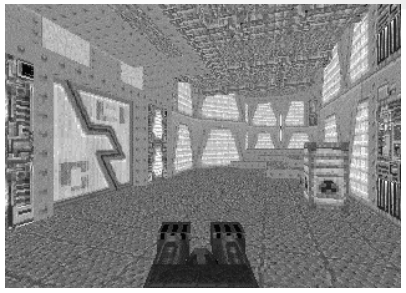
CIS 781
3D Raster Graphics

Roger Crawfis
Ohio State University

Realism Through Synthesis

Play Games …



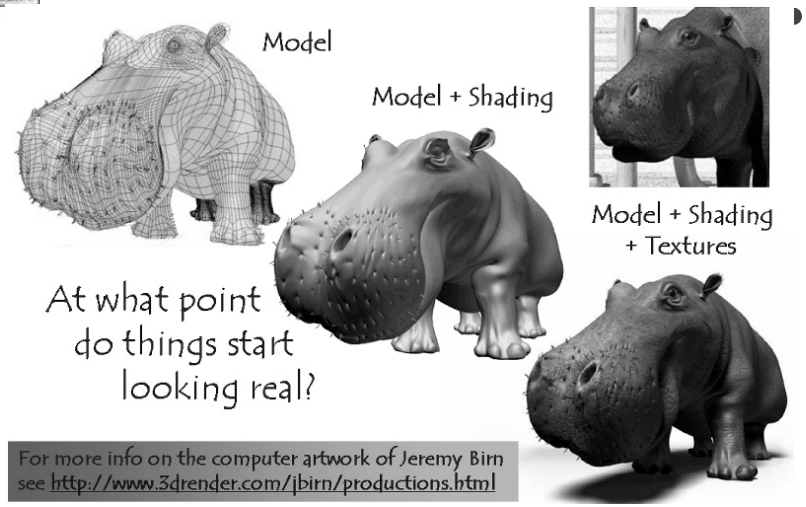
Goals of Computer Graphics

- 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

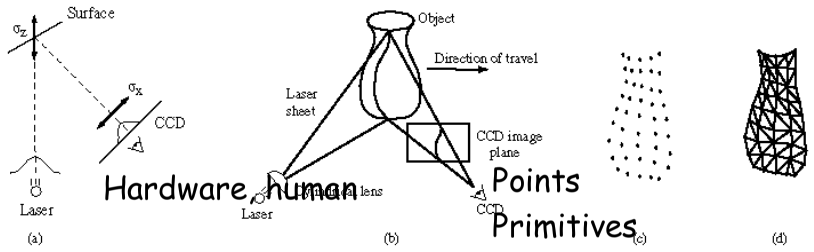


For more info on the computer artwork of Jeremy Birn see http://www.3drender.com/jbirn/productions.html

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

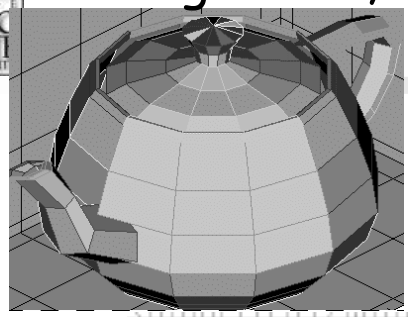


Hardware, human

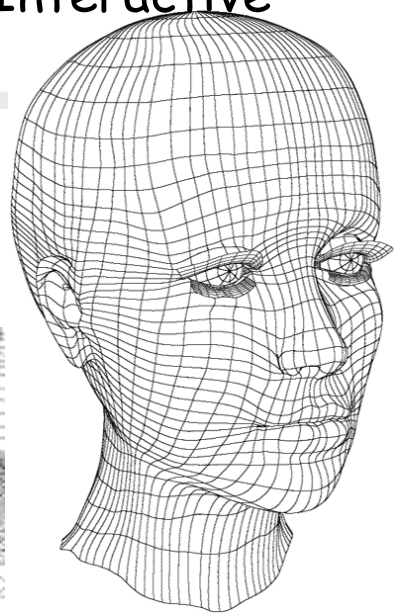
Points

Primitives

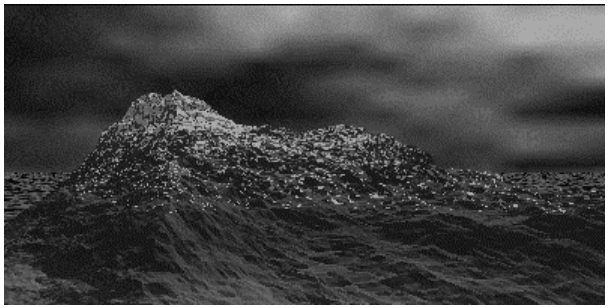
Algebraic, Interactive



Primitives ?



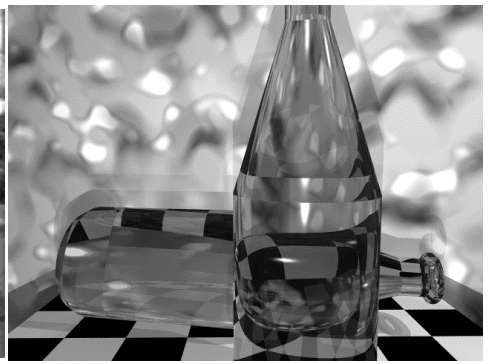
Modeling - Procedural



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

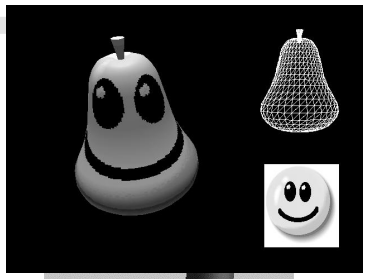
- 3D polygonal rendering
  - Rasterization
  - Clipping
  - Hidden surface determination
- Shadows
- Texture Mapping

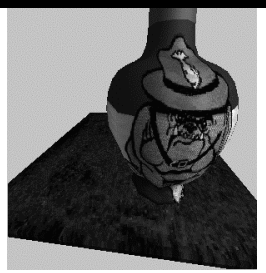
Other Courses @ OSU

- cis694 – Intro to 3D Graphics, OpenGL
- Cis681 – Ray Tracing, Local Illumination
- Cis782 – Global Illumination, Anti-aliasing, Special Topics
- Cis 881 - Geometric Modeling
- Cis 694? – Scientific Visualization (Crawfis/Shen/I)
- 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."

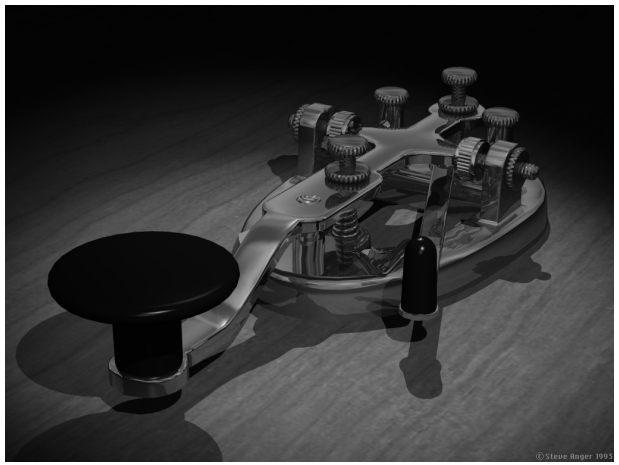
- Owen Demers
[digital] Texturing & Painting, 2002

Quote

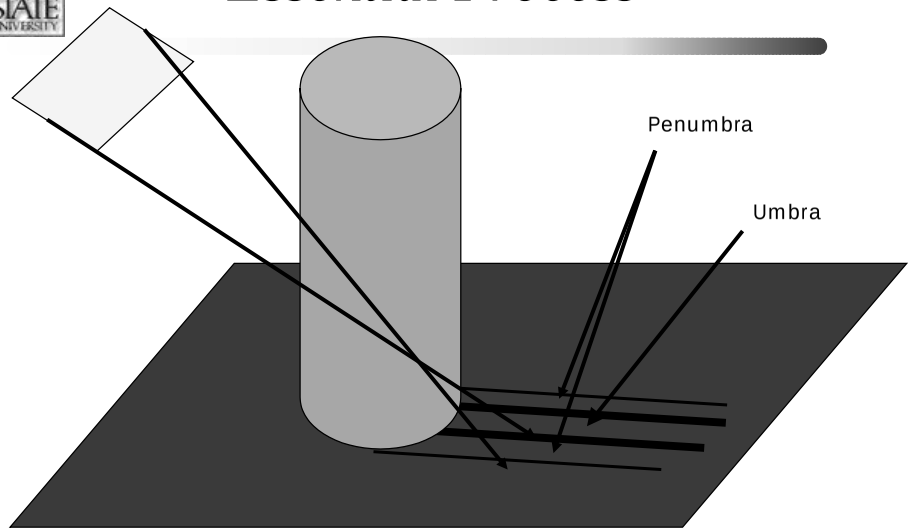
"I am interested in the effects on an object that speak of human intervention. This is another factor that you must take into consideration. How many times has the object been painted? Written on? Treated? Bumped into? Scraped? This is when things get exciting. I am curious about: the wearing away of 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

Shadows



Essential Process



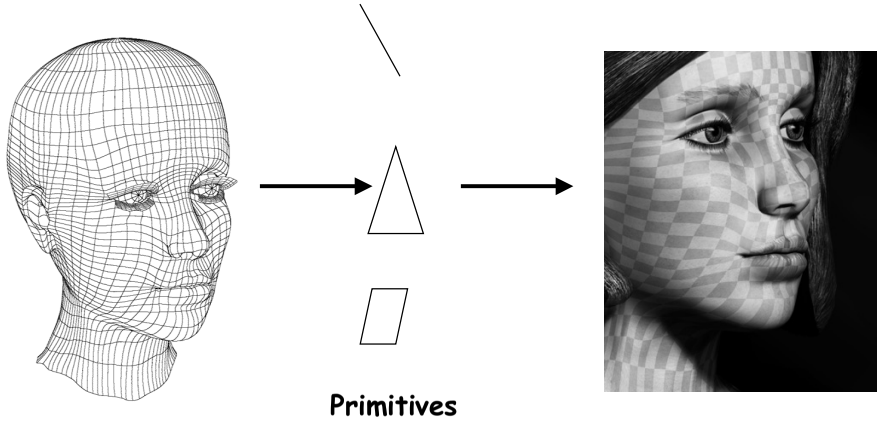
Texture Mapping

- Why use textures?

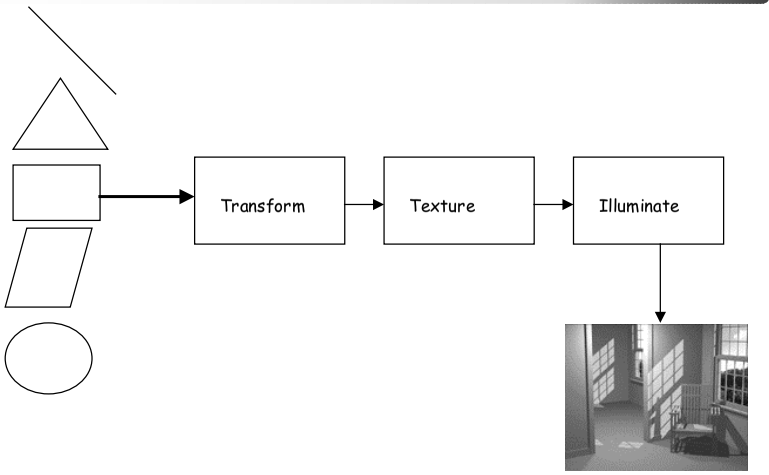
Texture Mapping

- Modeling complexity

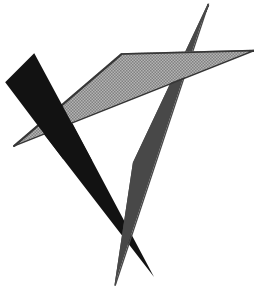
Essential Process

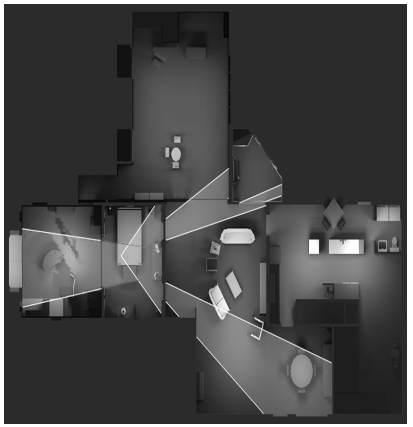


Graphics Pipeline (OpenGL)



The Problem of Visibility





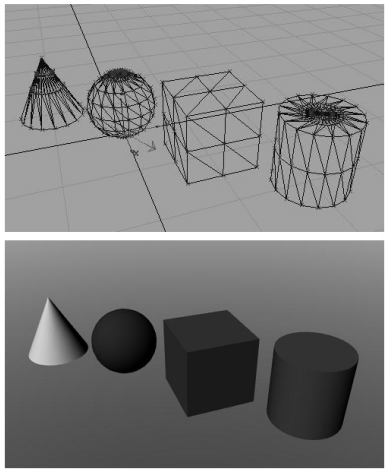
Light Material Interaction - ?

Modeling

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

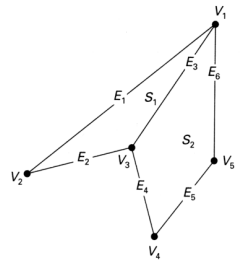
Polygon Mesh

- 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



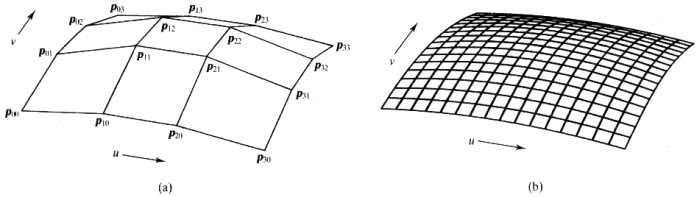
VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

POLYGON-SURFACE TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

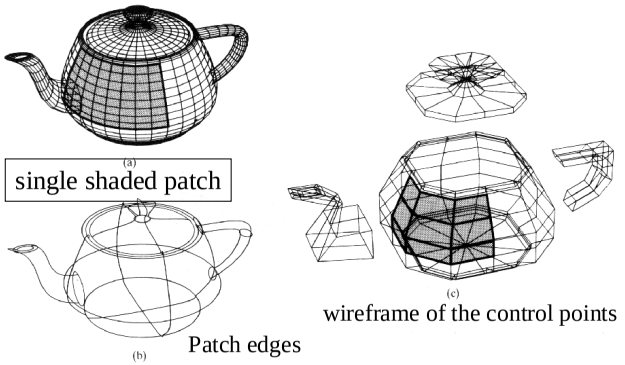
Arriving at a Mesh

- 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



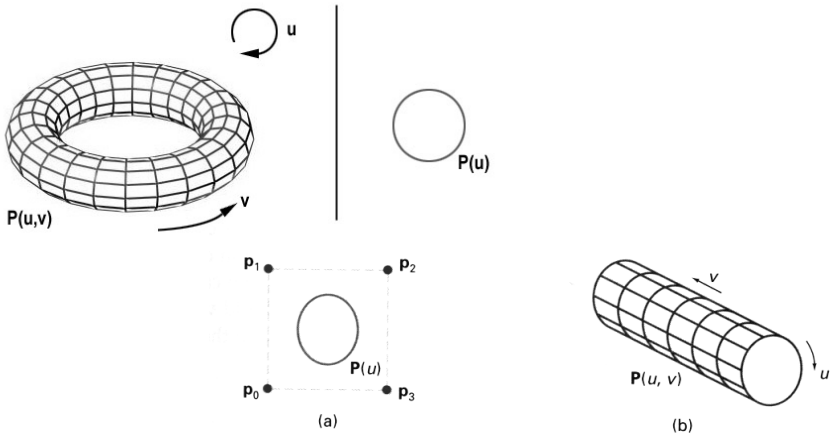
Patch Representation vs. Polygon Mesh

- 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

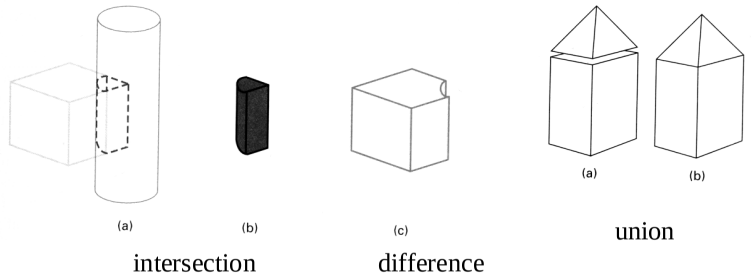
- **Extrude**: add a height to a flat polygon
- **Revolve**: Rotate a polygon around a certain axis
- **Sweep**: sweep a shape along a certain curve (a generalization of the above two)
- **Loft**: shape from contours (usually in parallel slices)
- Set operations (intersection, union, difference), **CSG**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

Sweep (Revolve and Extrude)



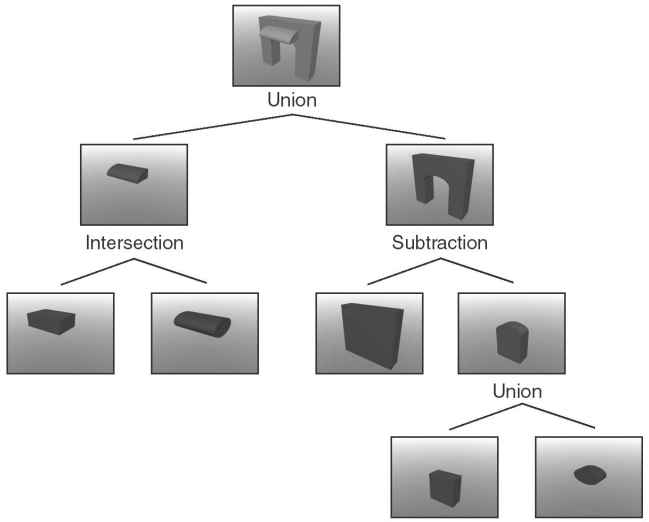
Constructive Solid Geometry (CSG)

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

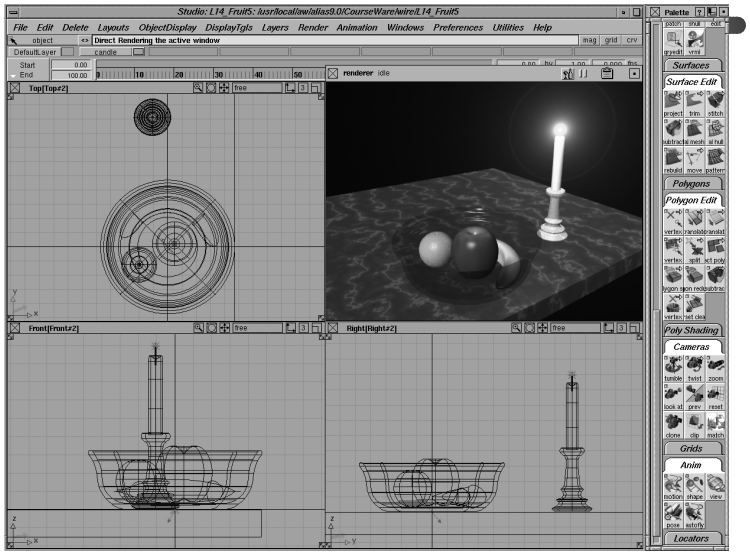


intersection difference union

A CSG Tree

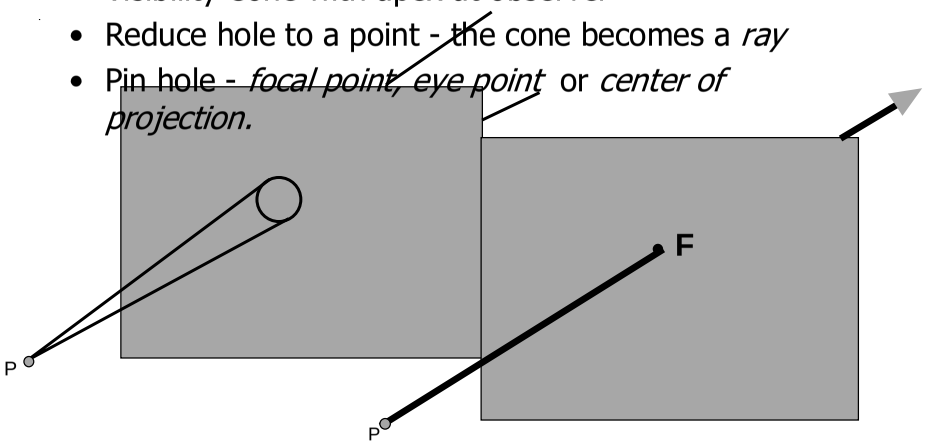


Example Modeling Package: Alias Studio



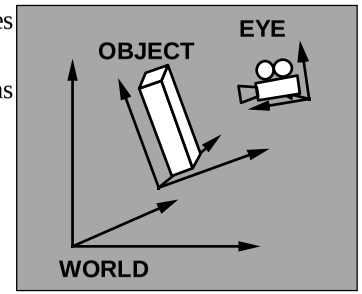
Pin Hole Model

- 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



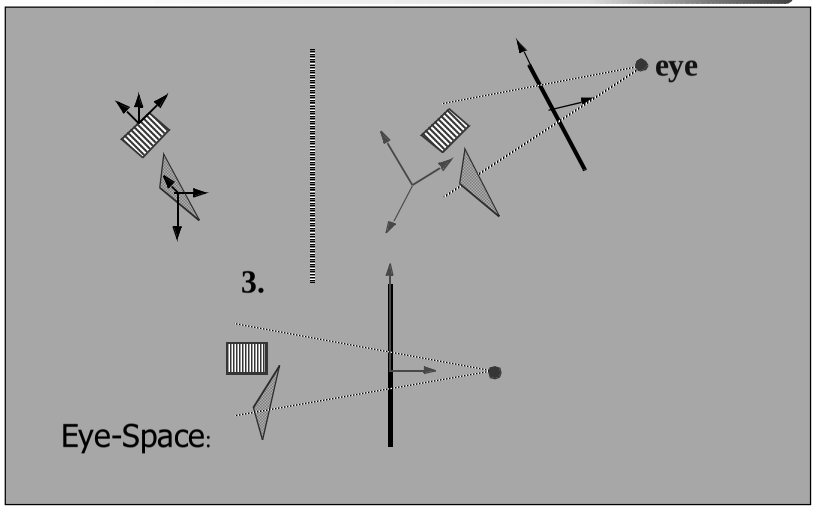
Viewing Pipeline

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 parallelpipied shape defined by (-1:1,-1:1,0,1). Objects in this space are distorted
- **Screen space:** x and y screen pixel coordinates

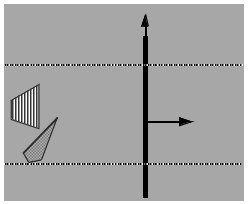
Model->Eye Space

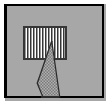
Object Space and World Space:

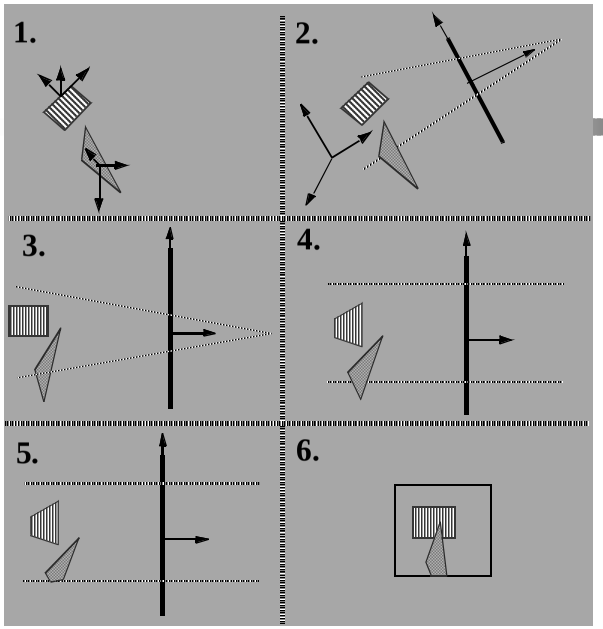


Clip and Image Spaces

- Clip Space
- Image Space

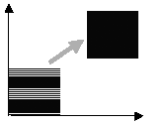






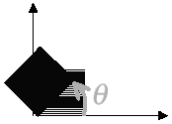
2D Transformation

- Translation



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

- Rotation



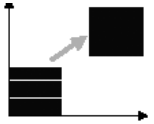
$$\begin{cases} x' = x \cdot \cos\theta - y \cdot \sin\theta \\ y' = x \cdot \cos\theta + y \cdot \sin\theta \end{cases}$$

Matrix and Vector format:

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

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

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

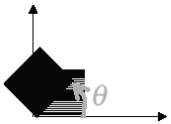
$$M\Big|_{\substack{t_x=0 \\ t_x=0}} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = Identity(I)$$

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.

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

Rotation in Homogeneous Space



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

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

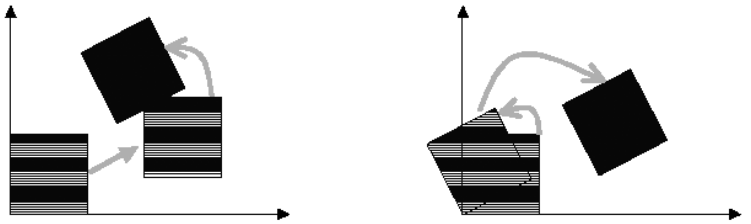
The two properties still apply.

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

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

Putting Translation and Rotation Together

- Order matters !!



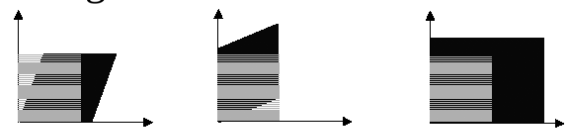
Affine Transformation

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

Affine Transformations

- Translation
- Rotation
- Scaling
- Shearing

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



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

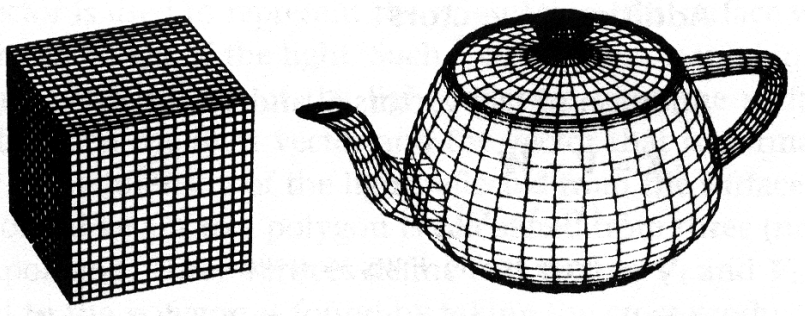
- Which axis of rotation?

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

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

Global Deformations

- Taper
- Twist
- Bend



Global Deformations

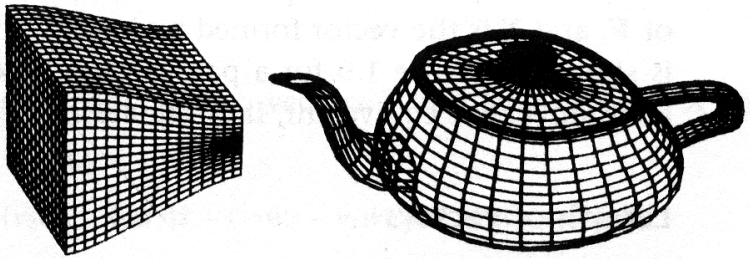
- Tapering:

$r = f(z)$

$x = r*x$

$y = r*y$

$z = z$



Global Deformations

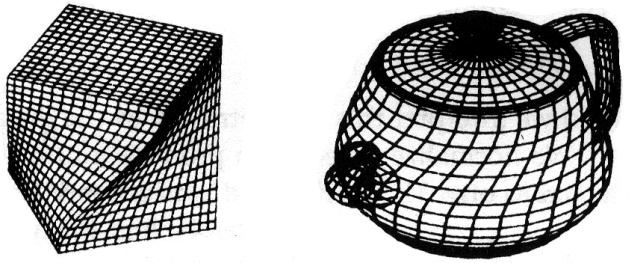
- Twisting:

$\theta = f(z)$

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

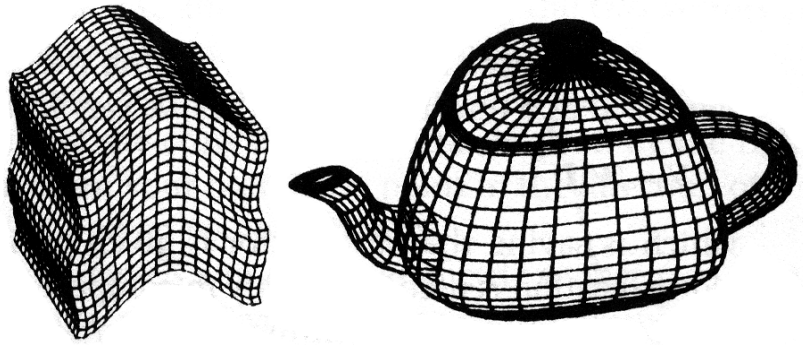
$y = x*\sin\theta + y*\cos\theta$

$z = z$



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)

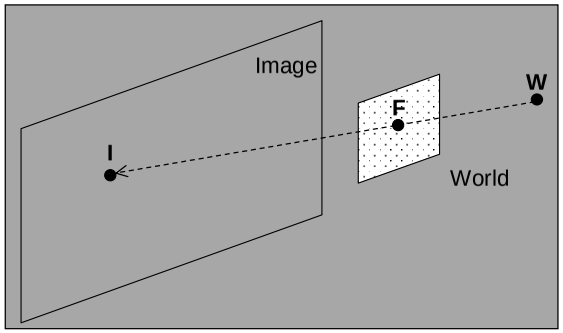
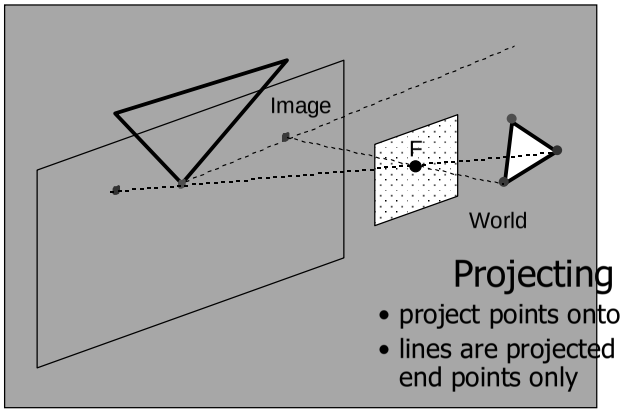


Image Formation

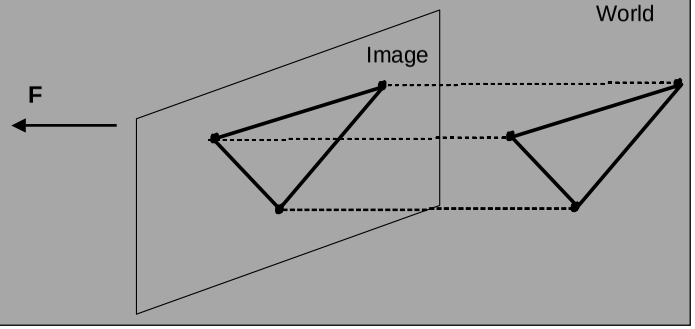


Projecting shapes

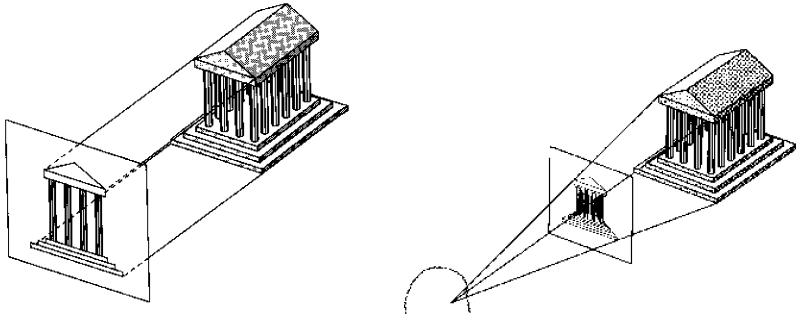
- project points onto image plane
- lines are projected by projecting their end points only

Orthographic Projection

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

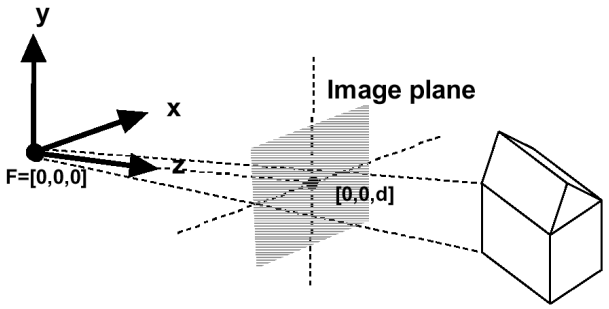


Comparison



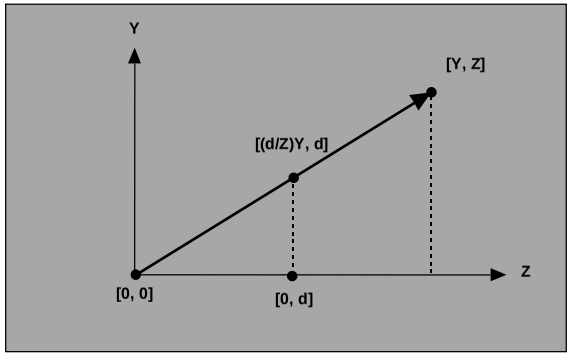
Simple Perspective Camera

- 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]

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

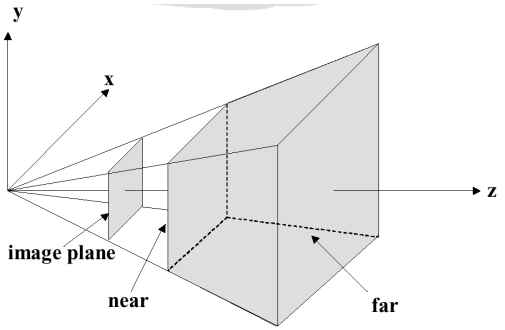
Divide by 4th coordinate (the "w" coordinate)

Image Space

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

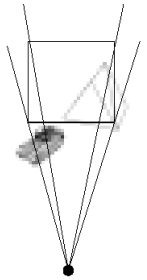
View Volume

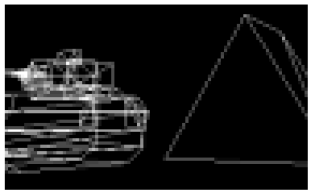
- Defines visible region of space, pyramid edges are clipping planes
- *Frustum* :truncated pyramid with near and far clipping planes
  - Near (Hither) plane ? Don't care about behind the camera
  - Far (Yon) plane, define field of interest, allows *z* to be scaled to a limited fixed-point value for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



Difficulty

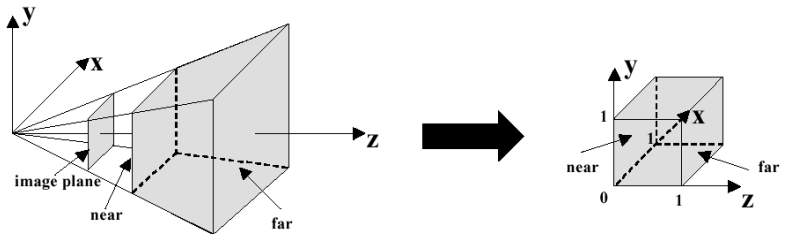
- It is difficult to do clipping directly in the viewing frustum





Canonical View Volume

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



Perspective Transform

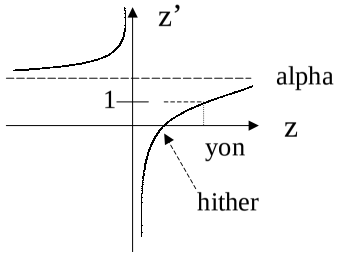
- The equations

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

alpha = hither/(yon-hither)

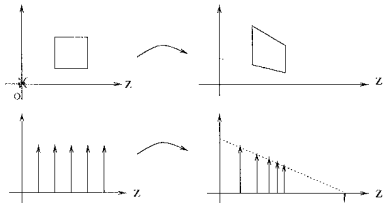
beta = yon*hither/(hither - yon)

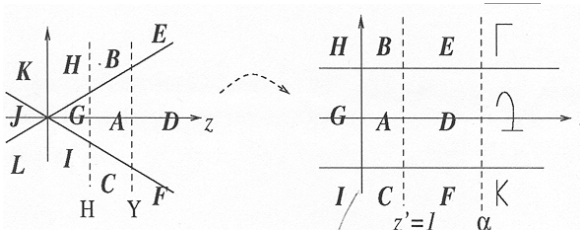
s: size of window on the image plane



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.

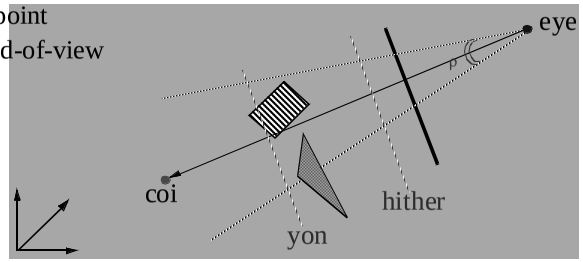
Perspective + Projection Matrix

- 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
  - ρ - 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 *znear* and *zfar* 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:
  - **v** = (*lookat-lookfrom*) (normalized) and **z** = [0,0,1]
  - rotation axis: **a** = (**v**x**z**)/|**v**x**z**|
  - rotation angle: cosθ = **a**•**z** and sinθ = |**r**x**z**|
- OpenGL: glRotate(θ, 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_t) be the upper-right corner and (v_l,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.

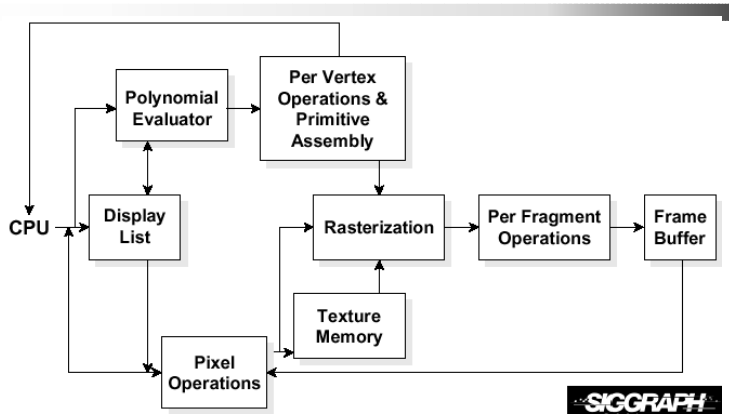
OpenGL an GLUT Overview

- 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



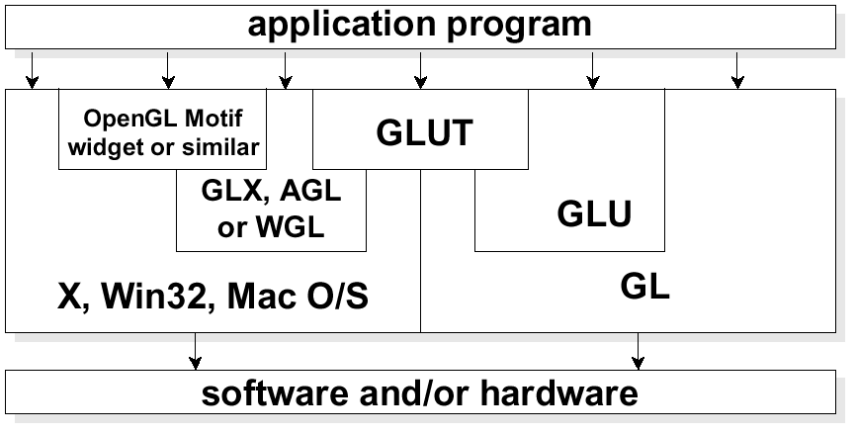
OpenGL as a Renderer

- 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

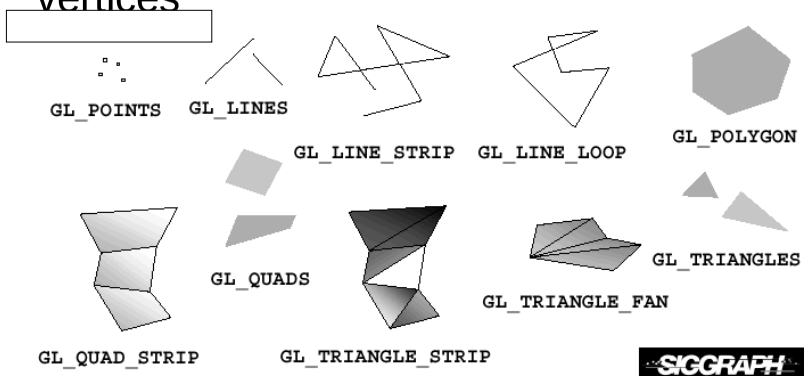
```
glutKeyboardFunc( my_key_events );
void my_key_events ( char key, int x, int y )
{
    switch( key ) {
        case 'q' : case 'Q' :
            exit( EXIT_SUCCESS );
            break;
        case 'r' : case 'R' :
            rotate = GL_TRUE;
            break;
        }
}
```

Elementary Rendering

- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers

OpenGL Geometric Primitives

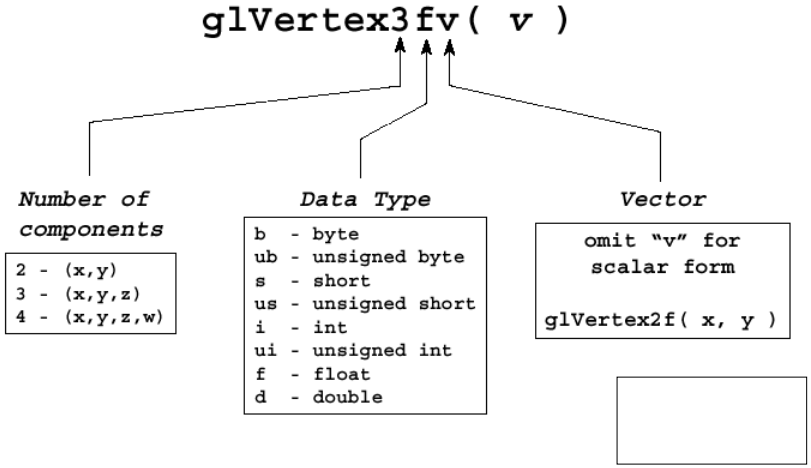
- All geometric primitives are specified by vertices



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



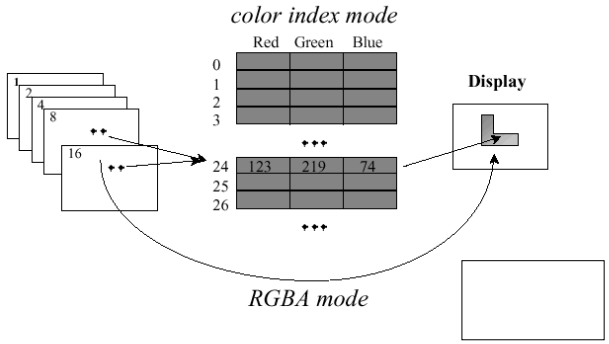
Specifying Geometric Primitives

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

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

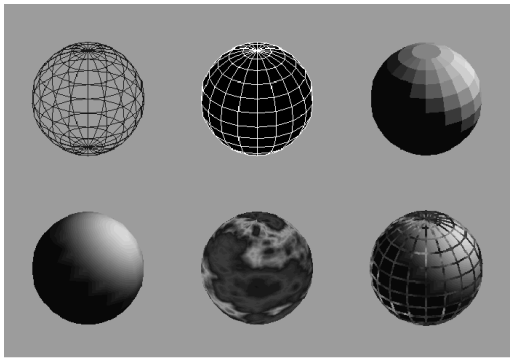
OpenGL Color Model

- 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.

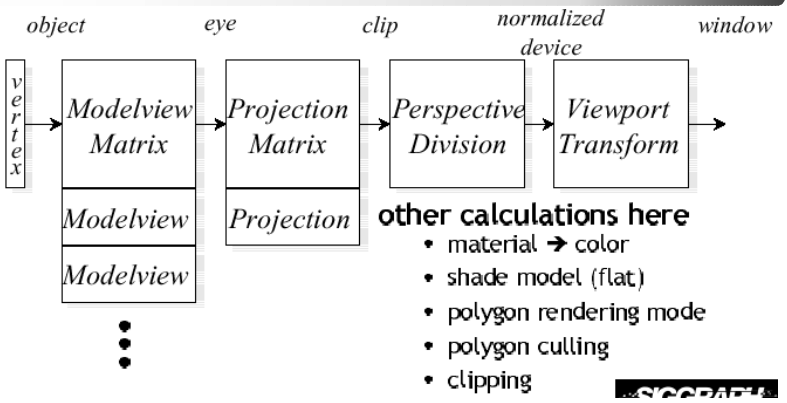
Specifying Transformations

- 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 camea/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)

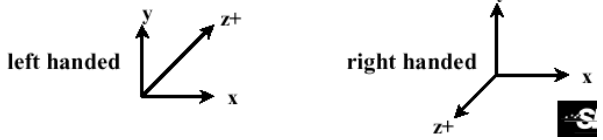
- up vector determines unique orientation
- careful of degenerate positions

Modeling Transformations

- Move object
  - **glTranslate{fd}(x, y, z)**
- Rotate object aro nd 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 );
}
```