CMT107 Visual Computing

III.1 Transformations

Xianfang Sun

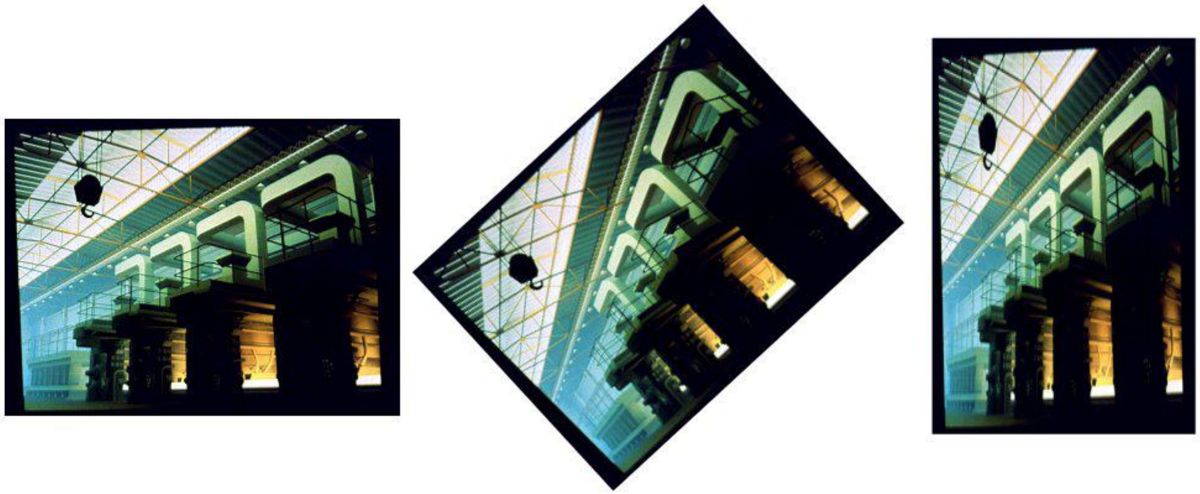
School of Computer Science & Informatics
Cardiff University

Overview

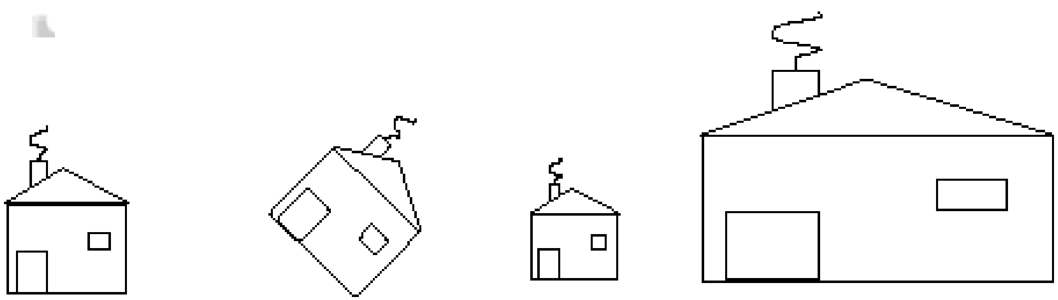
- Model transformations
  - 2D/3D linear transformations
  - 2D/3D affine transformations
- Homogeneous coordinates
  - Homogeneous affine transformations
- Coordinate transformations
  - Reference frames
  - Object vs. Frame Transformations
  - Camera Transformation
- OpenGL transformations

Model Transformations

- Transforming an object: transforming all its points



- Transforming a polygonal model: transforming its vertices



Basic 2D Transformations

- Scale:

$$x' = x \cdot s_x$$

$$y' = y \cdot s_y$$

(mirror: s_x and/or s_y = -1)

- Rotate:

$$x' = x \cdot \cos\phi - y \cdot \sin\phi$$

$$y' = x \cdot \sin\phi + y \cdot \cos\phi$$

- Shear:

$$x' = x + h_x \cdot y$$

$$y' = y + h_y \cdot x$$

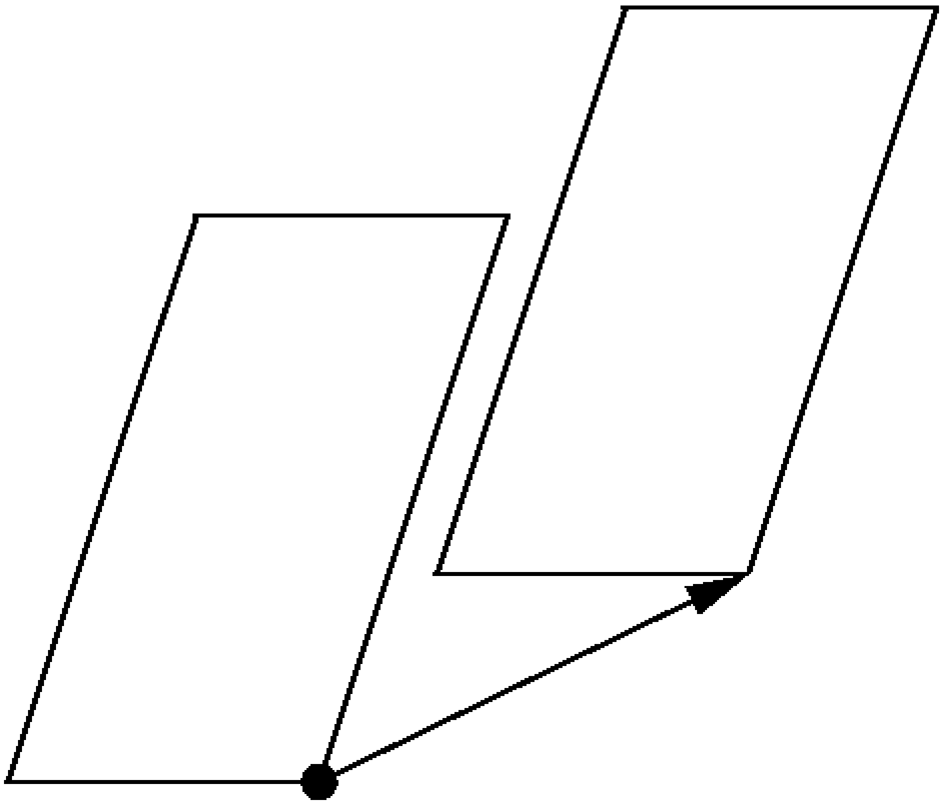
- Translate:

$$x' = x + t_x$$

$$y' = y + t_y$$

$$x''' = x'' + h_x y'' + t_x$$

$$y''' = y'' + h_y x'' + t_y$$



Matrix Representations

- Matrices are *convenient* to represent linear transformations:
  - Scale: $\begin{cases} x' = s_x \cdot x \\ y' = s_y \cdot y \end{cases}, \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
  - Rotate: $\begin{cases} x' = \cos\phi \cdot x - \sin\phi \cdot y \\ y' = \sin\phi \cdot x + \cos\phi \cdot y \end{cases}, \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
  - Shear: $\begin{cases} x' = x + h_x \cdot y \\ y' = y + h_y \cdot x \end{cases}, \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & h_x \\ h_y & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
  - In general: $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
- Efficient due to hardware matrix multiplication

Linear Transformations

- Linear transformations are combinations of
  - – scaling, mirroring, rotation, shearing
- Properties of linear transformations T:
  - Satisfies $\mathbf{T}(s_1\mathbf{v}_1 + s_2\mathbf{v}_2) = s_1\mathbf{T}(\mathbf{v}_1) + s_2\mathbf{T}(\mathbf{v}_2),\ s_1, s_2 \in R$
  - Origin maps to origin
  - Lines map to lines
  - Parallel lines remain parallel
  - Ratios are preserved
  - Closed under composition (The composition of two or more linear transformations is a linear transformation)

$$\mathbf{T}_0(\mathbf{T}_1(\mathbf{T}_2(\mathbf{v}))) = (\mathbf{T}_0 \circ \mathbf{T}_1 \circ \mathbf{T}_2)(\mathbf{v}) = \mathbf{T}(\mathbf{v})$$

- Translation is not linear transformation

Affine Transformations

- Affine transformations are combinations of
  - Linear transformations (matrices)
  - Translations (vectors)

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

  - General representation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

- Properties of affine transformations:
  - *Origin does not necessarily map to origin*
  - Lines map to lines
  - Parallel lines remain parallel
  - Ratios are preserved
  - Closed under composition

Homogeneous Coordinates

- *Homogeneous coordinates* in 2D
  - (x, y, w) represents a point at position $(x/w, y/w)$
  - $(x, y, 0)$ represents a point at infinity or direction
  - $(0, 0, 0)$ is not allowed
- We need a *3rd coordinate* for 2D points to represent translations solely with matrices
- 2D translation can be represented by a 3×3 matrix:

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ w \end{pmatrix}$$

Homogeneous 2D Transformations

- Basic 2D homogeneous transformation matrices
  - Scale: $$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ w \end{pmatrix}$$
  - Rotate: $$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ w \end{pmatrix}$$
  - Shear: $$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & h_x & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ w \end{pmatrix}$$
  - Translate: $$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ w \end{pmatrix}$$

3D Transformations

- *Same idea* as 2D transformations
  - Linear transformation: $\mathbf{p}' = \mathbf{T}\mathbf{p}$
  - Affine transformation: $\mathbf{p}' = \mathbf{T}\mathbf{p} + \mathbf{t}$
- Common 3D transformation matrices:

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}$$

Scale/mirror

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

Rotate around Z axis

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

Rotate around Y axis

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

Rotate around X axis

Homogeneous 3D Transformations

- Homogeneous coordinates in 3D:
  - (x, y, z, w) represents 3D position $(x/w, y/w, z/w)$
  - $(x, y, z, 0)$ represents a point at infinity or direction
  - $(0, 0, 0, 0)$ is not allowed
- Affine transformations represented by matrices

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Identity | Scale | Mirror over *x* axis

Homogeneous 3D Rotations

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

Rotate around z axis

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

Rotate around y axis

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

Rotate around x axis

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

Matrix Composition

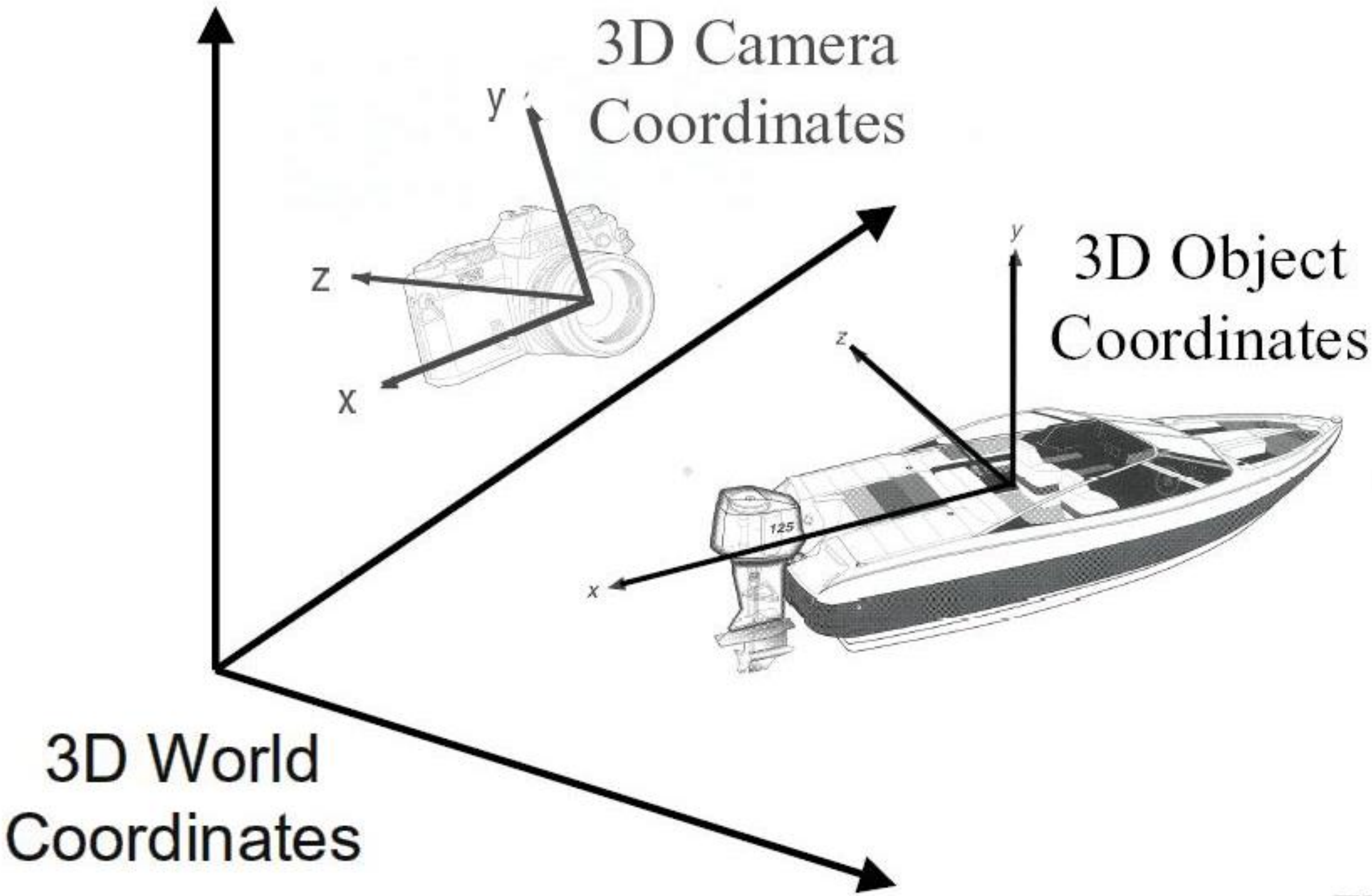
- Transformations can be *combined by matrix multiplication*
- Using homogeneous coordinates all affine transformations can be represented by matrices
  - Matrix multiplication is *associative*:

$$\mathbf{p}' = (\mathbf{T}_0 \cdot (\mathbf{T}_1 \cdot (\mathbf{T}_2(\mathbf{p})))) = ((\mathbf{T}_0 \cdot \mathbf{T}_1) \cdot \mathbf{T}_2)(\mathbf{p}) = (\mathbf{T}_0 \cdot \mathbf{T}_1 \cdot \mathbf{T}_2)(\mathbf{p})$$

  - Simple way to combine transformations
  - Only one matrix multiplication to transform vertices
- Beware: order of transformations matters
  - Matrix multiplication is *not commutative*:

$$(\mathbf{T}_1 \cdot \mathbf{T}_2)(\mathbf{p}) \neq (\mathbf{T}_2 \cdot \mathbf{T}_1)(\mathbf{p})$$

Reference Frames



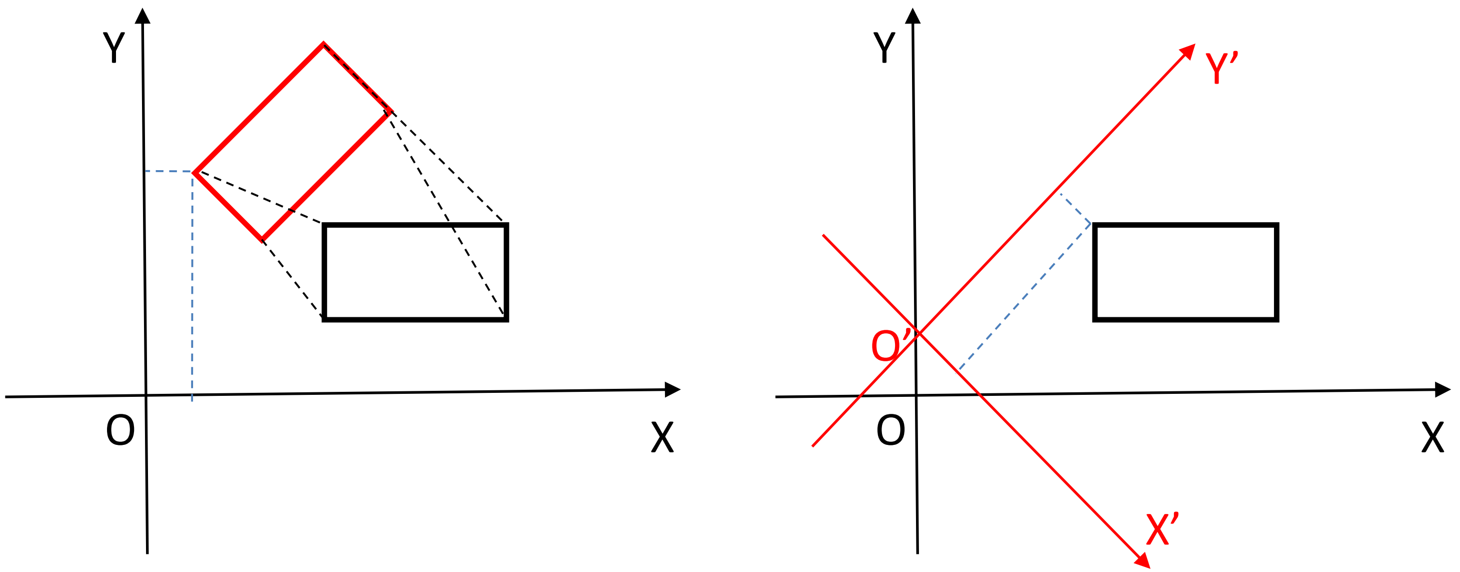
FVFHP Figure 6.1

Coordinate Transformations

- Scenes are defined in a *world-coordinate system*
- Objects in a scene are represented in a local *object coordinate system*
  - Transform local coordinates into other local coordinates
  - Ultimately transform local coordinates into world coordinates
- A camera is represented in a *camera coordinate system*
  - A scene is viewed by a camera from an arbitrary position and orientation
  - Transform world-coordinates into camera coordinates
- Transformation from object coordinate system to camera coordinate system can be represented by a signal matrix called model-view matrix in OpenGL.

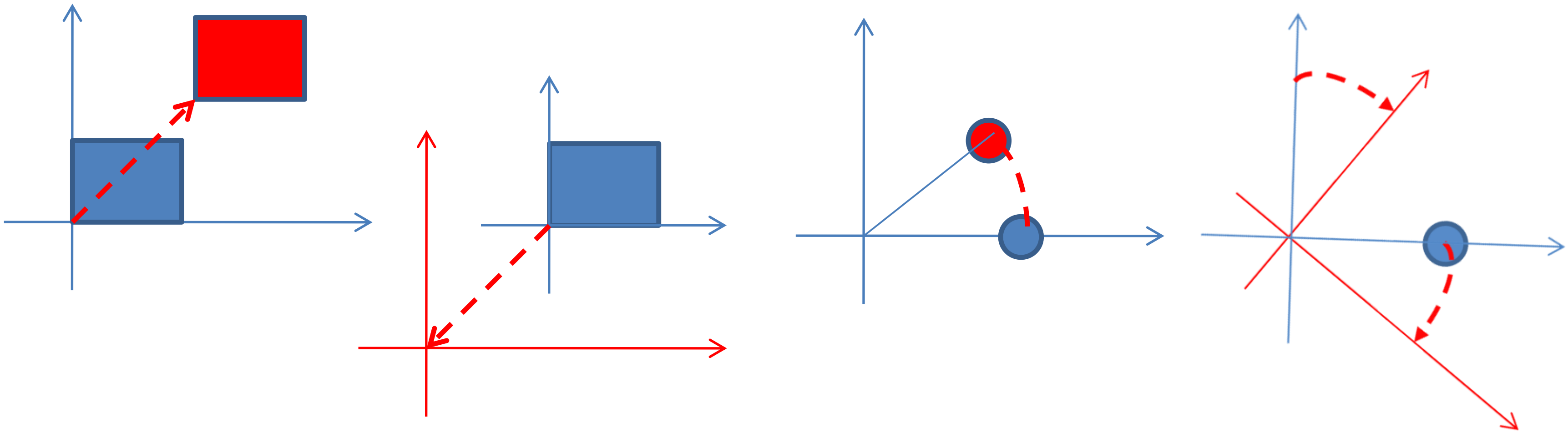
Object vs. Coordinate Transformations

- Object transformations transfer a object in a fixed coordinate system
- Coordinate transformations transform an object's coordinates from one coordinate system to another, while keep the object at its original position.
- The coordinates of a object transformation can be obtained equivalently by a coordinate transformation



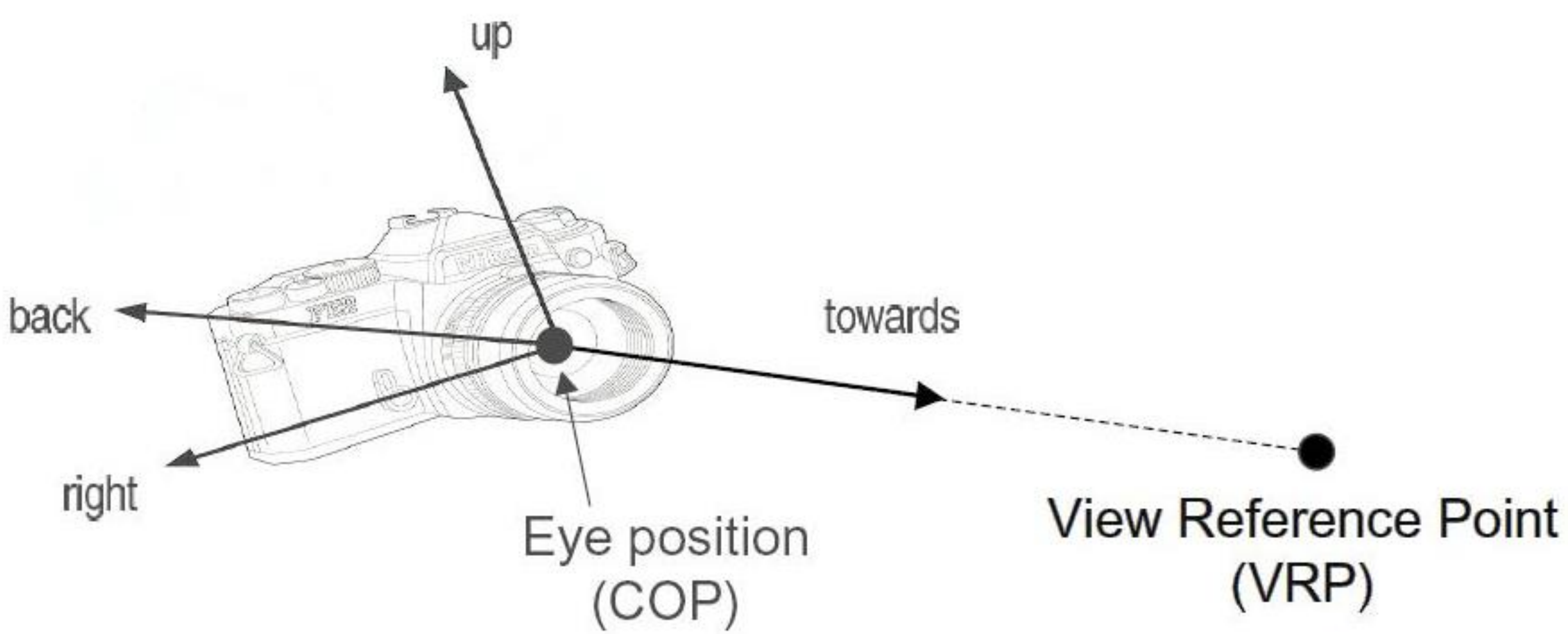
Object vs. Coordinate Transformations

- Translate an object by (t_x, t_y, t_z) is equivalent to translate the reference frame by $(-t_x, -t_y, -t_z)$
- Rotate an object around an axis by angle α is equivalent to rotate the reference frame around the same axis by angle $-\alpha$.
- Scale an object in a direction by value *s* is equivalent to scale the reference frame in the same direction by value 1/s.



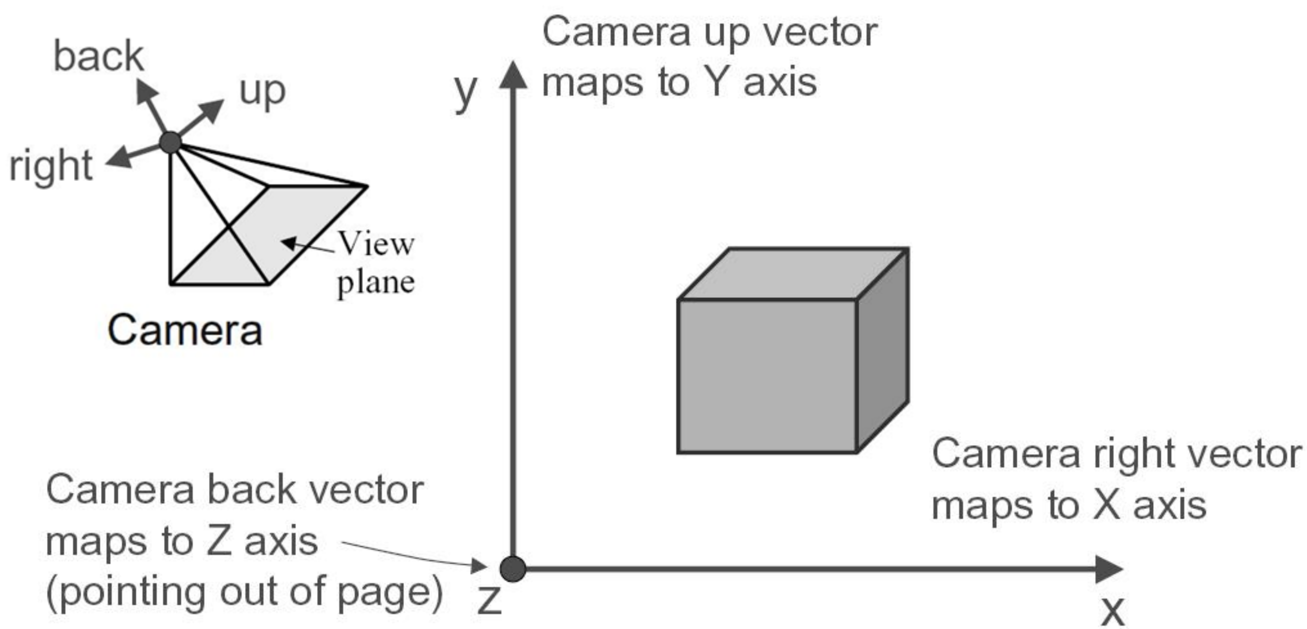
Camera Analogy

- Define a *synthetic camera* to determine view of a scene
- Camera parameters:
  - *Eye position* (x, y, z)
  - *View direction* (towards vector, up vector)
  - *Field of view* (xfov, yfov)



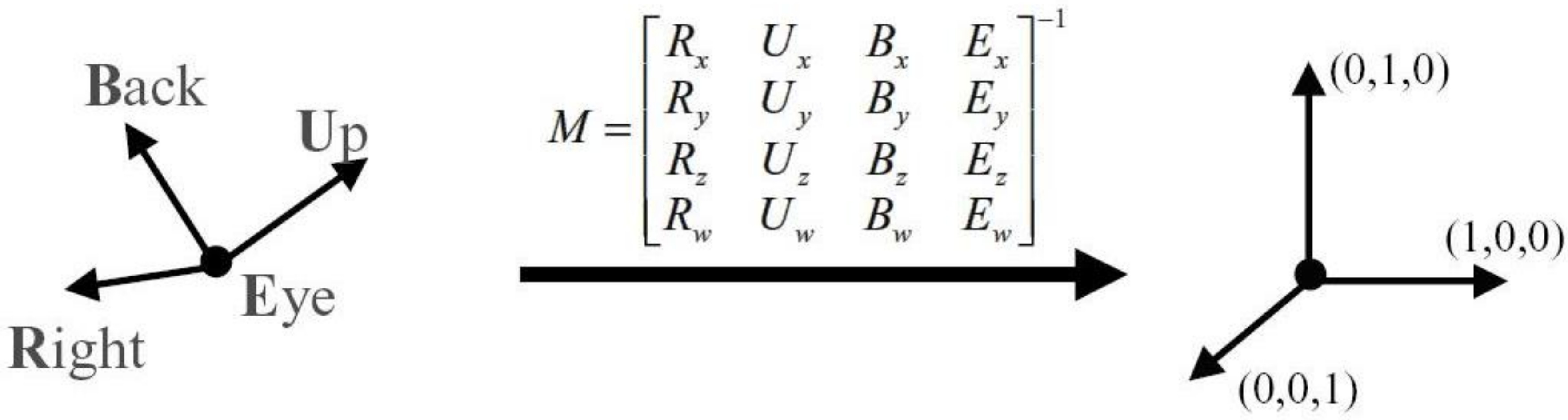
Camera Coordinates

- Mapping from world to camera coordinates (*normalisation*)
  - Origin moves to eye position
  - Up vector maps to Y axis, right vector maps to X axis
  - *Canonical* coordinate system for camera coordinates
  - Convention is *right-handed*
  - New versions of OpenGL adopts *left-handed* Frame



Camera Transformation

- Transformation matrix maps camera basis vectors to canonical vectors in camera coordinate system



world coordinates $(x_w, y_w, z_w, w_w)^t$ M→ camera coordinates $(x_c, y_c, z_c, w_c)^t$

Derivation of Camera Transformation

- Let the camera transformation matrix be **M**, then because **R**, **U**, **B**, and **E** are transformed to $[1\ 0\ 0\ 0]^T$, $[0\ 1\ 0\ 0]^T$, $[0\ 0\ 1\ 0]^T$, and $[0\ 0\ 0\ 1]^T$, respectively, we have

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = M \begin{pmatrix} R_x \\ R_y \\ R_z \\ R_w \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = M \begin{pmatrix} U_x \\ U_y \\ U_z \\ U_w \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = M \begin{pmatrix} B_x \\ B_y \\ B_z \\ B_w \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = M \begin{pmatrix} E_x \\ E_y \\ E_z \\ E_w \end{pmatrix}$$

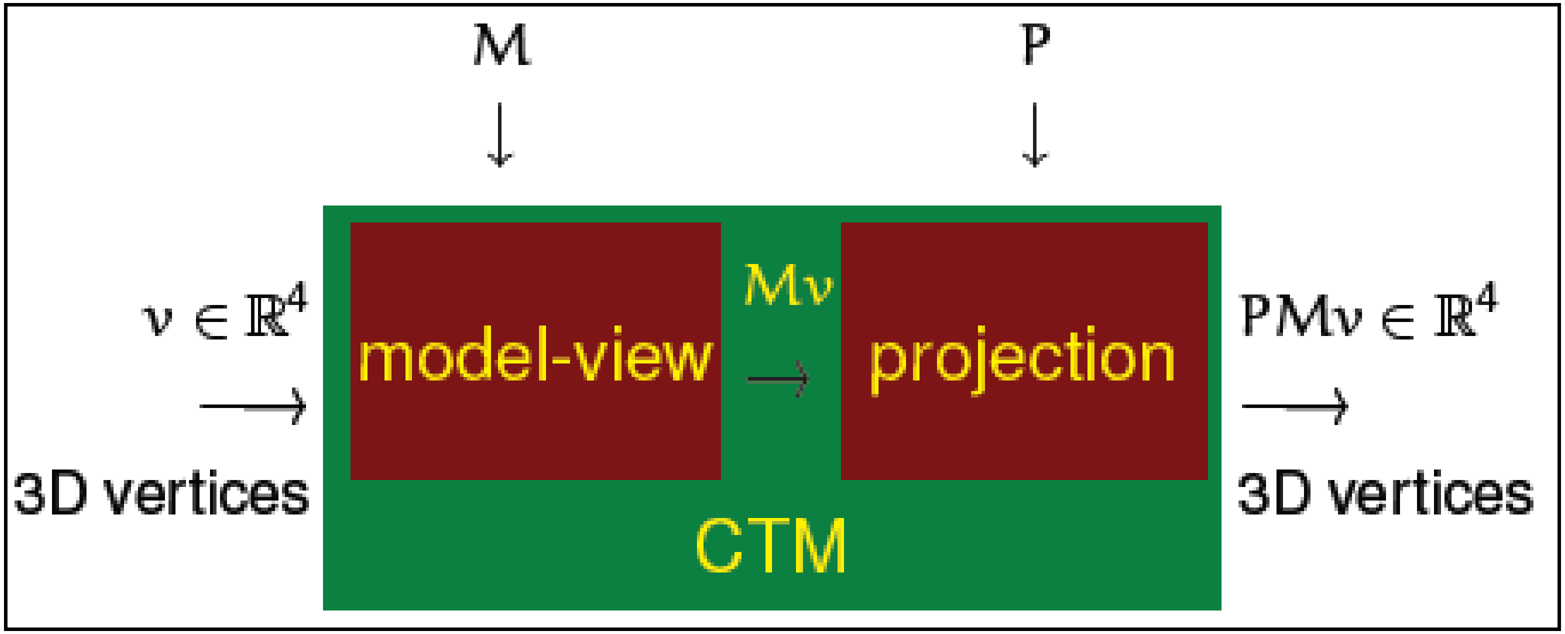
- Combine them together form the matrix equation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = M \begin{pmatrix} R_x & U_x & B_x & E_x \\ R_y & U_y & B_y & E_y \\ R_z & U_z & B_z & E_z \\ R_w & U_w & B_w & E_w \end{pmatrix}$$

- And thus matrix M is the inverse of the right matrix
- Note that **R**, **U**, and **B** represent direction, so $R_w = U_w = B_w = 0$
- **E** represents a point, so here $E_w = 1$

Current Transformation Matrix

- Conceptually two 4×4 matrices:
  - a *model-view* and a *projection* matrix in pipeline
  - Both matrices form the current transformation matrix (CTM)
  - All vertices are transformed by the CTM



OpenGL Transformations

- Early versions of OpenGL use some functions to represent transformations (matrix computations)
- Current OpenGL with shaders needs the programmers to write their own transformation code
- Maths libraries for matrix computations are available
  - vecmath from java package javax.vecmath
- An example simple matrix computation package is provided in the labs of this module
  - Vec3.java, Vec4.java, Mat4.java, Transform.java

Matrix Representation in OpenGL

- OpenGL uses 4x4 matrices to represent transformations
- A matrix is stored in a vector in the program
- Two orders to store a matrix in a vector
  - Row major (in row by row order)
  - column major (in column by column order)
- We use row major order in the package provided
- Shaders use column major order to represent matrices
- Post-multiplying with row-major matrices produces the same result as pre-multiplying with column-major matrices.

Transform Class

- In Transfrom.java, a class Transform is defined. T is the transformation matrix
- Constructor Transform(), or function initialize() will assign T as an identity matrix
- Functions scale(), translate(), rotateX(), rotateY(), rotateZ() perform as their names defined
- rotateA() performs rotation around an arbitrary axis
- reverseZ() is to convert right-hand frame to left-hand frame
- lookAt() is to locate the camera in the scene
  - Transform the model coordinates into camera frame
- ortho(), frustum(), and perspective() perform projection transformation (discuss later)

Function scale()

- Pre-multiply the current matrix T by a scaling transformation matrix
- For scale(sx, sy, sz), the scaling matrix is:

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Current matrix is modified as:

$$T' = ST = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{pmatrix}$$

$$= \begin{pmatrix} s_x M_{00} & s_x M_{01} & s_x M_{02} & s_x M_{03} \\ s_y M_{10} & s_y M_{11} & s_y M_{12} & s_y M_{13} \\ s_z M_{20} & s_z M_{21} & s_z M_{22} & s_z M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{pmatrix}$$

Function scale()

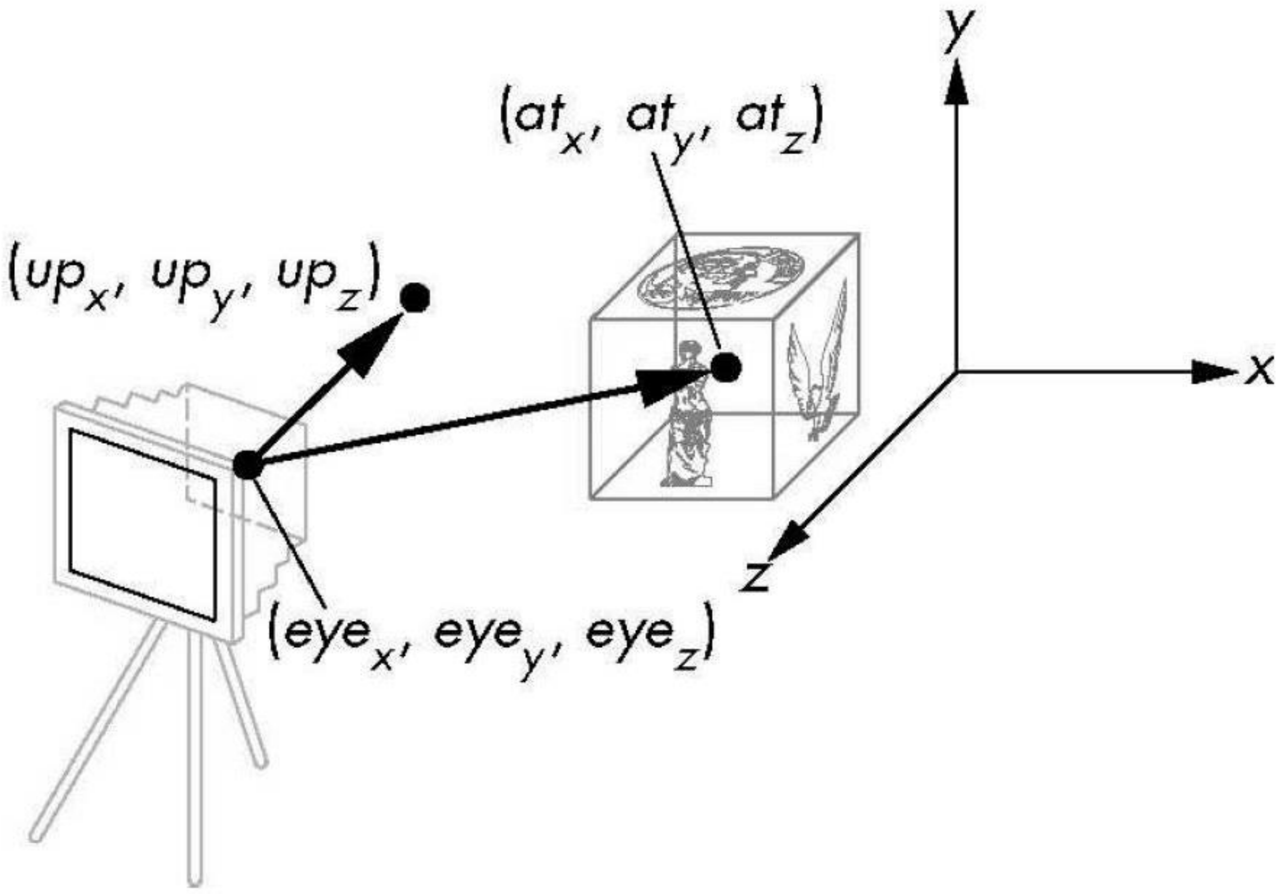
- Implementation of function scale(sx, sy, sz):

```
public void scale(float sx, float sy, float sz){
   for(int i=0;i<4;i++) {
          T.M[0][i] = T.M[0][i]*sx;
          T.M[1][i] = T.M[1][i]*sy;
          T.M[2][i] = T.M[2][i]*sz;
   }
}
```

lookAt()

Simulate gluLookAt() function in early versions of OpenGL

void lookAt(eye_x,eye_y,eye_z, at_x,at_y,at_z, up_x,up_y,up_z)



Angel: Interactive Computer Graphics 3E © Addison-Wesley 2002

Use Transform class

```
// Define a Transformation instance
// Transformation matrix is initialised as Identity;
Transform T = new Transform();

// In display(), load Identity matrix
T.initialize();

//Do transformations
T.scale(scale, scale, scale);
T.rotateX(rx);
T.rotateY(ry);
T.translate(tx, ty, 0);

//set up the camera
T.lookAt(0, 0, 0, 0, 0, -100, 0, 1, 0); //default parameters

// Send model_view matrix to shader. Here true for transpose
//means converting the row-major matrix to column major one
gl.glUniformMatrix4fv( ModelView, 1, true, T.getTransformv(), 0 );
```

Summary

- What is a reference frame? How can points in space be represented?
- What are linear and affine transformations?
- What are homogeneous coordinates? For what are they used?
- List some common/basic linear and affine 2D/3D transformations and their representation for Cartesian and homogeneous coordinates.
- What is object transformation and what is frame transformation? What's their relation?
- How can one build more complex affine transformations from the basic transformations?

CMT107 Visual Computing

III.2 Viewing

Xianfang Sun

School of Computer Science & Informatics
Cardiff University

Overview

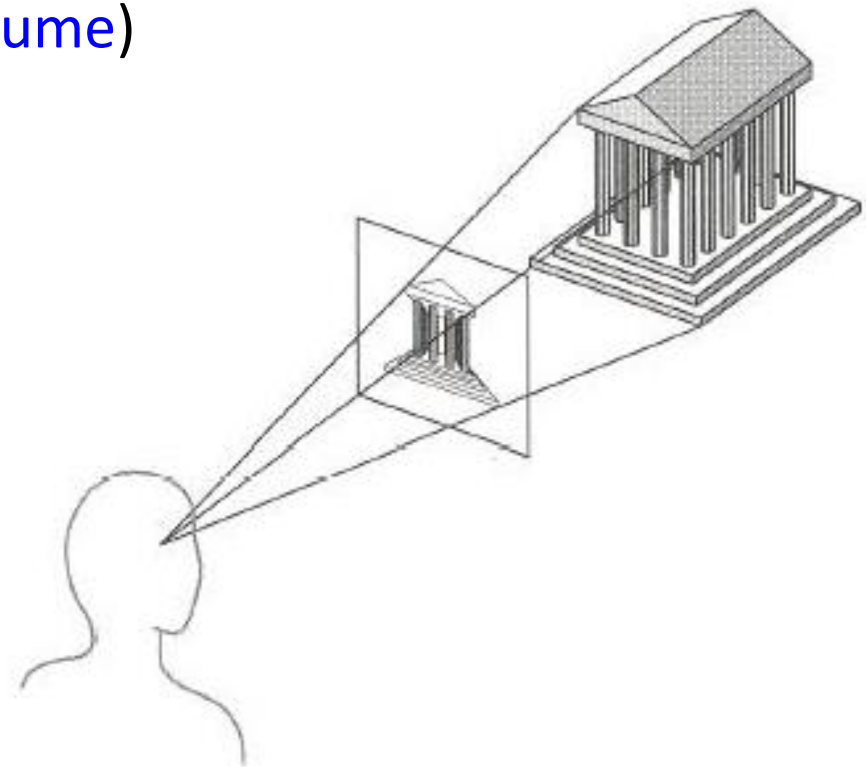
- Projection
  - Parallel projection
  - Perspective projection
- OpenGL viewing

Viewing Transformations

- *Viewing transformations*:
  - *Camera transformation*: 3D world coordinates to 3D camera coordinates
  - *Projection transformation*: Define a viewing volume, and transform 3D camera coordinates onto the view plane
  - *Viewport transformation*: The image on the view plane is translated and scaled to be fitted in the viewport on the screen

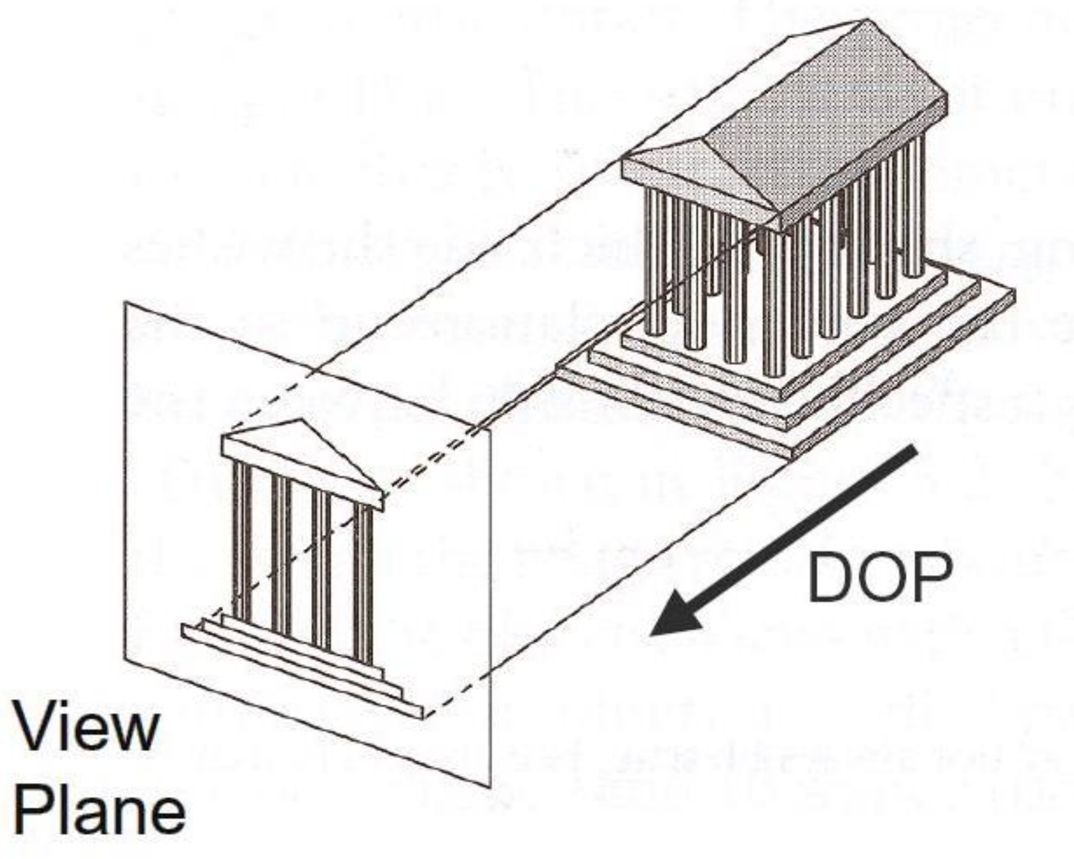
Projection

- General definition
  - Transform points in nD space to mD space, $n > m$
- In computer graphics:
  - Map 3D camera coordinates to 2D view plane coordinates
  - Also map depth to a specific range ([0, 1], related to viewing volume)



Parallel Projection

- Centre of projection is at *infinity*
- Direction of projection (DOP) is the *same* for all points



Parallel Projection Matrix

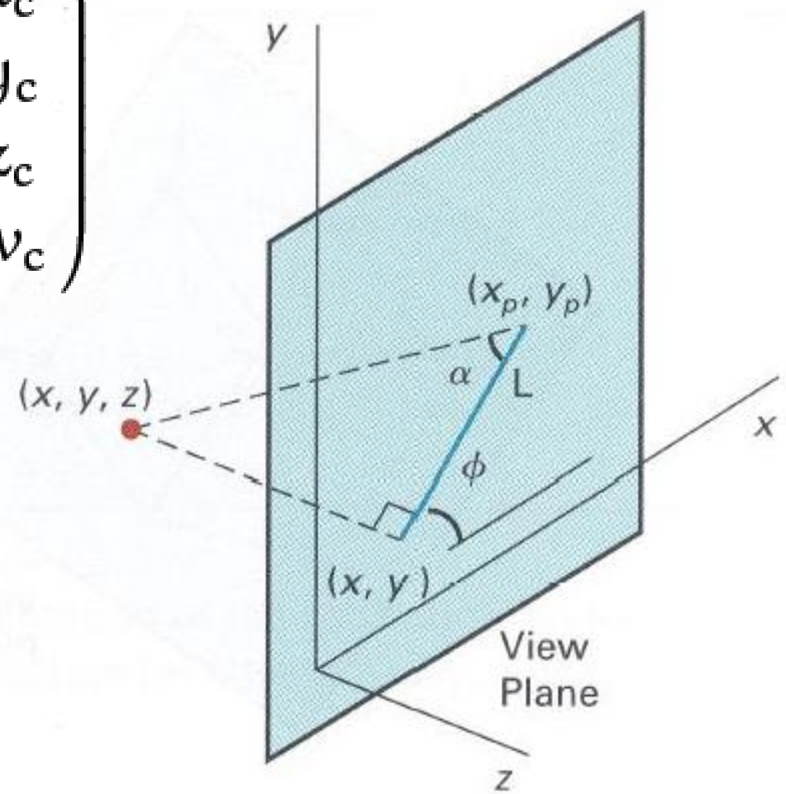
- General parallel projection transformation (defined by α, ϕ)
  - Orthogonal (orthographic) projection for $\alpha = 90°$

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ w_p \end{pmatrix} = \begin{pmatrix} 1 & 0 & -L_1 \cos\phi & 0 \\ 0 & 1 & -L_1 \sin\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$$

$$z_p = 0 \qquad \tan\alpha = \frac{-z_c}{L}$$

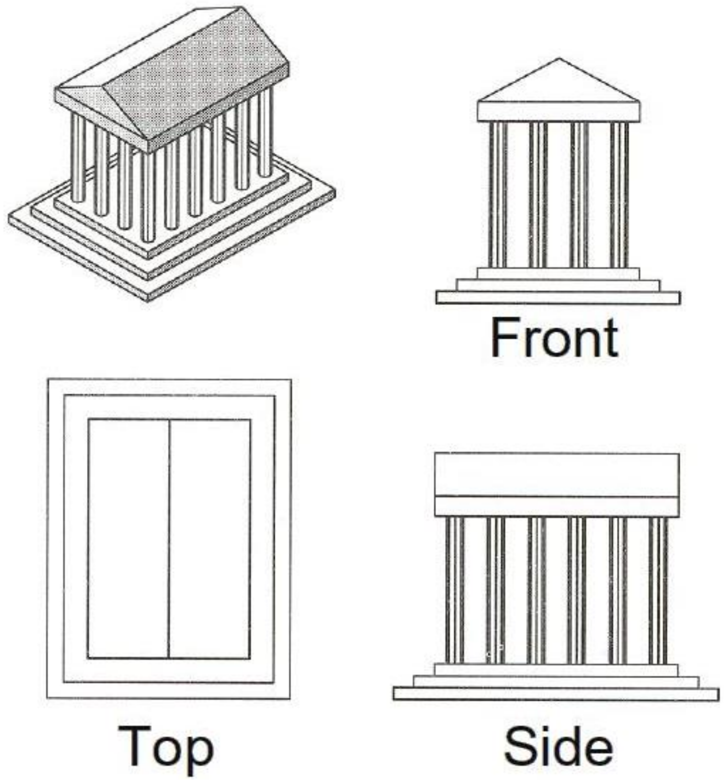
$$L_1 = \frac{1}{\tan\alpha} \quad (\text{for } \alpha \neq 90°)$$

$$L_1 = 0 \quad (\text{for } \alpha = 90°)$$



Orthographic Projection

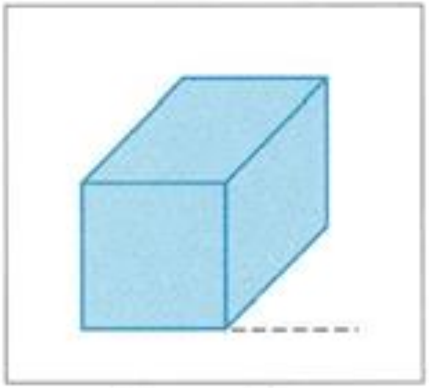
- Direction of projection *orthogonal* to view plane
  - Points with the same (x, y) coordinates will project at the same point on the view plane



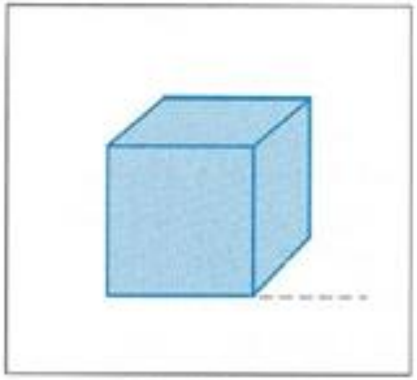
- Applications: for exact scaling the object like CAD etc

Oblique Projection

- Direction of projection *not orthogonal* to view plane
  - For cavalier projection ($\alpha = 45°$), two points with the same (x, y) coordinates will keep their distance on the view plane
  - For cabinet projection (α= 63.4°), two points with the same (x, y) coordinates will half their distance on the view plane
- Applications: for technical drawing and illustration like in furniture, or architecture, etc.



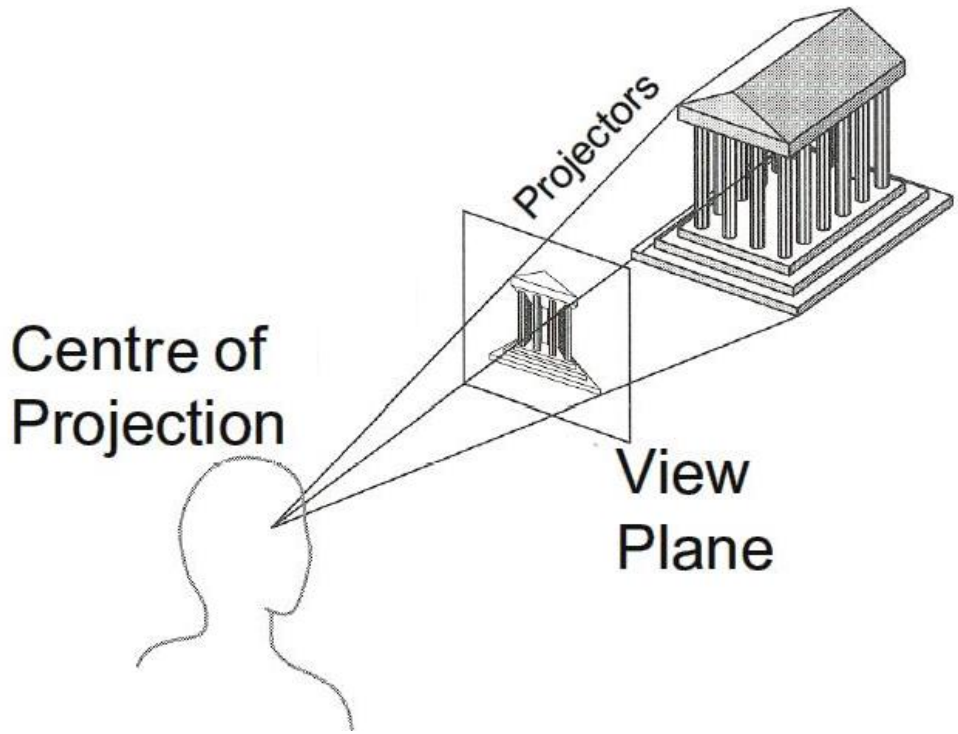
Cavalier
(DOP at 45°)



Cabinet
(DOP at 63.4°)

Perspective Projection

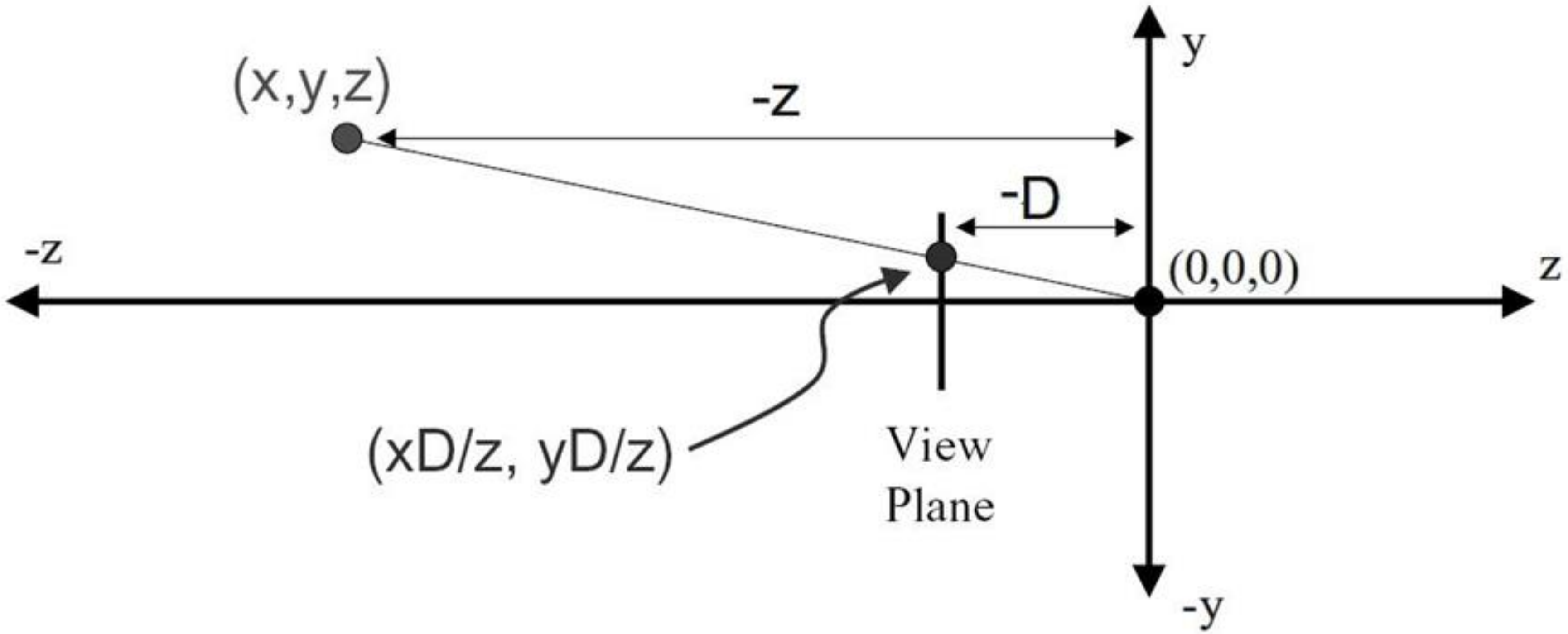
- Map points onto view plane along projectors emanating from centre of projection



- Application : for art drawings, especially for representing large scenes.

Perspective Projection

- Compute 2D coordinates from 3D coordinates using *similar triangles*



$$\frac{y_c}{z_c} = \frac{y_p}{D} \qquad \frac{x_c}{z_c} = \frac{x_p}{D} \qquad \text{for } D < 0$$

Perspective Projection Matrix

- 4×4 homogeneous coordinates matrix representation

$$\begin{aligned} x_p &= x_c D / z_c \\ y_p &= y_c D / z_c \\ z_p &= D \\ w_p &= 1 \end{aligned} \quad \rightarrow \quad \begin{aligned} x'_p &= x_c \\ y'_p &= y_c \\ z'_p &= z_c \\ w'_p &= z_c / D \end{aligned}$$

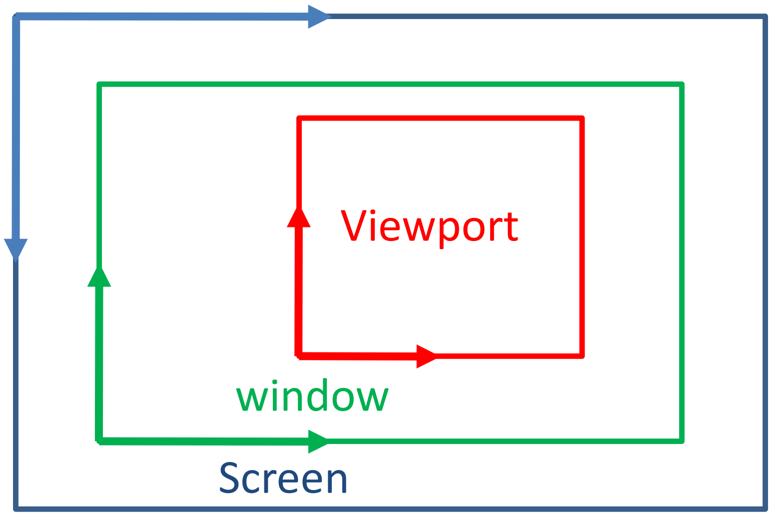
$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ w_p \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/D & 0 \end{pmatrix} \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$$

Perspective vs. Parallel Projection

- Perspective projection
  - Size varies inversely with distance – looks realistic
  - Distance and angles are not (in general) preserved
  - Parallel lines do not (in general) remain parallel
- Parallel projection
  - Good for exact measurements
  - Parallel lines remain parallel
  - Angles are not (in general) preserved
  - Less realistic looking

Viewport on Screen

- Coordinate systems on display:
  - Screen coordinate system: Origin at the upper-left corner of the screen, x direction from left to right, and y direction from top to bottom
  - Window coordinate system: Origin at the lower-left corner of the window, x direction from left to right, and y direction from bottom to top
  - Viewport: The rectangular region in the window where the image is drawn. Defined on window coordinate system by (x_0, y_0, w, h)



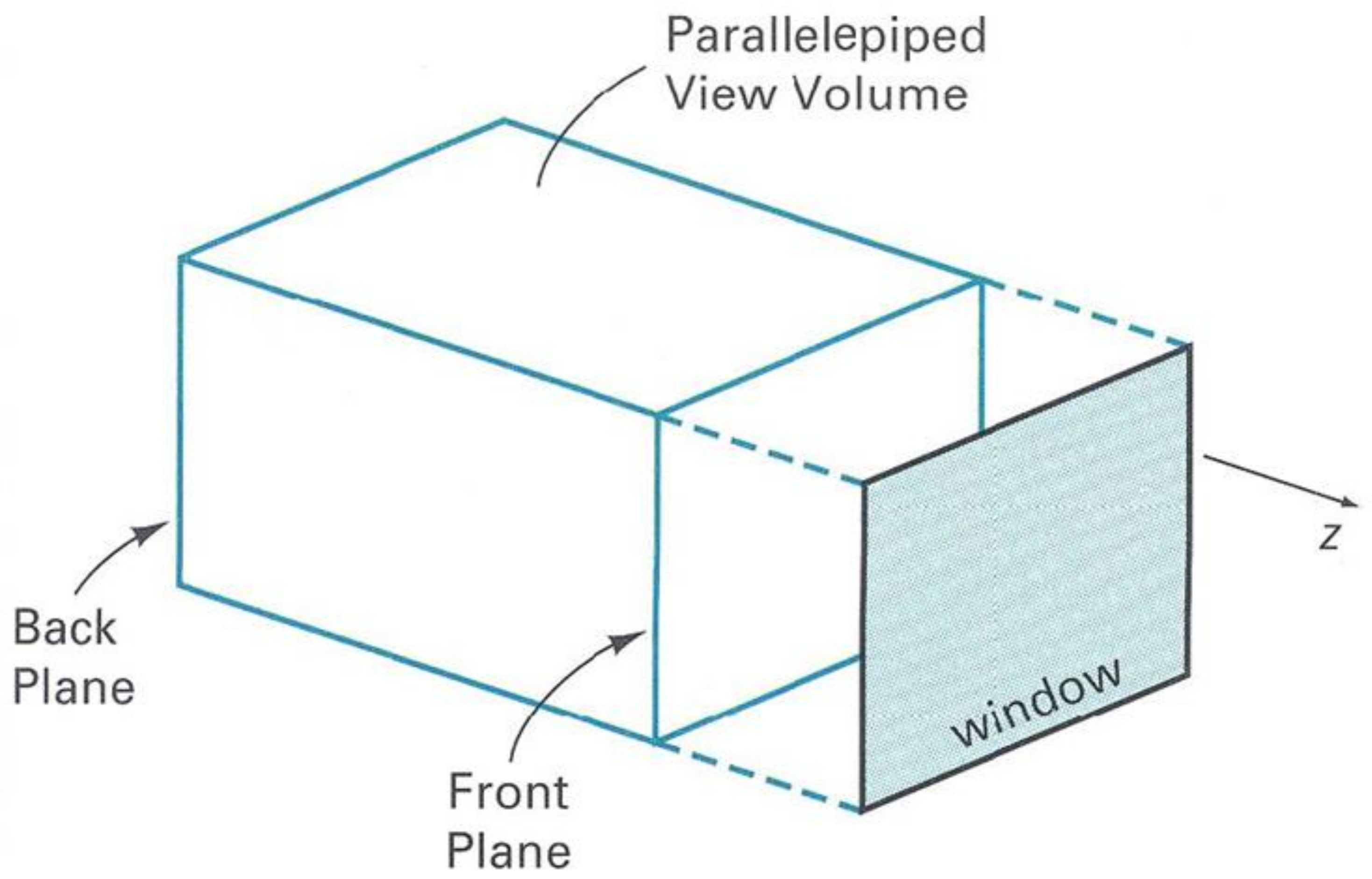
Viewport Transformation

- The whole image on the view plane are mapped on the whole viewport (by scaling and translating)
- To avoid distortion, the aspect ratio of the viewport should be equal to the aspect ratio of the viewing volume
  - aspect ratio: The ratio of the width to the height of a rectangle area (w/h)

OpenGL Projection

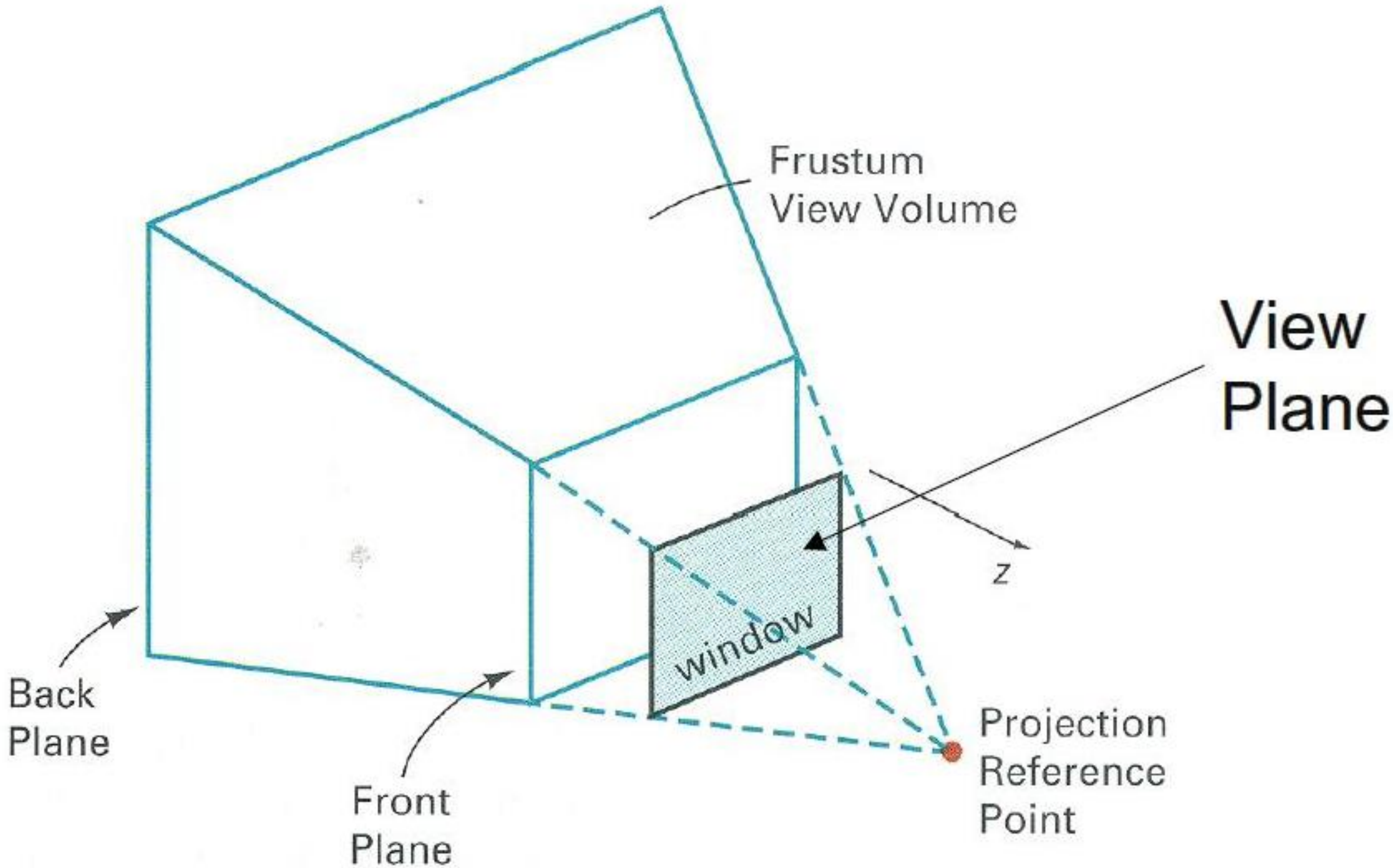
- Actual projection is set by projection matrix
- Projection matrix specifies parallel or perspective projection parameters
- Projection matrix is essentially defined by selecting a viewing volume (the region camera can see)
- Points inside the viewing volume are projected into a cube of edge length 2 (x, y, and z all range from -1 to 1)
  - Depths are maps of the z coordinate to the range [0, 1]
- Orthographic and perspective projections are implemented in class Transform, simulating the projection functions in OpenGL fixed-function pipeline

Parallel Projection Viewing Volume



H&B Figure 12.30

Perspective Projection Viewing Volume



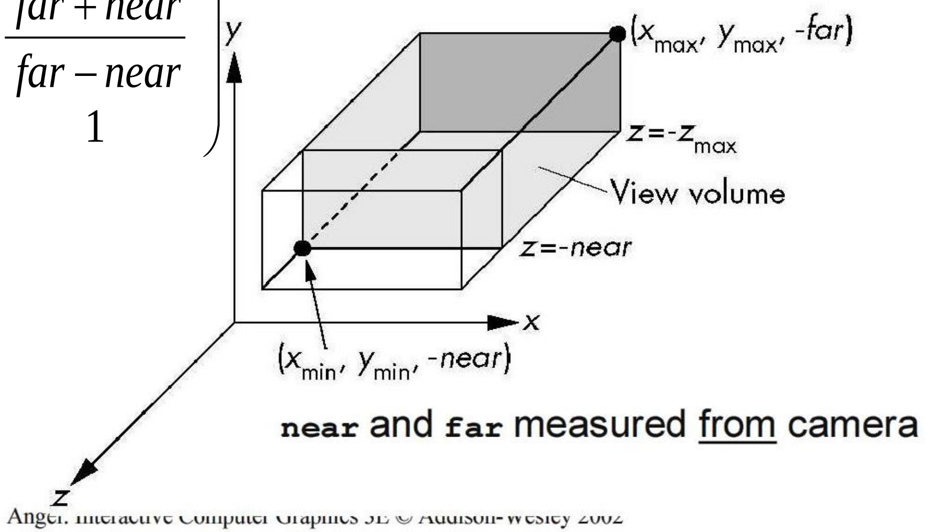
H&B Figure 12.30

Orthographic Viewing in Transform

ortho (xmin, xmax, ymin, ymax, near, far);

- Projection matrix:

$$P = \begin{pmatrix} \frac{2}{x_{\max} - x_{\min}} & 0 & 0 & -\frac{x_{\max} + x_{\min}}{x_{\max} - x_{\min}} \\ 0 & \frac{2}{y_{\max} - y_{\min}} & 0 & -\frac{y_{\max} + y_{\min}}{y_{\max} - y_{\min}} \\ 0 & 0 & \frac{2}{far - near} & \frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



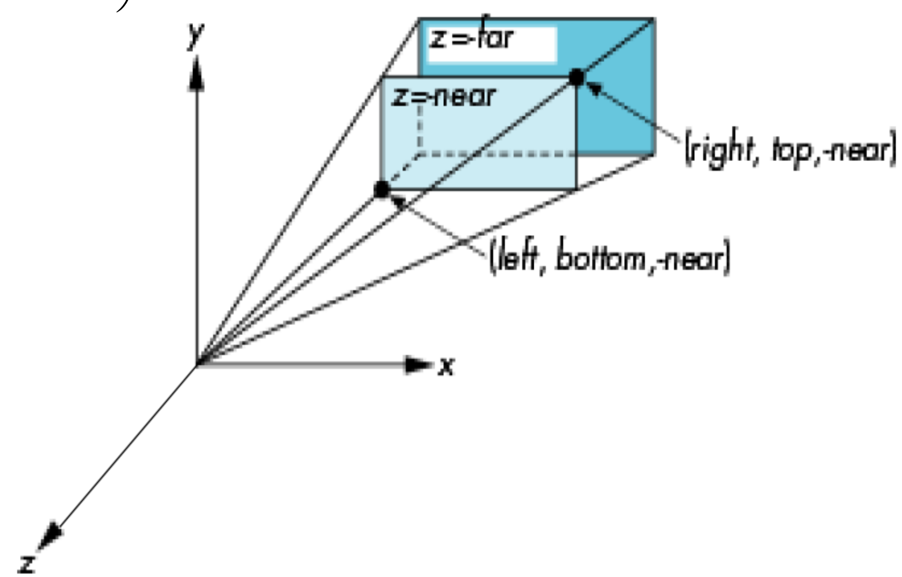
- No oblique projection is implemented

Perspective Viewing in Transform

frustum (xmin, xmax, ymin, ymax, near, far);

- Projection matrix:

$$P = \begin{pmatrix} \frac{2near}{x_{\max} - x_{\min}} & 0 & \frac{x_{\max} + x_{\min}}{x_{\max} - x_{\min}} & 0 \\ 0 & \frac{2near}{y_{\max} - y_{\min}} & \frac{y_{\max} + y_{\min}}{y_{\max} - y_{\min}} & 0 \\ 0 & 0 & \frac{far + near}{far - near} & \frac{2\,far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



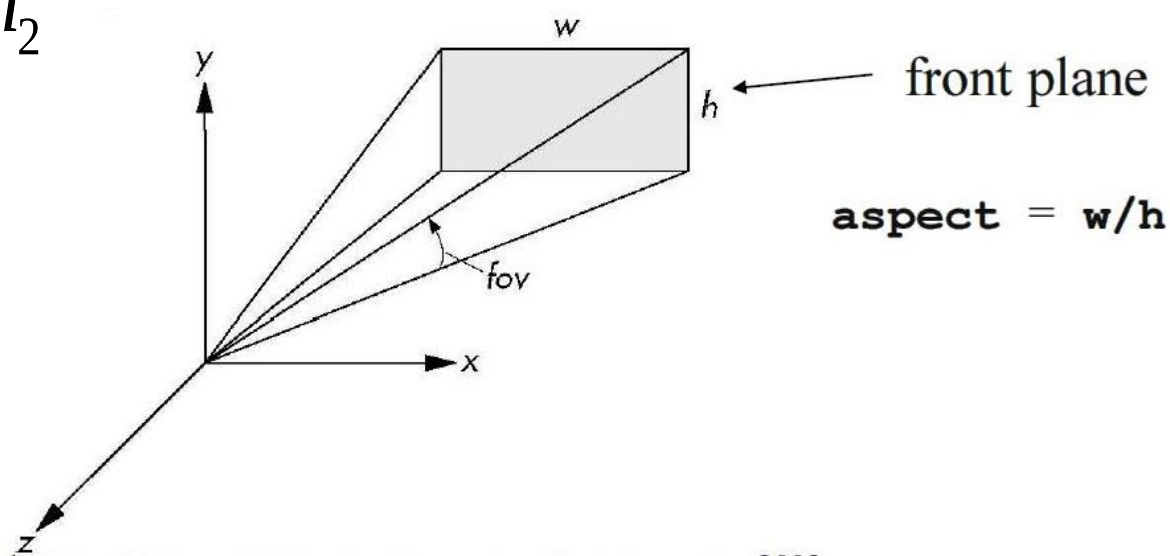
Using Field of View

- frustum not intuitive
- Better interface (for symmetric frustum):

  perspective (fovy, aspect, near, far) =
  frustum (-w2, w2, -h2, h2, near, far);

$$h_2 = near \cdot \tan(fovy / 2)$$

$$w_2 = aspect \cdot h_2$$



Angel: Interactive Computer Graphics 3E © Addison-Wesley 2002

OpenGL Viewport

glViewport (x, y, width, height);

- Default value (0, 0, winWidth, winHeight)
  - winWidth and winHeight specify the size of the window
- Map points drawn on the view plane into the viewport
  - Coordinate transforming from ([-1,-1] ~[1,1]) on the camera coordinate system to ([x,y] ~[x+width,y+height]) on the window coordinate system
- When combined with perspective(), either
  - glViewport (x, y, width, height);
    perspective(fovy, width/height, near, far);
  - glViewport (x, y, width, width/aspect);
    perspective(fovy, aspect, near, far);
- Similar when combined with ortho()

Summary

- How are world coordinates transformed into camera coordinates? Why is this done?
- What is parallel projection? How is it computed?
- What is perspective projection? How is it computed?