CS4247
Graphics Rendering Techniques

Semester 2, 2015/2016

Lecture 1

Raster Graphics Pipeline

School *of* Computing
National University *of* Singapore

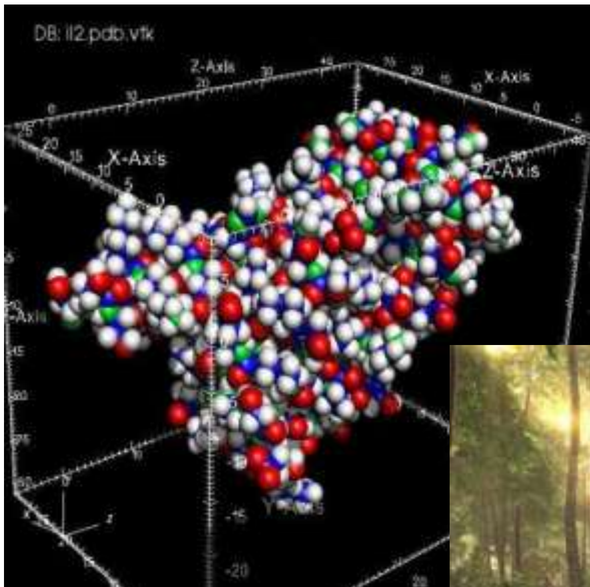
Lecture Outline

- Polygonal representation of 3D objects
  - (Watt 2000 Sec 2.1)
- Raster graphics pipeline
  - (Watt 2000 Chapter 5 & 6)

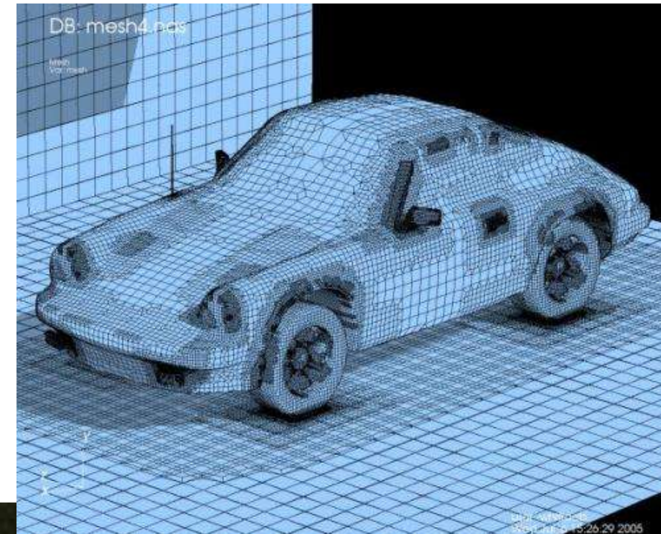
Example Applications

- Real-time interactive 3D graphics

Scientific Visualization



3D design



Courtesy of Crytek

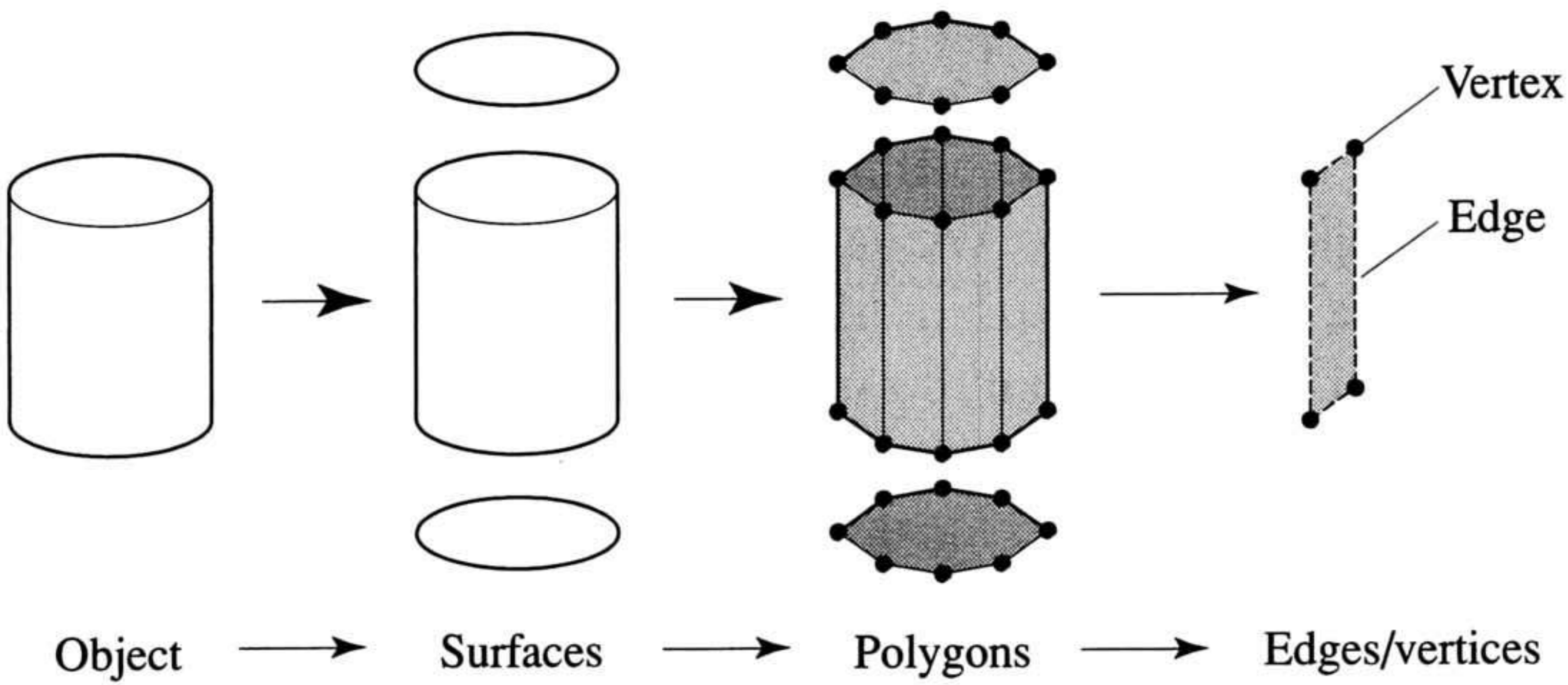
3D games

(Image of model Adrianne Curry rendered on a GeForce 8800 GTX GPU)

Polygonal Representation of 3D Objects

Polygonal Representation of 3D Objects

- 3D objects are approximated by a net or **mesh** of planar polygonal facets



Polygonal Representation of 3D Objects

- Polygonal representation and rendering is currently the mainstream for real-time 3D graphics
- Advantages
  - Simplicity & generality
  - Simple & effective shading algorithms available to reduce faceted appearance
  - Well-supported by current commodity graphics hardware
    - Video processors (GPUs) made by NVidia and AMD (ATI)
    - Popular game consoles, e.g. Sony PlayStation 1/2/3/4, MS XBox/360/One, Nintendo Wii

Polygonal Representation of 3D Objects

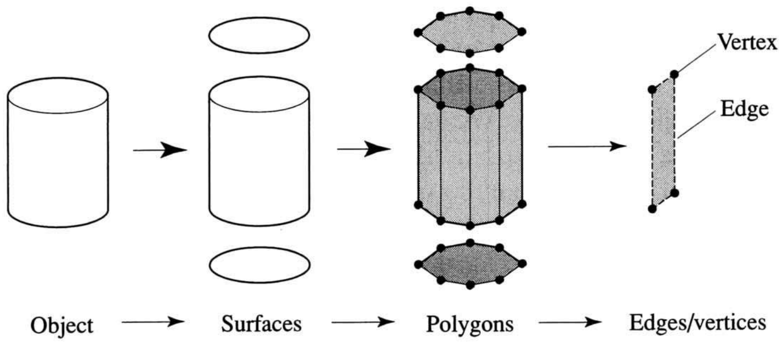
- Disadvantages
  - Inaccuracy of representation
  - Complex objects (esp. with curved surfaces) need large number of polygons, but is wasteful when objects are projected to only a few pixels on screen
  - Not suitable for interactive manipulation and free-form sculpting of models

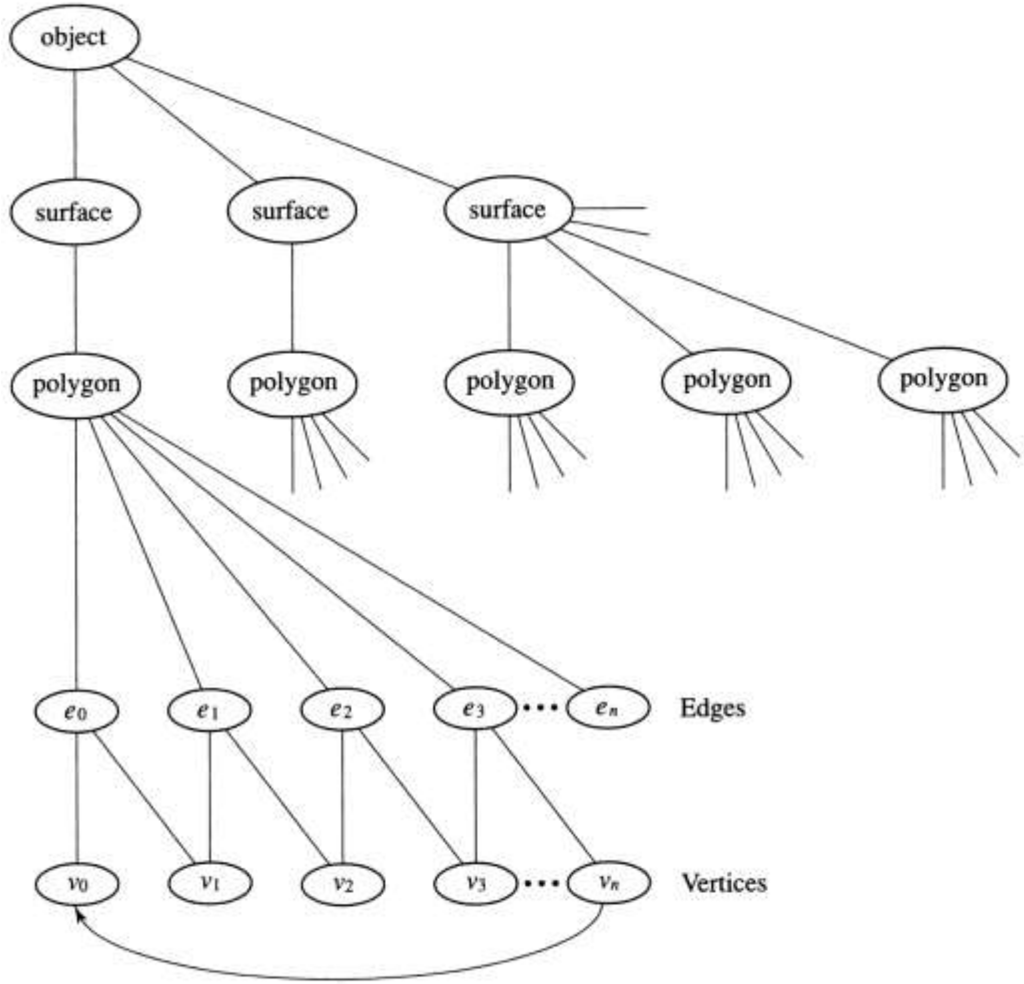
Data Structures for Polygonal Models

- The simplest is stored each polygon independently with a list of (x, y, z) coordinates that are the polygon vertices
  - Wasteful because vertices shared by multiple polygons are stored multiple times
  - Inefficient for geometric queries and manipulations
    - Find all faces adjacent to a face/vertex
    - Find all vertices adjacent to a face/vertex

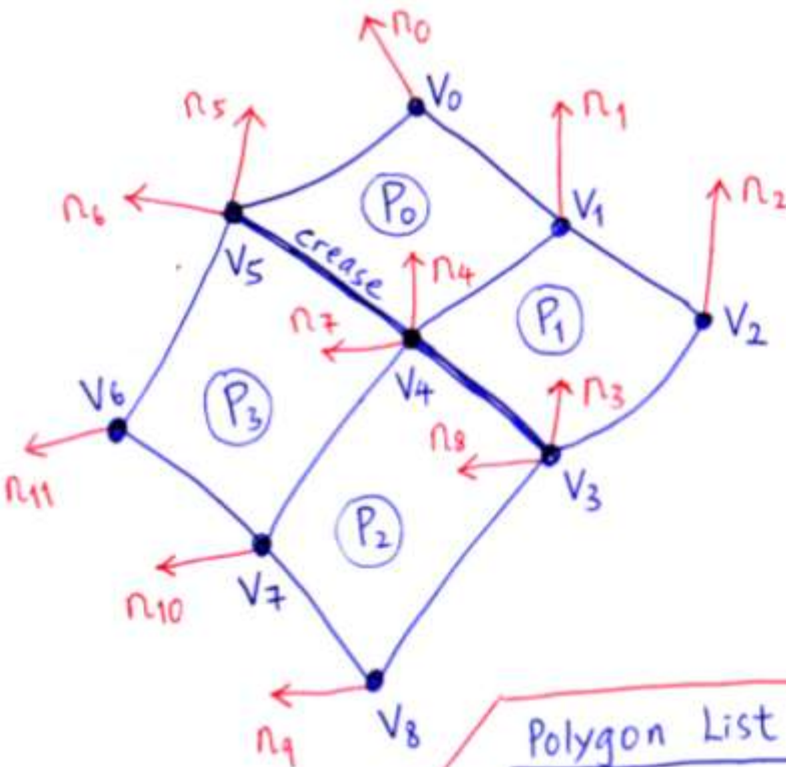
Data Structures for Polygonal Models

- Represent as a hierarchical structure





Data Structures for Polygonal Models



Vertex List

$0: x_0\ y_0\ z_0$

⋮

$8: x_8\ y_8\ z_8$

Normal List

$0: i_0\ j_0\ k_0$

⋮

$11: i_{11}\ j_{11}\ k_{11}$

Polygon List

$0: \underbrace{4}_{\#\ \text{sides}}\ \underbrace{1\ 0\ 5\ 4}_{\text{vertex indices}}\ \underbrace{1\ 0\ 5\ 4}_{\text{normal indices}}$

⋮

$3: 4\ \underbrace{5\ 6\ 7\ 4}\ \underbrace{6\ 11\ 10\ 7}$

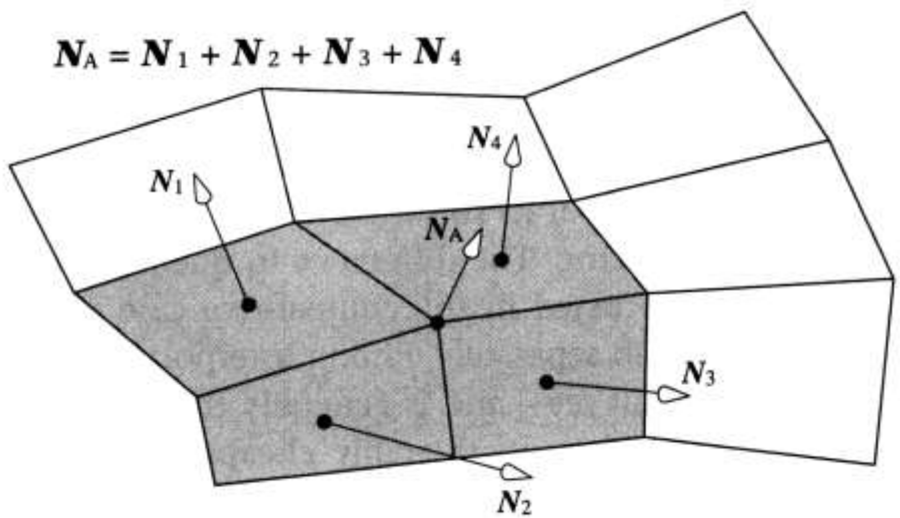
Vertex-list
data structure

Data Structures for Polygonal Models

- Winged-edge data structure
  - Can represent holes
  - Efficient for many geometric queries and manipulations, e.g.
    - Find all faces adjacent to a face/vertex
    - Find all vertices adjacent to a face/vertex

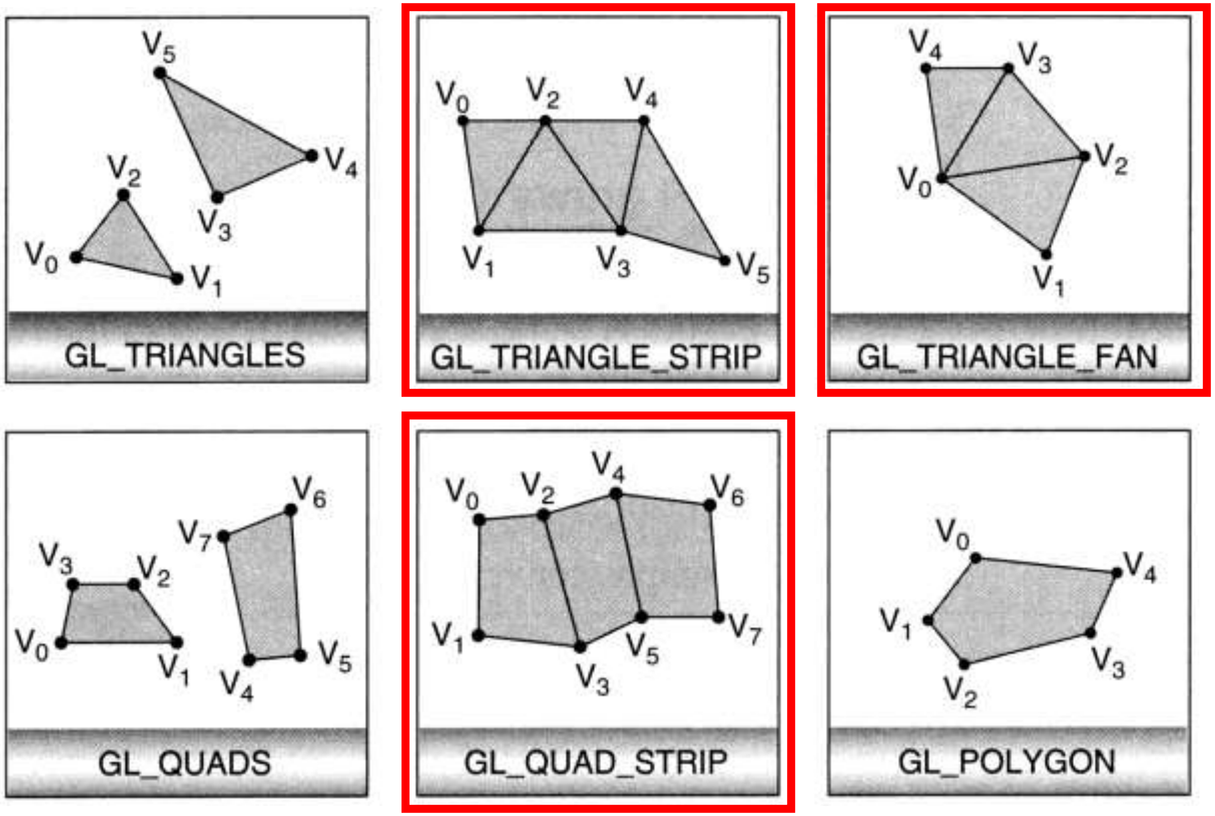
Vertex Normals

- Each polygon vertex is associated a vertex normal
  - For computation of light reflection intensity at the vertex
  - For "flat" shading, use polygon normal as vertex normal
  - For "smooth" shading,
    - use "true" surface normal (analytical) at the vertex, or
    - use average polygon normals of polygons sharing the vertex



Sending Polygons for Rendering

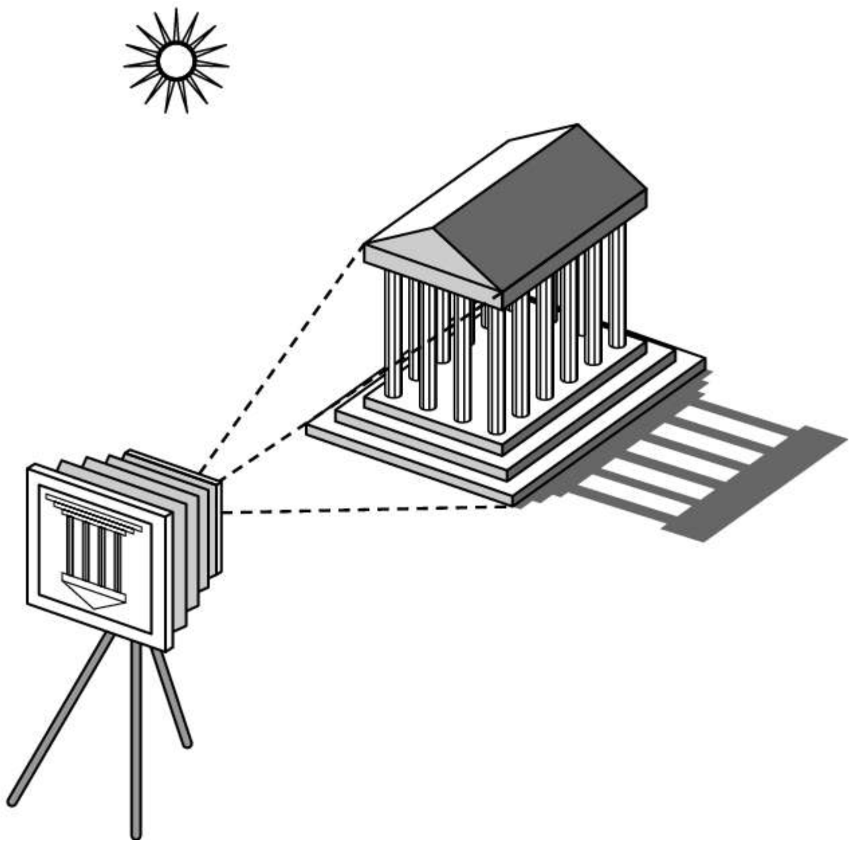
- It is inefficient to send polygons independently to the graphics pipeline for rendering
  - OpenGL provides functions to reduce sending duplicate data



OpenGL Rendering Pipeline

Elements of Image Formation

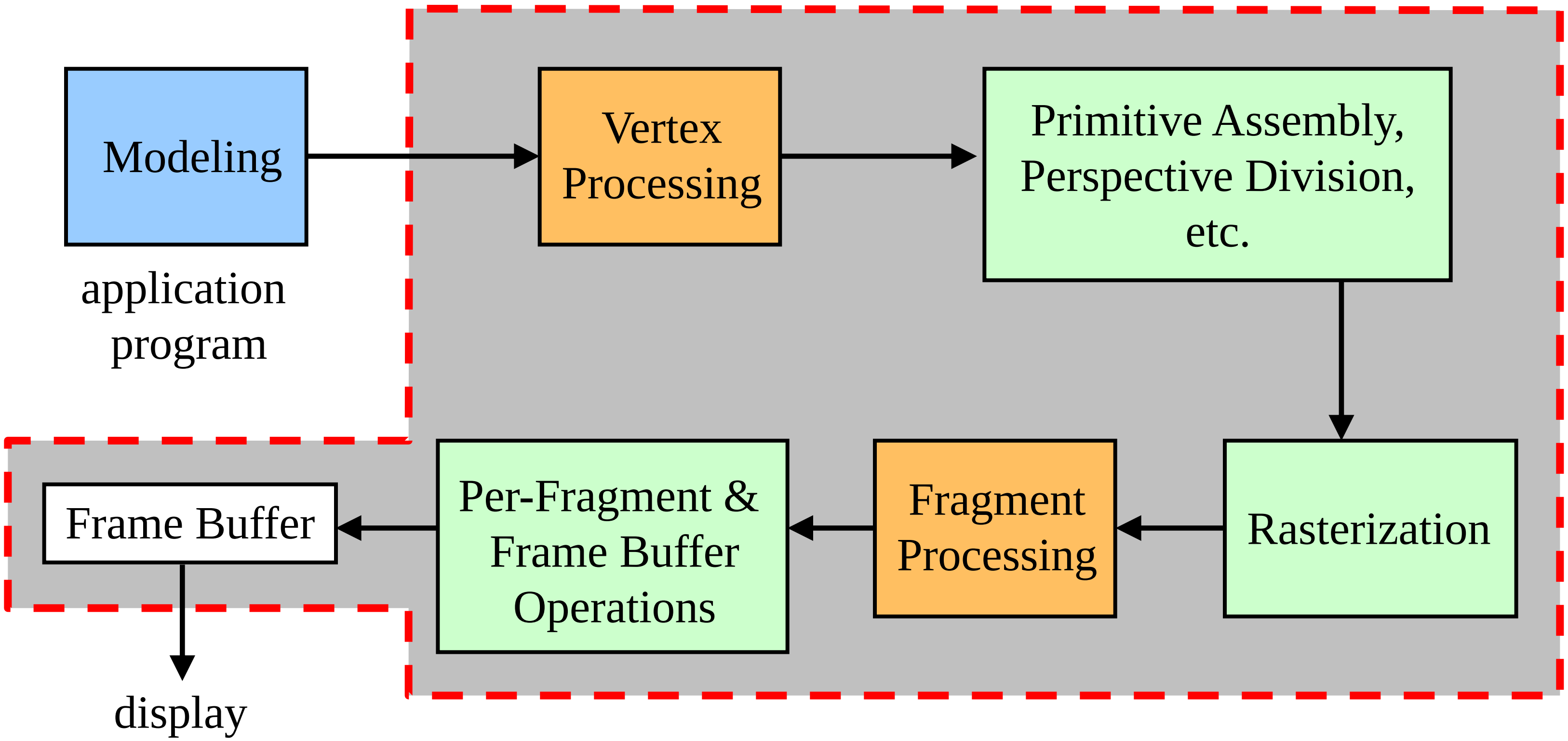
- Objects
- Viewer
- Light source(s)
- Materials
  - Attributes that govern how light interacts with the materials in the scene



Basic OpenGL 3D Rendering Pipeline

- To render a primitive using OpenGL, the primitive goes through the following main stages
  - Turning primitive into pixels

Important



Modeling

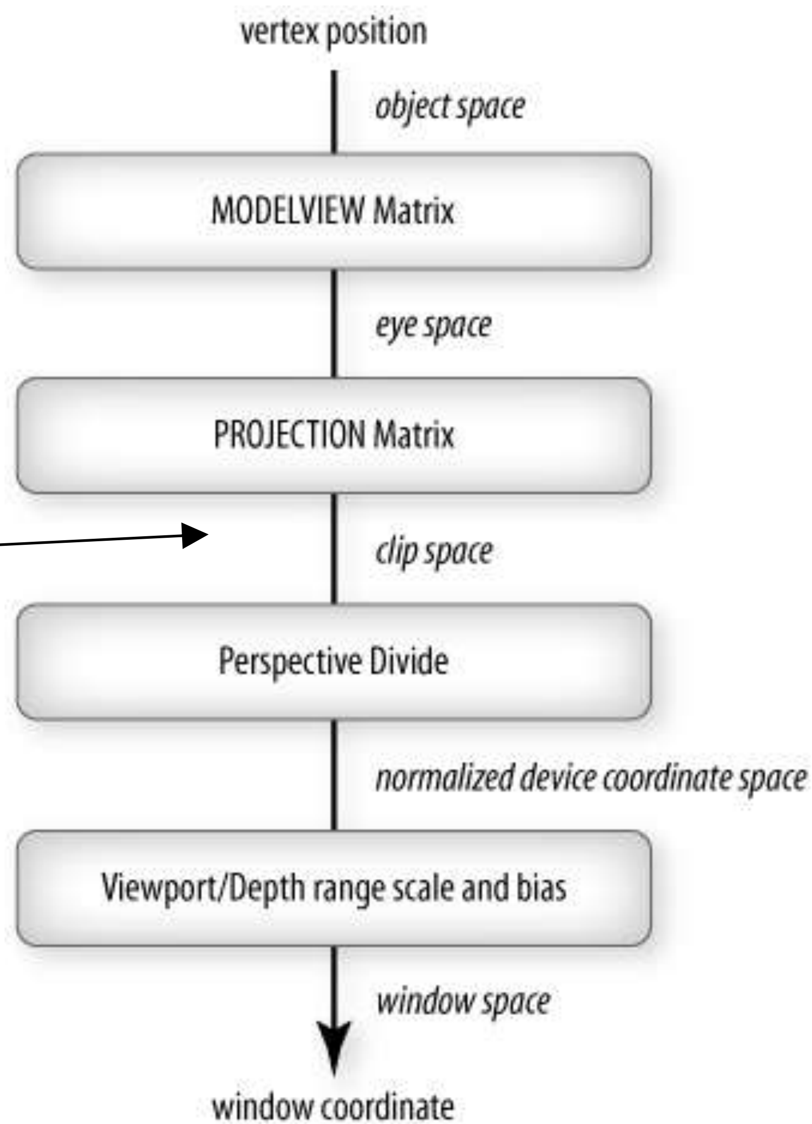
- Performed in the application program
- Sets up the OpenGL states (graphics/render context)
  - Light sources, materials, textures, view/projection, etc.
- Provides vertices to specify geometric primitives
  - Each vertex can be associated with attributes
    - Color, vertex normal, texture coordinates, etc.
  - Vertices and attributes can be provided using vertex array or display list
- May perform geometric processing to reduce amount of geometric data passed to rendering pipeline
  - E.g., view-frustum culling, occlusion culling

Vertex Processing

- The fixed-functioned stage performs
  - Vertex transformation
    - View transformation and projection
  - Normal transformation and normalization
    - Into camera/view/eye space
  - Texture coordinate generation
  - Texture coordinate transformation
  - Lighting computation
    - Computed in eye space
    - Resulting color replaces input vertex color attribute
  - Color material application
- Programmable by GLSL (vertex shader)

OpenGL Geometric Transformation Stages

Why is clipping performed before perspective division?

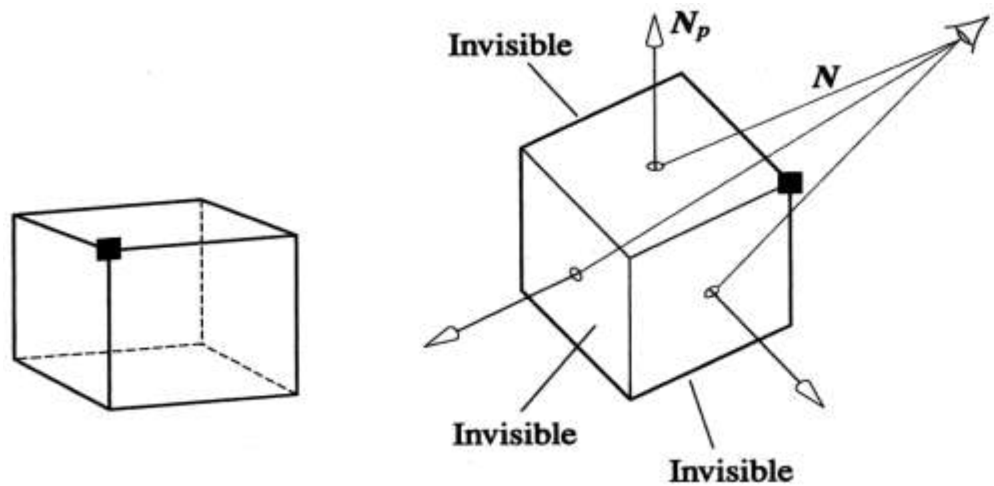


Primitive Assembly, etc.

- Primitive assembly
  - Vertex data is collected into complete primitives
  - Necessary for clipping and back-face culling
- Clipping
- Perspective division
  - To normalized device coordinate (NDC) space
- Viewport transformation
  - To windows space
  - Include depth range scaling
- Back-face culling

Back-Face Culling / Elimination

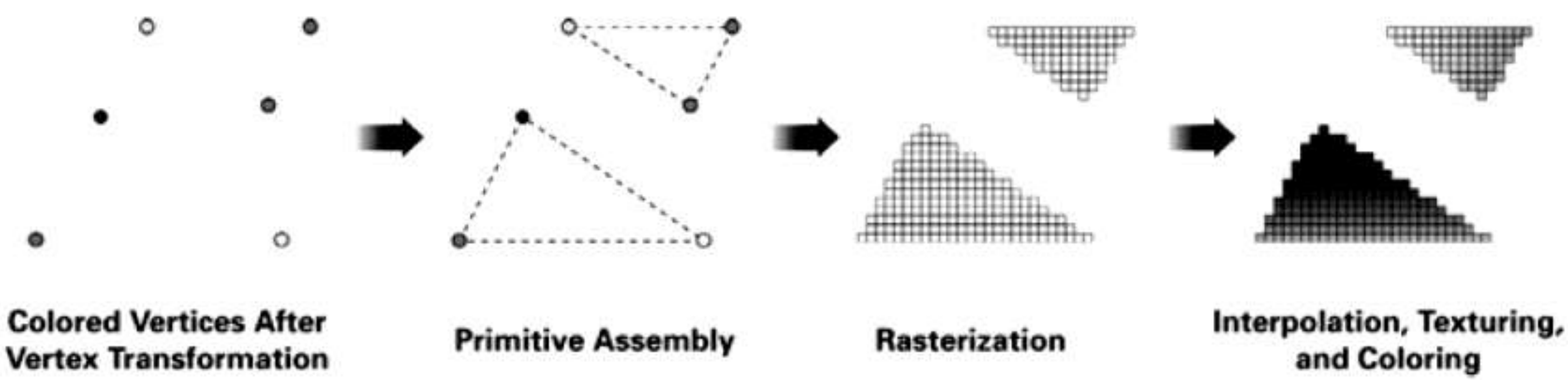
- Eliminate a polygon if it is back-facing and invisible
  - Polygon is back-facing if $\mathbf{N}_\mathrm{p} \cdot \mathbf{N} < 0$



  - By convention, when front-face of a polygon is visible, its vertices are listed in counter-clockwise order
  - In OpenGL, back-face culling is performed in the window space by computing the <u>signed</u> 2D area of the polygon

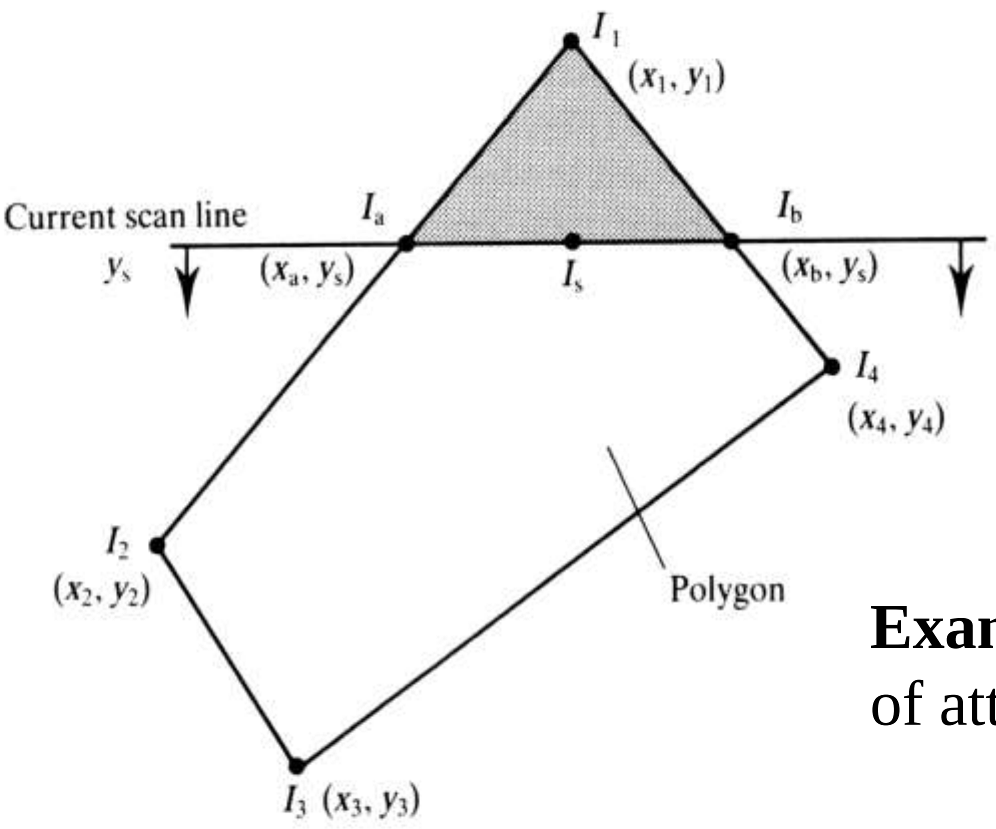
Rasterization (Scan Conversion)

- If geometric primitive is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each primitive
  - Fragments are “potential pixels”
    - Has a pixel/fragment location (in window space)
    - Has color and depth attributes (and others)
- Vertex attributes are interpolated over the primitive



Interpolation of Vertex Attributes

- Attribute values at fragments are computed by interpolating attribute values assigned to vertices
    - Interpolation is performed in window space (2D)



$$I_a = \frac{1}{y_1 - y_2}\left[I_1(y_s - y_2) + I_2(y_1 - y_s)\right]$$

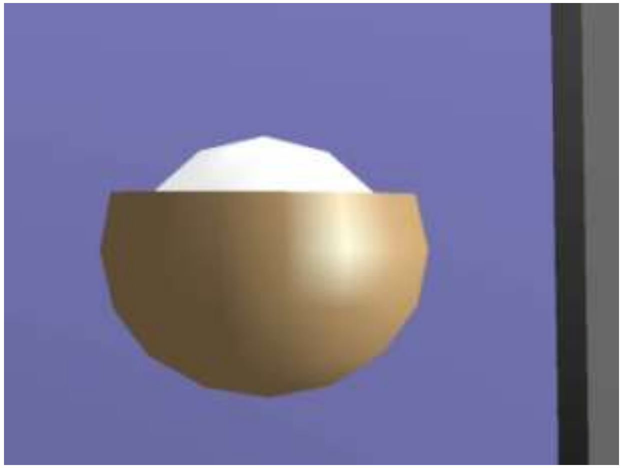
$$I_b = \frac{1}{y_1 - y_4}\left[I_1(y_s - y_4) + I_4(y_1 - y_s)\right]$$

$$I_s = \frac{1}{x_b - x_a}\left[I_a(x_b - x_s) + I_b(x_s - x_a)\right]$$

Example: Bilinear interpolation of attribute values

Example: Color Interpolation

Flat Shading
No interpolation of vertex colors

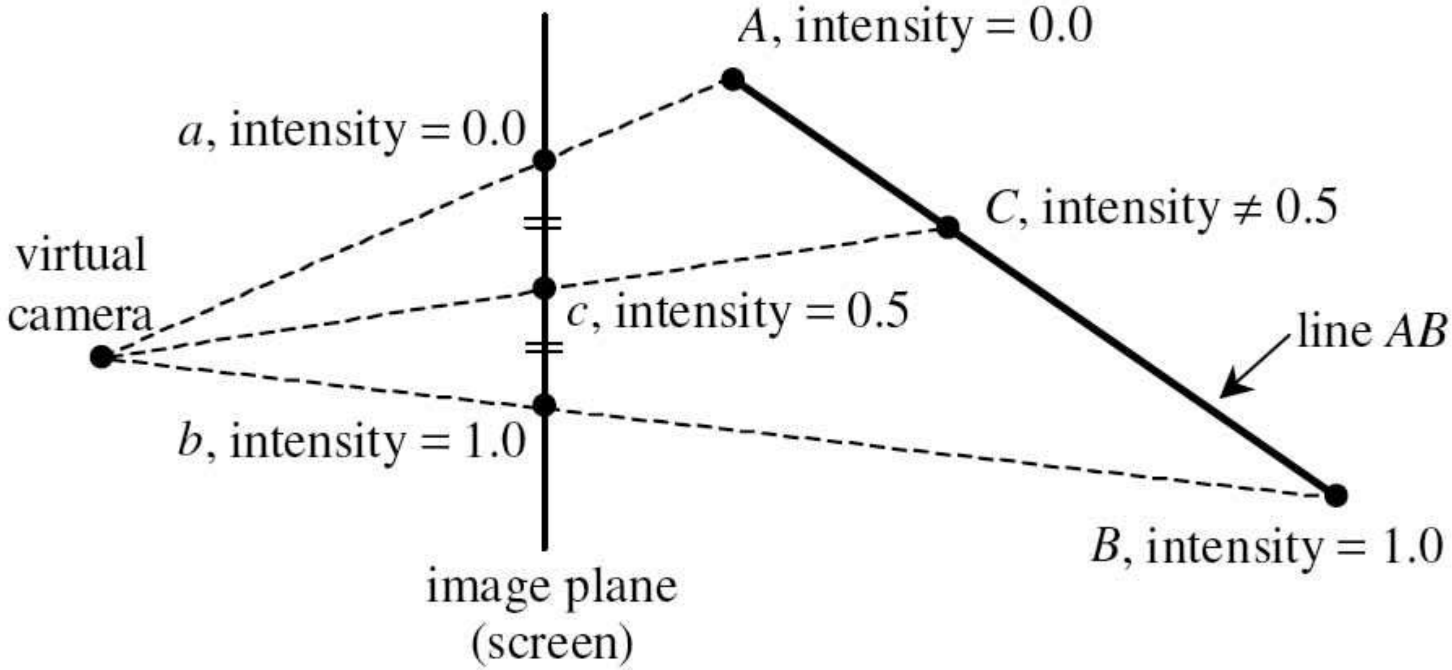


Smooth (Gouraud) Shading
Interpolation of vertex colors

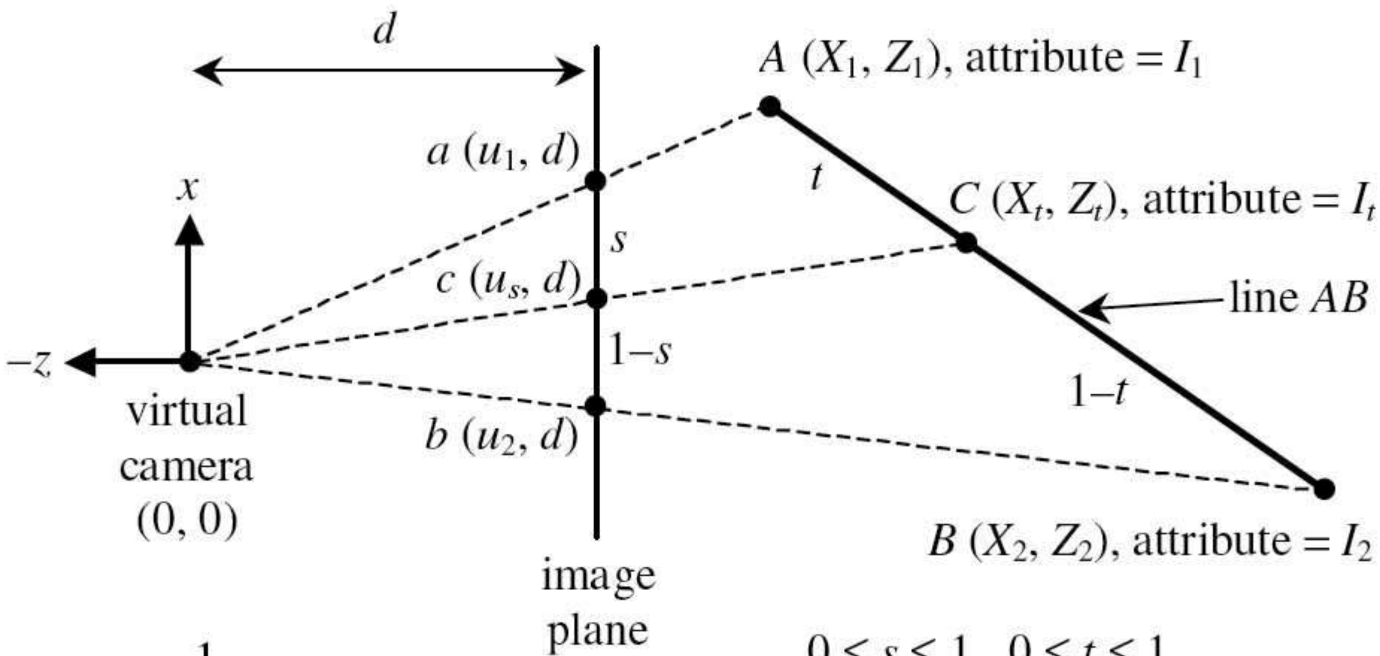
- When flat shading, only <u>vertex color and secondary color</u> are "flat", all other attributes are still smooth
  - Flat shading uses <u>color of last vertex</u> of the primitive
- Other vertex attributes that are interpolated
  - Z-value (depth), texture coordinates

Perspective-Correct Interpolation

- Note that direct bilinear interpolation in window space does not produce perspectively-correct result
  - Bilinearly varying attribute value in 3D space in general does not vary bilinearly in 2D window space



Perspective-Correct Interpolation



$0 \leq s \leq 1,\ \ 0 \leq t \leq 1$

$$Z_t = \frac{1}{\dfrac{1}{Z_1} + s\left(\dfrac{1}{Z_2} - \dfrac{1}{Z_1}\right)}$$

$$I_t = \left(\frac{I_1}{Z_1} + s\left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1}\right)\right) \Big/ \frac{1}{Z_t}$$

Refer to technical report "Perspective-Correct Interpolation" at http://www.comp.nus.edu.sg/~lowkl/publications/lowk_persp_interp_techrep.pdf

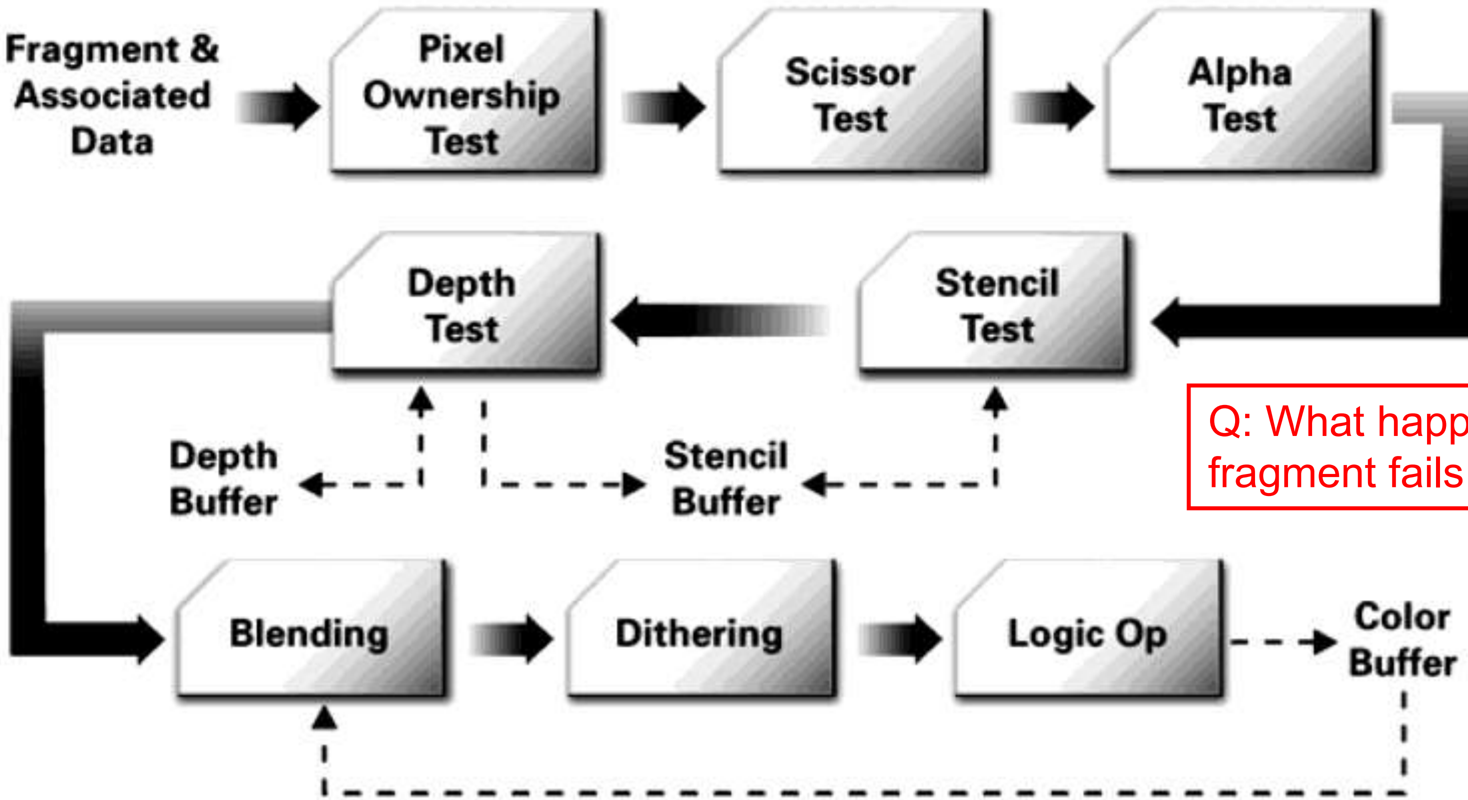
Fragment Processing

Q: What is a fragment?
Where are they from?

- Each generated fragment is processed to determine the color of the corresponding pixel in the frame buffer
- The fixed-functioned stage performs
  - Texture access (using interpolated texture coordinates)
    - May access multiple texture maps using multiple sets of texture coordinates (multi-texture)
  - Texture application
    - Texture color can be combined with the fragment color of the primitive (or with result from applying previous layer of texture)
    - Example: replace, modulate, decal, blend
  - Color sum
    - Combining primary and secondary colors
- Programmable by GLSL (fragment shader)

Per-Fragment Operations

- After fragment processing, each resulting fragment is submitted to a set of simple operations before reaching the frame buffer

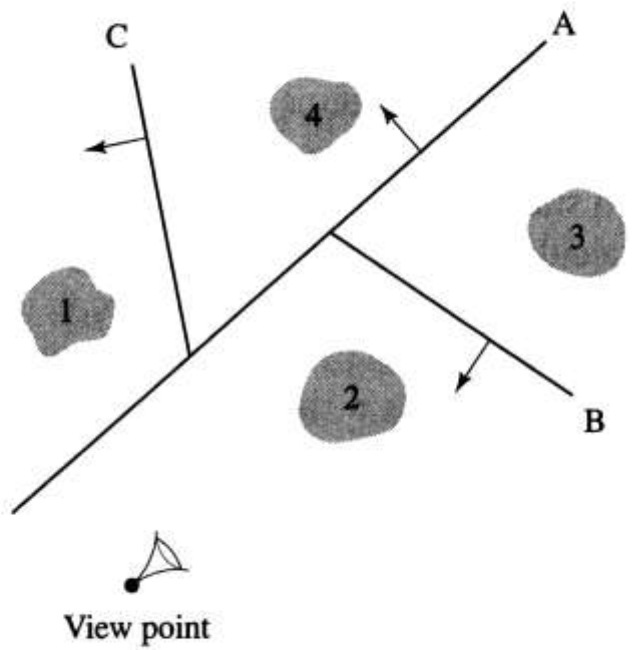


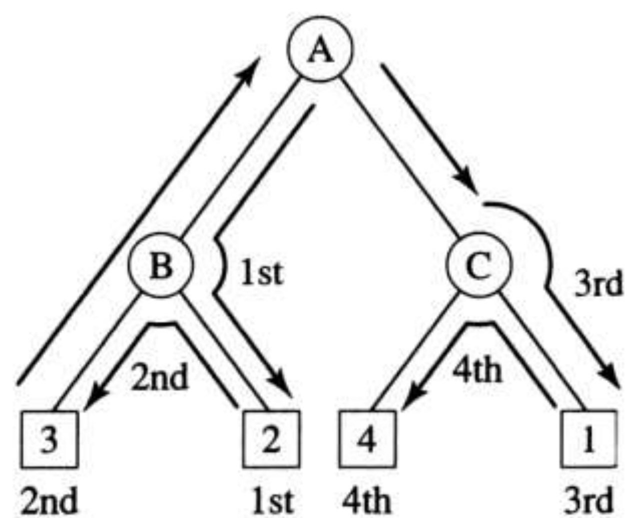
Q: What happens if a fragment fails a test?

[Figure from nVidia]

Hidden Surface Removal Using BSP Trees

- A binary space partitioning (BSP) tree partitions the scene (polygons) into hierarchical spatial regions
- Can be used to find the visibility order of the polygons from a given viewpoint
  - Allow back-to-front rendering of the polygons (no need z-buffer)



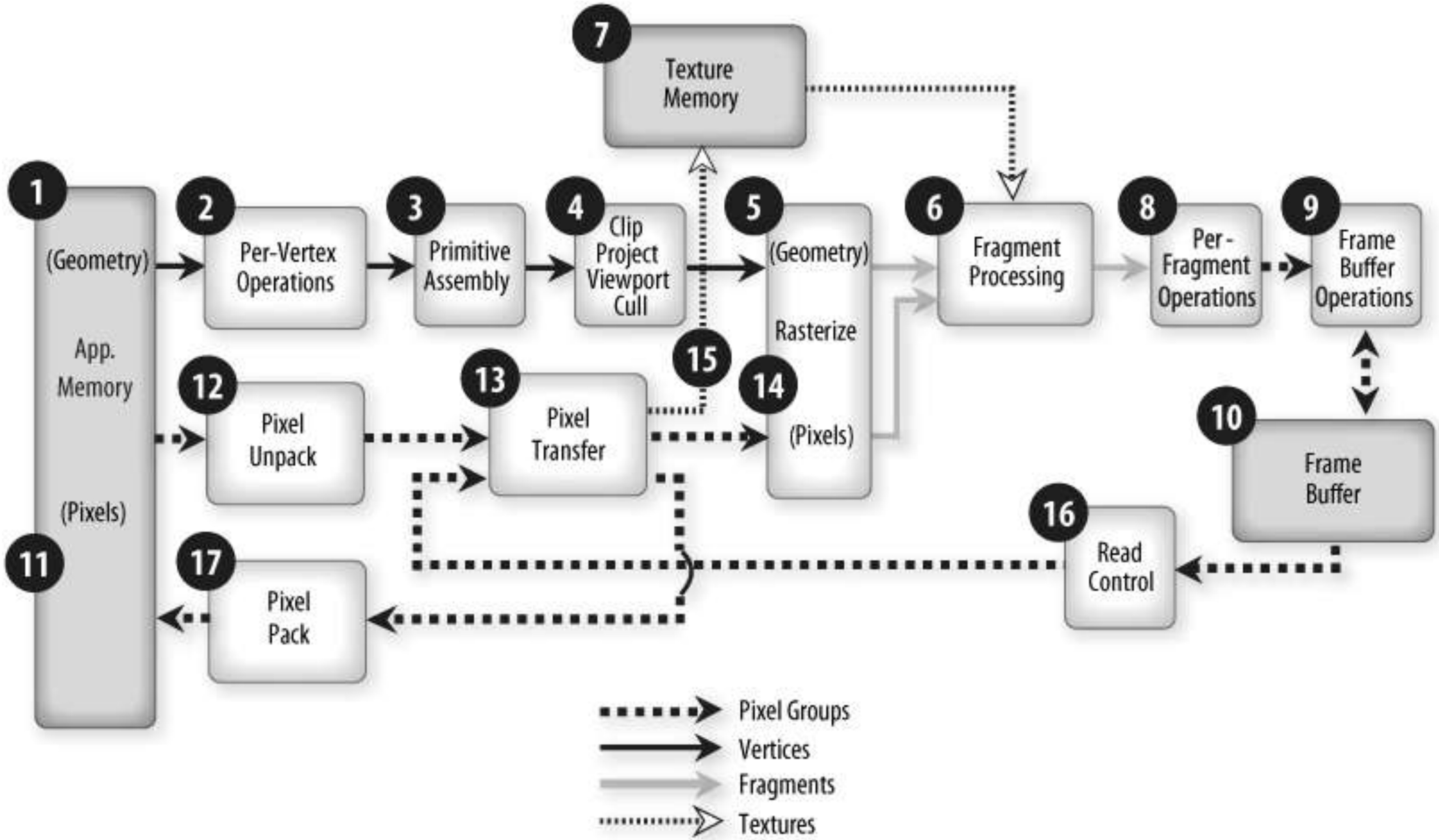


Frame Buffer Operations

- Operations that affect or control whole frame buffer
  - Example
    - Target color buffer (e.g. front or back buffer)
    - Enable/disable writes to color channels
    - Enable/disable writes to depth buffer
    - Enable/disable writes to stencil buffer bitplanes

Important for some rendering algorithms. E.g. shadow volume algorithm enables/disables writes to depth and color buffers.

More Detailed OpenGL Pipeline



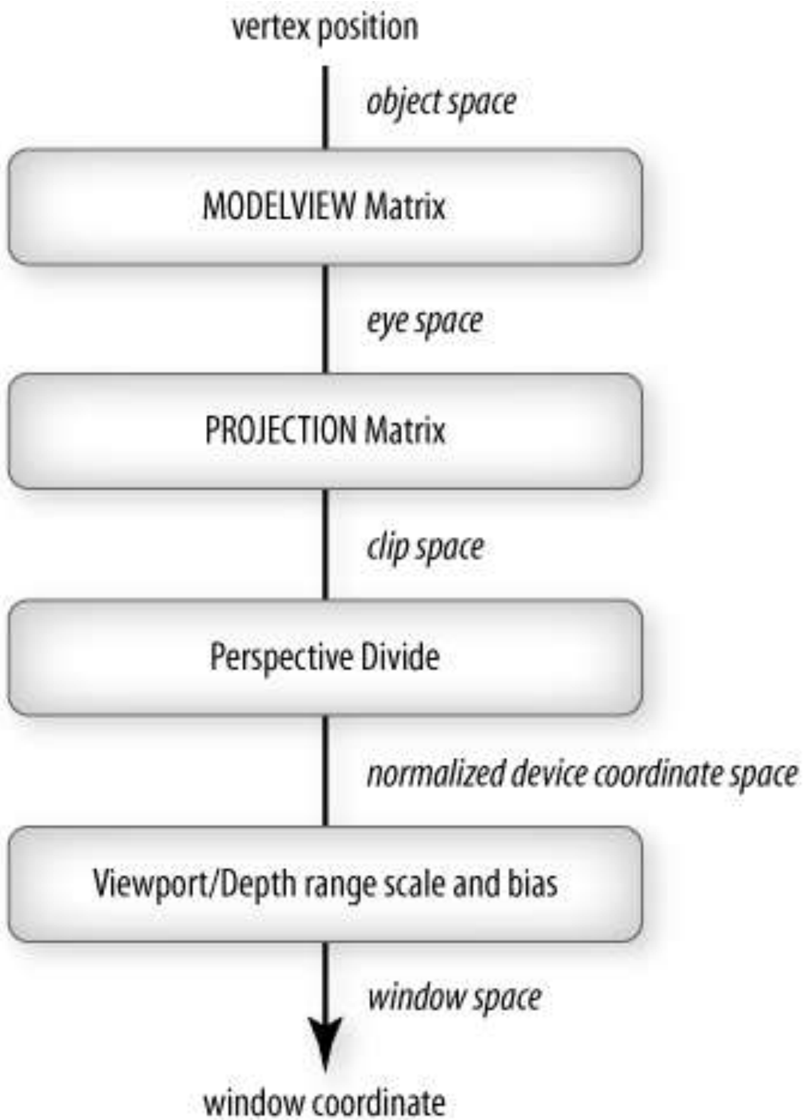
Images and Bitmaps

- OpenGL also has support for color/gray images (pixel rectangles) and bitmaps
  - Images and bitmaps from application can be rasterized to the frame buffer
  - Images in frame buffer can be read by application
  - Images from application can be read into texture memory
  - Images in frame buffer can be read into texture memory

- Read OpenGL "Redbook", Chapter 8 for more details

OpenGL
Geometric Transformations

More About OpenGL Geometric Transformations



OpenGL Code – Example 1

```
void display( void ) {
  ...
  glviewport( 0, 0, 800, 600 );
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective( viewerFovY, viewerAspect, viewerNear, viewerFar );

  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();
  gluLookAt( viewerPosX, viewerPosY, viewerPosZ,
             lookAtX, lookAtY, lookAtZ, upX, upY, upZ);

  glTranslate3d( 100.0, 200.0, 300.0 );
  glScale3d( 3.0, 3.0, 3.0 );

  glBegin(GL_QUADS);
    glColor3d( 1.0, 0.0, 0.0 );  glVertex3d( 0.0, 0.0, 0.0 );
    glColor3d( 0.0, 1.0, 0.0 );  glVertex3d( 1.0, 0.0, 0.0 );
    glColor3d( 0.0, 0.0, 1.0 );  glVertex3d( 1.0, 1.0, 0.0 );
    glColor3d( 1.0, 1.0, 1.0 );  glVertex3d( 0.0, 1.0, 0.0 );
  glEnd();

  glutSwapBuffers();
}
```

OpenGL Code – Example 2

```
void display( void ) {
  ...
  glviewport( 0, 0, 800, 600 );
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective( viewerFovY, viewerAspect, viewerNear, viewerFar );

  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();
  gluLookAt( viewerPosX, viewerPosY, viewerPosZ,
             lookAtX, lookAtY, lookAtZ, upX, upY, upZ);

  glPushMatrix();
    glTranslate3d( 100.0, 200.0, 300.0 );
    glScale3d( 3.0, 3.0, 1.0 );
    glColor3d( 1.0, 0.0, 0.0 );  // Red
    drawUnitSquare();
  glPopMatrix();

  glPushMatrix();
    glTranslate3d( 400.0, 400.0, 500.0 );
    glScale3d( 6.0, 6.0, 1.0 );
    glColor3d( 0.0, 1.0, 0.0 );  // Green
    drawUnitSquare();
  glPopMatrix();

  glutSwapBuffers();
}
```

```
void drawUnitSquare( void ) {
 glBegin(GL_QUADS);
   glVertex3d( 0.0, 0.0, 0.0 );
   glVertex3d( 1.0, 0.0, 0.0 );
   glVertex3d( 1.0, 1.0, 0.0 );
   glVertex3d( 0.0, 1.0, 0.0 );
 glEnd();
}
```

Local / Modeling / Object Coordinates

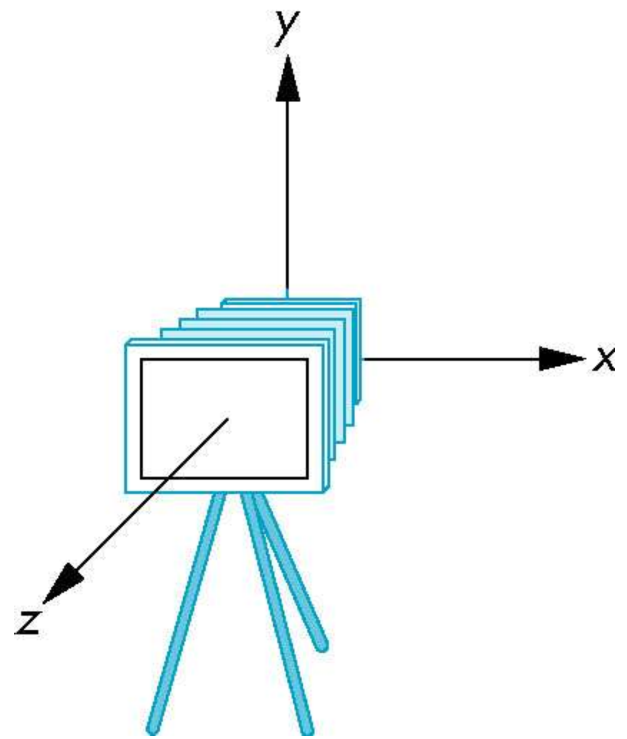
- Each object model has its own local coordinate frame
  - The coordinates of the vertices and vertex normals are specified with respect to the local coordinate frame
- Convenient for modeling of the object, e.g.
  - A sphere is centered at the origin
  - A cube with a corner located at the origin and edges parallel to the coordinate axes and

World Coordinates

- A common coordinate frame for all objects to form the scene to be rendered
  - Each object is transformed from its local space to a common world space
  - Vertex normals must also be transformed
- Lights are defined in this space
  - Positions and directions
- Camera pose is defined in this space

Camera Coordinates

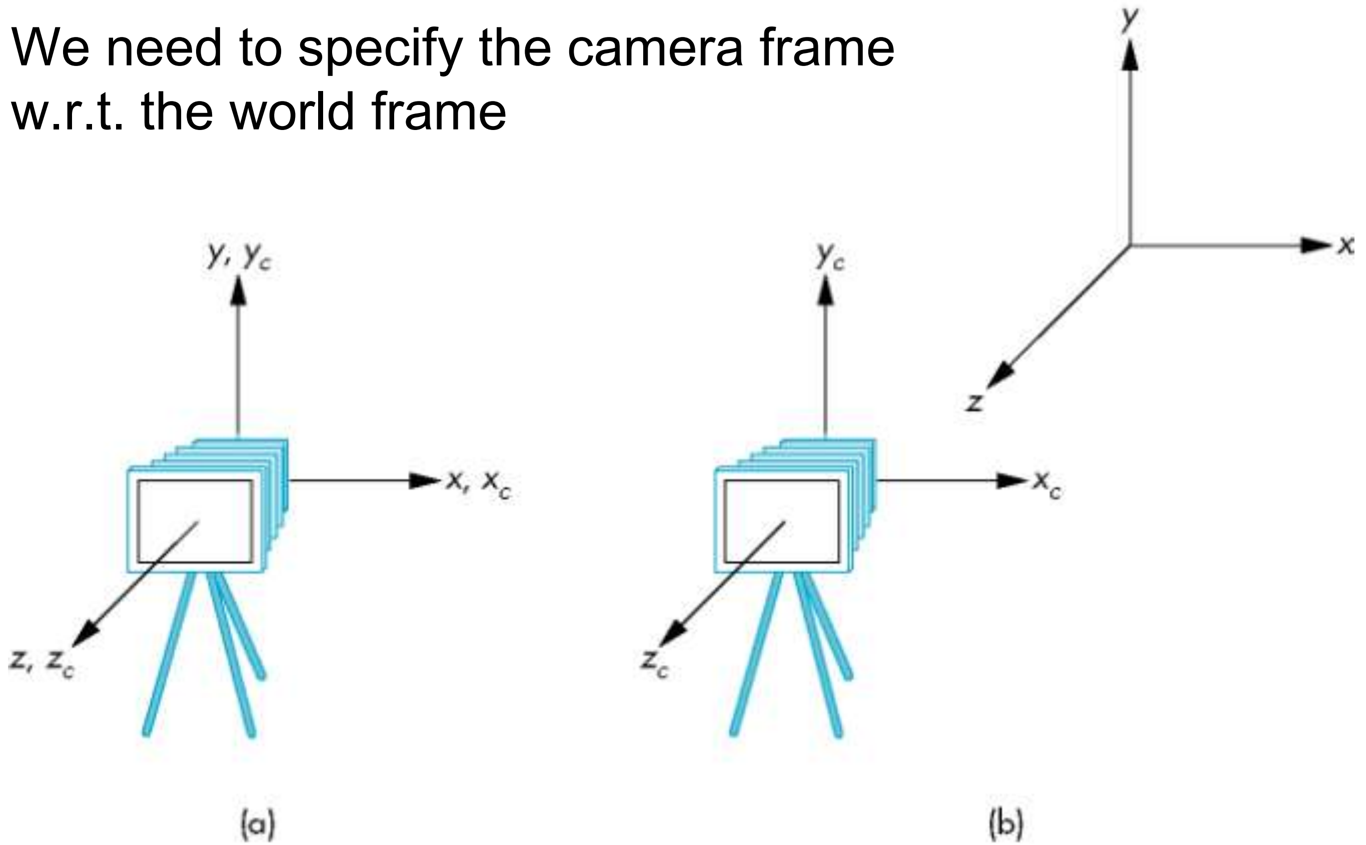
- The camera has a local coordinate frame, called the *camera coordinate frame*
  - Camera is located at the origin
  - Looking in negative z direction
  - $+y$-axis is the "*up-vector*"
- All projections are w.r.t. the camera frame
- Initially the world and camera frames are the same
  - Default model-view matrix is an identity



- To specify camera pose, we need to specify the camera coordinate frame w.r.t. the world coordinate frame

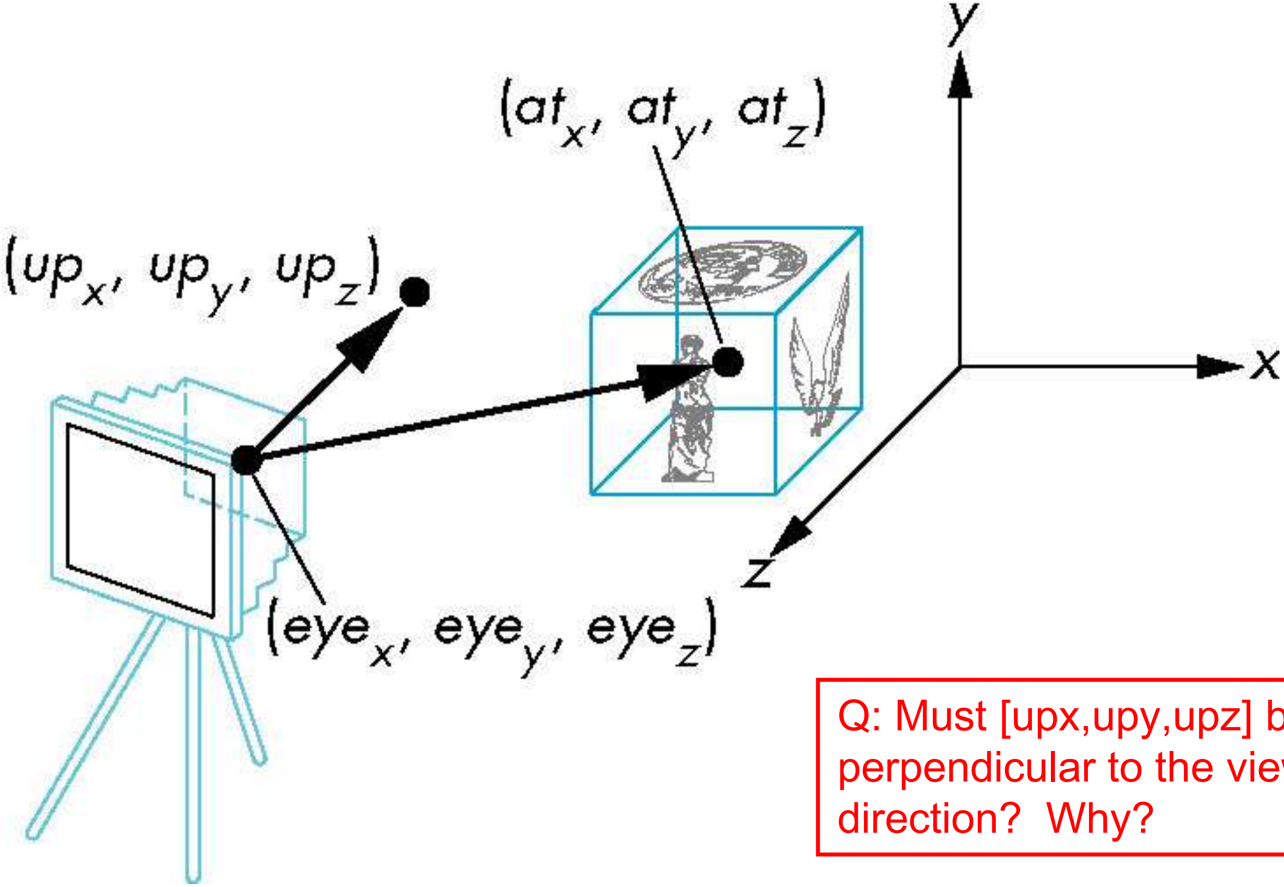
Specifying Camera Pose

- By default, the camera frame coincides with the world frame
- Often, we want to put the camera at other location and orientation
  - We need to specify the camera frame w.r.t. the world frame



Using the `gluLookAt()` Function

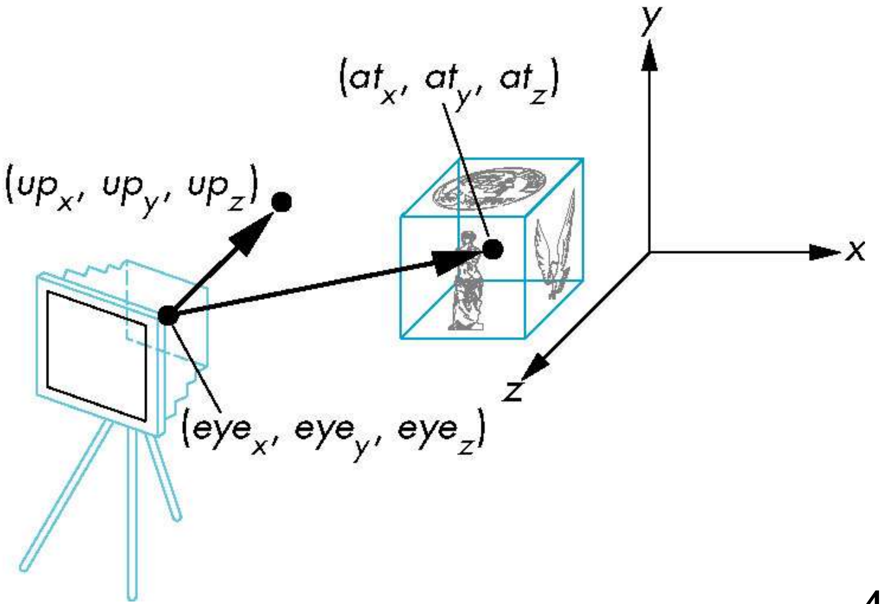
```
gluLookAt( eyex, eyey, eyez,
           atx, aty, atz,
           upx, upy, upz );
```



Q: Must [upx,upy,upz] be perpendicular to the view direction? Why?

The `gluLookAt()` Function

- Note that it does not directly specify the camera frame axes vectors **u**, **v**, **n** (correspond to x_c, y_c, z_c axes)
- The “up-vector” may not be perpendicular to the view direction
- The vectors **u**, **v**, **n** can be derived as follows
  - $\mathbf{n} = \text{normalize}(\,\mathbf{eye} - \mathbf{at}\,)$
  - $\mathbf{u} = \text{normalize}(\,\mathbf{up}\,) \times \mathbf{n}$
  - $\mathbf{v} = \mathbf{n} \times \mathbf{u}$



The `gluLookAt()` Function

- Conceptually, it positions the camera at the required location and orientation (w.r.t. the world frame)
- Internally, it generates a transformation matrix that can be used to express all points in the world frame w.r.t. the camera frame
- This is called *view transformation*
- This *view transformation matrix* is post-multiplied to the current model-view matrix
- In OpenGL, the view transformation matrix is normally the last transformation in the model-view matrix

```
glMatrixMode( GL_MODELVIEW ):
glLoadIdentity();
// specify view transformation matrix here;
...
```

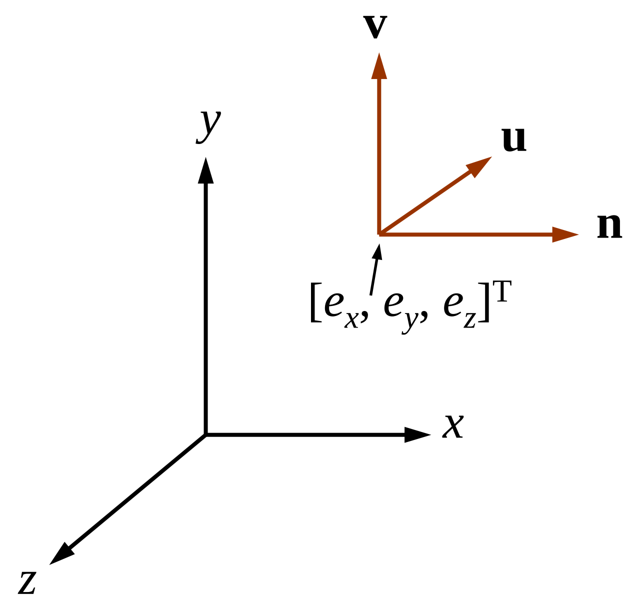
View Transformation

- All points in the world frame are expressed w.r.t. the camera frame
  - Can be performed using a 4x4 matrix
  - It is made up of a translation first, then a rotation
    - $\mathbf{M}_{view} = \mathbf{R}\ \mathbf{T}$
  - The translation $\mathbf{T}$ moves the camera position back to the world origin
  - The rotation $\mathbf{R}$ rotates the axes of the camera frame to coincide with the corresponding axes of the world frame
  - Multiply all points in the world frame by $\mathbf{M}_{view}$ and they will be expressed w.r.t. to the camera frame
  - Why view transformation?

Deriving View Transformation Matrix

- Suppose the camera has been moved to the location $[e_x, e_y, e_z]^T$, and its x_c, y_c, z_c axes are the unit vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{n}$, respectively, then

$$\mathbf{M}_{\text{view}} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- Note that $[e_x, e_y, e_z]^T$ and $\mathbf{u}$, $\mathbf{v}$, $\mathbf{n}$ are all specified w.r.t. to the world frame

Q: Why use homogeneous coordinates in computer graphics?

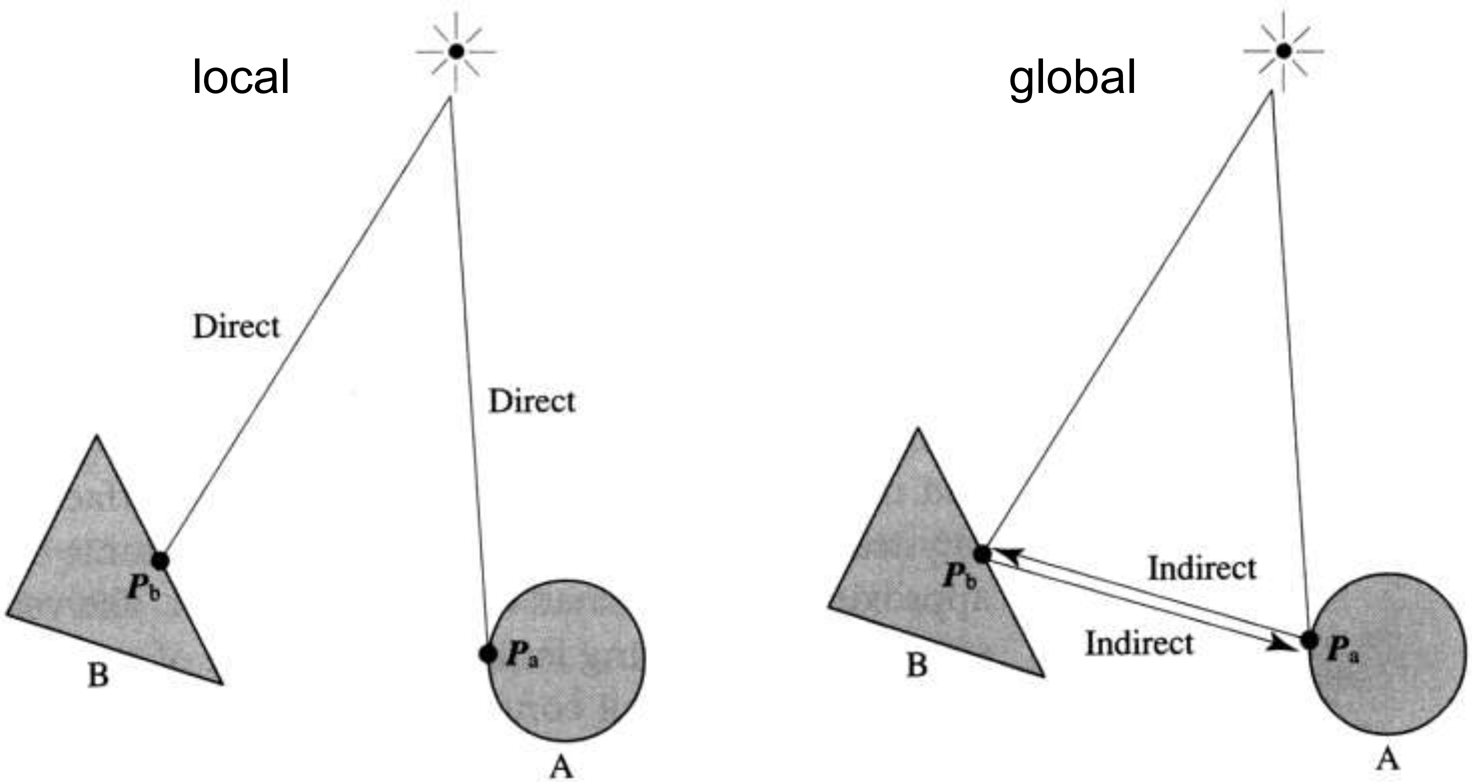
Transformation of Normal Vectors

- When vertices are transformed by a transformation $\mathbf{M}$, how should the normal vectors be transformed?
  - $\mathbf{M}_n = (\mathbf{M}_t^{-1})^T$
    - where $\mathbf{M}_t$ is the upper-left 3x3 submatrix of $\mathbf{M}$
  - What is $\mathbf{M}_n$ if $\mathbf{M}_t$ is just a rotation?

Note that the matrix $\mathbf{M}_n$ is computed by OpenGL automatically.

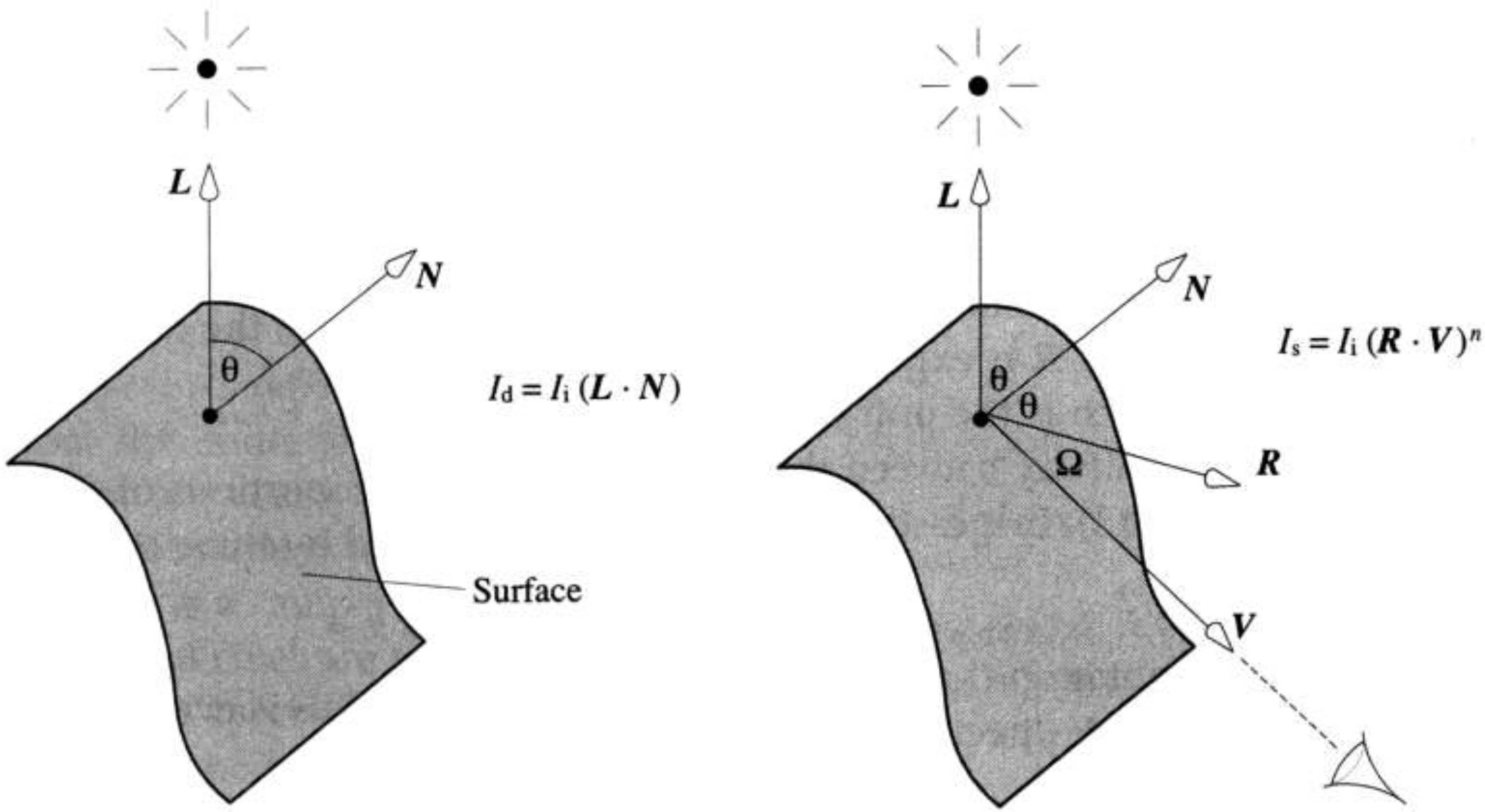
Lighting Computation in Eye Space

- Calculate reflected light intensity at each vertex
  - Using a local reflection model, e.g. Phong Model



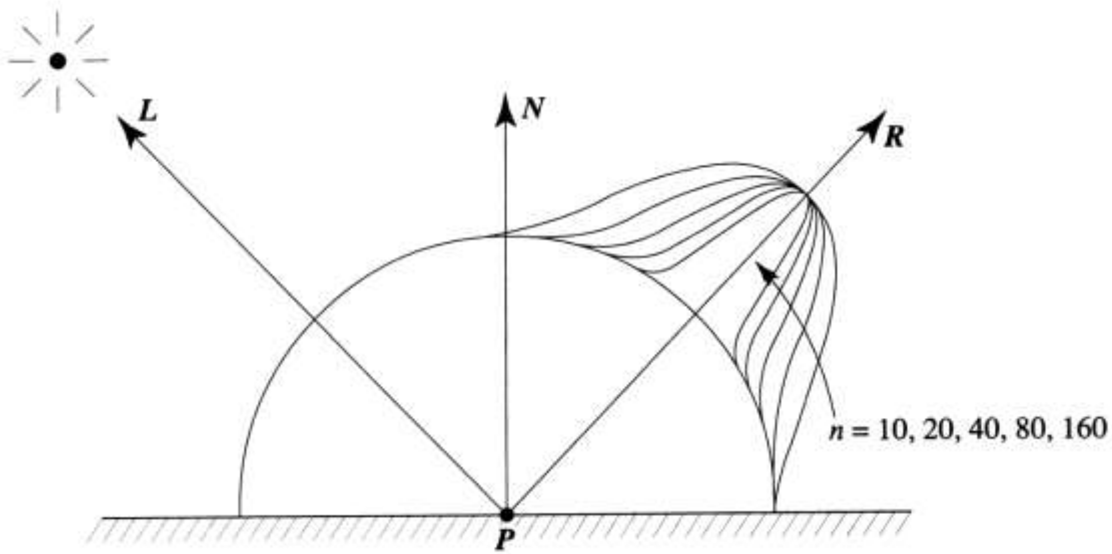
Phong Reflection Model

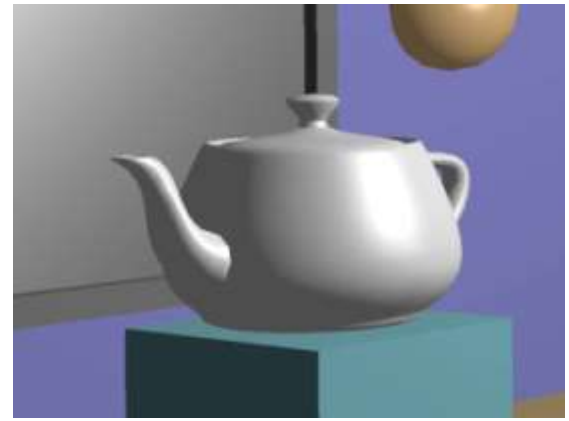
$$I = k_a I_a + I_i(k_d(\boldsymbol{L}\cdot\boldsymbol{N}) + k_s(\boldsymbol{R}\cdot\boldsymbol{V})^n)$$



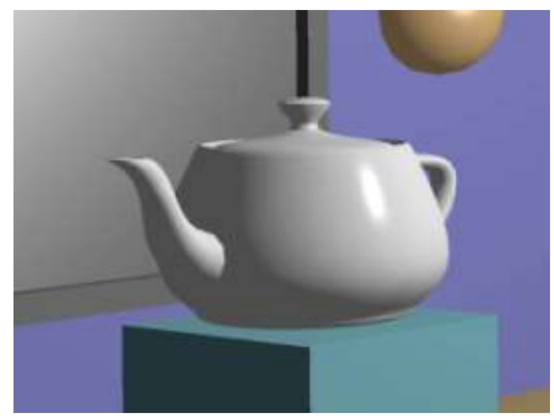
Phong Reflection Model

$$I = k_a I_a + I_i(k_d(\mathbf{L}\cdot\mathbf{N}) + k_s(\mathbf{R}\cdot\mathbf{V})^n)$$



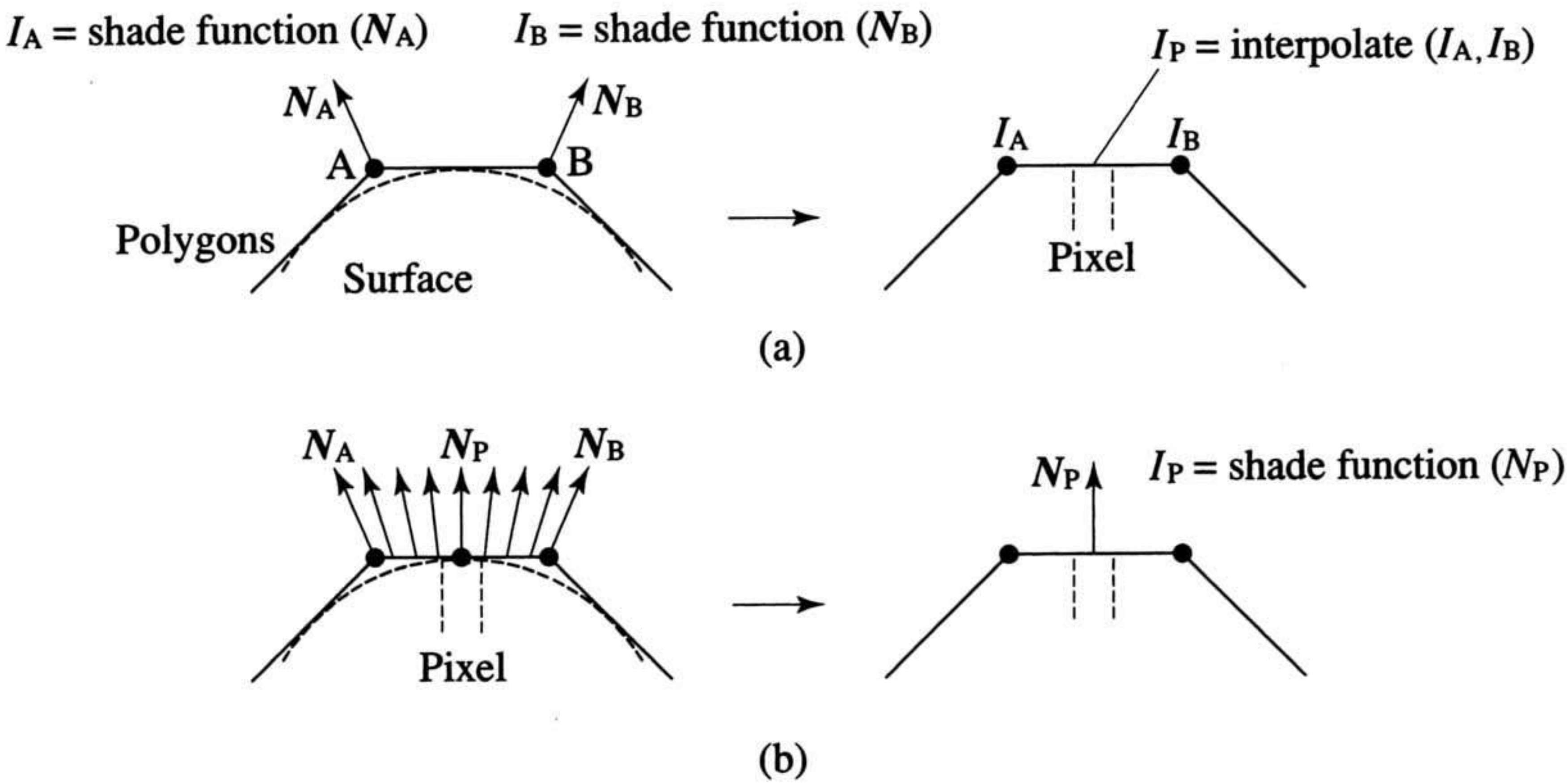


small exponent *n*



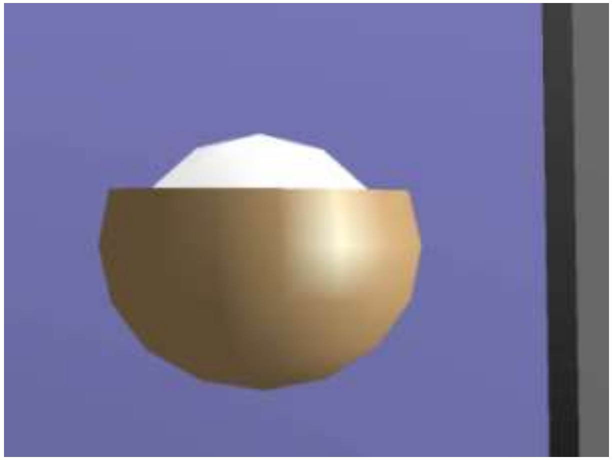
large exponent *n*

Gouraud & Phong Shading

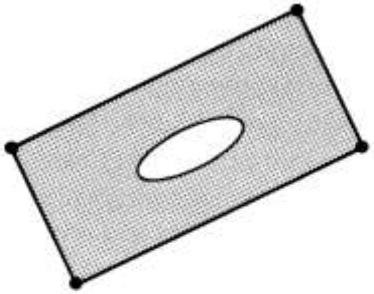


Gouraud & Phong Shading

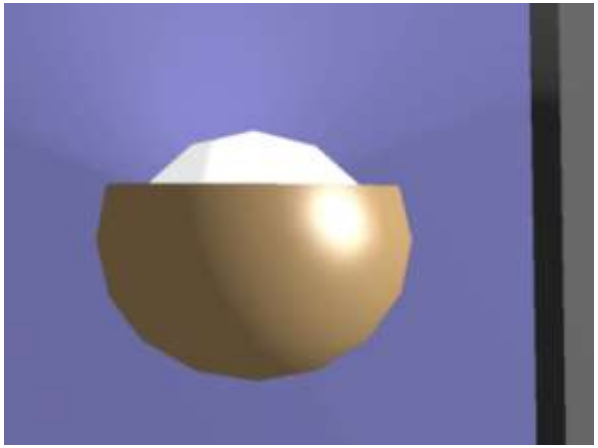
Flat Shaded



Gouraud Shaded



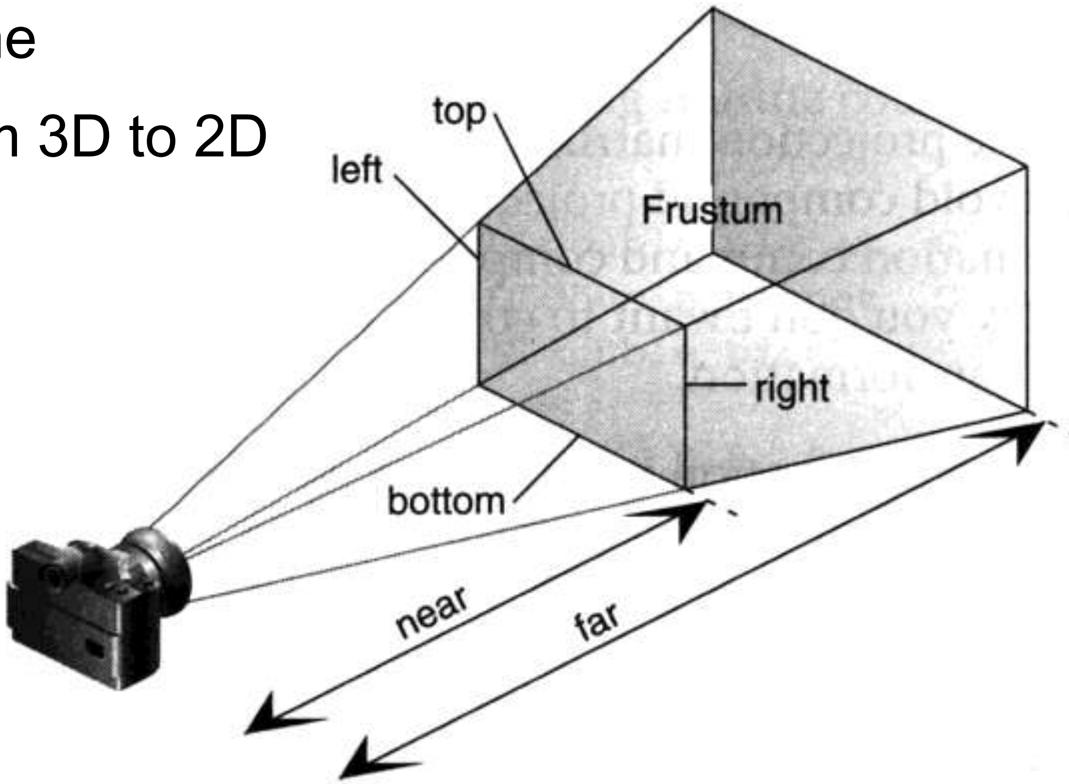
Gouraud shading may miss highlight in the interior of polygon.



Phong Shaded

Projections

- Specifying a view volume for projection
  - Specified w.r.t. the eye space
  - 3D region of the scene to appear in the rendered image
  - A clipping volume
  - A projection from 3D to 2D



Projections

- In OpenGL, after a vertex is multiplied by the model-view matrix, it is then multiplied by the projection matrix
- The projection matrix is a 4x4 matrix that defines the type of projection
- The projection matrix can be specified by first defining a *view volume* (or *clipping volume*) in the camera frame
  - For orthographic projection, use **`glOrtho()`**
  - For perspective projection, use **`glFrustum()`**

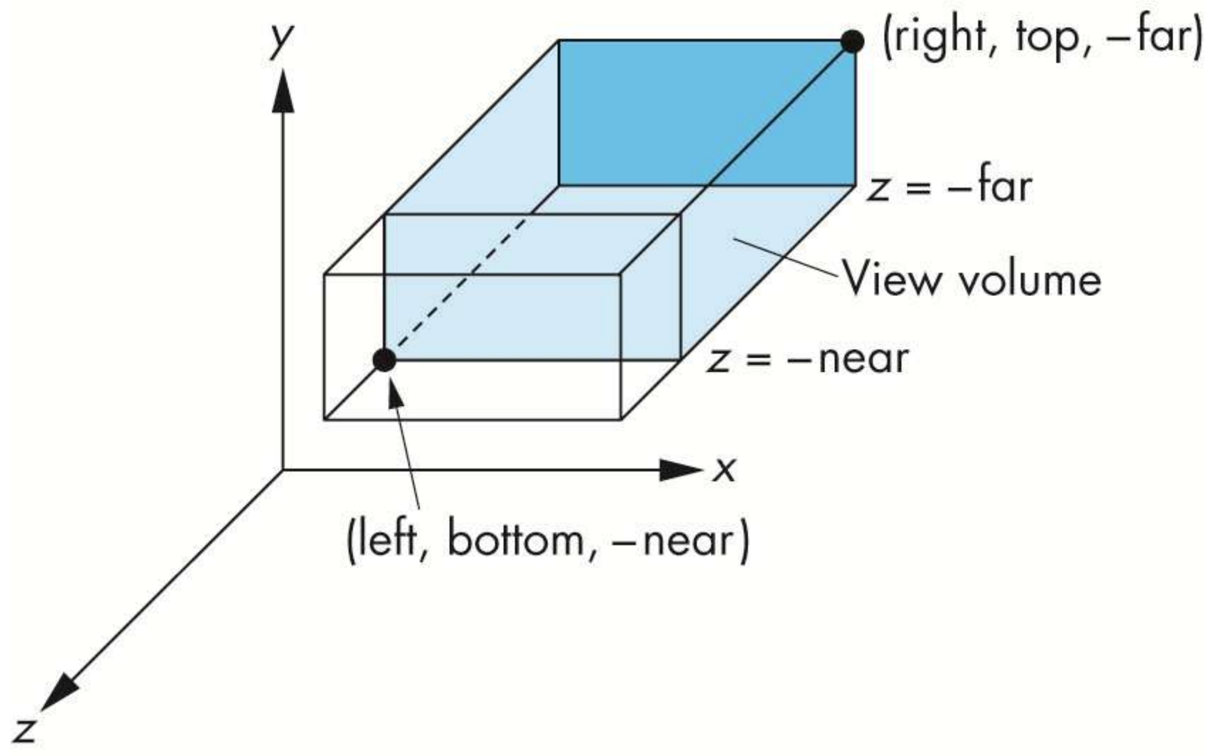
Projections

- A projection matrix is then computed such that it maps points in the view volume to a *canonical view volume*
    - The canonical view volume is the 2 x 2 x 2 cube defined by the planes $x = \pm 1$, $y = \pm 1$, $z = \pm 1$
    - Also called the *Normalized Device Coordinates* (NDC)
- Some criteria
    - Preserve depth order (in z coordinate)
    - Preserve lines
- The canonical view volume is then mapped to the viewport (*viewport transformation*)

OpenGL Orthographic Projection

- Can be specified by defining a view volume (in the camera frame) using

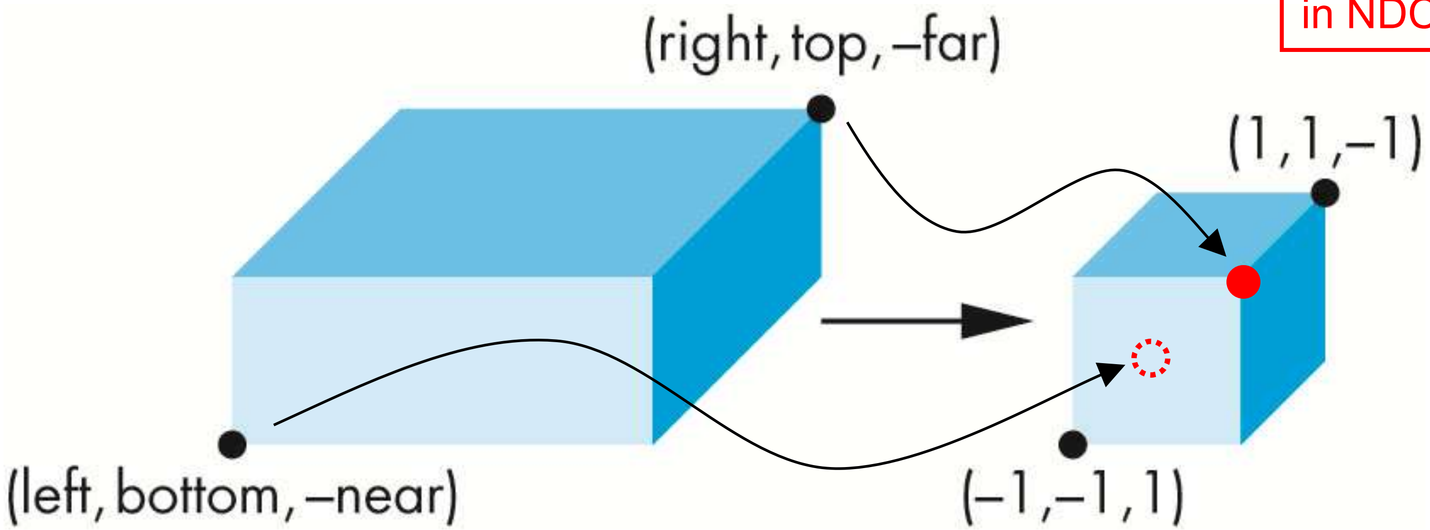
```
glOrtho( left, right, bottom, top, near, far );
```



OpenGL Orthographic Projection

- The **`glOrtho()`** function then generates a matrix that linearly maps the view volume to the canonical view volume, where
  - (left, bottom, –near) is mapped to (–1, –1, –1)
  - (right, top, – far) is mapped to (1, 1, 1)

Q: Why are the z values inverted in NDC?



Orthographic Projection Matrix

- The mapping can be found by
  - First, translating the view volume to the origin
  - Then, scaling the view volume to the size of the canonical view volume

$$\mathbf{M}_{\text{ortho}} = \mathbf{S}\left(\frac{2}{right-left}, \frac{2}{top-bottom}, \frac{2}{near-far}\right)\cdot$$
$$\mathbf{T}\left(\frac{-(right+left)}{2}, \frac{-(top+bottom)}{2}, \frac{(far+near)}{2}\right)$$

  - Note that $z = -near$ is mapped to $z = -1$, and $z = -far$ to $z = +1$

Orthographic Projection Matrix

$$\mathbf{M}_{\text{ortho}} = \begin{bmatrix} \dfrac{2}{right - left} & 0 & 0 & \dfrac{-(right + left)}{right - left} \\ 0 & \dfrac{2}{top - bottom} & 0 & \dfrac{-(top + bottom)}{top - bottom} \\ 0 & 0 & \dfrac{-2}{far - near} & \dfrac{-(far + near)}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

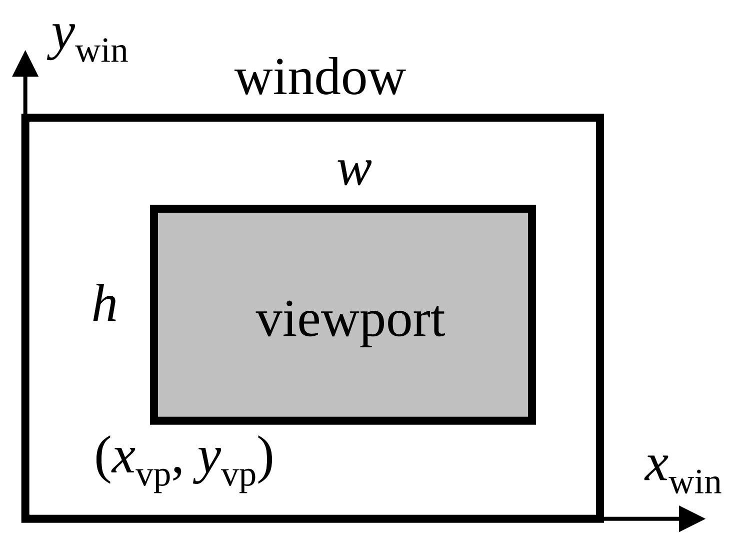
Viewport Transformation

- The canonical view volume is then mapped to the viewport (from NDC to window coordinates)

$$\frac{x_{\text{NDC}}-(-1)}{2}=\frac{x_{\text{win}}-x_{\text{vp}}}{w} \Rightarrow x_{\text{win}}=x_{\text{vp}}+\frac{w(x_{\text{NDC}}+1)}{2}$$

$$\frac{y_{\text{NDC}}-(-1)}{2}=\frac{y_{\text{win}}-y_{\text{vp}}}{h} \Rightarrow y_{\text{win}}=y_{\text{vp}}+\frac{h(y_{\text{NDC}}+1)}{2}$$

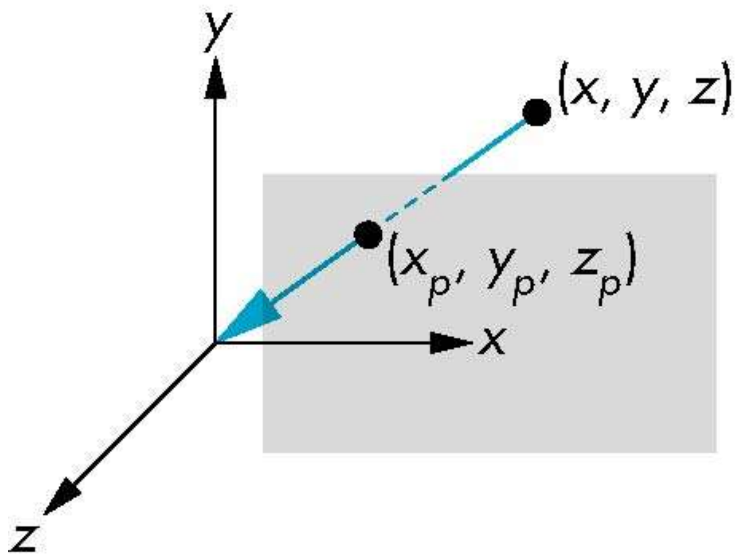
$$z_{\text{win}}=\frac{z_{\text{NDC}}+1}{2}$$



- By default, z_{win} is between 0 and 1
  - set using glDepthRange()
- It is needed for z-buffer hidden surface removal

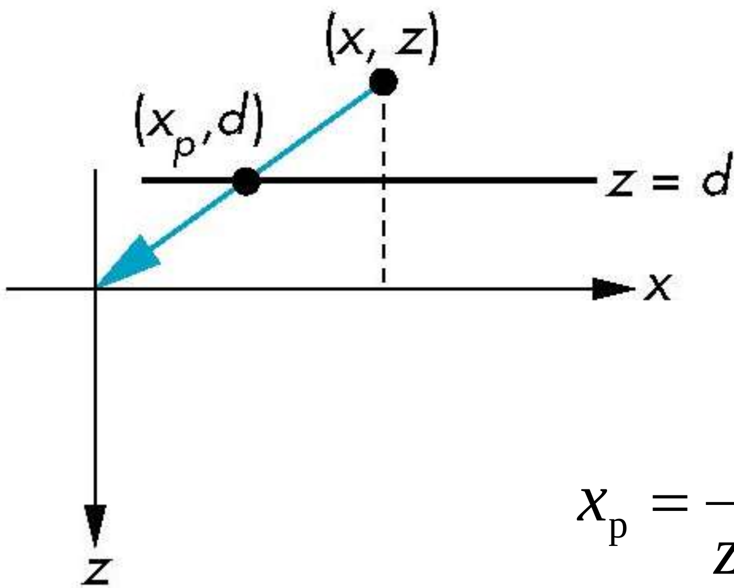
Simple Perspective Projection

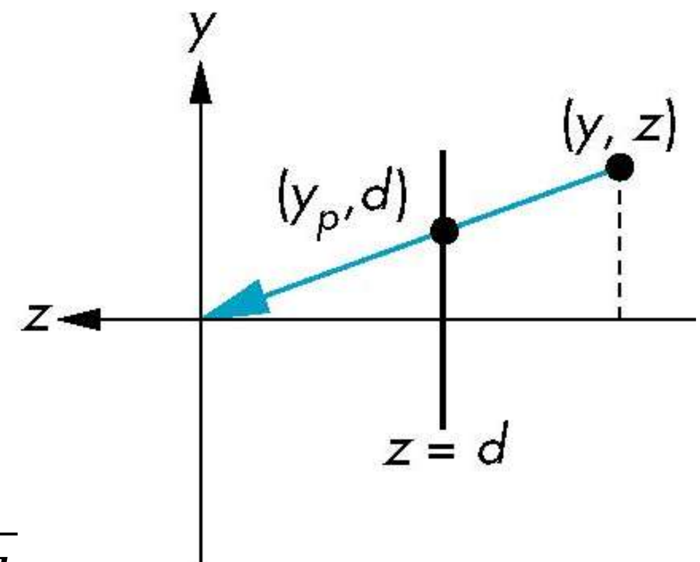
- Center of projection at the origin
- Projection plane is $z = d$, $d < 0$



Perspective Equations

- Consider top and side views





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

$$y_p = \frac{y}{z/d}$$

$$z_p = d$$

Using Matrix Multiplication

- Consider $\mathbf{p} = \mathbf{M}\mathbf{q}$ where

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \qquad \mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \qquad \mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Perspective Division

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

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \xrightarrow{\text{perspective division}} \mathbf{p}' = \begin{bmatrix} \dfrac{x}{z/d} \\ \dfrac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

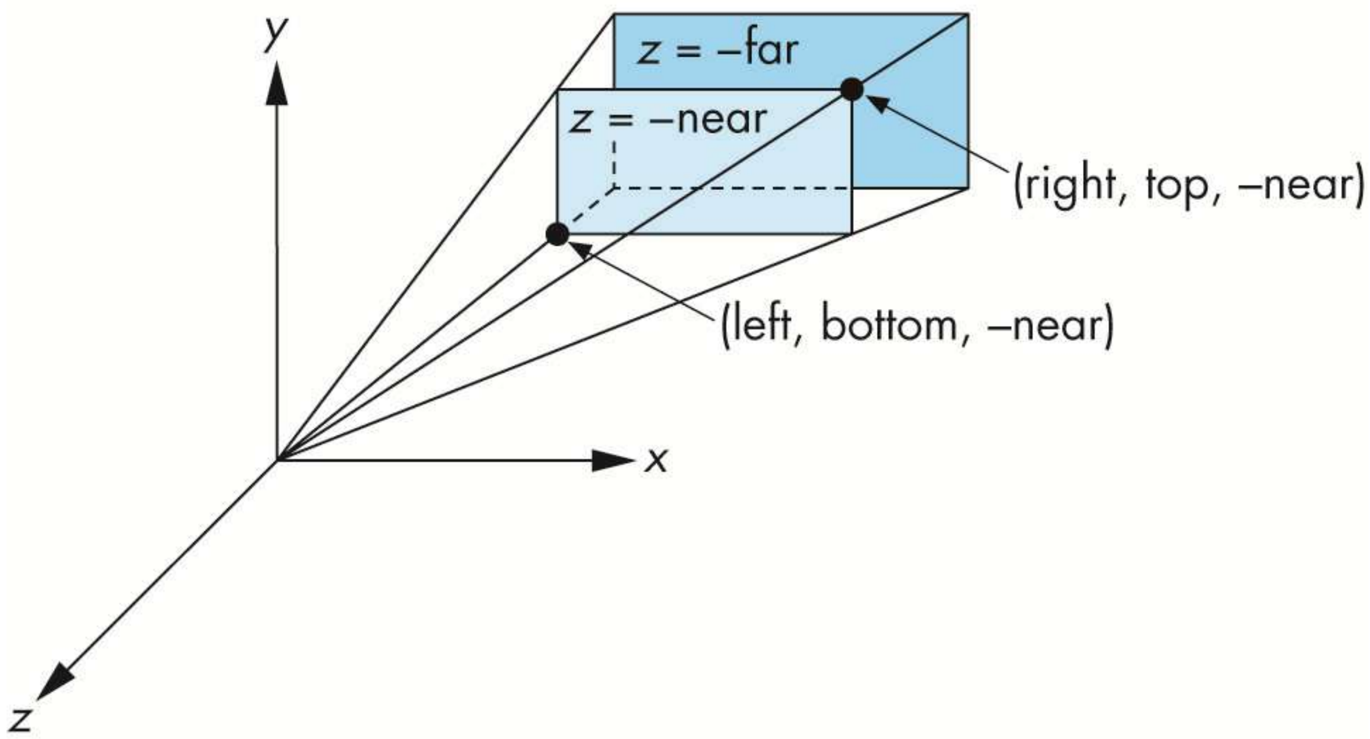
  the desired perspective equations

- This is one reason why 3D graphics API uses homogeneous coordinates

OpenGL Perspective Projection

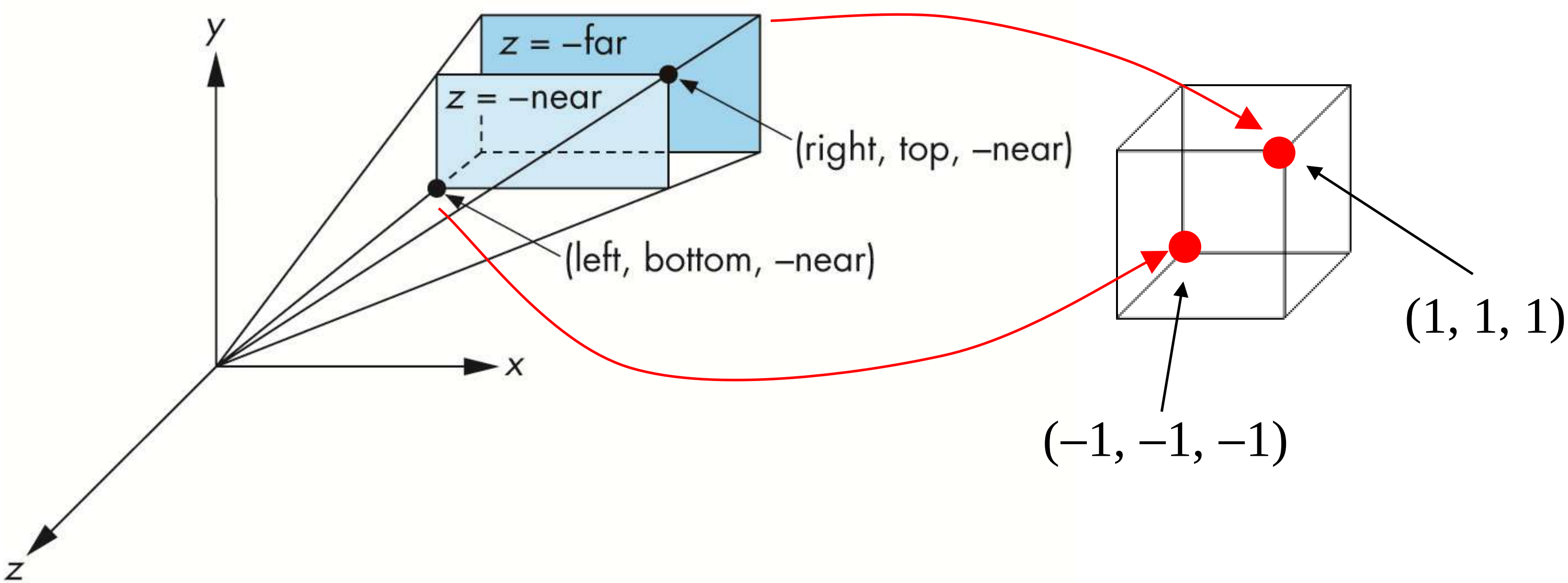
- Can be specified by defining a view volume (*view frustum*) in the camera frame using

```
glFrustum( left, right, bottom, top, near, far );
```



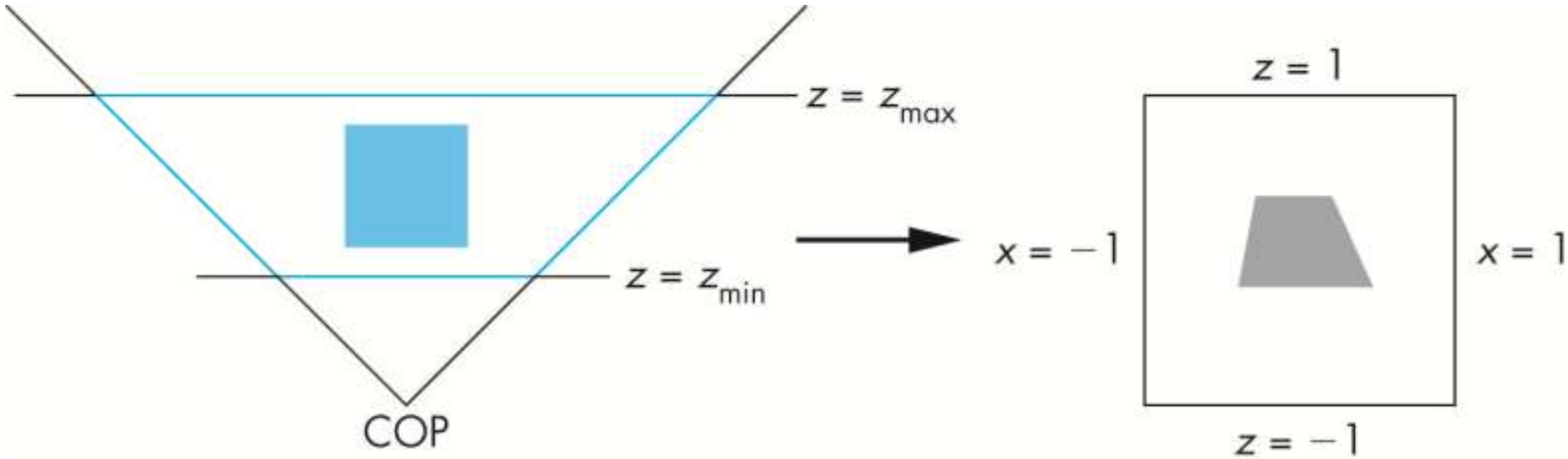
OpenGL Perspective Projection

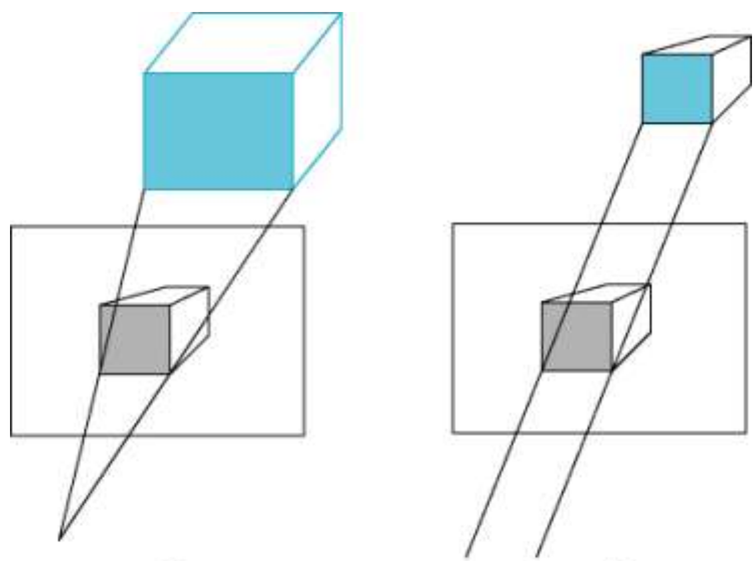
- The `glFrustum()` function then generates a matrix that maps the view frustum to the canonical view volume, where



Q: Why transform view volume to canonical cube?

OpenGL Perspective Projection





(a) (b)

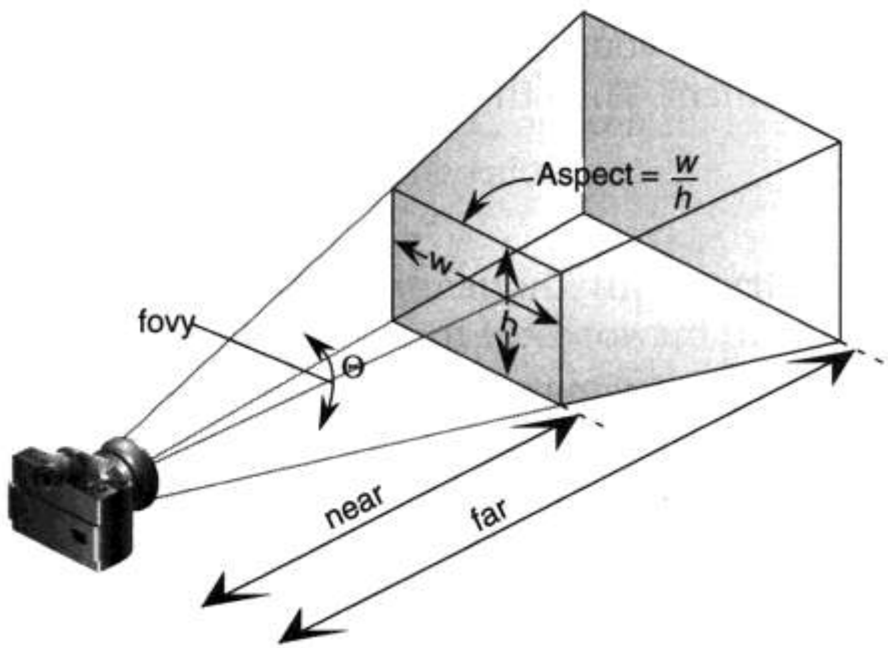
Perspective Projection Matrix

$$\mathbf{M}_{\text{persp}} = \begin{bmatrix} \frac{2 \cdot near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2}{top - bottom} & \frac{top + bottom}{top - bottomt} & 0 \\ 0 & 0 & \frac{-(far + near)}{far - near} & \frac{-2 \cdot far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective Projection Using Field of View

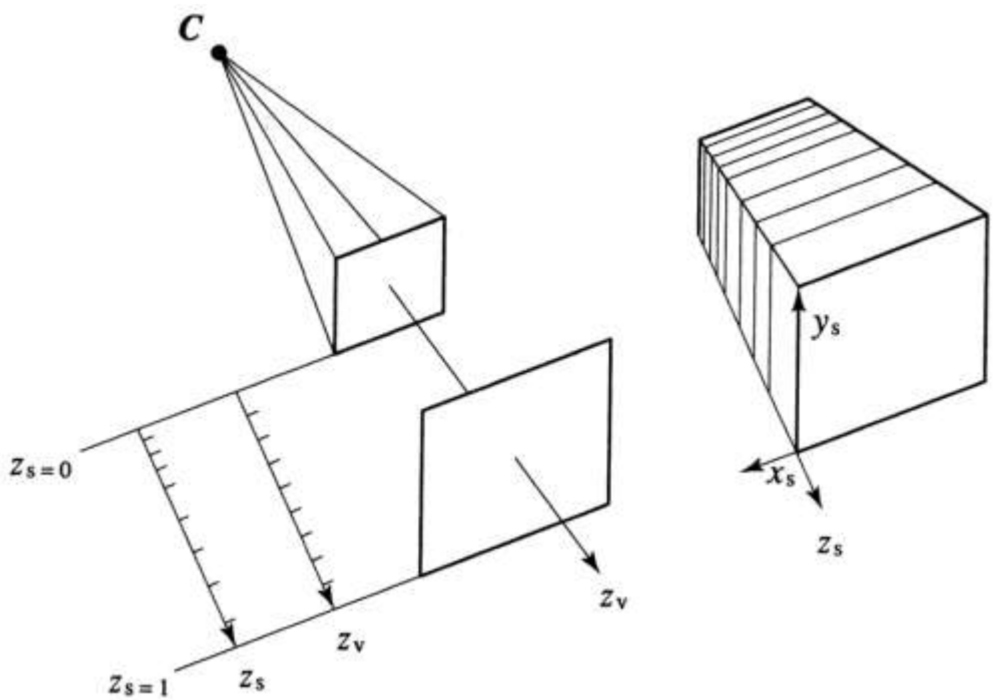
- The **`glFrustum()`** function allows (off-center) asymmetric view volume
  - What are the uses of off-center view volume?
- For symmetric view volume, we can also use

```
gluPerspective( fovy, aspect, near, far );
```



Distortion in Z Values

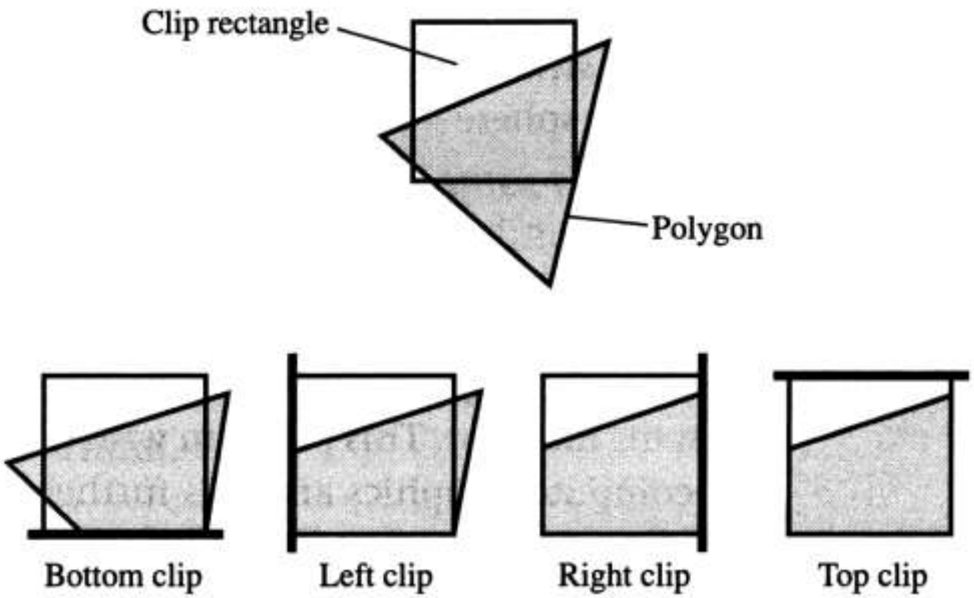
Distortion in z values after perspective projection.



Precision in z values.

Clipping Space

- It is the space after perspective transformation $\mathbf{M}_{\text{persp}}$ and before the perspective division by $w = -z_v$
- Polygons partially inside the view volume are clipped
- The clipping limits
  - $-w \leq x \leq w$
  - $-w \leq y \leq w$
  - $-w \leq z \leq w$



Framebuffer

- "Standard" window-system-provided framebuffer
  - Up to 4 color buffers
    - Front-left, front-right, back-left, back-right
  - A depth buffer
  - A stencil buffer
  - An accumulation buffer
  - A multisample buffer
  - One or more auxiliary buffers
    - Offscreen memory buffers
- Other offscreen memory and texture images can be used as framebuffer through framebuffer object
  - Render-to-texture
  - Multiple rendering targets (MRT)

OpenGL Execution Model

- Client-server
  - Client – application program
  - Server – OpenGL implementation (driver and hardware)
  - Majority of OpenGL states stored at server side
- OpenGL commands always processed in order
  - Out-of-order execution of OpenGL commands not allowed
  - Applies to queries of state and frame buffer read operations
- Data binding occurs when commands are issued
  - Data copied into OpenGL memory
  - Subsequent changes to this data by the application have no effect

End of Lecture 1