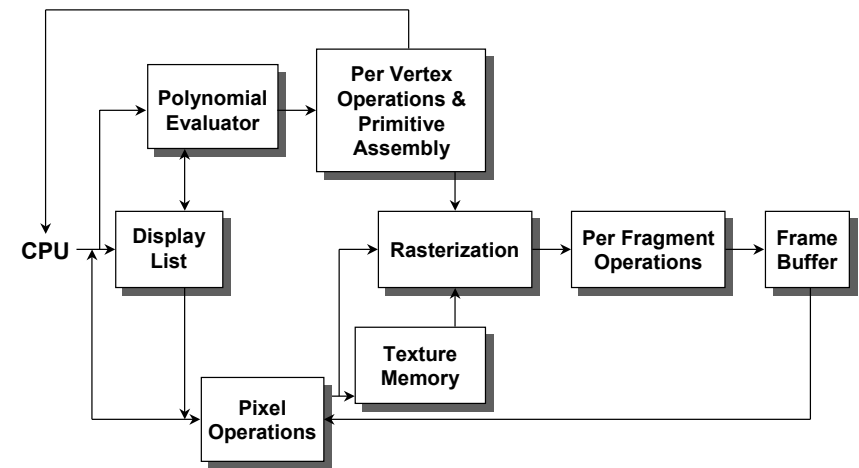


## What Is OpenGL?

- Graphics rendering API
  - high-quality color images composed of geometric and image primitives
  - window system independent
  - operating system independent



## OpenGL Architecture



## OpenGL as a Renderer

- Geometric primitives
  - points, lines and polygons
  - Image Primitives
  - images and bitmaps
- separate pipeline for images and geometry
  - linked through texture mapping
- Rendering depends on state
  - colors, materials, light sources, etc.

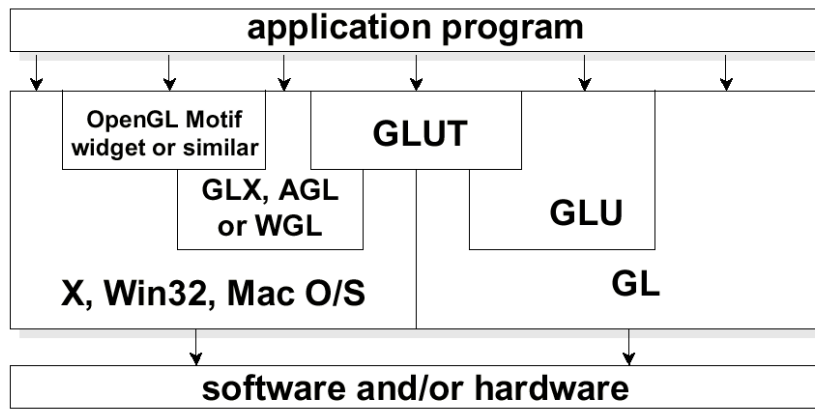


## Related APIs

- AGL, GLX, WGL
  - glue between OpenGL and windowing systems
- GLU (OpenGL Utility Library)
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc
- GLUT (OpenGL Utility Toolkit)
  - portable windowing API
  - not officially part of OpenGL



## OpenGL and Related APIs



## Preliminaries

- Header Files
  - #include <GL gl.h>
  - #include <GL glu.h>
  - #include <GL glut.h>
- Libraries
- Enumerated types
- OpenGL defines numerous types for compatibility
  - GLfloat, GLint, GLenum, etc.



## GLUT Basics

- Application Structure
- Configure and open window
- Initialize OpenGL state
- Register input callback functions
  - render
  - resize
  - input: keyboard, mouse, etc.
- Enter event processing loop



## Sample Program

```
void main( int argc, char** argv )
{
    glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE );
    glutCreateWindow( "Simple OpenGL Program" );
    my_init(); // initiate OpenGL states, program variables
    glutDisplayFunc( my_display ); // register callback routines
    glutReshapeFunc( my_resize );
    glutKeyboardFunc( my_key_events );
    glutIdleFunc( my_idle_func );
    glutMainLoop(); // enter the event-driven loop
}
```





## OpenGL Initialization

- Set up whatever state you're going to use

```
void my_init( void )
{
    glClearColor( 0.0, 0.0, 0.0, 1.0 );
    glClearDepth( 1.0 );
    glEnable( GL_LIGHT0 );
    glEnable( GL_LIGHTING );
    glEnable( GL_DEPTH_TEST );
}
```



## GLUT Callback Functions

- Routine to call when something happens
  - window resize or redraw
  - user input
  - animation
- “Register” callbacks with GLU
  - `glutDisplayFunc( my_display );`
  - `glutIdleFunc( my_idle_func );`
  - `glutKeyboardFunc( my_key_events );`



## Rendering Callback

- Do all of our drawing here

```
glutDisplayFunc( my_display );
void my_display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glBegin( GL_TRIANGLE );
        glVertex3fv( v[0] );
        glVertex3fv( v[1] );
        glVertex3fv( v[2] );
    glEnd();
    glutSwapBuffers();
}
```



## Idle Callbacks

- Used for animation, game AI and other continuous updates

```
glutIdleFunc( my_idle_func );
void my_idle_func ( void )
{
    if( rotate ) theta +=dt;
    glutPostRedisplay();
}
```

## User Input Callbacks

- Process user input

```

glutKeyboardFunc( my_key_events );
void my_key_events ( char key, int x, int y )
{
    switch( key ) {
        case 'q' : case 'Q' :
            exit( EXIT_SUCCESS );
            break;
        case 'r' : case 'R' :
            rotate = GL_TRUE;
            break;
    }
}

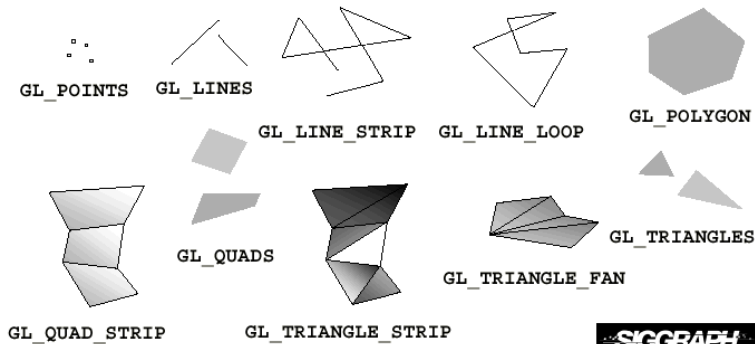
```

## Elementary Rendering

- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers

## OpenGL Geometric Primitives

- All geometric primitives are specified by vertices



## Simple Example

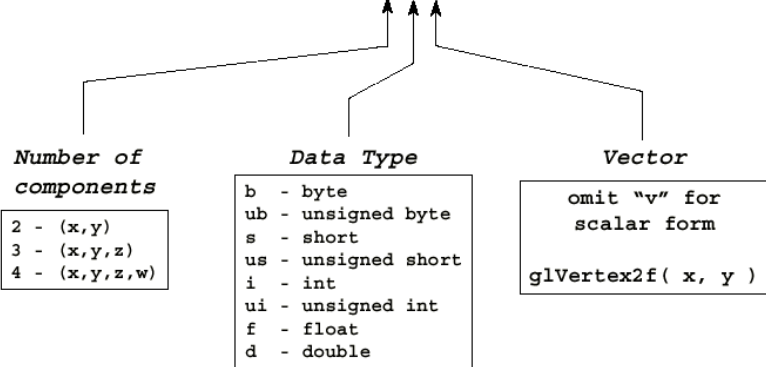
```

void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}

```

# OpenGL Command Formats

## glVertex3fv( v )



# Specifying Geometric Primitives

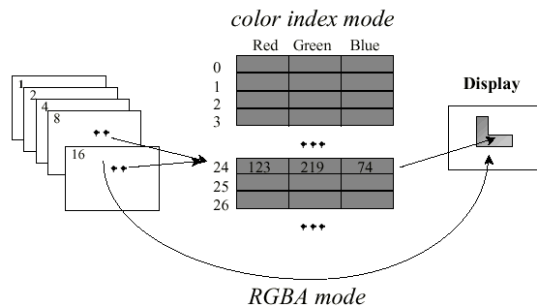
- Primitives are specified using
 

```
glBegin( primType );
glEnd();
```
- `primType` determines how vertices are combined
 

```
GLfloat red, green, blue;
GLfloat coords[3];
glBegin( primType );
for ( i=0; i<nVerts; i++ ) {
    glColor3f( red, green, blue );
    glVertex3fv( coords );
}
glEnd();
```

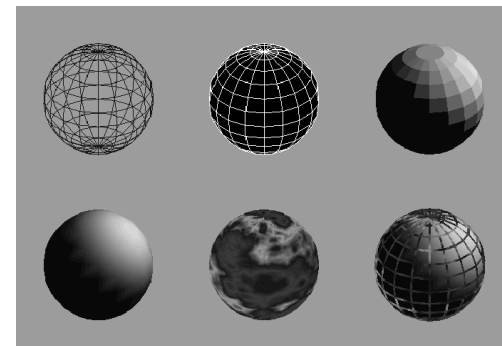
# OpenGL Color Model

- Both RGBA (true color) and Color Index



# Controlling Rendering

- Appearance
- From Wireframe to Texture mapped





## OpenGL's State Machine

- All rendering attributes are encapsulated in the OpenGL State
  - rendering styles
  - shading
  - lighting
  - texture mapping



## Manipulating OpenGL State

- Appearance is controlled by current state for each ( primitive to render ) {
  - update OpenGL state
  - render primitive}
- manipulating vertex attributes is most common way to manipulate state
  - glColor\*() / glIndex\*()
  - glNormal\*()
  - glTexCoord\*()



## Controlling current state

- Setting State
  - glPointSize( size );
  - glLineStipple( repeat, pattern );
  - glShadeModel( GL\_ SMOOTH );
- Enabling Features
  - glEnable( GL\_ LIGHTING );
  - glDisable( GL\_ TEXTURE\_ 2D );



## Transformations in OpenGL

- Modeling
- Viewing
  - orient camera
  - projection
- Animation
- Map to screen



## Coordinate Systems and Transformations



- Steps in Forming an Image
  - specify geometry (world coordinates)
  - specify camera (camera coordinates)
  - project (window coordinates)
  - map to viewport (screen coordinates)
- Each step uses transformations
- Every transformation is equivalent to a change in coordinate systems

## 3D Transformations



- A vertex is transformed by 4 x 4 matrices
- all affine operations are matrix multiplication
- matrices are stored column-major in OGL
- matrices are always post-multiplied
- OpenGL uses stacks of matrices, the programmer must remember that the last matrix specified is the first applied.

## Specifying Transformations



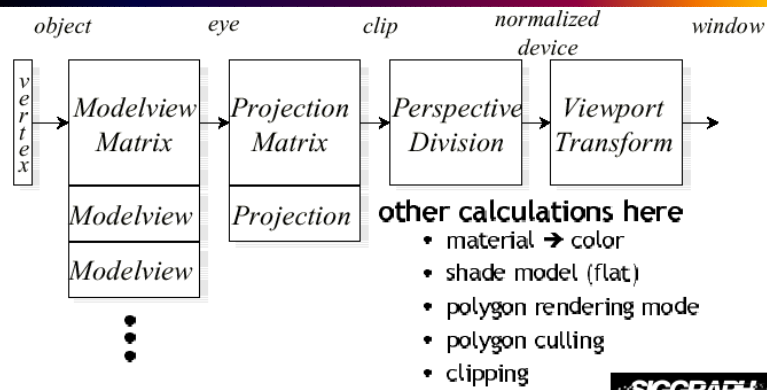
- Programmer has two styles of specifying transformations
  - specify matrices `glLoadMatrix`, `glMultMatrix`
  - specify operations `glRotate`, `glOrtho`
- Programmer does not have to remember the exact matrices
- Check Appendix of the Red Book

## Programming Transformations



- Prior to rendering, view, locate, and orient:
  - eye/camera position
  - 3D geometry
- Manage the matrices
  - including matrix stack
- Combine (composite) transformations
- Transformation matrices are part of the state, they must be defined prior to any vertices to which they are to apply.
- OpenGL provides matrix stacks for each type of supported matrix (ModelView, projection, texture) to store matrices.

## Transformation Pipeline



## Matrix Operations

- Specify Current Matrix Stack
  - `glMatrixMode( GL_MODELVIEW or GL_PROJECTION )`
- Other Matrix or Stack Operation
  - `glLoadIdentity() glPushMatrix() glPopMatrix()`
- Viewport
  - usually same as window size
  - viewport aspect ratio should be same as projection transformation or resulting image may be distorted
  - `glViewport( x, y, width, height )`

## Projection Transformation

- Perspective projection
  - `gluPerspective( fovy, aspect, zNear, zFar )`
  - `glFrustum( left, right, bottom, top, zNear, zFar )` (very rarely used)
- Orthographic parallel projection
  - `glOrtho( left, right, bottom, top, zNear, zFar )`
  - `gluOrtho2D( left, right, bottom, top )`
  - calls `glOrtho` with z values near zero
- *Warning:* for `gluPerspective()` or `glFrustum()`, don't use zero for zNear!

## Applying Projection

- Transformations
- Typical use ( orthographic projection )
 

```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( left, right, bottom, top, zNear, zFar );
```



## Viewing Transformations

- Position the camera/eye in the scene
- To “fly through” a scene
- change viewing transformation and redraw scene
 

```
gluLookAt( eye x ,eye y ,eye z ,
           aim x ,aim y ,aim z ,
           up x ,up y ,up z )
```
- up vector determines unique orientation
- careful of degenerate positions



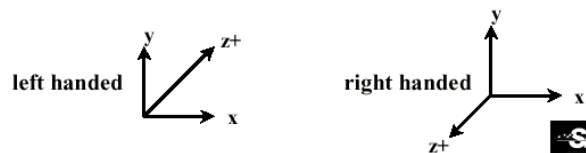
## Modeling Transformations

- Move object
  - `glTranslate{fd}( x, y, z )`
- Rotate object around arbitrary axis
  - `glRotate{fd}( angle, x, y, z )`
  - angle is in degrees
- Dilate (stretch or shrink) object
  - `glScale{fd}( x, y, z )`



## Projection is left handed

- Projection transformation (`gluPerspective`, `glOrtho`) are left handed
  - think of `zNear` and `zFar` as distance from view point
- Everything else is right handed, including the vertexes to be rendered



## Common Transformation Usage

- Example of `resize()` routine
  - restate projection & viewing transformations
- Usually called when window resized
- Registered a callback for `glutReshapeFunc()`



## resize(): Perspective & LookAt

```

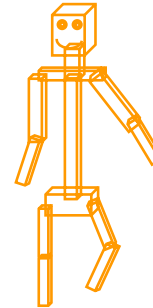
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat) w / h,
                  1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( 0.0, 0.0, 5.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0 );
}

```

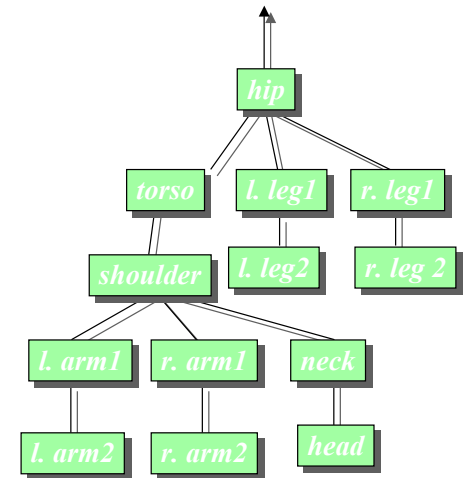


## Hierarchical Modeling

- A node represents:
  - rotation(s)
  - geometric primitive(s)
  - Transformations



The root can be anywhere (hip)  
Control for each joint angle, plus global position and orientation



## Relevant OpenGL Routines

- glPushMatrix(), glPopMatrix()
  - push and pop the stack. push leaves a copy of the current matrix on top of the stack
- glLoadIdentity(), glLoadMatrixd(M)
  - load the Identity matrix, or an arbitrary matrix, onto top of the stack
- glMultMatrixd(M)
  - multiply the matrix C on top of stack by M.  $C = CM$
- glOrtho(x0,y0,x1,y1,z0,z1)
  - set up parallel projection matrix
- glRotatef(theta,x,y,z), glRotated(...)
  - axis/angle rotate. “f” and “d” take floats and doubles, respectively
- glTranslatef(x,y,z), glScalef(x,y,z)
  - translate, scale. (also exist in “d” versions.)

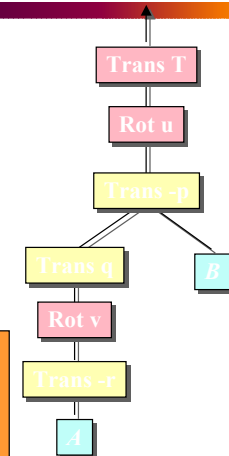
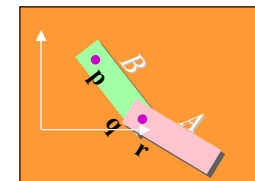


## OpenGL Example

```

glLoadIdentity();
glOrtho(...);
glPushMatrix();
glTranslatef(Tx,Ty,0);
glRotatef(u,0,0,1);
glTranslatef(-px,-py,0);
glPushMatrix();
glTranslatef(qx,qy,0);
glRotatef(v,0,0,1);
glTranslatef(-rx,-ry,0);
Draw(A);
glPopMatrix();
Draw(B);
glPopMatrix();

```







## Hierarchy methods

- Object Oriented
- Push matrix stack
  - Implies depth-first traversal
- Do type-specific transform
- Recurse on its children, and pops.



## Interactive Applications

- How do we add interactive control?
- Many different paradigms
  - Examiner => Object in hand
  - Fly-thru => In a virtual vehicle pod
  - Walk-thru => Constrained to stay on ground.
  - Move-to / re-center => Pick a location to fly to.
- Collision detection?
  - Can we pass thru objects like ghosts?



## Interactive Applications

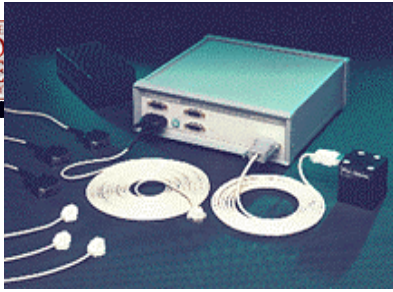
- What do we use to control the motion?
  - Mouse
    - One-button, two-button, three-button
    - What button does what?
    - Only when mouse is clicked down, released up, or continuously as the mouse moves?
  - Keyboard
    - Arrow keys?



## Input Devices

- Interactive user control devices
  - Mouse
  - 3D pointer - Polhemus, Microscribe, ...
  - Spaceball
  - Hand-held wand
  - Data Glove
  - Gesture
  - Custom

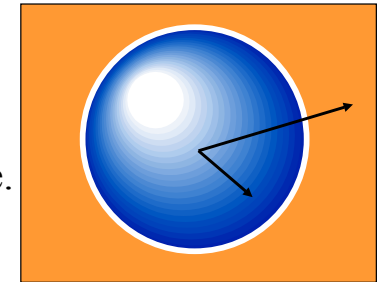




ALSO AVAILABLE  
The Spaceball 2003 FLX

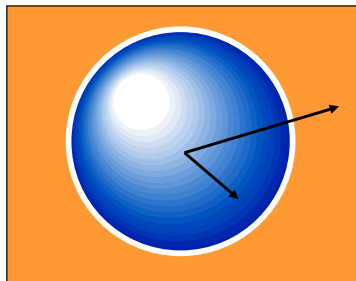
## A Virtual Trackball

- A rather standard and easy-to-use interface.
- Examiner type of interaction.
- Consider a hemi-sphere over the image-plane.
- Each point in the image is projected onto the hemi-sphere.



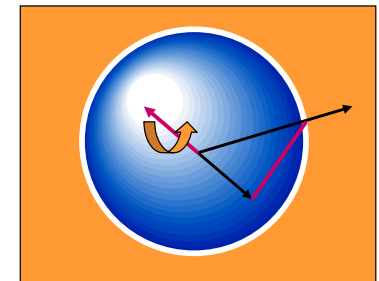
## A Virtual Trackball

- Points inside the projection of the hemi-sphere are mapped up to the surface.
  - Determine distance from point (mouse position) to the image-plane center.
  - Scale such that points on the silhouette of the sphere have unit length.
  - Add the z-coordinate to normalize the vector.



## A Virtual Trackball

- Do this for all points.
- Keep track of the last trackball (mouse) location and the current location.
- This is the direction we want the scene to move in.
- Take the direction perpendicular to this and use it as the axis of rotation.
- Use the distance between the two points to determine the rotation angle (or amount).

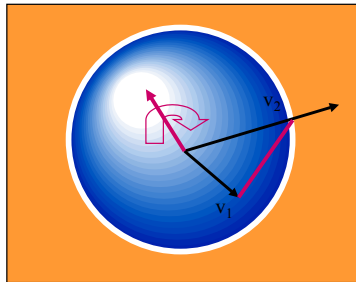


## A Virtual Trackball

- Rotation axis:

$$\vec{u} = \vec{v}_1 \otimes \vec{v}_2$$

Where,  $v_1$  and  $v_2$  are the mouse points mapped to the sphere.



## A Virtual Trackball

- Use `glRotatef( angle, u_x, u_y, u_z )`
- Slight problem: We want the rotation to be the last operation performed.
- Easily fixed:
  - Read out the current `GL_MODELVIEW` matrix
  - Load the identity matrix
  - Rotate
  - Multiply by the saved `GL_MODELVIEW` matrix

## Virtual Reality

Roger Crawfis



## Virtual reality technology

- many definitions of virtual reality (VR), for example:
- "the creation of the effect of immersion in a computer-generated three-dimensional environment in which objects have spatial presence" [Bryson & Feiner, 1994]
- "things as opposed to pictures of things"
- interaction, not content
- many variations, desktop VR, fish tank VR, augmented reality



## *Related terminology*

- virtual environment
- virtual world
- artificial reality
- augmented reality
- telepresence
- Teleoperation
- Collaborative Spaces



## *Performance requirements*

- wide-field stereoscopic display fill's the user's field of view
- head-tracking supports the illusion that the user is looking around in an environment
- 3D computer graphics fills the environment with objects
- 3D interaction gives users the feeling that they are interacting with real objects
- overall frame rate must be  $> 10$  frames/sec
- end-to-end delays must be  $< 0.1$  sec for interactive control



## *The problem with VR is...*

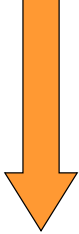
- that it is apparently simple
- NOT the unusual hardware
- many components must work together in real-time
- many criteria must be met
- unclear how to use the interface
- human factors issues not well understood



## *The evolution of VR*

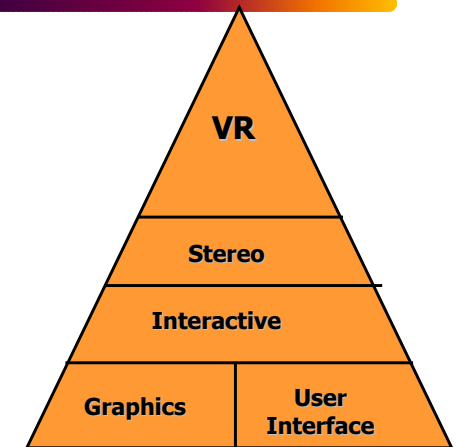
- 1960 Morton Heilig files patent to the US Patent Office  
**"Stereoscopic TV Apparatus for Individual Use"**  
My invention generally speaking comprises the following elements: a hollow casing, a pair of optical units, a pair of television tube units, a pair of earphones and a pair of air discharge nozzles, all coacting to cause the user to comfortably see the images, hear the sound effects and to be sensitive to the air discharge of the said nozzles.
- 1960-70 Sutherland's head-mounted display
- 1984 NASA Ames VIVED project
- 1986-90 NASA Ames VIEW lab and VPI
- 1990-onwards VR community fully formed and flourishing...

## Degrees of immersion

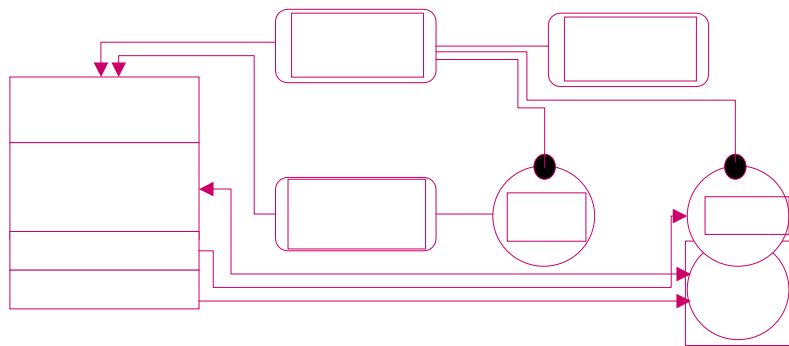
DISPLAY	INTERACTION	IMMERSION
colour	keyboard	low  High
high resolution	2D Mouse	
Stereo	6D input device	
head tracking	6D tracking + buttons	
wide field of view	6D tracking + gloves	
head coupled	force tracking	

## Virtual Environments

- Immersive
- Interactive
- User Centered



## Typical configuration



Patrick Olivier

## Displays

- primary technology underlying immersion
- many aspects: colour, resolution, field of view...
- display paradigms:
  - stereo via two displays
  - stereo via one display images synchronised (eyewear)
  - CAVE: immersion via surrounding large screens
  - head tracking (fish tank VR)
  - head tracking head-mounted

## Virtual Environments

- Display Technologies
  - HMD's - Head Mounted Displays
  - Large theater - Imax, Omnimax
  - Stereo displays
  - HUD's - Head's Up Displays
    - windshields
    - goggles
  - CAVE - Surround video projections



## Tracking paradigms

- usually a sensor determines position and orientation relative to source (calibration renders position of source irrelevant)
- sensor detects a signal from the source in such a way that the position and orientation can be determined
- either the source or the sensor can be fixed
- numerous technologies: electromagnetic, ultrasonic, mechanical, video, inertial

## What to track?

- head position and orientation
- any significant body part
- any articulations

### Other tracking technologies

- passive stereo vision systems
- marker systems (used in motion capture)
- structured light methods (light stripe)
- inertial tracking (using accelerometers)
- eye tracking (commonly optical - corneal reflection)

## Virtual Environments

- Interactive user navigation devices
  - Head tracker
  - Treadmill
  - Bicycle
  - Wheelchair
  - Boom
  - Video detection
- Anyone seen the new game at GameWorks?





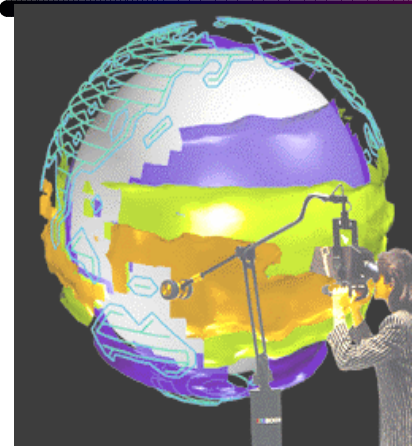
## Virtual Environments

- Interactive user control devices

- Mouse
- 3D pointer - Polhemus, Microscribe, ...
- Spaceball
- Hand-held wand
- Data Glove
- Gesture
- Custom



## Fakespace BOOM 3C



**Video Output**  
Full Color Stereo - or Monoscopic.

**Resolution**  
Up to 1280 x 1024 pixels per eye.

**Optics**  
User interchangeable modules offer from 40 to 110 degrees horizontal FOV

**Tracking**  
Opto-mechanical

**Accuracy**  
0.015" at 30"

**Latency**  
200ns

**Sampling Frequency**  
>70Hz

**Range**  
6' diameter horizontal circle (center 1 foot unavailable) 2.5' vertical.



ALSO AVAILABLE  
The Spaceball 2003 FLX

## Human factors of virtual reality

- Limits on motion frequencies:
  - head (5 Hz)
  - hand (10 Hz)
  - full body (5 Hz)
  - eye (100 Hz)

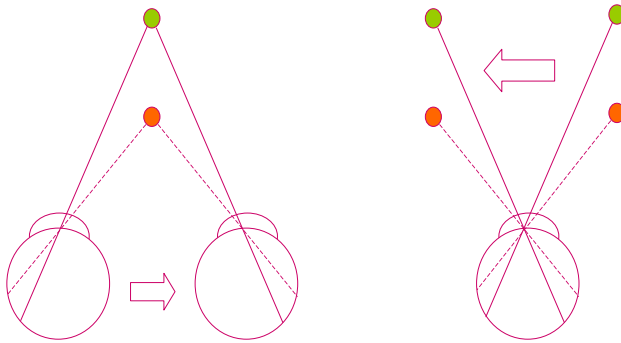
## Human factors of virtual reality

- Limits on Vision (optical resolution):
  - angular size of the smallest object that can be resolved:
    - essentially the angular size of a colour pixel
    - measure as a linear size in minutes of arc
    - full moon is 30 minutes of arc across its diameter
    - human visual system can resolve 0.5 minutes of arc in the central visual field
    - 2-3 minutes of arc in the peripheral visual field

## Cues to support the sense of immersion

- immersion: want to be in an environment that contains “things” and not looking at pictures
- spatial presence of virtual objects due to:
  - **spatial constancy**
    - 10 frames/sec minimum requirement
    - if your head moves and the scene doesn't it isn't VR
    - object behaviour (e.g. application of consistent physical laws)
  - **depth perception**
    - stereo
    - head motion parallax
    - many other depth cues
- wide field of view
  - environment seems to fill field of view (60° minimum threshold)

## Motion parallax

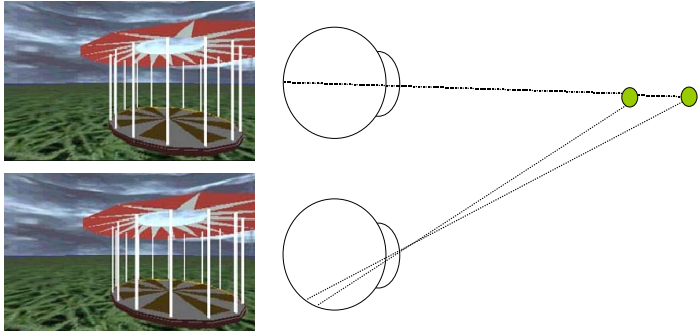


## Aspects of head-motion parallax

- due to change in visual scene as the head moves
- performed in a VR system by tracking the user's head and rendering the virtual scene from a moving point of view
- head-motion parallax is a monocular depth cue:
  - beyond 1m monocular cues dominate
  - within 1m binocular disparity and motion parallax is crucial
  - need 12 frames/sec for motion parallax

## Stereopsis

- fusion of images from two eyes
- projected rays of same points in world different for each eye
- points in the world are visible to one eye and not another



## Aspects of stereopsis

- People have different fusion capabilities (it is believed that as many as 20% have little capability)
- Effective out to 3-6m but critical < 1m
- Far-field, not that critical.

## Virtual Environments

- Draw at 120Hz
- Track user position/orientation at 120Hz
- Provide Haptic feedback at > 200Hz
- User tracking > 10Hz

## Augmented Reality

- Merged real imagery and computer generated imagery.
  - Video capture into visualization system
  - See-thru glasses



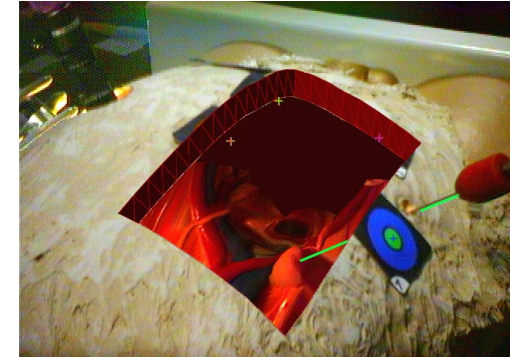
## Augmented Reality

University of North Carolina, Chapel Hill



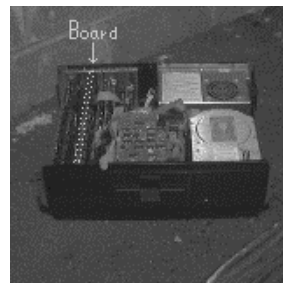
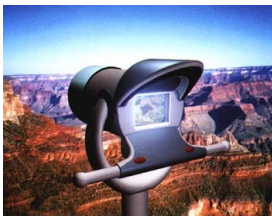
## Augmented Reality

University of North Carolina, Chapel Hill



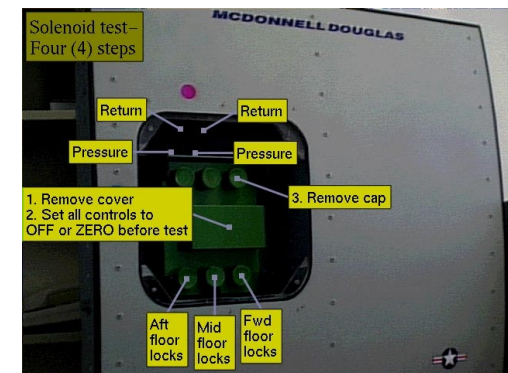
## Augmented Reality

- Also useful for non-medical
  - Mechanics drawing super-imposed over the actual machinery.
  - Guided tours.



## Augmented Reality

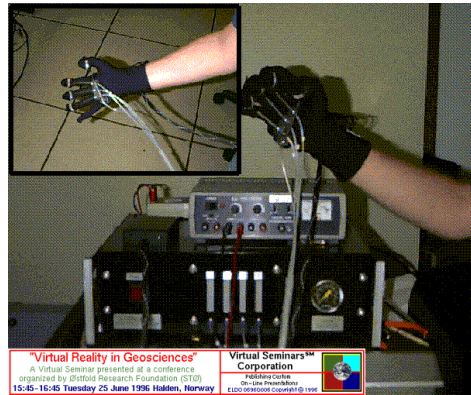
- Complex Instructional Manuals





## Haptics

- Force feedback is needed at very fast rates.
- Gloves
  - force resistant
  - nerve stimulated

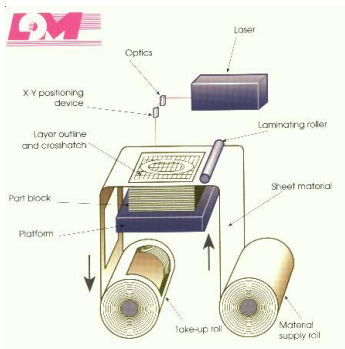


## Rapid Prototyping

- Build real models of the visualizations
- Stereo Lithography
  - Laser etching
- Laminated Object Manufacturing
  - Laminated paper layer, then cut with laser



## Laminated Object Manufacturing

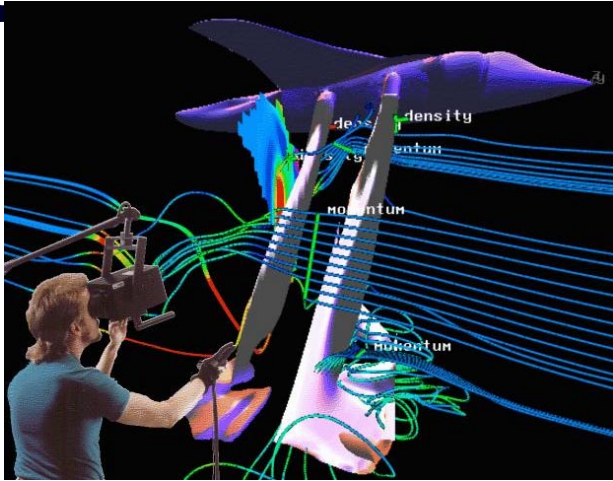


## Laminated Object Manufacturing

- Molecular Docking



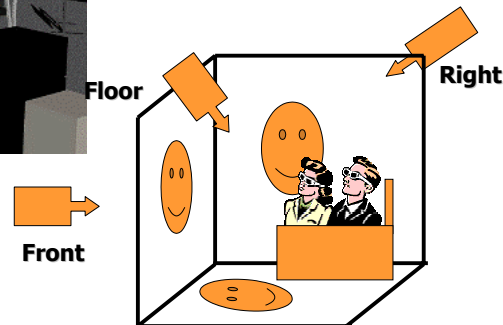
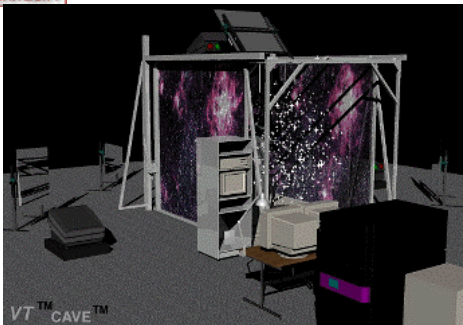
## NASA's Virtual Wind Tunnel



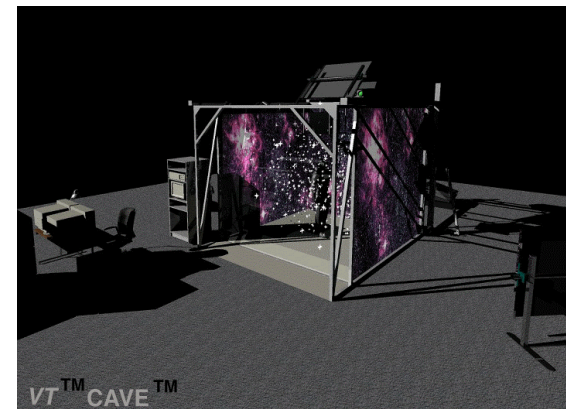
## The CAVE Architecture

- Four projection screens
- Four graphics rendering engines
- Stereo glasses
- Head-tracking of one user
- Hand held wand for input

## The CAVE Architecture

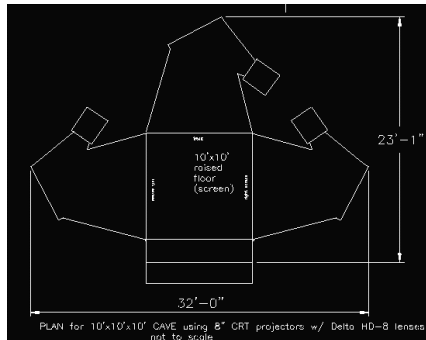


## The CAVE

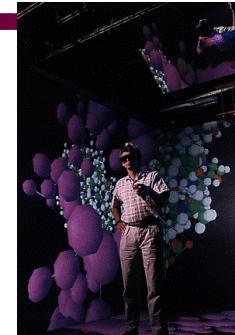
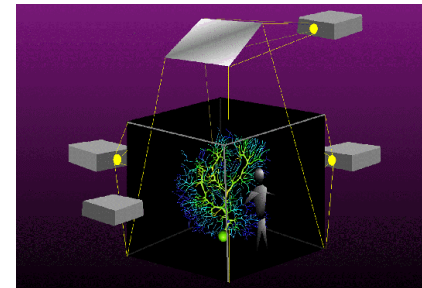




## The CAVE



## The CAVE



## The CAVE Architecture

- Several people can view at once
- The projections are only correct for one person.
- Laser's synch the stereo displays with Liquid Crystal shutter glasses on each viewer.

## The CAVE Architecture

- Benefits
  - Eye movement problems are avoided!!!
  - User's orientation does not matter.
  - Can see and examine real people and objects within the room



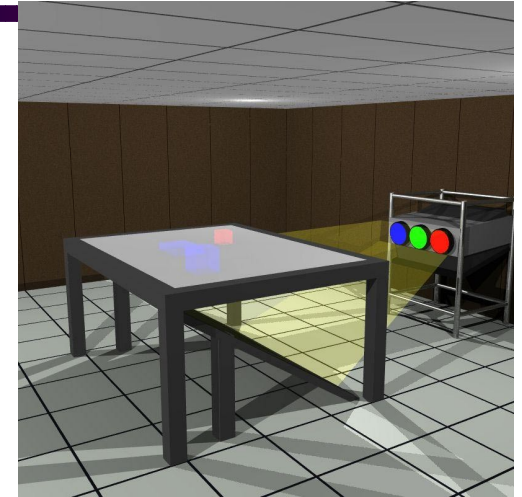
## *The CAVE Architecture*

- Problems
  - The light intensity on each projector varies
  - Precise alignment of the projectors is necessary for a smooth seam.
  - Viewing does not change for the other viewers.
  - Expensive.



## *Single Projector Systems*

- ImmersaDesk
- Responsive Workbench



## *Responsive Workbench*



## *Making VR Work*

- To ensure latency, many of the visualization techniques need to be streamlined or pre-computed.
- Examples, pre-computed iso-contours, precomputed stream lines and particle traces.



## Reading in Open Inventor files

- See:
- <http://www.cis.ohio-state.edu/~wenger/cis681/OSUInventorScene.html>
- **Download OSUInventor.C and OSUInventor.h:**
  - /usr/class/cis681/wenger/Src/OSUInventor
- **Download the sample read file sample\_read\_iv.C and Makefile:**
  - /usr/class/cis681/wenger/Src/sample\_read\_iv



## OpenGL - GLU

- GLUquadric\* gluNewQuadric( void )
- Sphere – gluSphere( quadric, radius, nslices, nstacks );
- Cylinder – Tapered cylinder
- Disk
- PartialDisk



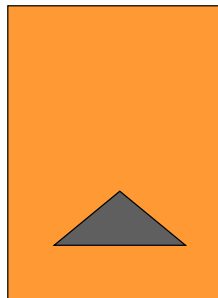
## General Polygons

- Allow for concave, self intersecting polygons:
- **EXAMPLE:** A quadrilateral with a triangular hole in it can be described as follows:

```

GLUtesselator* tobj gluNewTess()
gluTessBeginPolygon(tobj, NULL);
gluTessBeginContour(tobj);
gluTessVertex(tobj, v1, v1);
gluTessVertex(tobj, v2, v2);
gluTessVertex(tobj, v3, v3);
gluTessVertex(tobj, v4, v4);
gluTessEndContour(tobj);
gluTessBeginContour(tobj);
gluTessVertex(tobj, v5, v5);
gluTessVertex(tobj, v6, v6);
gluTessVertex(tobj, v7, v7);
gluTessEndContour(tobj);
gluTessEndPolygon(tobj);

```



## GLUT Geometric Primitives

- Sphere
  - glutSolidSphere( radius, slices, stacks )
  - glutWireSphere ( radius, slices, stacks )
- Cube – glutSolidCube( size ), ...
- Cone – glutSolidCone( base, height, slices, stacks), ...
- Torus – glutSolidTorus( inner, outer, nsides, rings), ...
- Tetrahedron – glutSolidTetrahedron(), ... (4-sided) ( $\sqrt{3}$ )
- Octahedron – glutSolidOctahedron(), ... (8-sided)
- Icosahedron – glutSolidOctahedron(), ... (12-sided)
- Dodecahedron – glutSolidOctahedron(), ... (20-sided) ( $\sqrt{3}$ )
- Teapot – glutSolidTeapot( size)

# Clipping

CIS 781

Roger Crawfis



# Why do clipping?

- Clipping is a visibility preprocess. In real-world scene clipping can remove a substantial percentage of the environment from consideration.
- Clipping offers an important optimization
- Also need to avoid setting pixel values outside of the range.

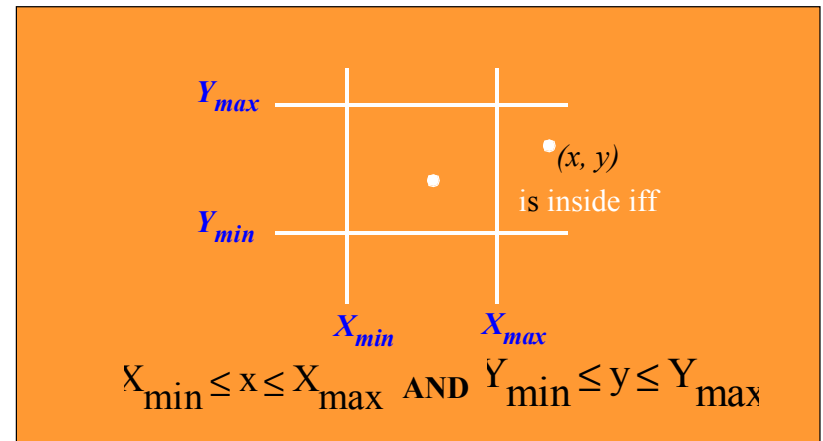


# What is clipping, two views

- Clipping *spatially partitions* geometric primitives, according to their containment within some region. Clipping can be used to:
  - Distinguish whether geometric primitives are inside or outside of a *viewing frustum* or *picking frustum*
  - Detect intersections between primitives
- Clipping *subdivides* geometric primitives. Several other potential applications.
  - Binning geometric primitives into spatial data structures
  - computing analytical shadows.

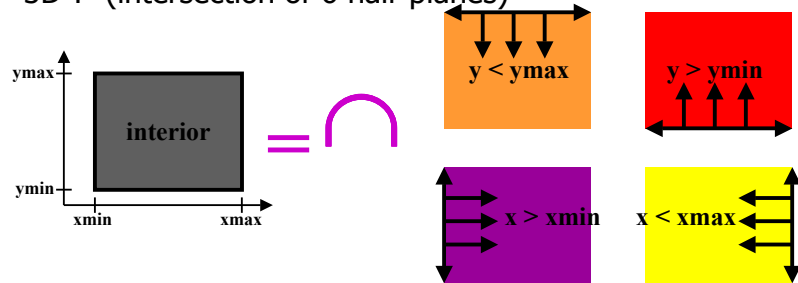


# Point Clipping



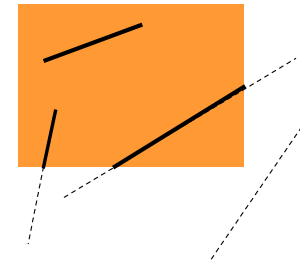
## Line Clipping - Half Plane Tests

- Modify endpoints to lie in rectangle
- "Interior" of rectangle?
- Answer: intersection of 4 half-planes
- 3D ? (intersection of 6 half-planes)



## Line Clipping

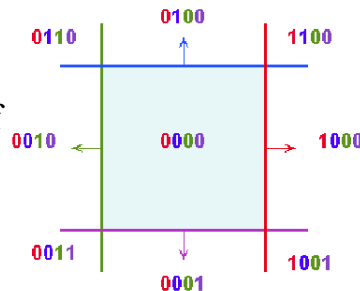
- Is end-point inside a clip region? - half-plane test
- If outside, calculate intersection between line and the clipping rectangle and make this the new end point



- Both endpoints inside: trivial accept
- One inside: find intersection and clip
- Both outside: either clip or reject (tricky case)

## Cohen-Sutherland Algorithm (Outcode clipping)

- Classifies each vertex of a primitive, by generating an *outcode*. An outcode identifies the appropriate half space location of each vertex relative to all of the clipping planes. Outcodes are usually stored as bit vectors.

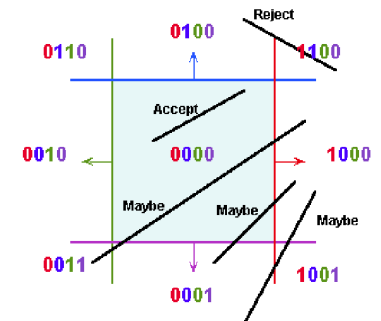


## Cohen-Sutherland Algorithm (Outcode clipping)

```

if (outcode1 == '0000' and outcode2 == '0000') then
    line segment is inside
else
    if ((outcode1 AND outcode2) == 0000) then
        line segment potentially crosses clip region
    else
        line is entirely outside of clip region
    endif
endif

```



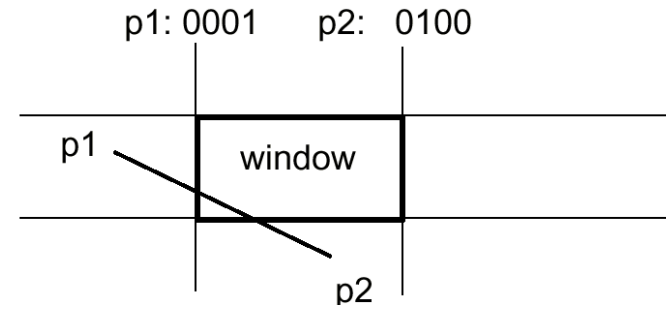


## The Maybe cases?

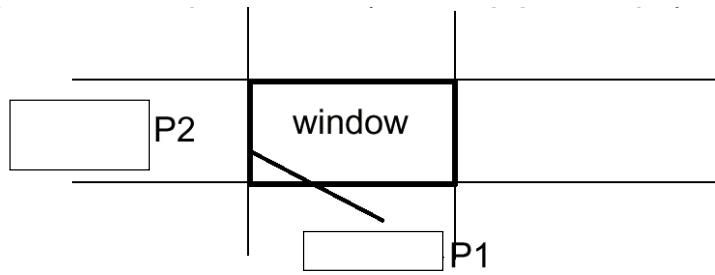
- If neither trivial accept nor reject:
  - Pick an outside endpoint (with nonzero outcode)
  - Pick an edge that is crossed (nonzero bit of outcode)
  - Find line's intersection with that edge
  - Replace outside endpoint with intersection point
  - Repeat until trivial accept or reject



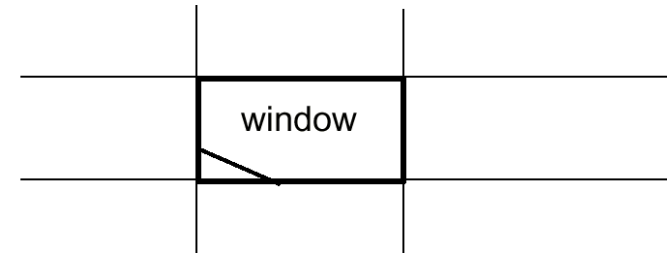
## The Maybe case



## The Maybe Case



## The Maybe Case







## Difficulty

- This clipping will handle most cases. However, there is one case in general that cannot be handled this way.
  - Parts of a primitive lie both in front of and behind the viewpoint. This complication is caused by our projection stage.
  - It has the nasty habit of mapping objects in behind the viewpoint to positions in front of it.



## One Plane At a Time Clipping

- (a.k.a. Sutherland-Hodgeman Clipping)
- The Sutherland-Hodgeman triangle clipping algorithm uses a *divide-and-conquer* strategy.
- Clip a triangle against a single plane. Each of the clipping planes are applied in succession to every triangle.
- There is minimal storage requirements for this algorithm, and it is well suited for pipelining.
- It is often used in hardware implementations.

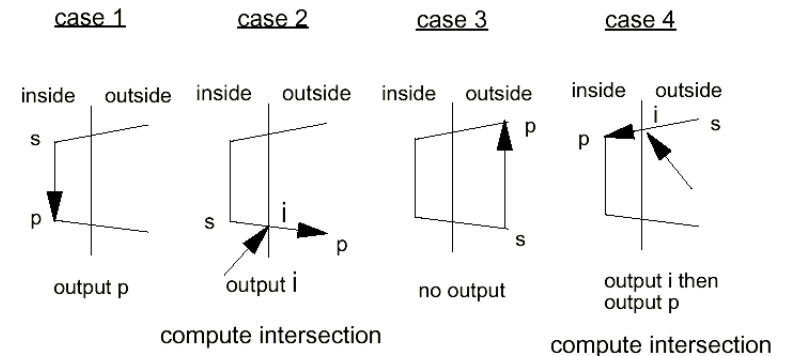


## Sutherland-Hodgman Polygon Clipping Algorithm

- Clip a polygon (input: vertex list) against a single clip edges
- Output the vertex list(s) for the resulting clipped polygon(s)
- Clip against all four planes
  - Generalizes to 3D (6 planes)
  - Generalizes to clip against any convex polygon/polyhedron
- Used in viewing transforms



## Sutherland-Hodgman Polygon Clipping Algorithm





## Sutherland-Hodgman

**SHclippedge**(var: ilit, olist: list; ilen, olen, edge : integer)

```

s = ilit[ilen];  olen = 0;
for i = 1 to ilen do
  d := ilit[i];
  if (inside(d, edge) then
    if (inside(s, edge) then          -- case 1 just add d
      addlist(d, olist);  olen := olen + 1;
    else                               -- case 4 add new intersection pt. and d
      n := intersect(s, d, edge);
      addlist(n, olist);  addlist(d, olist);  olen = olen + 2;
    else if (inside(s, edge) then      -- case 2 add new intersection pt.
      n := intersect(s, d, edge); addlist(n, olist);  olen ++; s = d;
  end_for;

```

Clip input polygon *ilit* to the edge, *edge*, and output the new polygon.



## Sutherland-Hodgman

**SHclip**(var: ilit, olist: list; ilen, olen : integer)

```

{
  SHclippedge(ilit, tmplist1, ilen, tlen1, RIGHT);
  SHclippedge(tmplist1, tmplist2, tlen1, tlen2, BOTTOM);
  SHclippedge(tmplist2, tmplist1, tlen2, tlen1, LEFT);
  SHclippedge(tmplist1, olist, tlen1, olen, TOP);
}

```



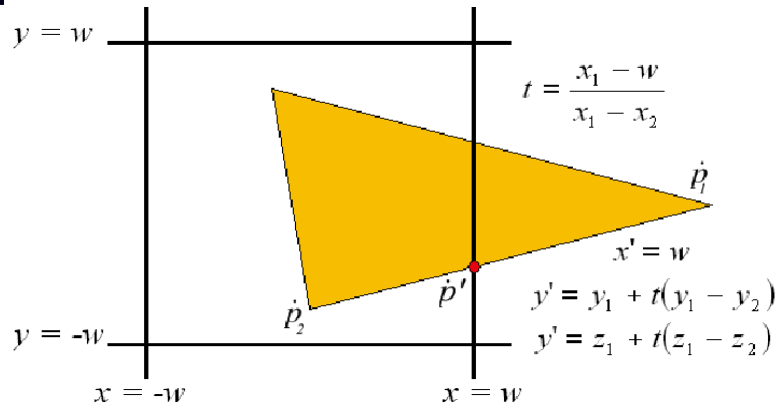
## Pictorial Example



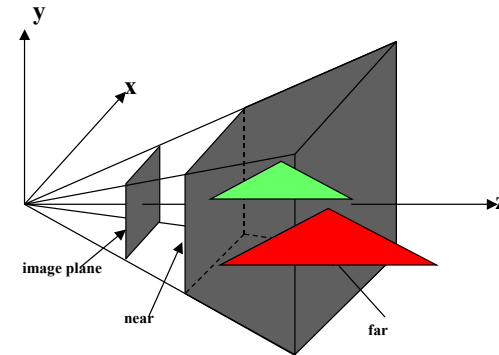
## Sutherland-Hodgman

- Advantages:
  - Elegant (few special cases)
  - Robust (handles boundary and edge conditions well)
  - Well suited to hardware
  - Canonical clipping makes fixed-point implementations manageable
- Disadvantages:
  - Only works for convex clipping volumes
  - Often generates more than the minimum number of triangles needed
  - Requires a divide per edge

## Interpolating Parameters



## 3D Clipping (Planes)



## 4D Polygon Clip

- Use Sutherland Hodgman algorithm
- Use arrays for input and output lists
- There are six planes of course !

## 4D Clipping

- OpenGL uses  $-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 1$
- We use:  $-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 0$
- Must clip in homogeneous coordinates:
  - $w > 0$ :  $-w \leq x \leq w, -w \leq y \leq w, -w \leq z \leq 0$
  - $w < 0$ :  $-w > x > w, -w > y > w, -w > z > 0$
- Consider each case separately
- What issues arise ?

## 4D Clipping

- Point A is inside, Point B is outside. Clip edge AB

$$x = Ax + t(Bx - Ax)$$

$$y = Ay + t(By - Ay)$$

$$z = Az + t(Bz - Az)$$

$$w = Aw + t(Bw - Aw)$$

- Clip boundary:  $x/w = 1$  i.e.  $(x-w=0)$ ;

$$w-x = Aw - Ax + t(Bw - Aw - Bx + Ax) = 0$$

Solve for t.

## Why Homogeneous Clipping

- Efficiency/Uniformity: A single clip procedure is typically provided in hardware, optimized for canonical view volume.
- The perspective projection canonical view volume can be transformed into a parallel-projection view volume, so the same clipping procedure can be used.
- But for this, clipping must be done in homogenous coordinates (and not in 3D). Some transformations can result in negative W : 3D clipping would not work.

## Difficulty (revisit)

- Clipping will handle most cases. However, there is one case in general that cannot be handled this way.
  - Parts of a primitive lie both in front of and behind the viewpoint. This complication is caused by our projection stage.
  - It has the nasty habit of mapping objects in behind the viewpoint to positions in front of it.
- Solution: clip in homogeneous coordinate

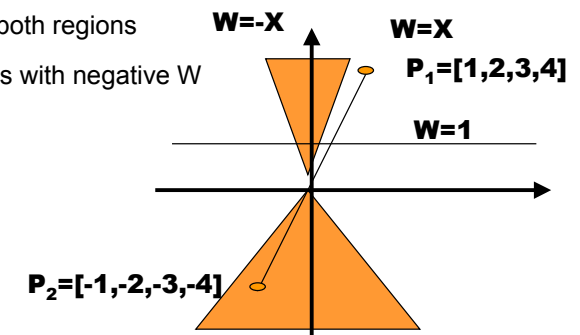
## 4D Clipping Issues

- $P_1$  and  $P_2$  map to same physical point !

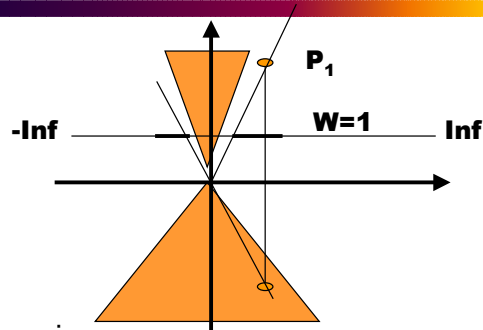
- Solution:

- Clip against both regions

- Negate points with negative W



# 4D Clipping Issues



- Line straddles both regions
- After projection one gets two line segments
- How to do this? Only before the perspective division

# Additional Clipping Planes



- At least 6 more clipping planes available
- Good for cross-sections
- Modelview matrix moves clipping plane
- $Ax + By + Cz + D < 0$  clipped
- `glEnable( GL_CLIP_PLANEi )`
- `glClipPlane( GL_CLIP_PLANEi, GLdouble* coeff )`

# Reversing Coordinate Projection



- Screen space back to world space
- `glGetIntegerv( GL_VIEWPORT, GLint viewport[4] )`
- `glGetDoublev( GL_MODELVIEW_MATRIX, GLdouble mvmatrix[16] )`
- `glGetDoublev( GL_PROJECTION_MATRIX, GLdouble projmatrix[16] )`
- `gluUnProject( GLdouble winx, winy, winz, mvmatrix[16], projmatrix[16], GLint viewport[4], GLdouble *objx, *objy, *objz )`
- `gluProject` goes from world to screen space

# Shaders

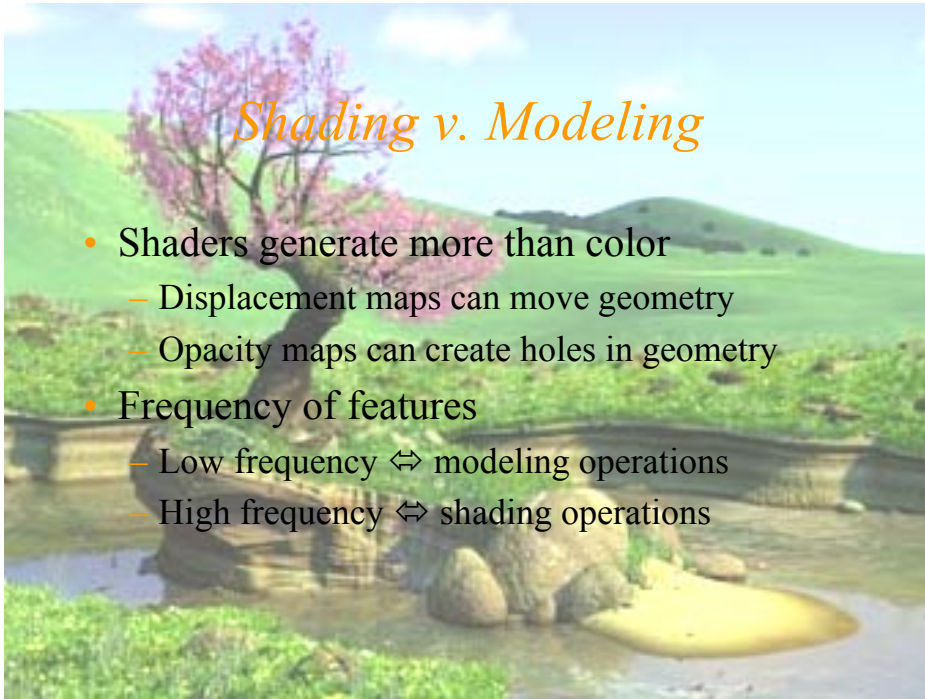


- Local illumination quite complex
  - Reflectance models
  - Procedural texture
  - Solid texture
  - Bump maps
  - Displacement maps
  - Environment maps
- Need ability to collect into a single shading description called a *shader*
- Shaders also describe
  - lights, e.g. spotlights
  - atmosphere, e.g. fog



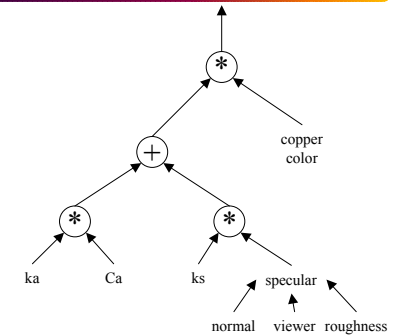
## Shading v. Modeling

- Shaders generate more than color
  - Displacement maps can move geometry
  - Opacity maps can create holes in geometry
- Frequency of features
  - Low frequency  $\Leftrightarrow$  modeling operations
  - High frequency  $\Leftrightarrow$  shading operations

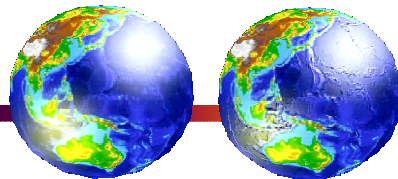


## Shade Trees

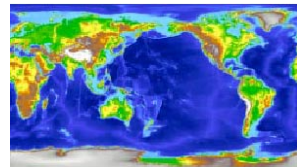
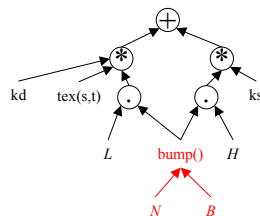
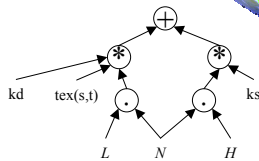
- Cook, SIGGRAPH 84
- Hierarchical organization of shading
- Breaks a shading expression into simple components
- Visual programming
- Modular
- Drag-n-drop shading components



## Texture v. Bump Mapping



- Texture mapping simulates detail with a color that varies across a surface
- Bump mapping simulates detail with a surface normal that varies across a surface



## Problems with Shade Trees

- Shaders can get *very* complex
- Sometimes need higher-level constructs than simple expression trees
  - Variables
  - Iteration
- Need to compile a program instead of evaluate an expression





## Renderman Shading Language

- Hanrahan & Lawson, SIGGRAPH 90
- High level little language
- Special purpose variables useful for shading
  - P – surface position
  - N – surface normal
- Special purpose functions useful for shading
  - smoothstep( $x_0, x_1, a$ ) – smoothly interpolates from  $x_0$  to  $x_1$  as  $a$  varies from 0 to 1
  - specular( $N, V, m$ ) – computes specular reflection given normal  $N$ , view direction  $V$  and roughness  $m$ .



## Types

- Colors
  - Multiplication is componentwise
  - e.g.  $Cd*(La + Ld) + Cs*Ls + Ct*Lt$
- Points
  - Built in dot ( $L.N$ ) and cross ( $N^L$ ) products
  - Transform to other coordinate systems: “raster,” “screen,” “camera,” “world,” and “object”
- Variables
  - Uniform – independent of position
  - Varying – changes across surface



## Lighting

- Constructs
  - illuminate() – point source with cone spread
  - solar() – directional source
- Variables
  - L – direction of light (independent)
  - Cl – color of light (dependent)
- Types
  - ambient – non-directional (but can vary with position)
  - point – equal in all directions
  - spot – focused around a given direction
  - shadowed – modulated by texture/shadow map
  - distant – directional source
  - environment map – distant source modulated by texture



## Local Illumination

- Construct
  - illuminance()
- Variables
  - L – incoming light direction
  - Cl – incoming light color
  - C – output color
- Example (hair diffuse)
 

```
color C = 0;
illuminance(P,N,Pi/2) {
  L = normalize(L);
  C += Kd * Cd * Cl * length(L^T);
}
```

## Texture Functions

- texture() returns float/color based on texture coordinates
- bump() returns normal perturbation based on texture coordinates
- environment() returns float/color based on a direction passed to it
- shadow() returns a float indicating the percentage a point's position is shadowed

## Renderman Example



```
Surface dent(float Ks=.4, Kd=.5, Ka=.1, roughness=.25, dent=.4) {
    float turbulence;
    point Nf, V;
    float I, freq;
    /* Transform to solid texture coordinate system */
    V = transform("shader",P);
    /* Sum 6 octaves of noise to form turbulence */
    turbulence = 0; freq = 1.0;
    for (i = 0; i < 6; i += 1) {
        turbulence += 1/freq + abs(0.5*noise(4*freq*V));
        freq *= 2;
    }
    /* sharpen turbulence */
    turbulence *= turbulence * turbulence;
    turbulence *= dent;
    /* Displace surface and compute normal */
    P -= turbulence * normalize(N);
    Nf = faceforward(normalize(calculatenormal(P)),I);
    V = normalize(-I);
    /* Perform shading calculations */
    Oi = 1 - smoothstep(0.03,0.05,turbulence);
    Ci = Oi*Cs*(Ka*ambient() + Ks*specular(Nf,V,roughness));
}
```

## Try It Yourself



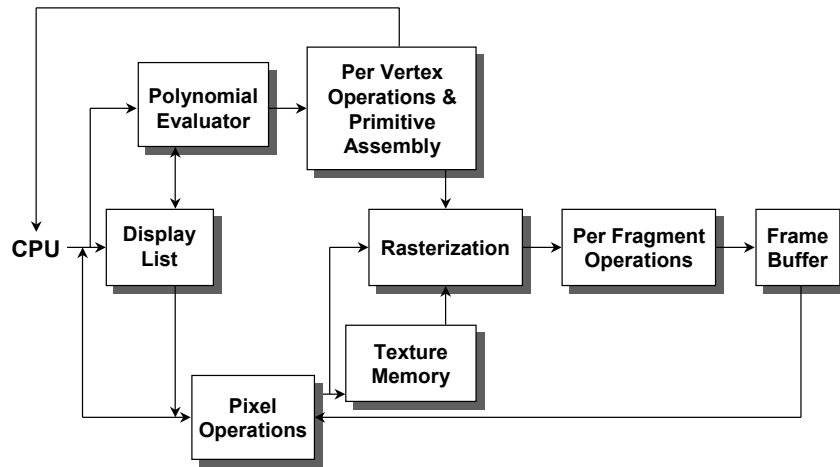
- Photorealistic Renderman
  - Based on REYES polygon renderer
  - Uses shadow maps
- Blue Moon Rendering Tools
  - Free
  - Uses ray tracer
  - No displacement maps
  - <http://www.exluna.com/products/bmrt/>

## Deferred Shading

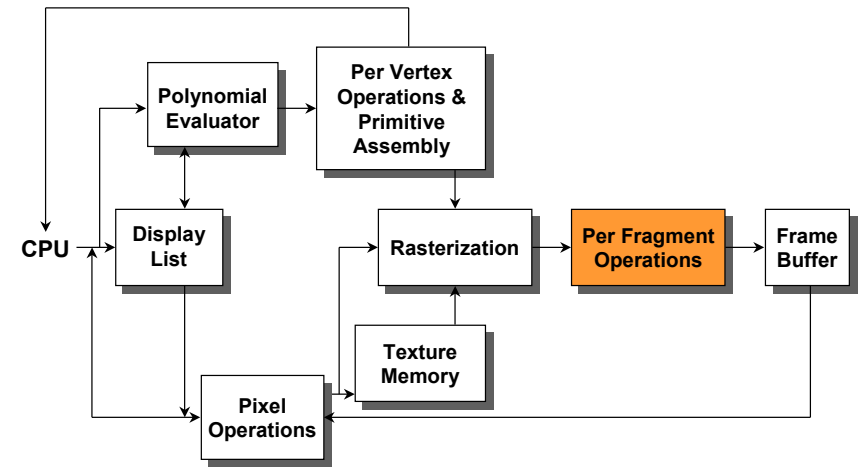
- Makes procedural shading more efficient
- Why apply shader to entire surface if only small portion is actually visible
- Separate rendering into two passes
  - Pass 1: Render geometry using Z-buffer
    - But rather than storing color in frame buffer
    - Store shading parameters instead
  - Pass 2: Shade frame buffer
    - Apply shading procedure to frame buffer
    - Replaces shading parameters with color
- Problem: Fat framebuffer



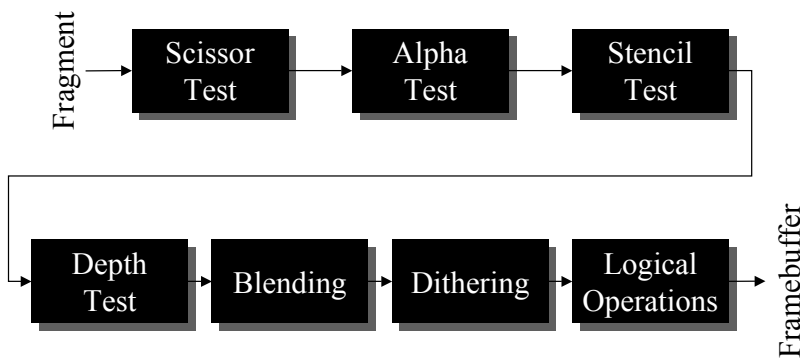
# OpenGL Architecture



# Per-Fragment Operations



# Getting to the Framebuffer



# Scissor Box

- Additional Clipping Test
  - `glScissor( x, y, w, h )`
    - any fragments outside of box are clipped
    - useful for updating a small section of a viewport
      - affects `glClear()` operations

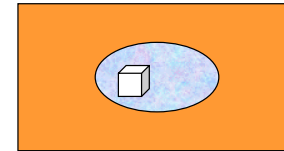
## Alpha Test

- Reject pixels based on their alpha value
  - `glAlphaFunc( func, value )`
  - `glEnable( GL_ALPHA_TEST )`
- use alpha as a mask in textures



## Stencil Buffer

- Used to control drawing based on values in the stencil buffer
  - Fragments that fail the stencil test are not drawn
  - Example: create a mask in stencil buffer and draw only objects not in mask area



## Stencil Testing

- Now broadly supports by both major APIs
  - OpenGL
  - DirectX 6
- RIVA TNT and other consumer cards now supporting full 8-bit stencil
- Opportunity to achieve new cool effects and improve scene quality

## What is Stenciling?

- Per-pixel test, similar to depth buffering.
- Tests against value from stencil buffer; rejects fragment if stencil test fails.
- Distinct stencil operations performed when
  - Stencil test fails
  - Depth test fails
  - Depth test passes
- Provides fine grain control of pixel update



## OpenGL API

- glEnable/glDisable(GL\_STENCIL\_TEST);
- glStencilFunc(function, reference, mask);
- glStencilOp(stencil\_fail, depth\_fail, depth\_pass);
- glStencilMask(mask);
- glClear(... | GL\_STENCIL\_BUFFER\_BIT);



## Controlling Stencil Buffer

- glStencilFunc( *func*, *ref*, *mask* )
  - compare value in buffer with **ref** using **func**
  - only applied for bits in **mask** which are 1
  - **func** is one of standard comparison functions
- glStencilOp( *fail*, *zfail*, *zpass* )
  - Allows changes in stencil buffer based on passing or failing stencil and depth tests: **GL\_KEEP**, **GL\_INCR**



## Request a Stencil Buffer

- If using stencil, request sufficient bits of stencil
- Implementations may support from zero to 32 bits of stencil
- 8, 4, or 1 bit are common possibilities
- Easy for GLUT programs:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |  
    GLUT_DEPTH | GLUT_STENCIL);  
glutCreateWindow("stencil example");
```



## Stencil Test

- Compares reference value to pixel's stencil buffer value
- Same comparison functions as depth test:
  - NEVER, ALWAYS
  - LESS, LEQUAL
  - GREATER, GEQUAL
  - EQUAL, NOTEQUAL
- Bit mask controls comparison ((ref & mask) op (svalue & mask))

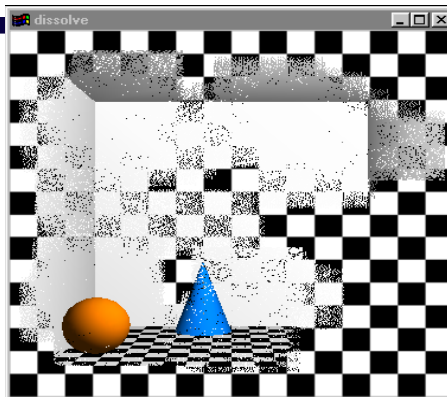
## Stencil Operations

- Stencil side effects of
  - Stencil test fails
  - Depth test fails
  - Depth test passes
- Possible operations
  - Increment, Decrement (saturates)
  - Increment, Decrement (wrap, DX6 option)
  - Keep, Replace
  - Zero, Invert
- Way stencil buffer values are controlled

## Stencil Write Mask

- Bit mask for controlling write back of stencil value to the stencil buffer
- Applies to the clear too!
- Stencil compare & write masks allow stencil values to be treated as sub-fields

## Very Complex Clip Window



Digital Dissolve

## Creating a Mask

- `gluInitDisplayMode (... | GLUT_STENCIL | ...);`
- `glEnable( GL_STENCIL_TEST );`
- `glClearStencil( 0x0 );`
- `glStencilFunc( GL_ALWAYS, 0x1, 0x1 );`
- `glStencilOp( GL_REPLACE, GL_REPLACE, GL_REPLACE );`
- *draw mask*





## Using Stencil Mask

- Draw objects where stencil = 1
  - `glStencilFunc( GL_EQUAL, 0x1, 0x1 )`
- Draw objects where stencil != 1
  - `glStencilFunc( GL_NOTEQUAL, 0x1, 0x1 )`;
  - `glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP )`;
- 



## Performance

- With today's 32-bit graphics accelerator modes, 24-bit depth and 8-bit stencil packed in *same* memory word
- RIVA TNT is an example
- Performance implication:  
  
if using depth testing, stenciling is at  
NO PENALTY



## Repeating that!

- On card like RIVA TNT2 in 32-bit mode  
  
if using depth testing, stenciling has  
NO PENALTY
- Do not treat stencil as “expensive” --  
in fact, treat stencil as “free” when already  
depth testing



## Pseudo Global Lighting Effects

- OpenGL's light model is a “local” model
  - Light source parameters
  - Material parameters
  - Nothing else enters the equation
- Global illumination is fancy word for real-world light interactions
  - Shadows, reflections, refractions, radiosity, etc.
- Pseudo global lighting is about clever hacks

## Planar Reflections

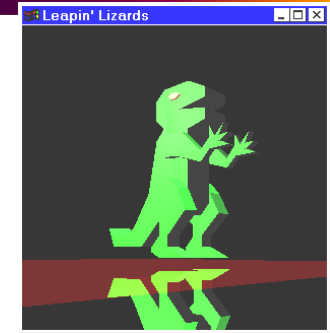


Dinosaur is reflected by the planar floor.  
 Easy hack, draw dino twice, second time has  
`glScalef(1, -1, 1)` to reflect through the floor

## Compare Two Versions



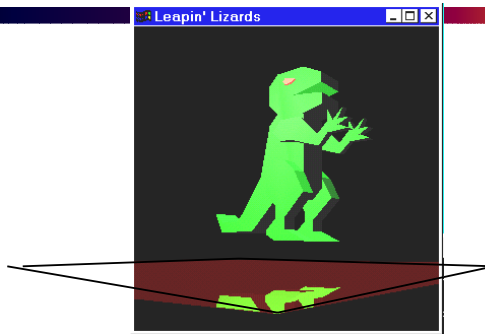
Good.



Bad.

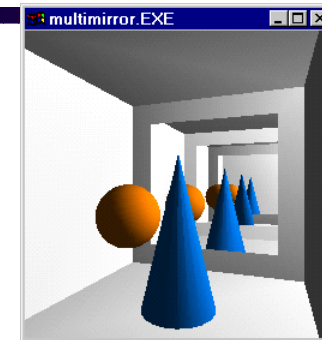
Notice right image's reflection falls off the floor!

## Stencil Maintains the Floor



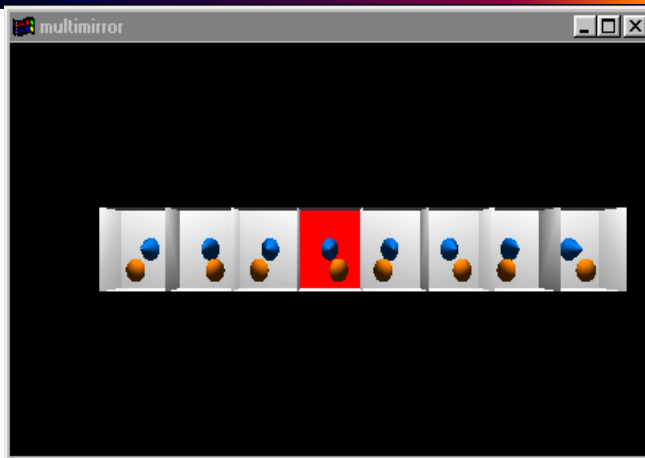
Clear stencil to zero.  
 Draw floor polygon with stencil set to one.  
 Only draw reflection where stencil is one.

## Recursive Planar Mirrors



Basic idea of planar reflections can be applied recursively. Requires more stencil bits.

## The Trick (bird's eye view)



## Next: Planar Shadows



Shadow is projected into the plane of the floor.

## Constructing a Shadow Matrix

```
void shadowMatrix(GLfloat shadowMat[4][4], GLfloat groundplane[4], GLfloat lightpos[4])
{
    GLfloat dot;
    /* Find dot product between light position vector and ground plane normal. */
    dot = groundplane[X] * lightpos[X] +
          groundplane[Y] * lightpos[Y] +
          groundplane[Z] * lightpos[Z] +
          groundplane[W] * lightpos[W];
    shadowMat[0][0] = dot - lightpos[X] * groundplane[X];
    shadowMat[1][0] = 0.f - lightpos[X] * groundplane[Y];
    shadowMat[2][0] = 0.f - lightpos[X] * groundplane[Z];
    shadowMat[3][0] = 0.f - lightpos[X] * groundplane[W];
    shadowMat[X][1] = 0.f - lightpos[Y] * groundplane[X];
    shadowMat[1][1] = dot - lightpos[Y] * groundplane[Y];
    shadowMat[2][1] = 0.f - lightpos[Y] * groundplane[Z];
    shadowMat[3][1] = 0.f - lightpos[Y] * groundplane[W];
    shadowMat[X][2] = 0.f - lightpos[Z] * groundplane[X];
    shadowMat[1][2] = 0.f - lightpos[Z] * groundplane[Y];
    shadowMat[2][2] = dot - lightpos[Z] * groundplane[Z];
    shadowMat[3][2] = 0.f - lightpos[Z] * groundplane[W];
    shadowMat[X][3] = 0.f - lightpos[W] * groundplane[X];
    shadowMat[1][3] = 0.f - lightpos[W] * groundplane[Y];
    shadowMat[2][3] = 0.f - lightpos[W] * groundplane[Z];
    shadowMat[3][3] = dot - lightpos[W] * groundplane[W];
}
```

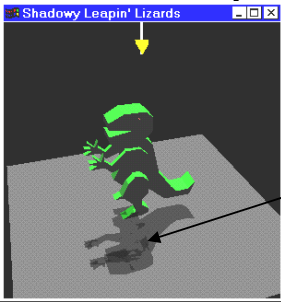
## How to Render the Shadow

```
/* Render 50% black shadow color on top of whatever
the floor appearance is. */
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
            GL_ONE_MINUS_SRC_ALPHA);
glDisable(GL_LIGHTING); /* Force the 50% black. */
glColor4f(0.0, 0.0, 0.0, 0.5);

glPushMatrix();
/* Project the shadow. */
glMultMatrixf((GLfloat *) floorShadow);
drawDinosaur();
glPopMatrix();
```

## Note Quite So Easy (1)

Without stencil to avoid double blending of the shadow pixels:

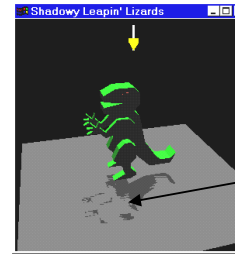


Notice darks spots on the planar shadow.

Solution: Clear stencil to zero. Draw floor with stencil of one. Draw shadow if stencil is one. If shadow's stencil test passes, set stencil to two. No double blending.

## Note Quite So Easy (2)

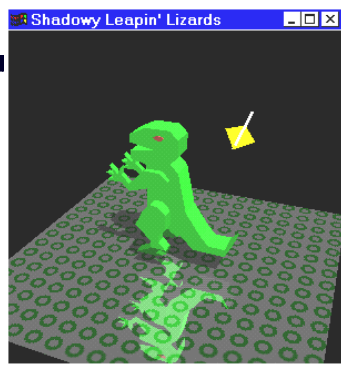
There's still another problem even if using stencil to avoid double blending.



depth buffer Z fighting artifacts

Shadow fights with depth values from the floor plane. Use polygon offset to raise shadow polygons slightly in Z.

## Everything All At Once

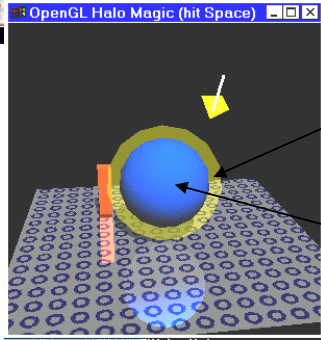


Lighting, texturing, planar shadows, and planar reflections all at one time. Stencil & polygon offset eliminate aforementioned artifacts.

## Pseudo Global Lighting

- Planar reflections and shadows add more than simplistic local lighting model
- Still not really global
  - Techniques more about hacking common cases based on knowledge of geometry
  - Not really modeling underlying physics of light
- Techniques are “multipass”
  - Geometry is rendered multiple times to improve the rendered visual quality

## Bonus Stenciled Halo Effect



Halo is blended with objects behind haloed object.

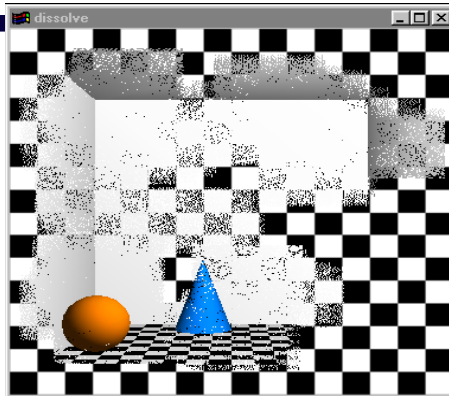
Halo does not obscure or blend with the haloed object.

Clear stencil to zero. Render object, set stencil to one where object is. Scale up object with `glScalef`. Render object again, but not where stencil is one.

## Other Stencil Uses

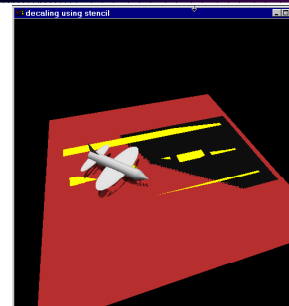
- Digital dissolve effects
- Handling co-planar geometry such as decals
- Measuring depth complexity
- Constructive Solid Geometry (CSG)

## Digital Dissolve

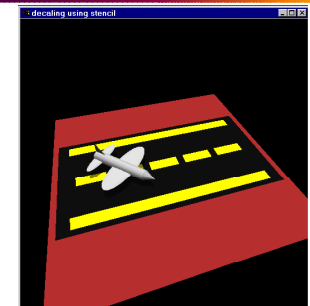


Stencil buffer holds dissolve pattern.  
Stencil test two scenes against the pattern

## Co-planar Geometry



Shows “Z fighting” of co-planar geometry



Stencil testing fixes “Z fighting”

## Visualizing Depth Complexity



Use stencil to count pixel updates,  
then color code results.

## Dithering

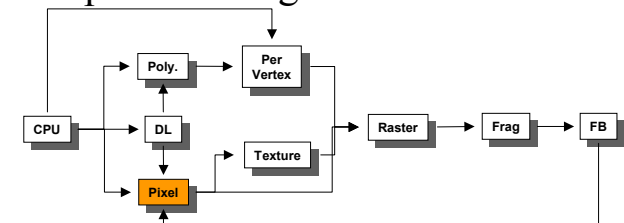
- `glEnable( GL_DITHER )`
- Dither colors for better looking results
  - Used to simulate more available colors

## Logical Operations on Pixels

- Combine pixels using bitwise logical operations
  - `glLogicOp( mode )`
- Common modes
  - `GL_XOR` – Rubberband user-interface.
  - `GL_AND`
- Others
  - `GL_CLEAR`, `GL_SET`, `GL_COPY`,
  - `GL_COPY_INVERTED`, `GL_NOOP`, `GL_INVERT`
  - `GL_AND`, `GL_NAND`, `GL_OR`
  - `GL_NOR`, `GL_XOR`, `GL_AND_INVERTED`
  - `GL_AND_REVERSE`, `GL_EQUIV`, `GL_OR_REVERSE`
  - `GL_OR_INVERTED`

## Imaging and Raster Primitives

- Describe OpenGL's raster primitives: bitmaps and image rectangles
- Demonstrate how to get OpenGL to read and render pixel rectangles



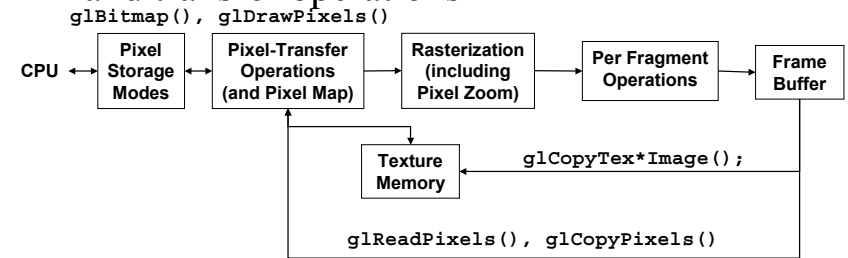


## Pixel-based primitives

- Bitmaps
  - 2D array of bit masks for pixels
    - update pixel color based on current color
- Images
  - 2D array of pixel color information
    - complete color information for each pixel
- OpenGL doesn't understand image formats

## Pixel Pipeline

- Programmable pixel storage and transfer operations



May 22-26, 2000

Dagstuhl Visualization

## Positioning Image Primitives

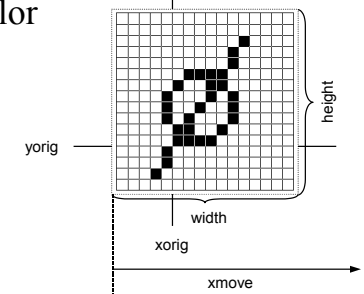
- `glRasterPos3f( x, y, z )`
  - raster position transformed like geometry
  - discarded if raster position is outside of viewport
    - may need to fine tune viewport for desired results



Raster Position

## Rendering Bitmaps

- `glBitmap( width, height, xorig, yorig, xmove, ymove, bitmap )`
  - render bitmap in current color at  $(\lfloor x - xorig \rfloor \lfloor y - yorig \rfloor)$
  - advance raster position by  $(xmove \ ymove)$  after rendering





## Rendering Fonts using Bitmaps

- OpenGL uses bitmaps for font rendering
  - each character is stored in a display list containing a bitmap
  - window system specific routines to access system fonts
    - `glXUseXFont()`
    - `wglUseFontBitmaps()`



## Rendering Images

- `glDrawPixels( width, height, format, type, pixels )`
  - render pixels with lower left of image at current raster position
  - numerous formats and data types for specifying storage in memory
    - best performance by using format and type that matches hardware



## Reading Pixels

- `glReadPixels( x, y, width, height, format, type, pixels )`
  - read pixels from specified (x,y) position in framebuffer
  - pixels automatically converted from framebuffer format into requested format and type
- Framebuffer pixel copy
  - `glCopyPixels( x, y, width, height, type )`



## Pixel Zoom

- `glPixelZoom( x, y )`
  - expand, shrink or reflect pixels around current raster position
  - fractional zoom supported

Raster Position → `glPixelZoom(1.0, -1.0);`





## Storage and Transfer Modes

- Storage modes control accessing memory
  - byte alignment in host memory
  - extracting a subimage
- Transfer modes allow modify pixel values
  - scale and bias pixel component values
  - replace colors using pixel maps

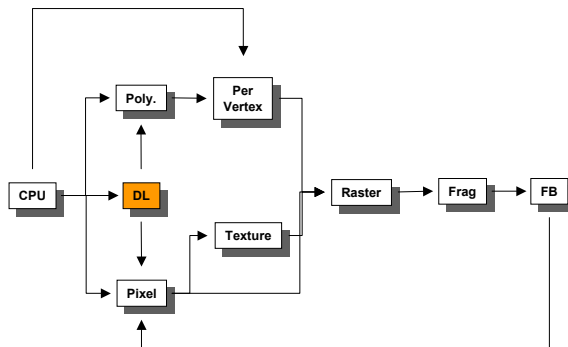


## Immediate Mode versus Display Listed Rendering

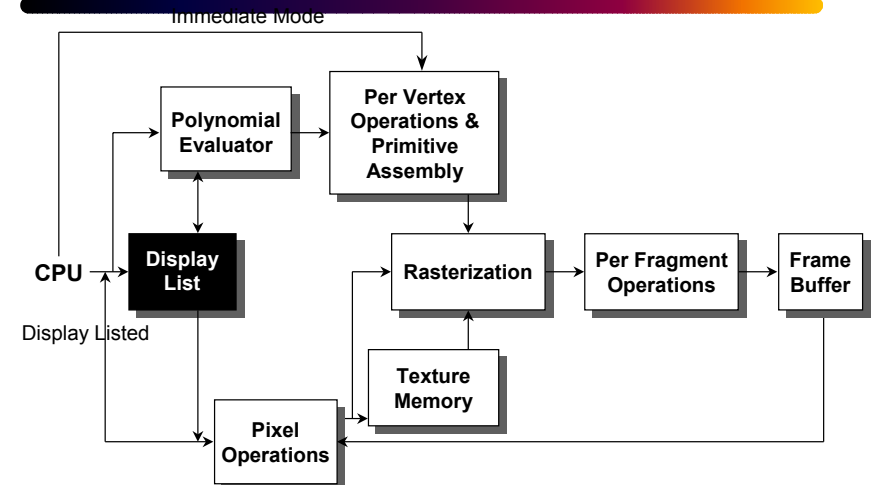
- Immediate Mode Graphics
  - Primitives are sent to pipeline and display right away
  - No memory of graphical entities
- Display Listed Graphics
  - Primitives placed in display lists
  - Display lists kept on graphics server
  - Can be redisplayed with different state
  - Can be shared among OpenGL graphics contexts



## Display Lists



## Immediate Mode versus Display Lists



## Display Lists

- Creating a display list

```
GLuint id;
void init( void )
{
    id = glGenLists( 1 );
    glNewList( id, GL_COMPILE );
    /* other OpenGL routines */
    glEndList();
}
```

- Call a created list

```
void display( void )
{
    glCallList( id );
}
```

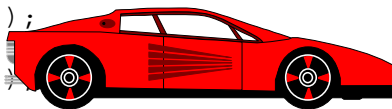
## Display Lists

- Not all OpenGL routines can be stored in display lists
- State changes persist, even after a display list is finished
- Display lists can call other display lists
- Display lists are not editable, but you can fake it
  - make a list (A) which calls other lists (B, C, and D)
  - delete and replace B, C, and D, as needed

## Display Lists and Hierarchy

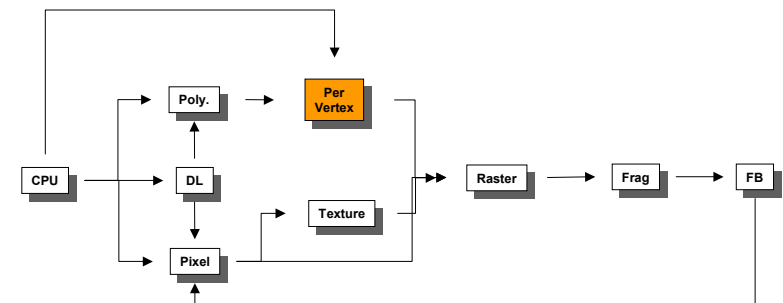
- Consider model of a car
  - Create display list for chassis
  - Create display list for wheel

```
glNewList( CAR, GL_COMPILE );
glCallList( CHASSIS );
glTranslatef( ... );
glCallList( WHEEL );
glTranslatef( ... );
glCallList( WHEEL );
...
glEndList();
```



## Advanced Primitives

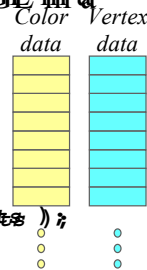
- **Vertex Arrays**



## Vertex Arrays

- Pass arrays of vertices, colors, etc. to OpenGL in a large chunk

```
glVertexPointer( 3, GL_FLOAT, 0, coords )
glColorPointer( 4, GL_FLOAT, 0, colors )
glEnableClientState( GL_VERTEX_ARRAY )
glEnableClientState( GL_COLOR_ARRAY )
glDrawArrays( GL_TRIANGLE_STRIP, 0, numVerts );
```



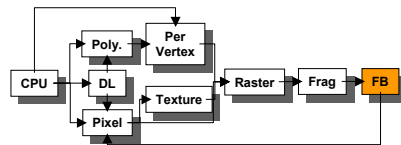
- All active arrays are used in rendering

## Why use Display Lists or Vertex Arrays?

- May provide better performance than immediate mode rendering
  - Avoid function call overheads and small packet sends.
- Display lists can be shared between multiple OpenGL context
  - reduce memory usage for multi-context applications
- Vertex arrays may format data for better memory access

## Alpha: the 4<sup>th</sup> Color Component

- Measure of Opacity
  - simulate translucent objects
    - glass, water, etc.
  - composite images
  - antialiasing
  - ignored if blending is not enabled



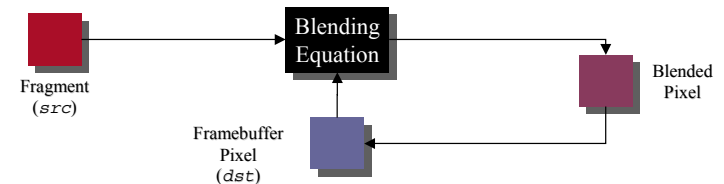
```
glEnable( GL_BLEND )
```

## Blending

- Combine pixels with what's in already in the framebuffer

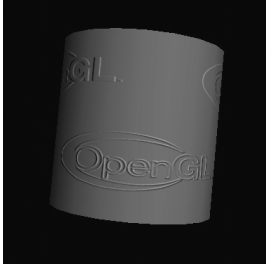
- `glBlendFunc( src, dst )`

$$\bar{C}_r = src \bar{C}_f + dst \bar{C}_p$$



## *Multi-pass Rendering*

- Blending allows results from multiple drawing passes to be combined together
  - enables more complex rendering algorithms



Example of bump-mapping  
done with a multi-pass  
OpenGL algorithm