• 2D viewing
  – Window
  – Viewport

• 3D viewing
  – Projection methods and viewing volumes
  – Normalization
  – Clipping
  – Perspective division
VIEWING IN 2D

• We specify a rectangular area in the modeling coordinates (world coordinates) and a viewport in the device coordinates on the display
  – window defines what to appear
  – viewport defines where to display

• The mapping of the window (modeling coordinates) to viewport (device coordinates) is a 2D viewing transformation
The 2D Viewing Pipeline

Specify a window in the modeling coordinates: area to be displayed

Transform the window and the models to the normalized coordinates. Clip against the square

Transform the square and the models to the device coordinates in the display viewport.

S(2/width, 2/height);
T(-center);
Transform(models);
// normalized models
// glOrtho()

Clipping();
// generating clipped models
// results not available

Ex: J2_12_RobotSolar

T(Center);
S(With/2, Height/2);
Transform(models);
// device models
// glViewport()
VIEWING IN 3D

• In 3D viewing, we specify a viewing volume with a projection method in the world coordinates, and a viewport on the display
  • Viewing volume specifies what to appear
  • viewport specifies where to display

• The viewing pipeline
  • … modeling transformation (Lighting calculation in 3D)
  • The 3D viewing volume is processed for projection methods
  • Objects are clipped against the 3D viewing volume (z values are preserved for clipping)
  • The contents are then transformed into the viewport for display
  • … scan-conversion (z values are preserved for hidden-surface removal)
The 3D Viewing Pipeline

Specify a volume in the modeling coordinates: volume to be displayed

Transform the volume and the models to the normalized coordinates. Clip against the cube

Transform the cube and the models to the device coordinates in the display viewport.

glFrustum(l,r,b,t,n,f);
//glOrtho(l,r,b,t,n,f);
Trans(models);
// normalized models

Clipping(models);
// clipped models

Ex: J2_12_RobotSolar

glViewport(x,y,w,h);
Trans(models);
Draw(models);
// device models
PROJECTION METHODS

• We deal with only planar geometric projections

• If distance from center of projection (viewpoint) to object is finite, projection is perspective; otherwise parallel
Parallel Projections

**Orthographic Parallel Projections**: the direction of proj is normal to the proj plane.

**Oblique Parallel Proj**: otherwise.

![Diagram of parallel projections]

---

Copyright @ 2002 by Jim X. Chen:

jchen@cs.gmu.edu

OpenGL Viewing
Perspective Projections

Vanishing point: the perspective projections of any set of parallel lines that are not parallel to the projection plane converge to a vanishing point (a point at infinity)

If a set of lines is parallel to a coordinate axis, the vanishing point is called the principle vanishing point

Perspective projections are categorized by their number of principal vanishing points; current graphics projection only deals with the one point projection

Copyright @ 2002 by Jim X. Chen:

jchen@cs.gmu.edu
OpenGL Orthographic Projection

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

- The proj is parallel to the z-axis, & the viewpoint faces the negative z-axis.
- Positive far and near are used as negative z values (they planes are in front of the viewpoint)

Key strokes:
Up Arrow: move forward
Down Arrow: move backward
Right Arrow: move to right
Left Arrow: move to left
Page up: move upward
Page down: move downward
OpenGL Perspective Projection

```c
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,
               GLdouble top, GLdouble near, GLdouble far);
```
Orthographic Projection Math
(if you really want to put a 3D point into a 2D plane)

Projection parallel to z axis and perpendicular to projection plane

\[ P_{ortho}(x, y, -n) \]
**Perspective Projection Math** (put a 3D point into a 2D plane)

The projection onto the near clipping plane:

\[
\begin{pmatrix}
  x_p
  \\
  y_p
  \\
  z_p
\end{pmatrix} = \begin{pmatrix}
  -\frac{x}{z/n}
  \\
  -\frac{y}{z/n}
  \\
  -n
\end{pmatrix}
\]

Given a point in Perspective Proj, you can consider its \((x,y)\) values are at its projection, and its \(z\) value stays:

**Perspective Proj \(\Rightarrow\) Orthographic Proj.** (almost true)
In **homogeneous coordinates**

In matrix form, perspective projection matrix:

\[
M_{\text{per}} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & -\frac{1}{n} & 0
\end{bmatrix}
\]

Transforming in homogeneous coordinates yields the **general homogeneous point**:

\[
\begin{bmatrix}
x \\ y \\ z \\ w
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & -\frac{1}{n} & 0
\end{bmatrix} \begin{bmatrix}
x \\ y \\ z \\ 1
\end{bmatrix} = M_{\text{per}}P;
\]

where \( w = -\frac{z}{n} \)

The 3D coordinates can be calculated from the homogeneous coordinates (**perspective division**, or divide by \( w \)):

\[
(x_p, y_p, z_p) = \left( \frac{x}{w}, \frac{y}{w}, \frac{z}{w} \right) \Rightarrow \left( -\frac{x}{z/n}, -\frac{y}{z/n}, -n \right)
\]
However, we want to keep the z for hidden-surface removal: no linear solution without separating xy and z.

\[
\begin{bmatrix}
  x' \\
y' \\
z' \\
w
\end{bmatrix} =
\begin{bmatrix}
n & 0 & 0 & 0 & 0 \\
0 & n & 0 & 0 & 0 \\
0 & 0 & f + n & fn & 0 \\
0 & 0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} = M_{per}P;
\]

where \( \Rightarrow \) \( w = -z \)

The 3D coordinates can be calculated from the homogeneous coordinates (perspective division, or divide by \( w \)): \( n\leq-z\leq f \)

\[
(x_p, \ y_p, \ z_p) = \left(\frac{x'}{w}, \ \frac{y'}{w}, \ \frac{z'}{w}\right) \Rightarrow \left(\frac{-nx}{z}, \ \frac{-ny}{z}, \ -(f + n + \frac{fn}{z})\right)
\]
Perspective projection – z forshortening

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  w
\end{bmatrix} =
\begin{bmatrix}
  n & 0 & 0 & 0 \\
  0 & n & 0 & 0 \\
  0 & 0 & f + n & fn \\
  0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix} = M_{\text{per}} P;
\]

where \( w = -z \)

The 3D coordinates can be calculated from the homogeneous coordinates (perspective division, or divide by \( w \)): \( n \leq z \leq f \)

\[
(x_p, y_p, z_p) = \left( \frac{x'}{w}, \frac{y'}{w}, \frac{z'}{w} \right) = \left( -\frac{nx}{z}, -\frac{ny}{z}, -(f + n + \frac{fn}{z}) \right)
\]

\[
z_p = -(f + n + \frac{fn}{z})
\]

Copyright @ 2002 by Jim X. Chen:
jchen@cs.gmu.edu
OpenGL Viewing Pipeline

**Modeling Coordinates**

1. Normalize the viewing volume \((w \text{ may not be } 1)\)
2. Clip against the normalized viewing volume
3. Divide by \(w\) for perspective projection
4. Transform into the viewport

```c
glOrtho();
glFrustum();
```

…and (3D transformation in homogeneous coordinates)

- **Specifying a viewing volume**
  - Projection method
  - Normalization
- Clipping
- Perspective division
- Viewport transformation

… (scan-conversion)
Normalization (Viewing Volume)

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

Which transform an arbitrary viewing volume into a normalized viewing volume:

(-1,-1,-1) to (1,1,1)

\[ R = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{-r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{r-l}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{r-l}{t-b} \\ 0 & 0 & 0 & \frac{-f+n}{f-n} \end{bmatrix} \]

\[ R = S(2/(r-l), 2/(t-b), -2/(f-n)) \cdot T(-(r+l)/2, -(t+b)/2, (f+n)/2); \]
glOrtho(l, r, b, t, n, f):
(same as translate and scale)

glOrtho(l, r, b, t, n, f) = 

\[
R = S(\frac{2}{r-l}, \frac{2}{t-b}, -\frac{2}{f-n}) \cdot \begin{pmatrix}
1 & 0 & 0 & -\frac{r+l}{2} \\
0 & 1 & 0 & -\frac{t+b}{2} \\
0 & 0 & 1 & -\frac{f+n}{2} \\
0 & 0 & 0 & 1
\end{pmatrix},
\]

Which transform an arbitrary viewing volume into a normalized viewing volume: (-1,-1,-1) to (1,1,1)
Normalization

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

(\(\text{left, bottom, } -\text{near}\)) and (\(\text{right, top, } -\text{near}\)) specify the \((x,y,z)\) coord of the lower left and upper right corners of the near clipping plane.

\[
R = \begin{bmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{t+b}{t-b} & \frac{r-l}{t-b} & 0 \\
0 & 0 & -(f+n) & -2fn \\
0 & 0 & f-n & f-n
\end{bmatrix}
\]

Which transform an arbitrary viewing volume into a normalized viewing volume.
Perspective Projection $\Rightarrow$ Orthographic Projection

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w
\end{bmatrix} = \begin{bmatrix}
n & 0 & 0 & 0 \\
0 & n & 0 & 0 \\
0 & 0 & f+n & fn \\
0 & 0 & -1 & 0
\end{bmatrix} \begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} = M_{\text{per}}P;
\]

where $\Rightarrow w = -z$;

$z' = (f+n)z + fn$

The 3D coordinates can be calculated from the homogeneous coordinates (perspective division, or divide by $w$). Note that $z$ is linear, but $1/z$ is not, which means when $z$ is big, the difference in 3D coordinates varies very little.

\[
(x_p, \ y_p, \ z_p) = \left(\frac{x'}{w}, \ \frac{y'}{w}, \ \frac{z'}{w}\right) = \left(-\frac{nx}{z}, \ -\frac{ny}{z}, \ -(\frac{fn}{z} + f + n)\right)
\]

Copyright @ 2002 by Jim X. Chen:

jchen@cs.gmu.edu
glFrustum(l, r, b, t, n, f): same as glOrtho after perspective transformation

\[
glFrustum(l, r, b, t, n, f) = \]

\[
R=S\left(\frac{2}{r-l}, \frac{2}{t-b}, \frac{-2}{f-n}\right) T\left(-\frac{r+l}{2}, -\frac{t+b}{2}, \frac{n+f}{2}\right)M_{\text{per}}; \\
\begin{bmatrix}
x' \\
y' \\
z' \\
w
\end{bmatrix} = S\left(\frac{2}{r-l}, \frac{2}{t-b}, \frac{-2}{f-n}\right) T\left(-\frac{r+l}{2}, -\frac{t+b}{2}, \frac{n+f}{2}\right)M_{\text{per}} P
\]

where \( \Rightarrow w = -z \)

The 3D coordinates can be calculated from the homogeneous coordinates (perspective division, or divide by \(w\)).
glFrustum(l, r, b, t, n, f): same as glOrtho after perspective transformation

\[
R = \begin{bmatrix}
\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\
0 & \frac{2}{t-b} & 0 & -\frac{r+l}{t+b} \\
0 & 0 & \frac{-2}{f-n} & -\frac{t-b}{f-n} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
n & 0 & 0 & 0 \\
0 & n & 0 & 0 \\
0 & 0 & f + n & fn \\
0 & 0 & -1 & 0
\end{bmatrix};
\]

\[
\text{glFrustum}(l, r, b, t, n, f) = R =
\begin{bmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{t-b} & \frac{r-l}{t-b} & 0 \\
0 & 0 & -(f+n) & -2fn \\
0 & 0 & \frac{f-n}{f-n} & f-n \\
\end{bmatrix}
\]
CLIPPING AGAINST THE CUBE

A point \((x, y, z)\) is represented by 6 bits:
- Bit 6 =1 if \(x<\text{left}\);
- Bit 5 =1 if \(x>\text{right}\);
- Bit 4 =1 if \(y<\text{bottom}\);
- Bit 3 =1 if \(y>\text{top}\);
- Bit 2=1 if \(z<\text{near}\);
- Bit 1=1 if \(z>\text{far}\);

A line (2 points) logic OR=0, trivially accepted; logic AND !=0, trivially rejected;

Otherwise intersection is calculated and a line is cut into two; the process continue until all segments are trivially accepted/rejected;

\[
x = x_1 + (x_2 - x_1)\alpha; \quad y = y_1 + (y_2 - y_1)\alpha; \quad z = z_1 + (z_2 - z_1)\alpha;
\]

plus the plane equations;

2D polygon clipping algorithm can be easily extended to 3D as well.
Line Clipping against a Plane

// how about a line against a plane?

GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};
glClipPlane (GL_CLIP_PLANE0, eqn);
glEnable (GL_CLIP_PLANE0);

• A plane equation
  \(Ax + By + Cz + D = 0\)

• A 3D line equation
  \(x = x_0 + t(x_1 - x_0);\ y = y_0 + t(y_1 - y_0);\ z = z_0 + t(z_1 - z_0);\)

The intersection of the line with the plane is at a specific \(t\)

Example: J2_12_Clipping
\texttt{glViewport (x, y, width, height);}

T(Center);
S(with/2, height/2, 1);
Trans(models);
// device models
\texttt{// glDepthRange( GLclampd znear, GLclampd zfar );// 0-1}

• The viewport transformation calculates each vertex’s \( (x, y, z) \)
corresponding to the pixels, and invokes scan-conversion algorithms to draw the model into the viewport.

• Projecting into 2D is nothing more than ignoring the \( z \) values when scan-converting the model’s pixels into the frame buffer. It is not necessary but we may consider that the projection plane is at \( z=0 \).

Example: J2_13_ViewPort
Viewing & Transformation in OpenGL

• Mechanism
• Programming
Modeling Transformations

• Consider the following code sequence:

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);   /* apply transformation N */
glMultMatrixf(M);   /* apply transformation M */
glMultMatrixf(L);   /* apply transformation L */
glBegin(GL_POINTS);
    glVertex3f(v); /* draw transformed vertex v */
glEnd();
```

The vertex transformation is \( N(M(Lv)) \); (As we know, the matrices are multiplied together first)

• Special matrix multiplications:
  - `void glScale{fd}(TYPE x, TYPE y, TYPE z);`
  - `void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);`
  - `void glTranslate{fd}(TYPE x, TYPE y, TYPE z);`
Viewing Transformation

• Analogous to positioning and aiming a camera.
• The camera is originally situated at the origin and points down the negative z-axis.

• Viewing transformation are called first modeling transformation take effect first.
MODELVIEW MATRIX

Another way of looking at the MODELVIEW matrix is that the matrix transforms the viewing method instead of the model.

Translating a model along the negative $z$ axis is like moving the viewing volume along the positive $z$ axis.

Rotating a model along an axis by a positive angle is like rotating the viewing volume along the axis by a negative angle.

Example: J2_13_TravelSolar
Viewing & Modeling (are the same)

- When we analyze a model’s transformations, logically speaking, the order of transformation steps are bottom-up from the closest transformation above the drawing command to where we specify the viewing volume.

- When we analyze a model’s transformation by thinking about transforming its viewing, the order of transformation steps are top-down from where we specify the viewing volume to where we specify the drawing command.

- The signs of the transformation are logically negated.
Examples of moving the camera

• Going backwards to the moon in generalized solar system: J2_13_TravelSolar

• A flight simulator to display the world from the point of view of the pilot:

```c
void pilotView(GLdouble planex, GLdouble planey,
               GLdouble planez, GLdouble roll, GLdouble pitch,
               GLdouble heading) {
    glRotated(roll, 0.0, 1.0, 0.0);
    glRotated(pitch, 1.0, 0.0, 0.0);
    glRotated(heading, 0.0, 0.0, 1.0);
    glTranslated(-planex, -planey, -planez);
}
```

• Orbiting the camera around an object that’s centered at the origin:

```c
void polarView(GLdouble distance, GLdouble twist,
               GLdouble elevation, GLdouble azimuth) {
    glTranslated(0.0, 0.0, -distance);
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(elevation, 1.0, 0.0, 0.0);
    glRotated(azimuth, 0.0, 0.0, 1.0);
}
```
Another projection method

```c
void gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble znear, GLdouble zfar);
```

- `fovy` -- angle of the field of view in x-z plane (0-180 degree);
- `aspect` -- w/h;
- `znear, zfar` -- the distances between the viewpoint and the clipping planes along the negative z-axis. The should be always positive.
public void myPerspective(double fovy, double aspect, double near, double far) {
    double left, right, bottom, top;
    fovy = fovy*Math.PI/180; // convert degree to arc
    top = near*Math.tan(fovy/2);
    bottom = -top;
    right = aspect*top;
    left = -right;
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glFrustum(left, right, bottom, top, near, far);
}
Viewing from an arbitrary viewpoint

OpenGL viewing is always from the center of the coordinates looking down the negative z axis, with the view-up in the y axis direction

```c
void gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centrery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);
```

eye -- viewpoint (PRP);
center -- reference point (VRP);
up -- view up direction (VUP)
public void myLookAt(double eX, double eY, double eZ,
    double cX, double cY, double cZ, double upX, double upY, double upZ) {
   // eye and center are points, but up is a vector

   // 1. change center into a vector: goal?
   // glTranslated(-eX, -eY, -eZ);
   cX = cX-eX;
   cY = cY-eY;
   cZ = cZ-eZ;

Example: J2_15_LookAt(eX, eY, eZ);
//2. The \textbf{angle} of \textit{center on xz plane} and \textit{x axis}
// i.e. angle to rot so center in the neg. yz plane
double \texttt{a} = \texttt{Math.atan(cZ/cX)};
if (\texttt{cX} >= 0) {
    \texttt{a} = \texttt{a} + \texttt{Math.PI}/2;
} else {
    \texttt{a} = \texttt{a} - \texttt{Math.PI}/2;
}
// \texttt{a} is now the angle to rotate \texttt{c} into yz plane
// 3. The **angle** between the **center** and **y axis**
// i.e. angle to rot around x axis so center in the negative z axis
    double b = Math.acos(cY/Math.sqrt(cX*cX+cY*cY+cZ*cZ)); b = b-Math.PI/2;

// 4. **up** rotate around **y axis** (a) radians (up rots accordingly)
    double upx = upX*Math.cos(a)+upZ*Math.sin(a);
    double upz = -upX*Math.sin(a)+upZ*Math.cos(a);
    upX = upx;    upZ = upz;

// 5. **up** rotate around **x axis** (b) radians (up rots accordingly)
    double upy = upY*Math.cos(b)-upZ*Math.sin(b);
    upz = upY*Math.sin(b)+upZ*Math.cos(b);
    upY = upy;    upZ = upz;
// 6. the angle between up on xy plane and y axis
    double c = Math.atan(upX/upY);
    if (upY<0) { c = c+Math.PI; }

    gl.glRotated(Math.toDegrees(c), 0, 0, 1);
    // up in yz plane
    gl.glRotated(Math.toDegrees(b), 1, 0, 0);
    // center in negative z axis
    gl.glRotated(Math.toDegrees(a), 0, 1, 0);
    // center in yz plane
    gl.glTranslated(-eX, -eY, -eZ);
    // eye at the origin
HW

Implement two commands yourself: myPerspective and myLookat just like gluPerspective and gluLookAt:

- **Void** gluPerspective (double fovy, double aspect, double znear, double zfar);
  - fovy -- angle of the field of view in x-z plane (0-180 degree);
  - aspect -- w/h;
  - znear, zfar -- the distances between the viewpoint and the clipping planes along the negative z-axis. The should be always positive.

- **Void** gluLookAt (double eyex, double eyey, double eyez, double centerx, double centery, double centerz, double upx, double upy, double upz);
  - eye -- viewpoint (PRP);
  - center -- reference point (VRP);
  - up -- view up direction (VUP)
Viewing & Transformation in OpenGL

Review

SpaceView    MoonView

Copyright @ 2002 by Jim X. Chen:
jchen@cs.gmu.edu
**REVIEW: THE CAMERA ANALOGY**

<table>
<thead>
<tr>
<th>With a Camera</th>
<th>With a Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>tripod</td>
<td>viewing</td>
</tr>
<tr>
<td>move</td>
<td>positioning the viewing volume in the world</td>
</tr>
<tr>
<td>lens</td>
<td>positioning the models in the world</td>
</tr>
<tr>
<td>photograph</td>
<td>determining shape of viewing volume</td>
</tr>
<tr>
<td></td>
<td>viewport</td>
</tr>
</tbody>
</table>

Key strokes:
- Up Arrow: move forward
- Down Arrow: move backward
- Right Arrow: move to right
- Left Arrow: move to left
- Page up: move upward
- Page down: move downward

jchen@cs.gmu.edu
To specify viewing, modeling, and projection transformations, you construct a $4 \times 4$ matrix $M$: $v' = Mv$. (Vertices have four coordinates $(x,y,z,w)$, though in most cases $w$ is 1)
• **Modelview matrix**: object coordinates to eye coordinates.  
  *(specifying camera & obj position)*

• **Projection matrix**: eye to clip coordinates.  
  *(view volume, normalization, clipping, & projection)*

• **Perspective division**: divide coordinate values by $w$ to produce normalized device coordinates.  
  *(part of projection)*

• **Viewport matrix**: to window coordinates *(shrunk or stretched, continuous, float)* by applying the viewport transformation.

• All the transformations are performed on the $z$ coordinates as well.
Suppose the current matrix is $C$ and we call `glMultMatrixf(M)`. After multiplication, the matrix is $CM$. 
Viewing Transformation

• Analogous to positioning and aiming a camera.
• By default, the camera is originally situated at the origin and points down the negative z-axis.

• Viewing transformation are called first so that modeling transformation take effect first.
Viewport Transformation

The aspect ratio of a viewport should generally equal the aspect ratio of the viewing volume.

```c
    gluPerspective(myFovy, 1.0, myNear, myFar);
    glViewport (0, 0, 400, 400);

    gluPerspective(myFovy, 2.0, myNear, myFar);
    glViewport (0, 0, 400, 400);
```

To avoid the distortion, this line could be used:

```c
    glViewport(0, 0, 400, 200);
```
To create two side-by-side viewports, you issue these commands, along with the appropriate modeling, viewing, and projection transformation:

```c
glViewport (0, 0, sizex/2, sizey);

...  

glViewport (sizex/2, 0, sizex/2, sizey);
```

**Manipulating the Matrix Stacks**