8 Viewing and Projection

1. OpenGL Rendering Pipeline
2. OpenGL tools for Modeling and Viewing
3. Orthographic and Perspective Cameras
4. View Transformation
5. Specifying View Position and Orientation
6. Perspective Transformation
7. Clipping Edges after Perspective Transformation

- Textbook Readings: Hill 5.6.1, 5.6.2; Chapter 7.1 – 7.4

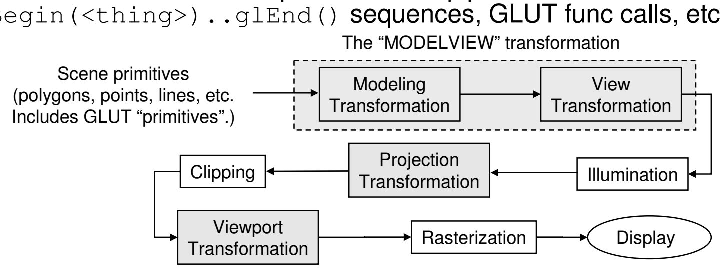
8 Viewing and Projection

- **Learning objectives and problems to be solved**
  - **Transformations** and **projections** needed to render a 3D scene: What are the modeling, viewing , and projection transformations and how are they applied in the rendering pipeline? How are they invoked in OpenGL?
  - **Viewport:** What is the viewport transformation, how is it used, how do we create multiple viewports?
  - **View Transformation**
    - What are some different ways a view transformation can be specified, what is the matrix for the transformation, and how is it implemented in OpenGL?
    - How can we specify a view attached to an object in the scene?
  - **View Projection**
    - What are the transformations for orthographic and perspective projection?
    - How are homogeneous coordinates used for perspective scaling?
    - How are 3D objects clipped in 4D space?

8.1 OpenGL Rendering Pipeline

Hill Chapter 5.6.1 (review)

- User sets up state of transformation matrices in pipeline with calls like `glOrtho`, `glTranslatef`, `glRotatef`, etc
- Then user sends scene components down pipeline with `glBegin(<thing>)..glEnd()` sequences, GLUT func calls, etc



- After a program sets the transformations to be used OpenGL automatically applies transformations to all vertices.
- These notes discuss various transformation stages of pipeline
  - MODEL_VIEW, PROJECTION and Viewport transformations

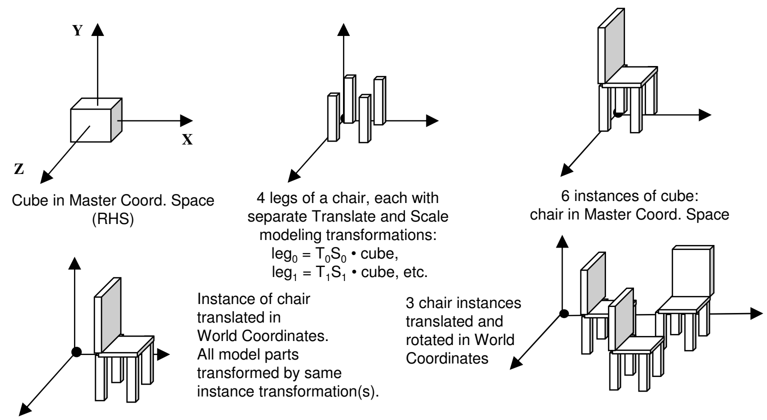
Rendering Pipeline: ModelView Matrix

- **Modelview matrix:** combines **modeling** and **viewing** transforms.
  - **Modeling transforms: M**, translate, rotate, and scale applied to **primitives** to compose objects of 3D scene. *** **Different transforms** for **each object**.
  - **Viewing transforms: V**, translate and rotate applied to position the camera (eye) for viewing. *** **Same viewing transforms** applied **all objects**.
  - **V** and **M** combined into one modelview matrix, $\mathbf{M}_{ModelView}$
    $\mathbf{M}_{ModelView} = \mathbf{V}\,\mathbf{M} = (\mathbf{Rz_V Ry_V Rx_V T_V})(\mathbf{T_0 Rx_0 Ry_0 Rz_0 S_0})$ – when object 0 drawn
    $\mathbf{M}_{ModelView} = \mathbf{V}\,\mathbf{M} = (\mathbf{Rz_V Ry_V Rx_V T_V})(\mathbf{T_1 Rx_1 Ry_1 Rz_1 S_1})$ – when object 1 drawn.
  - Transforming a 3D point *** **ORDER OF MATRICES IS IMPORTANT!!!**
    Mathematically: model transformations applied 1st, view transformations 2nd.
    Transformed $P' = \mathbf{M}_{ModelView}\,P = (\mathbf{Rz_V Ry_V Rx_V T_V})(\mathbf{T_0 Rx_0 Ry_0 Rz_0})\,\mathbf{S_0}\,P$

$$= (\mathbf{Rz_V Ry_V Rx_V T_V})(\mathbf{T_0 Rx_0 Ry_0})\,\mathbf{Rz_0}\,P^{(1)}$$
$$= (\mathbf{Rz_V Ry_V Rx_V T_V})(\mathbf{T_0 Rx_0})\,\mathbf{Ry_0}\,P^{(2)}$$
$$\ldots$$
$$= (\mathbf{Rz_V})\,\mathbf{Ry_V}\,P^{(5)}$$
$$= \mathbf{Rz_V}\,P^{(6)} = P'$$

Rendering Pipeline: Modeling Transf.

- **Modeling Transformation Examples:**



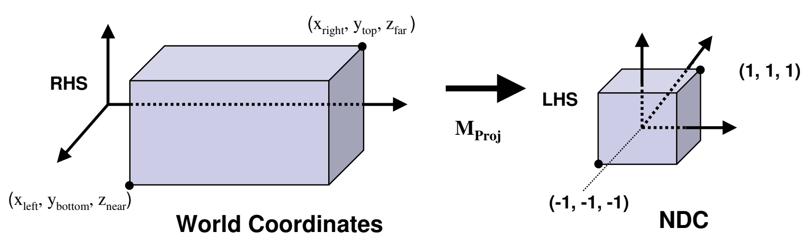
- **OpenGL demo** – Instance (modeling) and view transformations.

Rendering Pipeline: Projection Matrix

- **Projection matrix:** specifies transformation from **3D World coordinate space** to **normalised 3D camera/eye coord. space**.
  - Defines **3D viewing volume** that will be mapped onto the 2D drawing window, i.e., the **viewport** (actually still in 3D viewport, because 3D ⇨ 2D projection occurs after 3D clipping and visibility computations during rasterization stage).
  - Projection transformation matrix, $\mathbf{M}_{\mathbf{Proj}}$, maps **3D World Coordinate** values into **3D Normalised Device Coordinates** (**NDC**). View volume boundaries (rectangular block) mapped to {-1, +1} cube in X, Y, and Z. 3D clipping performed most efficiently in NDC.
  - **Aspect ratio** (width/height ratio) of **view volume** must match aspect ratio of **viewport** to preserve correct x,y,z proportionality of objects.

Projection Matrix (cont'd)

- In OpenGL **window coords.** are **relative to eye position**.
- In OpenGL **World Coords**. are **RHS** and **NDC** are **LHS**, so projection transformation also inverts Z values. Allows Z clipping planes to be specified as positive distances from the eye position. For example, Z coord. of near clip plane = $Z_{eye} - Z_{near}$, Z coord. of far clip plane = $Z_{eye} - Z_{far}$



- Similar mapping for perspective projection (see later slides)
- **OpenGL demo** – orthographic and perspective view volumes

Rendering Pipeline: Viewport Matrix

- **Viewport transformation :** specifies mapping from **normalised window** (3D viewing volume in NDC) to a 3D **viewport**.
  - After passing through the MODEL_VIEW and PROJECTION matrices, all vertex coordinates x,y *and* z are in range -1 to 1.
  - Finally, these floating point values have to be mapped to integer screen coordinates (becomes input values for rasterization stage).
  - Mapping: from range {-1, +1} usually
    to range {0, WINDOW_WIDTH} and {0, WINDOW_HEIGHT}
    - But user can override this with a call to

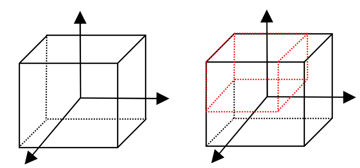
```
glViewport(x, y, width, height);     // or alternately
glViewport(xmin, ymin, xmax-xmin, ymax-ymin);
```

    - We used this command in the GLUT window reshape callback function.
- **Viewport matrix**
  - Maps **NDC boundaries** onto **viewport boundaries** (also called **Device Coordinates, DC**).

Rendering Pipeline: Viewport Matrix

- In OpenGL viewport matrix includes inverting Y coordinates because viewport coordinate origin is at upper left.
- Viewport transformation is the world-to-viewport mapping from chapter 3. Rewrite the equation from chapter 3 in homogeneous coordinates and replace w.l, w.r, w.b, w.t with -1, 1, -1, 1, respectively.
- Denote the normalised World Coordinates (NDC coords. in range {-1, +1}) by **x** = $(x, y, z, 1)^T$ and the 3D screen coordinates (in range {-1, +1}) by **u** = $(u, v, n, 1)^T$ then the **world-to-viewport mapping** (NDC-to-DC) is:

$$\begin{pmatrix} u \\ v \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{v.r - v.l}{2} & 0 & 0 & \frac{v.r + v.l}{2} \\ 0 & \frac{v.t - v.b}{2} & 0 & \frac{v.t + v.b}{2} \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$



- UDOO the transformation for device coordinates, DC, in range: {0, (maxScreenX-1)}, (maxScreenY-1), 0}, {0, (maxZbuffer-1)}:

Viewport Matrix (cont'd)

- **Problem:** How to write a GL program that displays multiple views of a scene, each one in a different viewport?
- **Solution: Multiple viewports**
  Multiple views of a scene, e.g., architectural drawing front, side, and top views
  Loop: repeat for each viewport
  - Set this viewport: call OGL function
    `glViewport( x, y, width, height );`
  - Set view projection for this viewport (might be the same for all viewports, if so do this before loop)
    `glOrtho(left, right, bottom, top, zNear, zFar );`
    or other such as `gluPerspective( … );`
  - Set camera view position and orientation for this viewport
    `gluLookAt(left, right, bottom, top, zNear, zFar );`
    or other such as `glTranslatef( … ); glRotatef( … );`
  - Draw scene

Viewport Matrix (cont'd)

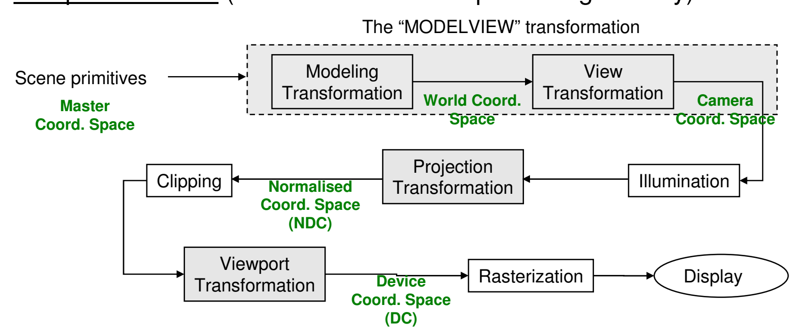
- **Multiple viewports code example:**
  4 views: perspective, front, side, and top (ortho). window = 1000 x 1000, viewports = 250 x 250.
  Demonstrating use of glTranslatef/glRotatef and gluLookAt.

```
// bottom left: perspective
glViewport( 0, 250, 250, 250 );
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective(yfov, aspect, zNear, zFar );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glRotatef( viewXAngle, 1.0f, 0.0f, 0.0f );
glTranslatef( viewX, viewY, viewZ );
drawScene();

// set orthographic projctn (all 3 vp)
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho(left, right, bottom, top,
        zNear, zFar );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();

// top left: orthographic, side view
glViewport( 0, 0, 250, 250 );
glLoadIdentity();
glRotatef( -90.0f, 0.0f, 1.0f, 0.0f );
glTranslatef( -10.f, 0.0f, 0.0f );
drawScene();

// top right: orthographic, front view
glViewport( 250, 0, 250, 250 );
glLoadIdentity();
gluLookAt(0.0f, 0.0f, 10.0f,
    0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
drawScene();
// bottom right: orthographic, top view
glViewport( 250, 250, 250, 250 );
glLoadIdentity();
gluLookAt(0.0f, 10.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
drawScene();
```

- **OpenGL demo program** – 1, 2, and 4 viewports

OpenGL Rendering Pipeline: Revision

- **Summary**: Rendering Pipeline Coordinate Spaces and Transformations
  - Note: to render multiple viewports with perhaps different view transformations and projections traverse the pipeline once for each viewport and view (redraw/resend same primitive geometry).



8.2 OpenGL Tools for Modeling, Viewing

- **Set camera for parallel (orthographic) projection:**
  - set matrix mode: `glMatrixMode( GL_PROJECTION );`
  - reset top of stack : `glLoadIdentity();`
  - multiply by 3D ortho matrix (values relative to eye position, LHS coords.): `glOrtho( left, right, bottom, top, near, far );`
  - $\mathbf{CTM_{Proj}}$[top of stack] ⇦ $\mathbf{CTM_{Proj}}$ post-multiplied by transf. matrix
- **Position and orient camera:**
  - `glMatrixMode( GL_MODELVIEW ); // select` $CTM_{ModelView}$ `stack`
  - reset top of stack : `glLoadIdentity();`
  - multiply by view transformation matrix, may use `glTranslatef()` and `glRotatef()`, or

```
gluLookAt(eye.x, eye.y, eye.z, look.x, look.y, look.z,
          up.x, up.y, up.z);
```

  - $\mathbf{CTM_{ModelView}}$[top of stack] ⇦ $\mathbf{CTM_{ModelView}}$ post-multiplied by transf. matrix

OpenGL Modeling, Viewing Tools (cont'd)

- **Set Model (instance) transformations:**
  - `glMatrixMode( GL_MODELVIEW ); // select` $CTM_{ModelView}$ `stack`
  - apply:

```
glTranslatef( tx, ty, tz );
glRotatef( angle, ux, uy, uz);   // angle in degrees
glScalef( sx, sy, sz );
```

  - $\mathbf{CTM_{ModelView}}$[top of stack] ⇦ $\mathbf{CTM_{ModelView}}$ post-multiplied by each transf. matrix
  - use `glPushMatrix()` and `glPopMatrix()` to save/restore matrix state for different model objects (but same view transformation).

OpenGL Modeling, Viewing Tools: Hints

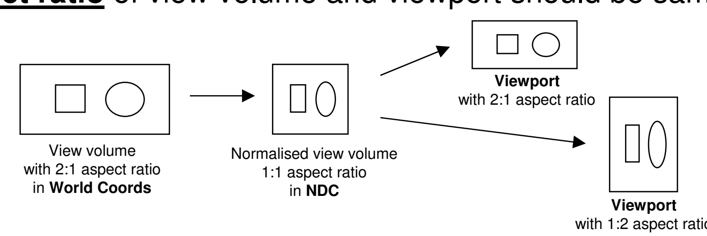
- **Modeling**, **viewing**, and **projection** functions merely set OpenGL **state**. Actual drawing occurs only when primitive functions are called.
- Normal order is **GL_PROJECTION** transf., then **GL_MODELVIEW** transf. (doesn't matter which is set first as long as both before drawing)
- **Order** of **model** and **view transf** calls (applied in opposite order):
  - **identity**: $\mathbf{CTM_{ModelView}}$[top] = **I**
  - **view transformations**: $\mathbf{CTM_{ModelView}}$[top] = $\mathbf{M_{View}}$
  - **push** matrix stack: $\mathbf{CTM_{ModelView}}$[top] and $\mathbf{CTM_{ModelView}}$[top-1] now both = $\mathbf{M_{View}}$
  - **$\mathbf{model_0}$ transformations**: $\mathbf{CTM_{ModelView}}$[top] = $\mathbf{M_{View}\ M_{Model_0}}$
  - draw 1st object: all vertices transformed by $\mathbf{CTM_{Proj}}$[top] $\mathbf{CTM_{ModelView}}$[top]
  - **pop** matrix stack $\mathbf{CTM_{ModelView}}$[top] = $\mathbf{M_{View}}$
  - **push** matrix stack: $\mathbf{CTM_{ModelView}}$[top] and $\mathbf{CTM_{ModelView}}$[top-1] now both = $\mathbf{M_{View}}$
  - **$\mathbf{model_1}$ transformations**: $\mathbf{CTM_{ModelView}}$[top] = $\mathbf{M_{View}\ M_{Model_1}}$
  - draw 2nd object: all vertices transformed by $\mathbf{CTM_{Proj}}$[top] $\mathbf{CTM_{ModelView}}$[top]
  - **pop** matrix stack $\mathbf{CTM_{ModelView}}$[top] = $\mathbf{M_{View}}$
- **MUST** set **identity** before **projection** and also before **view**, but **NOT** before any of the **model** transformations (why?)

OpenGL Modeling, Viewing: Aspect Ratio

- Final pipeline transformation step (after 3D clipping) is **viewport transformation**.

```
glViewport(GLint x, GLint y,
           GLsizei width, GLsizei height );
```

  Default viewport is entire drawing window, (0, 0, winWidth, winHeight).
- **Aspect ratio** of view volume and viewport should be same.



- **Problem:** How to write a GLUT program that automatically resets the view volume aspect ratio when window (viewport) is resized?

Aspect Ratio: reshape callback function

- **Solution:** in GLUT, use **reshape callback** to adjust **viewport** and view volume **aspect ratio** after a **window resize event**.
  - **Register reshape callback function** (in main at prog. init.)

```
void reshape(GLsizei width, GLsizei height);         // prototype
glutReshapeFunc( reshape );               // callback registration
```

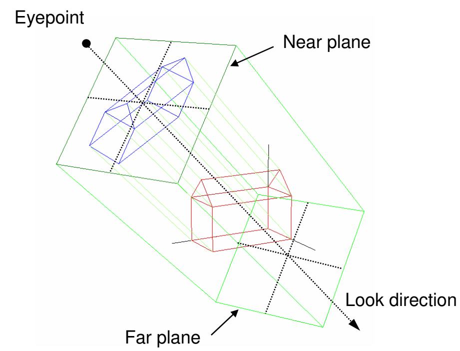
  - **Define reshape callback function** (in main prog. module)

```
// left, right, bottom, top = class member or global variables
void reshape( GLsizei width, GLsizei height ) {
   glViewport( 0, 0, width, height );         // set viewport size
   GLfloat aspect = (GLfloat)width / (GLfloat)height; // NOT int!!!
   GLdouble center = (left + right) / 2.0;
   GLdouble newHalfWidth = aspect * (top - bottom) / 2.0;
   left = center - newHalfWidth;      right = center + newHalfWidth;
   glMatrixMode(GL_PROJECTION);               // reset proj matrix
   glLoadIdentity();                          // 3D window->viewport
   glOrtho( left, right, bottom, top, zNear, zFar );     // mapping
   drawSceneObjects();                        // redraw all objects
}
```

- **OpenGL demo program** – viewport resize, aspect ratio resize

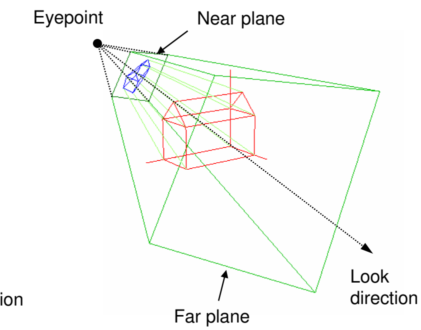
8.3 Orthographic & Perspective Cameras

- **Orthographic** (Hill 5.6.1/2)



a. View transformation
b. Transforming coords to range (-1,+1)
c. 3D → 2D Projection

- **Perspective** (Hill 7.2, 7.4, …)



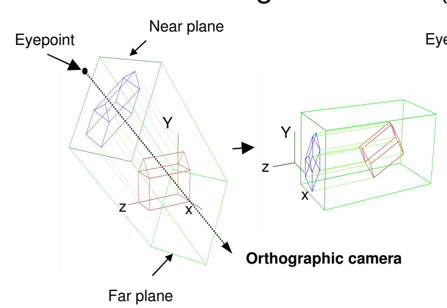
a. View transformation
b. Perspective transformation (includes scaling)
c. 3D → 2D Projection

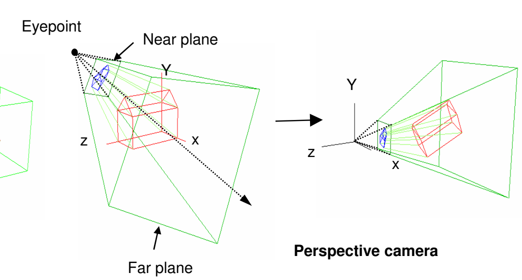
Ortho, Perspective Cameras: OpenGL

- **Orthographic**
  - `void glOrtho( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar )`
  - View volume boundaries in World Coord units, relative to eyepoint in the look direction. Z is positive distance from eye (along negative Z axis)
  - View volume ***may be* symmetric** about look direction vector (typical).
- **Perspective**
  - `void gluPerspective( GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar )`
  - Vertical field of view angle specified in degrees.
  - Horizontal fov determined by `aspect` ratio = width/height
    `fovx = aspect * fovy;`
  - View volume (frustum, or truncated pyramid) ***always* symmetric** about eyepoint towards the look direction.

8.4 View Transformation

- Default view transformation is identity: eye at origin looking down negative Z axis **[OpenGL demo program]**
- **View transformation** is combination of a **translation** that moves eye to World Coord. origin and a **rotation** that aligns look direction with negative Z axis (same for both projection types).





8.5 Specifying View Position & Orientation

- List of **Problems**
  How to write an OpenGL program that sets the view for a camera:
  1. Given an **camera (eye) position** and a **point to look at**?
  2. Given an **eye translation** and a **rotation**?
  3. For an **airplane flight simulator** (simulating the view out the front window) where the simulator position and orientation are controlled via pilot commands that set the plane's **pitch**, **yaw**, and **roll**?
  4. Mounted on a **pilot's helmet** (simulating the pilot's eye view such as in a virtual reality head mounted display) where the pilot can move (translate) and rotate his head **within the airplane's cockpit**?
  5. Mounted on the end of a **multi-jointed robot arm**, such as the NASA Space Shuttle Canadian arm?
- You already know the answer to #1, use gluLookAt(). But, there is no single gl, glu, or glut function for #2-#5 !

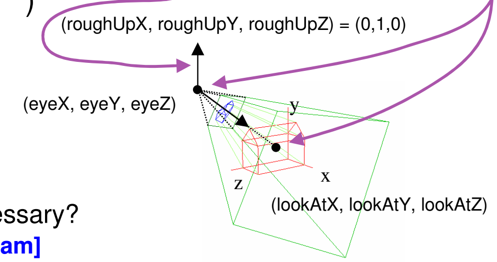
8.5 Specifying View Position & Orientation

- **Solution:** OpenGL program that sets view position & orientation given eye position and a point to look at. Use `gluLookAt()`
- Need:
  - Eyepoint
  - View direction
  - Something that specifies camera rotation around its axis `roughUp` may be any vector not parallel to (eye-look) vector. Along with the (eye-look) vector it defines the plane in which the true up vector must lie.

    Question: why is roughUp necessary?

    [OpenGL demo program]

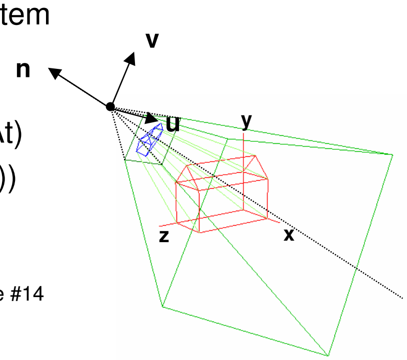
```
gluLookAt (
    eyeX,  eyeY,  eyeZ,
    lookAtX,  lookAtY,  lookAtZ,
    roughUpX,  roughUpY,  roughUpZ
)
```



The View Coordinate System (UVN)

a.k.a. *Eye Coordinate System* or *Camera Coordinate System*

- From the Eye and LookAt points plus the approximate Up vector, can derive UVN Coordinate system (Eye Coords.) basis vectors:
  - **n** = Normalised(Eye – LookAt)
  - **u** = Normalised(Cross(**Up, n**))
  - **v** = Cross(**n, u**)



Alternate definition: Burkhard's notes, 5.1 slide #14

The View Transformation Matrix, V

- Translates eye to World Coordinate origin (applied 1st)
- Rotates **uvn** to align with **xyz** (applied 2nd)

$$\mathbf{V} = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

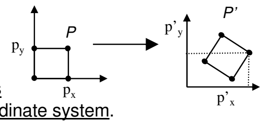
$$= \begin{pmatrix} u_x & u_y & u_z & -\mathbf{eye}\cdot\mathbf{u} \\ v_x & v_y & v_z & -\mathbf{eye}\cdot\mathbf{v} \\ n_x & n_y & n_z & -\mathbf{eye}\cdot\mathbf{n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This is what ***gluLookAt*** computes. Note that is a transformation applied to all *scene component vertices.*

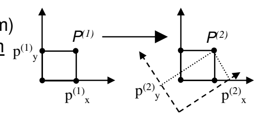
View Transformation Matrix (cont'd)

- Mathematical derivation of view matrix, $\mathbf{V}$ **(Hill, pp. 364-366)**
  - **Eye position** along with **vectors u, v, n** define the **UVN Coordinate System** (Eye/camera coords) *relative to* World Coord. System (WCS). **u**, **v**, and **n** are the **basis vectors** of coordinate system **UVN**.
  - Scene object vertices are defined *relative to* **WCS**. Therefore, $\mathbf{V}$ must be a transformation that converts them to be *relative to* **ECS**, i.e., $\mathbf{V} = \mathbf{M}_{W\text{->}E}$
  - 2 ways to think about a geometric transformation:

Transform old points to new points in same coordinate system.



Change (transform) coordinate system keeping same points.



  - **Camera analogy:** motion in each case is **inverse** of the other
    1. Real world, real visual effect: objects stationary, move camera
    2. Virtual world, rendering pipeline requirements: camera stationary, move objects
- **OpenGL demo program** – 2 ways to visualize view transformation

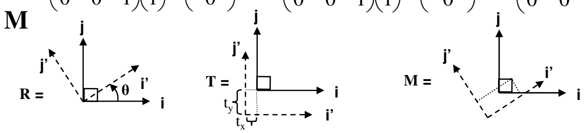
View Transformation Matrix (cont'd)

- **Review:** Hill text Ch. 5.4 "Changing Coordinate Systems", pg.244
  If **i, j** are basis vectors of System 1, and **i', j'** are basis vectors of System 2 (expressed relative to System 1), then **transformation matrix M**, with column vectors **i', j'** and the vector, **t**, the translation of the origins, is the matrix that transforms the coordinate frame (basis vectors) of System 1 into those of System 2. I.e., columns of $\mathbf{V}$ are the transformed System 1 coordinate frame basis vectors.

Example: $\mathbf{M} = \mathbf{T}\,\mathbf{R} = \begin{pmatrix} i_x' & j_x' & t_x \\ i_y' & j_y' & t_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} i_x' & j_x' & 0 \\ i_y' & j_y' & 0 \\ 0 & 0 & 1 \end{pmatrix}$ $\quad \mathbf{R} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Verify: Transform endpoints of vectors **i, j**, and origin point by $\mathbf{M}$

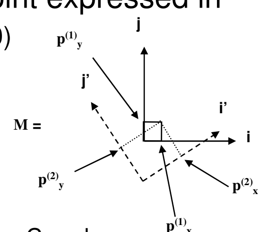
$$\begin{pmatrix} i_x' & j_x' & t_x \\ i_y' & j_y' & t_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i_x'+t_x \\ i_y'+t_y \\ 0 \end{pmatrix} \quad \begin{pmatrix} i_x' & j_x' & t_x \\ i_y' & j_y' & t_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} j_x'+t_x \\ j_y'+t_y \\ 0 \end{pmatrix} \quad \begin{pmatrix} i_x' & j_x' & t_x \\ i_y' & j_y' & t_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \\ 1 \end{pmatrix}$$



View Transformation Matrix (cont'd)

- By Hill's "theorem" (5.35) on page 245, if *P* is a point in System 2 and $\mathbf{M}$ is matrix that transforms System 1's coordinate frame to System 2's, then $\mathbf{M}\,P$ = coordinates of the point expressed in System 1. (see also, Burkhard's notes, 5.8 slide #40)

  So, if we know $P^{(2)} = (p^{(2)}_x, p^{(2)}_y, 1)$ in System 2, i.e., relative to the basis vectors **i', j'**, then we can solve for $P^{(1)} = (p^{(1)}_x, p^{(1)}_y, 1)$ in System 1, i.e., relative to the basis vectors **i, j** using, $P^{(1)} = \mathbf{M}\,P^{(2)}$



- But, for a **viewing transformation**:
  - System 1 corresponds to World Coords., System 2 to Eye Coords. (because the Eye Coordinate Frame. and 3D scene object vertices are expressed relative to World Coords).
  - Matrix $\mathbf{M}$ with basis vectors defined by the `gluLookAt()` parameters is the matrix that transforms from System 2's coordinate frame to System 1's (Eye to World).
  - However, we need a solution for the inverse: from System 1 (World) to System 2 (Eye) coordinates, $P^{(2)} = \mathbf{M}^{-1} P^{(1)}$, **not** $P^{(1)} = \mathbf{M}\,P^{(2)}$.

View Transformation Matrix (cont'd)

- Given $\mathbf{M} = \mathbf{T}\,\mathbf{R}$, how to find $\mathbf{M}^{-1} = (\mathbf{T}\,\mathbf{R})^{-1}$?
- Review: some axioms of linear algebra
  - $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$ (Hill Appendix 2, pg. 825)
  - an **orthogonal matrix** is a matrix such that: $col_i \cdot col_j = 0$, $i \neq j$, and $col_i \cdot col_i = 1$ (Hill Ch. 5, pg. 243)
  - a **rotation matrix** is **orthogonal** (Hill Ch. 5, pg. 243)
  - $\mathbf{M}^{-1} = \mathbf{M}^{T}$ (**transpose**) if $\mathbf{M}$ is **orthogonal** (Hill Ch. 5, pg. 243) (also, Burkhard's notes, 5.8, slide #44)
  - The **inverse** of a **translation matrix** is a matrix with the translation column terms **negated**. (by definition of translation)
- Thus, **View Transformation Matrix** $\mathbf{V}$, that transforms points from World Coordinates to Eye Coordinates is

$$\mathbf{V} = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{V} = \mathbf{M}^{-1} = (\mathbf{T}\,\mathbf{R})^{-1} = \mathbf{R}^{-1}\,\mathbf{T}^{-1} = \mathbf{R}^{T}\,\mathbf{T}^{-1} = \begin{pmatrix} u_x & u_y & u_z & -\mathbf{eye}\cdot\mathbf{u} \\ v_x & v_y & v_z & -\mathbf{eye}\cdot\mathbf{v} \\ n_x & n_y & n_z & -\mathbf{eye}\cdot\mathbf{n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

View Transformation Matrix Summary

- The rendering pipeline processing draws geometric primitives as seen from the World Coordinate origin looking down the –Z axis.
- The **View Transformation Matrix** $\mathbf{V}$, transforms points from World Coordinates to Eye Coordinates. The result is that primitives appear as they would if the eye were at the origin looking down the –Z axis.
- If a view's position and orientation are specified by a translation matrix and a rotation matrix, $\mathbf{M} = \mathbf{T}\,\mathbf{R}$, then the view transformation matrix $\mathbf{V}$ is the **inverse** of the matrix $\mathbf{M}$:

$$\mathbf{V} = \mathbf{M}^{-1} = (\mathbf{T}\,\mathbf{R})^{-1} = \mathbf{R}^{-1}\,\mathbf{T}^{-1} = \mathbf{R}^{T}\,\mathbf{T}^{-1}$$

- $\mathbf{M}$ is the same as an **instance transformation** (modeling transformation) without any scale transformation. So, if we specify a view as part of our model, we can determine the corresponding view transformation from the **inverse of the view's modeling transformation**!
- We now have a solution to **problems** # 1 and 2 (slide #21).
  Do you think you now know the answer to problems #3, #4, and #5?

Alternative View Transform Specifications

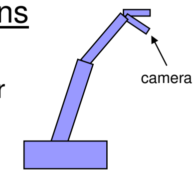
- View specified as a general **instance transformation**
  - Calls to `glRotatef()` for Euler angle rotations and to `glTranslatef()` for a translation to orient and position the camera (but, no scale).
  - Transformation matrix $\mathbf{M}$ that transforms System 1's coordinate frame (World Coord.) to System 2's frame (Eye Coord.) is:

    $$\mathbf{M} = \mathbf{T}\,\mathbf{R}_x\,\mathbf{R}_y\,\mathbf{R}_z$$

    Matrix $\mathbf{V}$, that transforms points from World to Eye Coordinates is

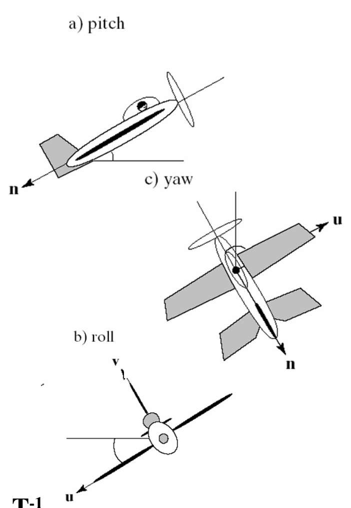
    $$\mathbf{V} = \mathbf{M}^{-1} = (\mathbf{T}\,\mathbf{R}_x\,\mathbf{R}_y\,\mathbf{R}_z)^{-1} = \mathbf{R}_z^{-1}\,\mathbf{R}_y^{-1}\,\mathbf{R}_x^{-1}\,\mathbf{T}^{-1}$$
  - **Note:** t_x, t_y, t_z are in World Coords., NOT relative to camera orientation.
- Specified as a **hierarchy** of instance transformations
  Example: camera on the gripper of a robot arm
  - Arm hierarchy, joints: base, lower arm, upper arm, gripper
  - Instance transformation of gripper

    $$\mathbf{M} = \mathbf{T}_B\,\mathbf{R}_{By}\,\mathbf{T}_{LA}\,\mathbf{R}_{LAx}\,\mathbf{R}_{LAy}\,\mathbf{T}_{UA}\,\mathbf{R}_{UAx}\,\mathbf{R}_{UAy}\,\mathbf{T}_G\,\mathbf{R}_{Gx}\,\mathbf{R}_{Gy}\,\mathbf{R}_{Gz}$$
  - View transformation for camera attached to gripper, $\mathbf{V} = \mathbf{M}^{-1}$



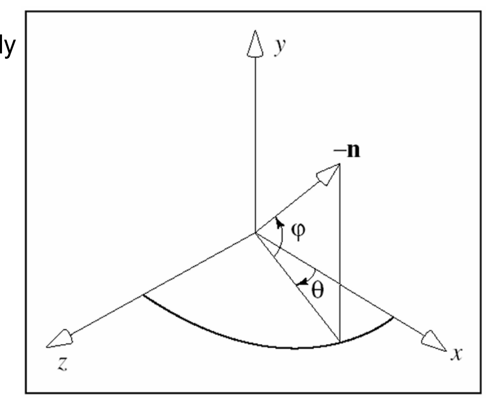
View Transformations for Aerospace

- View specified as **pitch, yaw, roll**
  - Euler angle specification, normally applied:
    $\mathbf{R}_{roll}\,\mathbf{R}_{yaw}\,\mathbf{R}_{pitch}$
  - *pitch* = angle **n** axis makes with plane $\mathbf{Y} = 0$ (horizontal)
    same as rotation about **u** axis
  - *yaw* = angle **u** axis makes with plane $\mathbf{Z} = 0$
    same as rotation about **v** axis
    (also known as *heading* or *bearing*)
  - roll = angle **u** axis makes with plane $\mathbf{X} = 0$
    same as rotation about **n** axis
  - Graphics applications often use a "no-roll" camera – pitch and yaw only
  - $\mathbf{M} = \mathbf{T}\,\mathbf{R}_{roll}\,\mathbf{R}_{yaw}\,\mathbf{R}_{pitch}$, $\mathbf{V} = \mathbf{M}^{-1}$
    $\mathbf{V} = (\mathbf{T}\,\mathbf{R}_{roll}\,\mathbf{R}_{yaw}\,\mathbf{R}_{pitch})^{-1} = \mathbf{R}^{-1}_{pitch}\,\mathbf{R}^{-1}_{yaw}\,\mathbf{R}^{-1}_{roll}\,\mathbf{T}^{-1}$



View Transformations Aerospace (cont'd)

- View specified as **azimuth, elevation** (tilt, optional but uncommon)
  - Euler angle specification, normally applied:
    $\mathbf{R}_{elevation}\,\mathbf{R}_{azimuth}$
  - *azimuth* = angle **u** axis makes with the plane $\mathbf{Z} = 0$
    same as rotation about **v** axis, same as yaw
  - *elevation* = angle **n** axis makes with the plane $\mathbf{Y} = 0$ (horizontal)
  - $\mathbf{M} = \mathbf{T}\,\mathbf{R}_{elevation}\,\mathbf{R}_{azimuth}$
    $\mathbf{V} = \mathbf{M}^{-1}$
    $= (\mathbf{T}\,\mathbf{R}_{elevation}\,\mathbf{R}_{azimuth})^{-1}$
    $= \mathbf{R}^{-1}_{azimuth}\,\mathbf{R}^{-1}_{elevation}\,\mathbf{T}^{-1}$



View Transformations Aerospace (cont'd)

- **Question:** Are these transformations really correct?
  - if $\mathbf{M} = \mathbf{T}\,\mathbf{R}_{roll}\,\mathbf{R}_{yaw}\,\mathbf{R}_{pitch}$
    then $\mathbf{V} = (\mathbf{T}\,\mathbf{R}_{roll}\,\mathbf{R}_{yaw}\,\mathbf{R}_{pitch})^{-1} = \mathbf{R}^{-1}_{pitch}\,\mathbf{R}^{-1}_{yaw}\,\mathbf{R}^{-1}_{roll}\,\mathbf{T}^{-1}$?
    if $\mathbf{M} = \mathbf{T}\,\mathbf{R}_{elevation}\,\mathbf{R}_{azimuth}$
    then $\mathbf{V} = (\mathbf{T}\,\mathbf{R}_{elevation}\,\mathbf{R}_{azimuth})^{-1} = \mathbf{R}^{-1}_{azimuth}\,\mathbf{R}^{-1}_{elevation}\,\mathbf{T}^{-1}$?
  - **M** = camera's instance transformation = position and orientation in World Coords. Translation by $(t_x, t_y, t_z)^T$ will be applied after the rotations. Result: first rotates camera about its origin and then **translates in World Coordinates! [OpenGL demo program]**
  - But – want to translate relative to look direction, i.e., in Eye Coordinates.
  - Examples:
    - For the default view orientation: "forward" = translate (0, 0, -dt), and "pitch up" = rotate dAngle about X axis, (1, 0, 0).
    - But, if camera has been rotated 90 degrees left and rolled 45 degrees, then "forward" = translate (-dt, 0, 0), and "pitch up" = rotate ??? about (?, ?, ?) axis.

View Transformations Aerospace (cont'd)

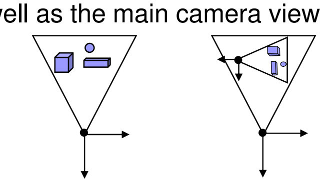
- **Problem:** How to "fly" a view using motion relative to view direction
  - Need to convert *changes* in position and orientation that are specified *relative to* current view orientation into changes *relative to* World Coords.
  - **"slide" function**: translation (+/-) for back/forward, right/left, and up/down relative to current orientation. Equivalent to motion along the **n, u,** and **v** axes in the camera's **UVN** coordinate system.
    - Given: displacement vector $\mathbf{d_2} = (d_u, d_v, d_n)$ in **UVN** Coord. System (System 2)
      Find: displacement vector $\mathbf{d_1} = (d_x, d_y, d_z)$ in **XYZ** Coord. System (System 1)
    - $$\mathbf{M} = \begin{pmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{pmatrix}$$
      matrix that transforms System 1 basis vectors into System 2 basis vectors. Then, by Hill's theorem (slide #24-25), $\mathbf{d_1} = \mathbf{M}\,\mathbf{d_2}$ (also, Burkhard's notes, 5.8 slide #40)
  - For slide() and roll() functions C++ code, see Hill text pg. 368

ModelView Transformation Summary

- **Summary:** Rendering pipeline ModelView transformation
  - Vertex coordinate points automatically transformed by ModelView matrix
    $P' = \mathbf{M}_{ModelView}\,P = (\mathbf{V}\,\mathbf{M})\,P$
    where $\mathbf{V}$ = `gluLookAt` matrix or $\mathbf{V} = \mathbf{R}{v_z}^{-1}\,\mathbf{R}{v_y}^{-1}\,\mathbf{R}{v_x}^{-1}\,\mathbf{T}v^{-1}$
    and $\mathbf{M} = \mathbf{T}\,\mathbf{R}_x\,\mathbf{R}_y\,\mathbf{R}_z\,\mathbf{S}$ or combination of several **T R S** matrices.
  - Points are transformed so that they are *relative to* the Eye Coordinate system with eye point at the origin. Thus, vertices that fall within the eye's viewing volume have a **negative Z coordinate** value (in RHS coordinate system).
- **OpenGL demo program**
  - transforming eye/camera relative to the World versus transforming the objects in the World relative to the eye/camera
  - flying the camera view

Questions about View Transformations

1. Why does *gluLookAt()* fail if you try looking vertically down and set *up = (0,1,0)*?
2. Write your own version of *gluLookAt.*
3. *gluLookAt* takes 9 float parameters. What would be the *minimum* number to specify the camera position and orientation?
4. What significance, if any, does the eyePoint have when specifying an orthographic view?
5. In the demo program shown in lecture you saw views in additional viewports that showed side, front, and top views as well as the main camera view of the 3D scene. Thus, the demo showed a "view" and a "view of a view". Write a program that shows two views in two viewports, a main view and another view, either a side, a front, or a top view.



Main view (shown from the top) — Side view of main view (shown from the top)

Questions View Transformations (cont'd)

6. Example from NASA Ames Mars Virtual Planetary Exploration project. The project's graphics system renders a polygon mesh model of the surface of Mars. A space exploration scientist wears a virtual reality head mounted display (HMD). The rendered view is displayed on 2 small LCD screens inside the HMD. Attached to the HMD is a 6 degree-of-freedom (DOF) magnetic tracker that measures the helmet's (x,y,z) position and (pitch, yaw, roll) orientation 30 times/sec. These are measured relative to the tracking device's fixed coordinate system in the laboratory (same as World Coords.). The scientist has a 5 DOF no-roll joystick (left/right, up/down, forward/back plus 2 twist rotations for pitch and yaw). This controls the position and orientation of a virtual hover craft on which the virtual explorer is sitting on virtual Mars. Translations and rotations of the joystick are measured relative to joystick's fixed coordinate system (same as World Coords.).

   Write the function that sets the hover craft transformations and the view transformation for this system. Joystick transformations applied to the hover craft should be relative to the craft's current orientation. Camera transformations applied to the view of the scene should be relative to the scientist's current head rotation and the hover craft rotation.

Perspective Projection

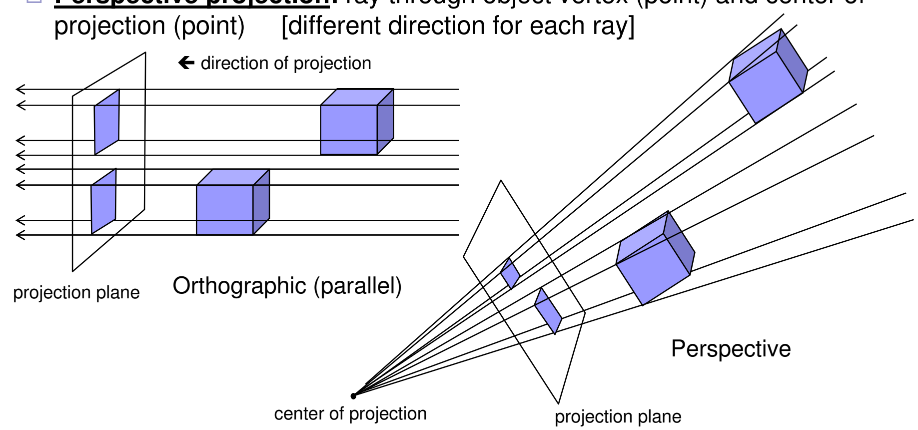
- **Problems to be solved:**
  - How can a graphics system simulate real world perspective depth (distant objects rendered smaller than near objects)?
  - How can a graphics system convert 3D objects to 2D perspective corrected objects?
  - Can this be done with a transformation matrix that can be applied in the rendering pipeline (with hardware)?

Principles of Geometric Projections

- **Projection:** a mapping of coordinate values from a higher dimension to lower dimension, usually N ⇨ N-1, e.g., 3D ⇨ 2D.
- **Requirements:**
  - **Projection surface:** plane or hyperplane (linear projection) or surface such as a sphere or conic section (non-linear projection).
  - **Projection rays**, or ***projectors***: lines from object projected towards projection surface.
  - **Direction of projection:** orientation of each projector
    - Perspective projection: all projectors pass through a center of projection (3D point), but have different directions.
    - Orthographic (parallel) projection: all projectors parallel to a common direction of projection.
- **How to project:**
  - Projection ray through object vertex intersects with the projection plane.

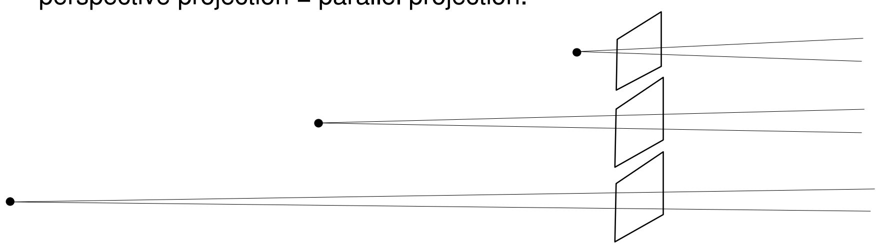
Principles Geometric Projections (cont'd)

- **Parallel projection:** ray through object vertex (point) in the projection direction (vector) [same direction for all rays].
- **Perspective projection:** ray through object vertex (point) and center of projection (point) [different direction for each ray]



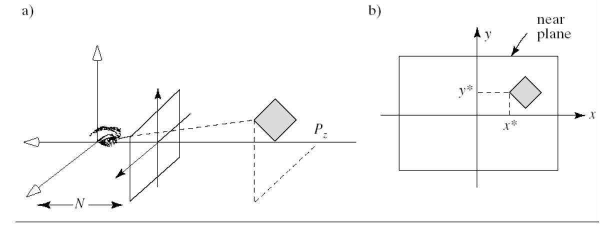
Principles Geometric Projections (cont'd)

- **Observation about perspective projection:** as center of projection moves farther and farther away, lines of projection become more nearly parallel. In the limit, when center of projection is at an infinite distance, perspective projection ≡ parallel projection.

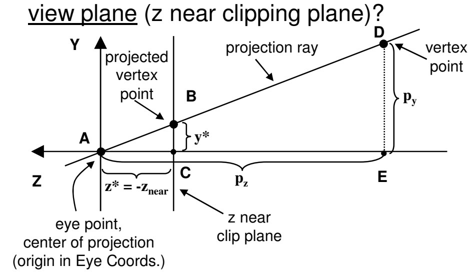


- Rays of light from a point source shining on an opaque object forming a shadow on a projection plane are similar to perspective projection rays.
- Rays of light from a point source at "infinite distance" (e.g., the Sun 93×10^6 miles from the Earth) forming a shadow are similar to parallel projection.

Perspective Projection of a 3D Point



- What are the coordinates (x^*, y^*, z^*) of the point $P = (p_x, p_y, p_z)$ when projected onto the view plane (z near clipping plane)?



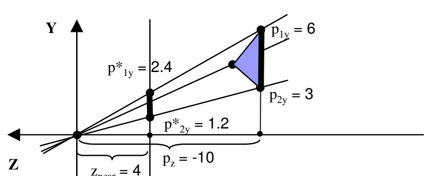
Similar triangles: ABC and ADE
Ratios of similar sides are equal.

$y^* / z^* = p_y / p_z$

Therefore:	$y^* = z^* (p_y / p_z)$
Substituting	$-N = -z_{near}$ for z^*
get	$y^* = (-z_{near}/p_z)\, p_y$
and	$x^* = (-z_{near}/p_z)\, p_x$

Perspective Projection of 3D Point (cont'd)

- Observe:
  - Perspective projection is just a scaling of a point's x and y coordinate by the factor $s_{persp} = (-z_{near}/p_z)$, e.g., $x^* = s_{persp}\, p_x$, $y^* = s_{persp}\, p_y$
  - For all points that are farther away than z_{near}, $-p_z >= z_{near}$. Thus, $s_{persp} = (-z_{near}/p_z) <= 1.0$ and the **larger** the magnitude of p_z (point's distance from the eye) the **smaller** the perspective scale factor: "perspective *foreshortening*".



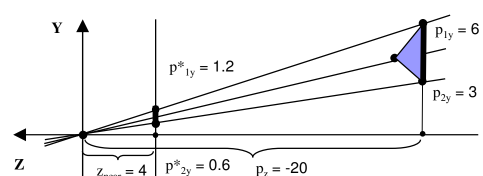
s_{persp} **= 0.4**

$d_y^* = (p^*_{1y} - p^*_{2y})$
$= s_{persp}\, d_y$
$= (\mathbf{0.4})\, 3 = \mathbf{1.2}$

$d_y = (p_{1y} - p_{2y}) = \mathbf{3}$

Perspective Projection of 3D Point (cont'd)

- Observe:
  - Perspective projection is just a scaling of a point's x and y coordinate by the factor $s_{persp} = (-z_{near}/p_z)$, e.g., $x^* = s_{persp}\, p_x$, $y^* = s_{persp}\, p_y$
  - For all points that are farther away than z_{near}, $-p_z >= z_{near}$. Thus, $s_{persp} = (-z_{near}/p_z) <= 1.0$ and the **larger** the magnitude of p_z (point's distance from the eye) the **smaller** the perspective scale factor: "perspective *foreshortening*".



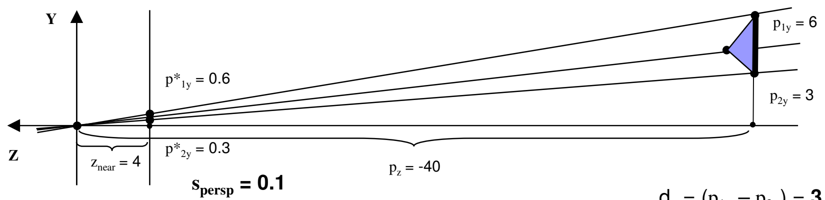
s_{persp} **= 0.2**

$d_y^* = (p^*_{1y} - p^*_{2y})$
$= s_{persp}\, d_y$
$= (\mathbf{0.2})\, 3 = \mathbf{0.6}$

$d_y = (p_{1y} - p_{2y}) = \mathbf{3}$

Perspective Projection of 3D Point (cont'd)

- Observe:
  - Perspective projection is just a scaling of a point's x and y coordinate by the factor $s_{persp} = (-z_{near}/p_z)$, e.g., $x^* = s_{persp}\, p_x$, $y^* = s_{persp}\, p_y$
  - For all points that are farther away than z_{near}, $-p_z >= z_{near}$. Thus, $s_{persp} = (-z_{near}/p_z) <= 1.0$ and the **larger** the magnitude of p_z (point's distance from the eye) the **smaller** the perspective scale factor: "perspective *foreshortening*".



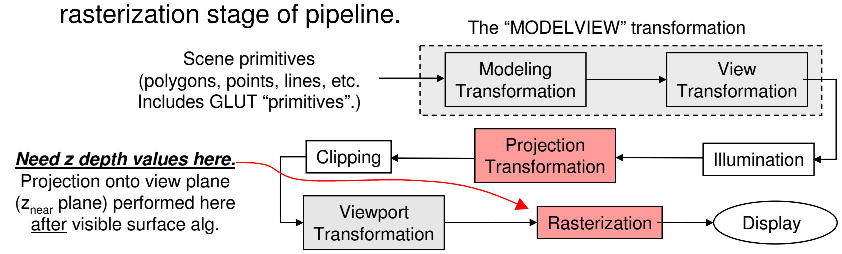
$d_y^* = (p^*_{1y} - p^*_{2y})$
$= s_{persp}\, d_y$
$= (\mathbf{0.1})\ 3 = \mathbf{0.3}$

$d_y = (p_{1y} - p_{2y}) = \mathbf{3}$

OpenGL demo program

Perspective Projection

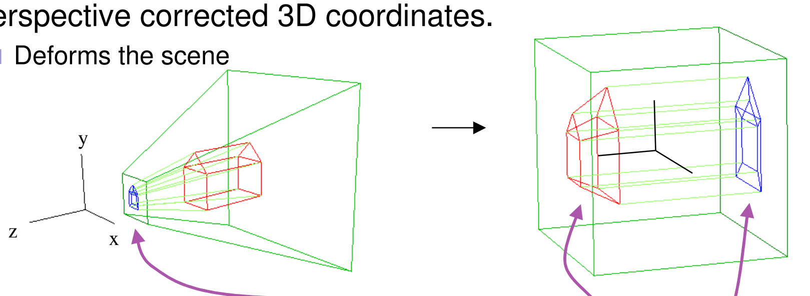
- Perspective projection CANNOT be used in 3D graphics pipeline!
  - Why not? Because it sets all projected z coordinates to same value, z_{near}. But, visible surface algorithm (Z buffer alg.) needs z depth values during rasterization stage of pipeline.



  - Therefore, pipeline uses **perspective *transformation***, not **perspective *projection***. Perspective transformation scales x, y, **and z** coordinates by a scale factor dependent upon 1/z. Then, projection is performed during rasterization stage after hidden surface removal.

Perspective Transformation and Projection

- Perspective **transformation**: converts 3D coordinates to perspective corrected 3D coordinates.
  - Deforms the scene



  - We want perspective projection to look like this
  - But, we actually perform it in a 2 step process:
    - Perspective **transformation**: 3D → 3D
    - Orthographic **projection**: 3D → 2D

Perspective Transformation (cont'd)

- Perspective transformation requirements:
  1. x and y values must be scaled by same factor as derived in perspective projection equations.
  2. z values must maintain depth ordering (monotonic increasing)
  3. z values must map: $-\mathbf{z_{near}}$ → -1 and $-\mathbf{z_{far}}$ → +1, view volume → NDC cube.
- In other words, we need a transformation that given a point *P* results in a transformed point *P'* such that P'_x and P'_y meet requirement 1 and $f(p_z)$ meets requirements 2 and 3.

$$P' = \left(\frac{-z_{near}\, p_x}{p_z},\ \frac{-z_{near}\, p_y}{p_z},\ f(p_z) \right)$$

- **Question:** Is there any matrix, $\mathbf{P}$, such that $\mathbf{P}\ P = P'$?
- **Answer: Not possible** because no linear combination of p_x, p_y, p_z, can result in a term with p_z in the denominator!

$$\begin{pmatrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{-z_{near}\, p_x}{p_z} \\ \frac{-z_{near}\, p_y}{p_z} \\ f(p_z) \\ 1 \end{pmatrix}$$

Perspective Transformation (cont'd)

- But, there is a matrix that can produce this result,

$$\begin{pmatrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} -z_{near}\, p_x \\ -z_{near}\, p_y \\ f(p_z)\, p_z \\ p_z \end{pmatrix}$$

- Then, after conversion from homogeneous to ordinary coordinates (division by w coordinate), we get result we need, $P' = \left(\frac{-z_{near}\, p_x}{p_z}, \frac{-z_{near}\, p_y}{p_z}, \frac{-f(p_z)}{p_z} \right)$
- This is the **homogeneous coordinate matrix** that performs perspective transformation

$$\mathbf{P} = \begin{pmatrix} z_{near} & 0 & 0 & 0 \\ 0 & z_{near} & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

with $a = -\frac{z_{far} + z_{near}}{z_{far} - z_{near}}, \quad b = \frac{-2\, z_{far} * z_{near}}{z_{far} - z_{near}}$

then $P' = \left(\frac{-z_{near} p_x}{p_z}, \frac{-z_{near} p_y}{p_z}, \frac{-(a\, p_z + b)}{p_z} \right)$

- So, **perspective transformation** can be applied via matrix multiplication in rendering pipeline (using hardware!)

Perspective Transformation (cont'd)

- Verify that the matrix meets our requirements:

$$\mathbf{P} = \begin{pmatrix} z_{near} & 0 & 0 & 0 \\ 0 & z_{near} & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad P = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad P_{znear} = \begin{pmatrix} x \\ y \\ -z_{near} \\ 1 \end{pmatrix} \quad P_{zfar} = \begin{pmatrix} x \\ y \\ -z_{far} \\ 1 \end{pmatrix}$$

- After multiplying by matrix **P**, we get these strange results!

$$P^* = \mathbf{P}\,P = \begin{pmatrix} z_{near}\, x \\ z_{near}\, y \\ a\,z + b \\ -z \end{pmatrix} \quad P^*_{near} = \mathbf{P}\,P_{znear} = \begin{pmatrix} z_{near} x \\ z_{near} y \\ -a\, z_{near} + b \\ z_{near} \end{pmatrix} \quad P^*_{far} = \mathbf{P}\,P_{zfar} = \begin{pmatrix} z_{near} x \\ z_{near}\, y \\ -a\, z_{far} + b \\ z_{far} \end{pmatrix}$$

- What to do about the homogeneous coordinates, w ≠ 1.0 ??? Up till now we've ignored w term in $P = (x, y, z, w)^T$ when w = 1.0.

Perspective Transformation (cont'd)

- Refer back to Burkhard's notes on 2D homogeneous coordinates. To convert a homogeneous coordinate point, P_{homog} = (a, b, c), to an "ordinary" point, P_{ord} = (x, y), use (a, b, c) → (a/c, b/c).
  - Use same conversion for 3D homogeneous points: P_{homog} = (x, y, z, w) → P_{ord} = (x/w, y/w, z/w). Also called perspective division.
  - Thus, for these transformed points,

$$P^* = \mathbf{P}\,P = \begin{pmatrix} z_{near}\, x \\ z_{near}\, y \\ a\,z + b \\ -z \end{pmatrix} \quad P^*_{near} = \mathbf{P}\,P_{znear} = \begin{pmatrix} z_{near} x \\ z_{near} y \\ -a\, z_{near} + b \\ z_{near} \end{pmatrix} \quad P^*_{far} = \mathbf{P}\,P_{zfar} = \begin{pmatrix} z_{near}\, x \\ z_{near}\, y \\ -a\, z_{far} + b \\ z_{far} \end{pmatrix}$$

  - Using, $a = -\frac{z_{far} + z_{near}}{z_{far} - z_{near}}, \quad b = \frac{-2\, z_{far} * z_{near}}{z_{far} - z_{near}}$

Ordinary form of the x and y components:

$z_{near}\, x / z = (-z_{near}/z)\, x$

$z_{near}\, y / z = (-z_{near}/z)\, y$

Ordinary form of the z components:

(a z + b) / (-z)

$(-a\, z_{near} + b) / z_{near} = -1.0$

$(-a\, z_{far} + b) / z_{far} = +1.0$

Check this out!

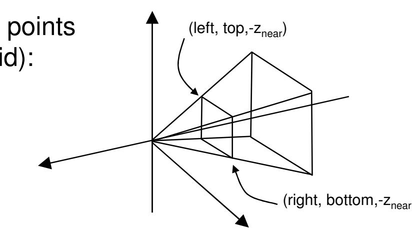
Perspective Transformation in OpenGL

- OpenGL perspective transformation: combined with view volume → NDC cube transformation. Thus, matrix =

$$\mathbf{P} = \begin{pmatrix} \frac{2\, z_{near}}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2\, z_{near}}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & \frac{-(z_{far} + z_{near})}{z_{far} - z_{near}} & \frac{-2\, z_{far}\, z_{near}}{z_{far} - z_{near}} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

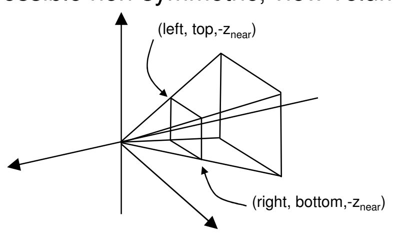
- View volume corners are specified by points on a ***frustum*** (a.k.a. truncated pyramid):

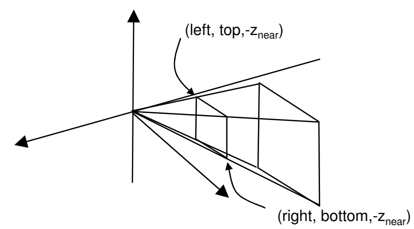
```
glFrustum(left, right,
          bottom, top,
          znear, zfar)
```



Perspective Transf. OpenGL (cont'd)

- `gluPerspective` computes these terms from its parameters (Hill, page 385):
  `top = zNear * tan((π/180)viewAngle/2);`
  `bottom = -top;`
  `right = top * aspect;   left = -right;`
- **Note:** with `gluPerspective` the view volume is always symmetric about the view direction vector. With `glFrustum` is is possible to specify an arbitrary, possible non-symmetric, view volume (useful for some stereo viewers).



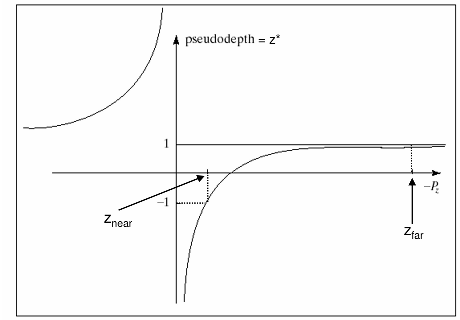


Perspective Transformation: pseudodepth

- Z coordinate transformation: "***pseudodepth***"
  - Transformed z* not linear function of z

$$z^* = \frac{az+b}{-z} = \frac{\left(-\frac{z_{far}+z_{near}}{z_{far}-z_{near}}\right)z + \frac{-2z_{far}*z_{near}}{z_{far}-z_{near}}}{-z}$$

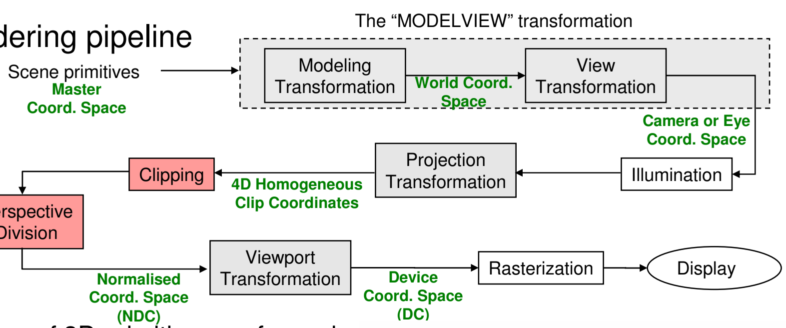
$$z^* = \frac{(z_{far}+z_{near})z + 2z_{far}*z_{near}}{(z_{far}-z_{near})z}$$



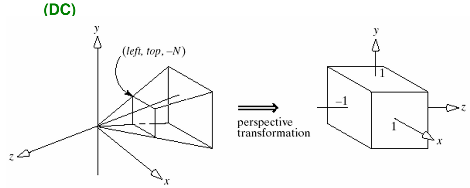
  - This is OK (sort of) because z* meets our 2 requirements:
    1. monotonic increasing, and
    2. z* = -1 and +1 for $z = z_{near}$ and $z = z_{far}$, respectively.
  - But, can cause z-buffer precision problems! (values usually 32 bit integers)
  - **WARNING:** avoid
    - very small z_{near} (**NEVER** use $z_{near} = 0$)
    - very large z_{far}

OpenGL Rendering Pipeline: Final Revision

- Rendering pipeline



- Clipping of 3D primitives performed in 4D clip space after view projection transformation (but before actual 3D ⇨ 2D projection). Transformed view volume is now a {-1, +1} cube (greatly simplifies clipping algorithm).
- CanonicalView Volume (CVV)



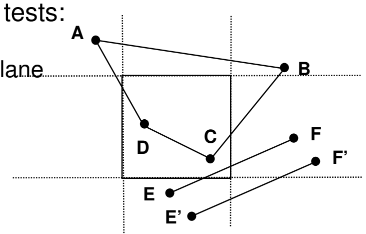
Canonical View Volume: Clipping

- But, there are **2 problems:**
  1. In some *strange cases* points that are **behind the eye** can have projected z values (pseudodepth) that are in front of the eye after perspective division! (because: for $P_{homog} = (x, y, z, w)^T \rightarrow P_{ord} = (x/w, y/w, z/w)^T$ z/w is the same result for negative and positive values, i.e., -z/w = z/-w and -z/-w = z/w)
  2. Division is a slow operation (even in hardware). Would be nice to clip away as many primitives as possible BEFORE performing perspective division on vertices.
- **Solution:** Clip in 4D Homogeneous Coordinate Space (whoa!)
- First, review how clipping to {-1, +1} NDC is performed in 3D
  1. Check if points lie on inside or outside of each of the 6 clipping planes
     - Example, test for point inside left plane: if $p_x > -1$, same as $\mathbf{(p_x+1) > 0}$. Other clip planes: $\mathbf{(p_x-1) > 0, (p_y+1) < 0, (p_y-1) > 0, (p_z+1) < 0, (p_z-1) > 0}$.
     - So, algorithm just adds or subtracts 1 and compares result to 0.
     - Very efficient and fast, especially fast in hardware!
  2. Assign result of boundary tests to **outcode** values for each end point of a line using one bit for each clip plane, left, right, bottom, top, near, far.

Canonical View Volume: Clipping (cont'd)

- Outcode examples (2D figure): **LRBTNF**
  Point A, outside the left and top boundaries = 1 0 0 1 0 0
  Point B, outside the right and top boundaries = 0 1 0 1 0 0
  Point C, inside all boundaries = 0 0 0 0 0 0
  Point D, inside all boundaries = 0 0 0 0 0 0
  Points E, E', outside bottom boundary = 0 0 1 0 0 0
  Points F, F', outside right boundary = 0 1 0 0 0 0

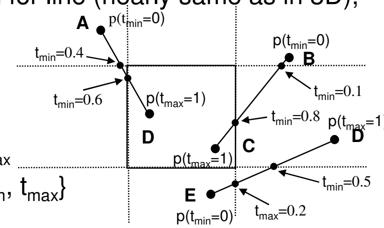
3. Perform **trivial accept** and **trivial reject** tests:
   - **trivial reject**
     = both endpoints outside some one clip plane
     = any 2 outcode bits both 1
     = (outcode A & outcode B) != 0
   - **trivial accept**
     = both endpoints inside all clip planes
     = all outcode bits 0
     = (outcode C | outcode D) == 0



4. For remaining endpoint pairs, must find intersection of line with clip planes to determine portion of line that is clipped.

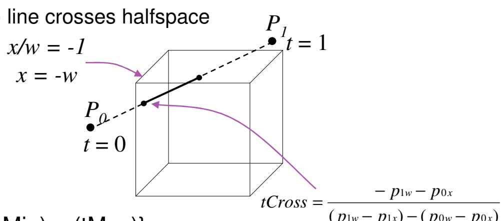
Canonical View Volume: Clipping (cont'd)

- Clipping to {-1, +1} NDC in 4D
  1. Check if points lie on inside or outside of each of the 6 clipping planes
     - test for point inside left plane: if $p_x/p_w > -1$ or $p_x > -p_w$ or $(\mathbf{p_w+p_x}) > 0$.
       Other planes: $(\mathbf{p_w-p_x}) > 0$, $(\mathbf{p_w+p_y}) < 0$, $(\mathbf{p_w-p_y}) > 0$, $(\mathbf{p_w+p_z}) < 0$, $(\mathbf{p_w-p_z}) > 0$.
  2. Assign result of tests to **outcode** values for each point (same as in 3D).
  3. Perform trivial reject and trivial accept tests (same as in 3D).
  4. Find clipped line by computing intersection point of line with each clipping plane using parametric equation for line (nearly same as in 3D), $p(t) = p_0 + t(p_1 - p_0)$, $0 <= t <= 1$
     1. Endpoint $p_0 = p(t_{min}=0)$: for each plane if it is outside the plane find t value at intersection, save largest t_{min}
     2. Endpoint $p_1 = p(t_{max}=1)$: save smallest t_{max}
     3. If $t_{max} > t_{min}$ reject, else clip line to $\{t_{min}, t_{max}\}$
     4. Compute $p_x(t)$, $p_y(t)$, $p_z(t)$, **and** $\mathbf{p_w(t)}$.
  5. Perform perspective division of clipped end points.



Canonical View Volume: Clipping (cont'd)

- Liang-Barsky and Cyrus-Beck clippers (Hill 7.4.4, simplified)
  for a single edge from point p_0 to p_1:
  Each edge is represented as: p(t) = p0 + t(p1 – p0)
  Compute outcodes; perform trivial reject and accept.
  If not rejected and not accepted:
  Initialize: [tMin, tMax] = [0,1]
  For each halfspace {x/w > – 1, x/w < +1, y/w > – 1, y/w < +1, z/w < +1, z/w > – 1}
  while tMin < tMax
  Compute **tCross** where (extended) line crosses halfspace
  if entering half-space
  tMin = max(tMin, tCross)
  else
  tMax = min(tMax, tCross)
  if tMin > tMax
  Edge is outside CVV
  else
  Compute new edge {p0, p1} = p(tMin), p(tMax)},
  p0x = p0x + tMin(p1x – p0x)
  p1x = p0x + tMax(p1x – p0x)
  and same for y, z, and w



Questions about View Projections

- For the case of $z_{near}=1$, $z_{far}=10$, plot a graph of pseudodepth versus *z*.
- Why isn't it a good idea to always use a very small number for z_{near} and a very large number for z_{far}?
- Work out what the matrix **P** should be for an orthographic camera.
- Assuming left = -right and bottom = -top (slide #52), work out formulae for the scaling factors required in matrix **P**. Compare with Hill's formulae.
- Why does OpenGL have two separate matrices (MODELVIEW and PROJECTION)? Why can't we just multiply the P matrix into the MODELVIEW matrix (as we do with the V matrix)?
- How would you compute the two view transformations for left eye and right eye stereo views for the HMD helmet of the NASA Ames VPE project?
- Would the projection transformation for each eye's view be the same or different? If different, how would they differ?