Computer Graphics

Prof. Feng Liu

Fall 2015

http://www.cs.pdx.edu/~fliu/courses/cs447/

10/27/2015
Last time

- More 2D Transformations
- Homogeneous Coordinates
- 3D Transformations
- The Viewing Pipeline
Today

☐ Perspective projection
☐ Clipping
☐ Homework 2 due in class today
☐ In-class Middle-Term
  ■ Thursday, Nov. 5
  ■ Close-book exam
  ■ Notes on 1 page of A4 or Letter size paper
  ■ To-know list available online
Graphics Pipeline

Local Coordinate Space → World Coordinate Space → View Space → Canonical View Volume → Display Space
Defining Cameras

- View Space is the *camera’s* local coordinates
  - The camera is in some location
  - The camera is looking in some direction
  - It is tilted in some orientation
- It is inconvenient to model everything in terms of View Space
  - Biggest problem is that the camera might be moving - we don’t want to have to explicitly move every object too
- We specify the camera, and hence View Space, with respect to World Space
  - How can we specify the camera?
Specifying a View

- The **location** of View Space with respect to World Space
  - A point in World Space for the origin of View Space, \((e_x, e_y, e_z)\)
- The **direction** in which we are looking: gaze direction
  - Specified as a vector: \((g_x, g_y, g_z)\)
  - This vector will be normal to the image plane
- A **direction** that we want to *appear up* in the image
  - \((up_x, up_y, up_z)\), this vector does not have to be perpendicular to \(g\)
- We also need the size of the view volume - \(l, r, t, b, n, f\)
  - Specified with respect to the eye and image plane, not the world
Subtle point: it doesn’t precisely matter where we put the image plane.
Getting there...

- We wish to end up in View Space, so we need a coordinate system with:
  - A vector toward the viewer, View Space $z$
  - A vector pointing right in the image plane, View Space $x$
  - A vector pointing up in the image plane, View Space $y$
  - The origin at the eye, View Space (0,0,0)

- We must:
  - Say what each of these vectors are in **World Space**
  - Transform points from the World Space into View Space
  - We can then apply the orthographic projection to get to Canonical View Space, and so on
View Space in World Space

Given our camera definition, in World Space:

- Where is the origin of view space? It will transform into $(0,0,0)_{\text{view}}$
- What is the normal to the view plane, $w$? It will become $z_{\text{view}}$
- How do we find the right vector, $u$? It will become $x_{\text{view}}$
- How do we find the up vector, $v$? It will become $y_{\text{view}}$

Given these points, how do we do the transformation?
View Space

- The origin is at the eye: \((e_x, e_y, e_z)\)
- The normal vector is the normalized viewing direction
  \[ \mathbf{w} = -\hat{\mathbf{g}} \]
- We know which way up should be, and we know we have a right handed system, so \(\mathbf{u} = \mathbf{up} \times \mathbf{w}\), normalized: \(\hat{\mathbf{u}}\)
- We have two vectors in a right handed system, so to get the third: \(\mathbf{v} = \mathbf{w} \times \mathbf{u}\)
World to View

- We must translate so the origin is at \((e_x, e_y, e_z)\)
- To complete the transformation we need to do a rotation
- After this rotation:
  - The direction \(u\) in world space should be the direction \((1,0,0)\) in view space
  - The vector \(v\) should be \((0,1,0)\)
  - The vector \(w\) should be \((0,0,1)\)
- The matrix that does the rotation is:
  - It’s a “change of basis” matrix

\[
\begin{bmatrix}
u_x & u_y & u_z & 0 \\
v_x & v_y & v_z & 0 \\
w_x & w_y & w_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
All Together

- We apply a translation and then a rotation, so the result is:

\[
M_{\text{world} \rightarrow \text{view}} = \begin{bmatrix}
  u_x & u_y & u_z & 0 \\
  v_x & v_y & v_z & 0 \\
  w_x & w_y & w_z & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix} \begin{bmatrix}
  1 & 0 & 0 & -e_x \\
  0 & 1 & 0 & -e_y \\
  0 & 0 & 1 & -e_z \\
  0 & 0 & 0 & 1 \\
\end{bmatrix} = \begin{bmatrix}
  u_x & u_y & u_z & -u \cdot e \\
  v_x & v_y & v_z & -v \cdot e \\
  w_x & w_y & w_z & -w \cdot e \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

- And to go all the way from world to screen:

\[
M_{\text{world} \rightarrow \text{canonical}} = M_{\text{view} \rightarrow \text{canonical}} M_{\text{world} \rightarrow \text{view}}
\]

\[
x_{\text{canonical}} = M_{\text{world} \rightarrow \text{canonical}} x_{\text{world}}
\]
Graphics Pipeline

Local Coordinate Space → World Coordinate Space → View Space → Canonical View Volume → Display Space
OpenGL and Transformations

- OpenGL internally stores two matrices that control viewing of the scene
  - The **GL_MODELVIEW** matrix is intended to capture all the transformations up to view space
  - The **GL_PROJECTION** matrix captures the view to canonical conversion
- You also specify the mapping from the canonical view volume into window space
  - Directly through a **glViewport** function call
- Matrix calls, such as **glRotate**, multiply some matrix M onto the current matrix C, resulting in CM
  - Set view transformation first, then set transformations from local to world space - last one set is first one applied
  - This is the convenient way for modeling, as we will see
Graphics Pipeline

Local Coordinate Space → World Coordinate Space → View Space → Canonical View Volume → Display Space

GL_MODELVIEW  GL_PROJECTION  glViewport
OpenGL Camera

- The **default** OpenGL image plane has u aligned with the x axis, v aligned with y, and n aligned with z
  - Means the default camera looks along the negative z axis
  - Makes it easy to do 2D drawing (no need for any view transformation)

- `glOrtho(...)` sets the view->canonical matrix
  - Modifies the `GL_PROJECTION` matrix

- `gluLookAt(...)` sets the world->view matrix
  - Takes an image center point, a point along the viewing direction and an up vector
  - Multiplies a world->view matrix **onto the current `GL_MODELVIEW` matrix**
  - You could do this yourself, using `glMultMatrix(...)` with the matrix from the previous slides
Typical Usage

GLU functions, such as `gluLookAt(...)` are not part of the core OpenGL library

- They can be implemented with other core OpenGL commands
- For example, `gluLookAt(...)` uses `glMultMatrix(...)` with the matrix from the previous slides
- They are not dependent on a particular graphics card

```c
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(l, r, b, t, n, f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(ex, ey, ez, cx, cy, cx, ux, uy, uz);
```
Left vs Right Handed View Space

- You can define $u$ as right, $v$ as up, and $n$ as toward the viewer: a right handed system $u \times v = w$
  - Advantage: Standard mathematical way of doing things

- You can also define $u$ as right, $v$ as up and $n$ as into the scene: a left handed system $v \times u = w$
  - Advantage: Bigger $n$ values mean points are further away

- OpenGL is right handed

- Many older systems, notably the Renderman standard developed by Pixar, are left handed
Graphics Pipeline

Local Coordinate Space → World Coordinate Space → View Space → Canonical View Volume → Display Space
Review

- **View Space** is a coordinate system with the viewer looking down the -z axis, with x to the right and y up.
- The World->View transformation takes points in world space and converts them into points in view space.
- The Projection matrix, or View->Canonical matrix, takes points in view space and converts them into points in Canonical View Space.
  - **Canonical View Space** is a coordinate system with the viewer looking along -z, x to the right, y up, and everything to be drawn inside the cube [-1, 1]x[-1, 1]x[-1, 1] using parallel projection.
Perspective Projection

- Abstract camera model - box with a small hole in it
- Pinhole cameras work in practice
Distant Objects Are Smaller
Parallel lines meet
Vanishing points

- Each set of parallel lines (direction) meets at a different point: The vanishing point for this direction
  - Classic artistic perspective is 3-point perspective
- Sets of parallel lines on the same plane lead to collinear vanishing points: the horizon for that plane
- Good way to spot faked images
Basic Perspective Projection

- We are going to temporarily ignore canonical view space, and go straight from view to window.
- Assume you have transformed to view space, with $x$ to the right, $y$ up, and $z$ back toward the viewer.
- Assume the origin of view space is at the center of projection (the eye).
- Define a focal distance, $d$, and put the image plane there (note $d$ is negative).
  - You can define $d$ to control the size of the image.
Basic Perspective Projection

- If you know $P(x_v,y_v,z_v)$ and $d$, what is $P(x_s,y_s)$?
  - Where does a point in view space end up on the screen?
Basic Case

Similar triangles gives:

\[ \frac{x_s}{d} = \frac{x_v}{z_v} \quad \frac{y_s}{d} = \frac{y_v}{z_v} \]

\[ \frac{x_s}{d} = \frac{x_v}{z_v} \quad \frac{y_s}{d} = \frac{y_v}{z_v} \]

\[ P(x_s, y_s) \quad P(x_v, y_v, z_v) \]

View Plane
Simple Perspective Transformation

- Using homogeneous coordinates we can write:

  - Our next big advantage to homogeneous coordinates

\[
\begin{bmatrix}
x_s \\
y_s \\
d
\end{bmatrix} \equiv \begin{bmatrix}
x_v \\
y_v \\
z_v \\
z_v/d
\end{bmatrix} \quad \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 0
\end{bmatrix}
\]
Parallel Lines Meet?

- Parallel lines are of the form: \( \mathbf{x} = \mathbf{x}_0 + t\mathbf{d} \)
  - Parametric form: \( \mathbf{x}_0 \) is a point on the line, \( t \) is a scalar (distance along the line from \( \mathbf{x}_0 \)) and \( \mathbf{d} \) is the direction of the line (unit vector)
  - Different \( \mathbf{x}_0 \) give different parallel lines

- Transform and go from homogeneous to regular:

\[
\begin{bmatrix}
  x \\
  y \\
  z \\
  w
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 1/f & 0
\end{bmatrix} \begin{bmatrix}
  x_0 \\
  y_0 \\
  z_0 \\
  1
\end{bmatrix} + t \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 1/f & 0
\end{bmatrix} \begin{bmatrix}
  x_d \\
  y_d \\
  z_d \\
  0
\end{bmatrix} = f \begin{bmatrix}
  x_0 + tx_d \\
  z_0 + tz_d \\
  y_0 + ty_d \\
  z_0 + tz_d
\end{bmatrix}
\]

- Limit as \( t \to \infty \) is

\[
\begin{bmatrix}
  fx_d/f & fy_d/f \\
  z_d/f & z_d/f & f
\end{bmatrix}
\]
The basic equations we have seen give a flavor of what happens, but they are insufficient for all applications. They do not get us to a Canonical View Volume. They make assumptions about the viewing conditions. To get to a Canonical Volume, we need a Perspective Volume.
Perspective View Volume

- Recall the orthographic view volume, defined by a near, far, left, right, top and bottom plane.
- The perspective view volume is also defined by near, far, left, right, top and bottom planes - the *clip planes*
  - Near and far planes are parallel to the image plane: $z_v=n, z_v=f$
  - Other planes all pass through the center of projection (the origin of view space)
  - The left and right planes intersect the image plane in vertical lines
  - The top and bottom planes intersect in horizontal lines
Clipping Planes

Near Clip Plane

Far Clip Plane

View Volume

Left Clip Plane

Right Clip Plane

$n$
Where is the Image Plane?

- Notice that it doesn’t really matter where the image plane is located, once you define the view volume.
  - You can move it forward and backward along the z-axis and still get the same image, only scaled.

- The left/right/top/bottom planes are defined according to where they cut the near clip plane.

- Or, define the left/right and top/bottom clip planes by the field of view.
Field of View

Assumes a *symmetric* view volume

- Near Clip Plane
- Far Clip Plane
- Left Clip Plane
- Right Clip Plane

$\text{FOV}$

$x_v$

$-z_v$

$f$
Perspective Parameters

- We have seen several different ways to describe a perspective camera
  - Focal distance, Field of View, Clipping planes
- The most general is clipping planes - they directly describe the region of space you are viewing
- For most graphics applications, field of view is the most convenient
  - It is *image size invariant* - having specified the field of view, what you see does not depend on the image size
- You can convert one thing to another
Focal Distance to FOV

- You must have the image size to do this conversion
  - Why? Same $d$, different image size, different FOV

\[
\tan\left(\frac{FOV}{2}\right) = \frac{\text{height}}{2d}
\]

\[
FOV = 2\tan^{-1}\left(\frac{\text{height}}{2d}\right)
\]
OpenGL

- **gluPerspective(…)**
  - Field of view in the y direction, *FOV*, (vertical field-of-view)
  - Aspect ratio, *a*, **should match window aspect ratio**
  - Near and far clipping planes, *n* and *f*
  - Defines a symmetric view volume

- **glFrustum(…)**
  - Give the near and far clip plane, and places where the other clip planes cross the near plane
  - Defines the general case
  - Used for stereo viewing, mostly
As noted previously, *glu* functions don’t add basic functionality, they are just more convenient.

- So how does *gluPerspective* convert to *glFrustum*?
- Symmetric, so only need *t* and *l*.

\[ \text{FOV} / 2 \]
We want a matrix that will take points in our perspective view volume and transform them into the orthographic view volume.

- This matrix will go in our pipeline before an orthographic projection matrix.
Mapping Lines

- We want to map all the lines through the center of projection to parallel lines.
  - This converts the perspective case to the orthographic case, we can use all our existing methods.

- The relative intersection points of lines with the near clip plane should not change.

- The matrix that does this looks like the matrix for our simple perspective case.
General Perspective

This matrix leaves points with \( z=n \) unchanged

It is just like the simple projection matrix, but it does some extra things to \( z \) to map the depth properly

We can multiply a homogenous matrix by any number without changing the final point, so the two matrices above have the same effect

\[
M_P = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & (n+f)/n & -f \\
0 & 0 & 1/n & 0
\end{bmatrix} \equiv \begin{bmatrix}
n & 0 & 0 & 0 \\
0 & n & 0 & 0 \\
0 & 0 & n+f & -nf \\
0 & 0 & 1 & 0
\end{bmatrix}
\]
Complete Perspective Projection

After applying the perspective matrix, we map the orthographic view volume to the canonical view volume:

\[
\begin{pmatrix}
\frac{2}{(r-l)} & 0 & 0 & -(r+l) \\
0 & \frac{2}{(t-b)} & 0 & -(t+b) \\
0 & 0 & \frac{2}{(n-f)} & -(n+f) \\
0 & 0 & \frac{2}{(n-f)} & 1
\end{pmatrix}
\begin{pmatrix}
\begin{array}{cccc}
n & 0 & 0 & 0 \\
0 & n & 0 & 0 \\
0 & 0 & (n+f) & -nf \\
0 & 0 & 1 & 0
\end{array}
\end{pmatrix}
\]

\[
M_{\text{world} \rightarrow \text{canonical}} = M_{\text{view} \rightarrow \text{canonical}} M_{\text{world} \rightarrow \text{view}}
\]

\[
\mathbf{x}_{\text{canonical}} = M_{\text{world} \rightarrow \text{canonical}} \mathbf{x}_{\text{world}}
\]
Near/Far and Depth Resolution

- It may seem sensible to specify a very near clipping plane and a very far clipping plane
  - Sure to contain entire scene
- But, a bad idea:
  - OpenGL only has a finite number of bits to store screen depth
  - Too large a range reduces resolution in depth - wrong thing may be considered “in front”
  - See Shirley for a more complete explanation
- Always place the near plane as far from the viewer as possible, and the far plane as close as possible
OpenGL Perspective Projection

For OpenGL you give the distance to the near and far clipping planes.

The total perspective projection matrix resulting from a glFrustum call is:

\[
\begin{bmatrix}
\frac{2|n|}{(r-l)} & 0 & \frac{r+l}{(r-l)} & 0 \\
0 & \frac{2|n|}{(t-b)} & \frac{(t+b)}{(t-b)} & 0 \\
0 & 0 & \left(\frac{|n|+|f|}{|n|-|f|}\right) & \frac{2|f||n|}{|n|-|f|} \\
0 & 0 & -1 & 0
\end{bmatrix}
\]
Next Time

☐ Clipping
☐ Rasterization
☐