#6: Camera Perspective, Viewing, and Culling

CSE167: Computer Graphics
Instructor: Ronen Barzel
UCSD, Winter 2006
Outline for today

- Camera Space
- Projection
- View Volumes
- Culling
- Bounding Hierarchy
3-D Graphics Rendering Pipeline

- Primitives
  - Modeling Transformation
    - Object coordinates
  - Viewing Transformation
    - World space
    - Eye space (AKA Camera space)
  - Lighting & Shading
  - Clipping
  - Projection
    - Normalized view space
  - Scan conversion, Hiding
    - Image space, Device coordinates
- Image
From modeling to rendering

- So far, we’ve discussed the following spaces
  - Object space (local space)
  - World space (global space)
- Today we’ll add:
  - Camera space
  - Normalized view space
  - Image space (2D)
Camera

- Think of camera itself as a model
  - Place it in 3D space

- Camera’s frame:
  - origin at *eye point*
  - -z points in the viewing direction
  - x,y define the *film plane*
    - x is to the right on the film
    - y is up on the film
Remember…

- Local-to-world matrix, AKA Model Transform
Camera Matrix

- The local-to-world matrix for the camera
How to specify camera matrix

- Can construct it using our existing techniques.
- Common idiom: "Look-at transformation"
  - Given the eye point, AKA look-from point
  - Given a target point, AKA look-at point
  - Matrix points the camera toward the look-at point
- Which way is up?
  - There’s a degree of freedom available: spin the camera
  - Specify an up vector in the world, that will point along y in the camera.
Camera Look-At setup

- \text{look-from} = \text{eye point} e
- \text{look-at} = \text{target point} t
- \text{up vector} \overrightarrow{u}
- \text{view vector}

World coordinates

Camera Space

Camera Matrix \( C \)
“Look-at” Matrix calculation

- **Given:**
  - look-from: eye at position \( \mathbf{e} \)
  - look-at: target at position \( \mathbf{t} \)
  - up-vector: \( \mathbf{\bar{u}} \)

- **Fill the** \( \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \) **columns of the matrix with the world-space coordinates of the camera’s frame:**
  - \( \mathbf{d} \) is position of frame origin, i.e. the eye point:
    \[
    \mathbf{d} = \mathbf{e}
    \]
  - \( \mathbf{c} \) is the z axis of the frame, i.e. the view vector:
    \[
    \mathbf{\bar{c}} = \frac{\mathbf{e} - \mathbf{t}}{|\mathbf{e} - \mathbf{t}|}
    \]
“Look-at” Matrix calculation

- **a** is the camera frame’s x axis. We want it to be perpendicular to the view vector, and also perpendicular to the up vector:

\[
\vec{a} = \frac{\vec{u} \times \vec{c}}{|\vec{u} \times \vec{c}|}
\]

- **b** is the camera frame’s y axis. It must be perpendicular to **a** and **c**.

\[
\vec{b} = \vec{c} \times \vec{a}
\]

- Notes:
  - Cross product order is important to make sure the frame is right-handed.
  - Since **a** and **c** are unit length and perpendicular to each other, we don’t need to normalize **b**.
“Look-at” Matrix calculation, summary

Given: eye point \( e \), target point \( t \), and up vector \( \vec{u} \)
Construct: columns of camera matrix \( C \)

\[
\begin{align*}
\vec{d} &= e \\
\vec{c} &= \frac{e - t}{|e - t|} \\
\vec{a} &= \frac{\vec{u} \times \vec{c}}{|\vec{u} \times \vec{c}|} \\
\vec{b} &= \vec{c} \times \vec{a}
\end{align*}
\]

- Note: The up vector may not end up parallel to the camera y axis
  - The projection of the up vector onto the film plane lines up with camera y

- If the up vector is parallel to the view vector, the result is undefined!
  - the up vector will project to nothing in the image
  - no matter how you spin the camera, there’s no thing to line up with the camera y
  - it’s a user error!
Camera Space

- For rendering, we want to consider all objects in camera space
  - We have matrix $C$ that transforms from camera space into world space
  - View an object that was placed into world space using matrix $M$

- To go from object space to camera space:
  - First go from object to world via $M$
  - Then go backwards from world to camera, using the inverse of $C$
  - Compose these into a single matrix:

  \[
  \text{Object-to-camera} = C^{-1}M
  \]
Model-to-Camera transform

\[ \text{Model-to-Camera} = \mathbf{C}^{-1} \mathbf{M} \]

- Camera Space
- Camera-to-world \( \mathbf{C} \)
- World-to-camera \( \mathbf{C}^{-1} \)
- Model-to-world \( \mathbf{M} \)
- Object Space
- Model-to-Camera

World Space

13
In camera space

- We have things lined up the way we like them on screen:
  - X to the right
  - Y up
  - -Z going into the screen
  - Objects to look at are in front of us, i.e. have negative z values

- But the objects are still in 3D.
  - Now let’s look at how to project them into 2D to get them on screen
Outline for today

- Camera Space
- *Projection*
- View Volumes
- Culling
- Bounding Hierarchy
Normalized view space, Image Space

- We’re ultimately heading to 2D *image space*:
  - We’ll need a mapping from 3D space into 2D *image space*
  - But for rendering (hiding) we’ll need to know the depth information (z)
  - So we’ll really map into a 3D space, at least at first

- **Normalized view space**
  - A 3D space
  - Everything visible in the image will range from -1 to 1 in both x and y, with (0,0) in the center of the image
  - The z coordinate will also range from -1 to 1 for depth, with 1 being the nearest and -1 being farthest.
View Projections

- Transform from camera space to normalized view space

- Two basic kinds:
  - **Perspective projection**: make things farther away seem smaller
    - Most common for computer graphics
    - Simple model of human eye, or camera lens
    - (Actually, a model of an ideal *pinhole camera*)
  
  - **Orthographic projection**: simply flatten, without any perspective
    - Used for architectural or plan views (top, side, front)
    - Not used for realistic rendering

- Others, more complex:
  - lens, with focus & depth of field
  - fish-eye lens
  - dome projection
  - computations don’t easily fit into basic hardware rendering pipeline
Perspective Projection

- Things farther away get smaller
- Parallel lines no longer parallel: vanishing point
- Discovery/formalization attributed to Filippo Brunelleschi in the early 1400’s
- Earliest example: La Trinitá (1427) by Masaccio
Perspective Projection

- Assume that we have “film” at distance $d$ from the eye
- Distant tall object projects to same height as near small object
- By similar triangles, we have:
  \[
  \frac{y'}{d} = \frac{y_1}{z_1} = \frac{y_2}{z_2}
  \]

  Giving the transformation relations:
  \[
  y' = d \frac{y}{z}, \quad x' = d \frac{x}{z}
  \]

- Notice: divide by $z$
  - not a linear operation!
Aside: go the other way

- If you make some assumptions about what parts of the image are square, etc., it’s possible to recover the 3D geometry.

(Figure from a group at Oxford)
Perspective Projection

\[
\begin{bmatrix}
  x' \\
y' \\
z'
\end{bmatrix} =
\begin{bmatrix}
  d_1 \frac{x}{z} \\
  d_2 \frac{y}{z} \\
  A + \frac{B}{z}
\end{bmatrix}
\]

- Not a linear equation
  - not an affine transformation
  - doesn’t preserve angles-but *does* preserve straight lines
  - Note: it will blow up if z=0 (object at the eye)
- Z maps to *pseudo-distance*
  - necessary to preserve straight lines
  - maintains depth order when B<0: if \( z_1 < z_2 \) then \( z'_1 < z'_2 \)
- We’ll come up with values for \( d_1, d_2, A, \) and \( B, \) in a little while
  - will choose them to keep area of interest within -1 to 1 in x,y,z
- Ugly formula. Make it work with homogeneous matrices…
Homogeneous Perspective Projection

- The homogeneous perspective projection matrix. Notice the last row!

\[
P = \begin{bmatrix}
  d_1 & 0 & 0 & 0 \\
  0 & d_2 & 0 & 0 \\
  0 & 0 & A & B \\
  0 & 0 & 1 & 0
\end{bmatrix}
\]

- Multiply it by a homogeneous point

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} = P \begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} = \begin{bmatrix}
d_1x + 0 + 0 + 0 \\
0 + d_2y + 0 + 0 \\
0 + 0 + Az + B \\
0 + 0 + z + 0
\end{bmatrix} = \begin{bmatrix}
d_1x \\
d_2y \\
Az + B \\
z
\end{bmatrix}
\]

- Notice that the result doesn’t have \( w=1 \). So divide by \( w \):

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} \Rightarrow \begin{bmatrix}
x' / w' \\
y' / w' \\
z' / w' \\
w' / w'
\end{bmatrix} = \begin{bmatrix}
d_1x / z \\
d_2x / z \\
A + B / z \\
1
\end{bmatrix}
\]
Homogeneous Perspective Transform

- As always, there’s some deep math behind this…
  - 3D projective space
- For practical purposes:
  - Use homogeneous matrices normally
  - Modeling & viewing transformations use *affine matrices*
    - points keep $w=1$
    - no need to divide by $w$ when doing modeling operations or transforming into camera space
  - Projection transform uses *perspective matrices*
    - $w$ not always 1
    - divide by $w$ after performing projection transform
    - AKA *perspective divide, homogeneous divide*
- GPU hardware does this
Orthographic projection

- Simple: $x' = x$, $y' = y$, $z' = z$
- For scaling purposes, we’ll introduce $d_1$, $d_2$, A, and B
  - Again, will choose them so that region of interest is in -1 to 1
Homogeneous Orthographic Projection

- The orthographic projection matrix is:

\[
P = \begin{bmatrix}
  d_1 & 0 & 0 & 0 \\
  0 & d_2 & 0 & 0 \\
  0 & 0 & A & B \\
  0 & 0 & 0 & 1 
\end{bmatrix}
\]

- This is an ordinary affine transform (scale+translate)
  - when transforming a point, keeps \( w=1 \)
  - no need to divide by \( w \)
  - but… no harm done dividing by \( w \)

- Send the GPU either an orthonormal or perspective projection matrix
  - it can divide by \( w \) in either case
  - don’t need to special-case the two types of projections
Outline for today

- Camera Space
- Projection
- View Volumes
- Culling
- Bounding Hierarchy
View Volume

- A 3D shape in world space that represents the volume viewable by the camera
Perspective view volume

- A perspective camera with a rectangular image describes a pyramid in space
  - The tip of the pyramid is at the eye point
  - The pyramid projects outward in front of the camera into space
  - Nominally the pyramid starts at the eye point and goes out infinitely…
  - But, to avoid divide-by-zero problems for objects close to the camera
    - introduce a *near clipping plane*
    - objects closer than that are not shown
    - chops off the tip of the pyramid
  - Also, to avoid floating-point precision problems in the Z buffer
    - introduce a *far clipping plane*
    - objects beyond that are not shown
    - defines the bottom of the pyramid
- A pyramid with the tip cut off is a truncated pyramid, AKA a *frustum*
- The standard perspective view volume is called the *view frustum*
View Frustum

Parameterized by:
• left, right, top, bottom (generally symmetric)
• near, far
Or, when symmetric, by:
• Field of view (FOV), aspect ratio
• near, far
• Aspect ratio is the x/y ratio of the final displayed image. Common values:
  • 4/3 for TV & old movies; 1.66 for cartoons & European movies; 16/9 for American movies & HDTV; 2.35 for epic movies

\[
\text{aspect ratio} = \frac{\text{right} - \text{left}}{\text{top} - \text{bottom}} = \frac{\text{right}}{\text{top}}
\]

\[
\tan\left(\frac{\text{FOV}}{2}\right) = \frac{\text{top}}{\text{near}}
\]
Frustum Projection Matrix

- We can think of the view frustrum as a distorted cube, since it has six faces, each with 4 sides.
- The perspective projection warps this to a cube.
  - Everything inside gets distorted accordingly.
  - By setting the parameters properly, we get the cube to range from -1 to 1 in all dimensions: i.e., normalized view space.

\[
P_{\text{persp}}(\text{FOV}, \text{aspect}, \text{near}, \text{far}) = \begin{bmatrix}
1 & 0 & 0 & 0 \\
\frac{1}{\text{aspect} \cdot \tan(\text{FOV} / 2)} & 0 & 0 & 0 \\
0 & \frac{1}{\tan(\text{FOV} / 2)} & 0 & 0 \\
0 & 0 & \frac{\text{near} + \text{far}}{\text{near} - \text{far}} & 2 \cdot \frac{\text{near} \cdot \text{far}}{\text{near} - \text{far}} \\
0 & 0 & \frac{\text{near} + \text{far}}{\text{near} - \text{far}} & -1 \\
0 & 0 & -1 & 0
\end{bmatrix}
\]
class Perspective(Camera) {
    Vector3 Eye;
    Vector3 Target;
    float FOV, Aspect, NearClip, FarClip;
    Matrix getProjection();
};
Orthographic View Volume

- Parametrized by: right, left, top, bottom, near, far
- Or, if symmetric, by: width, height, near, far
Orthographic Projection Matrix

- Simply translates and scales to transform the view volume to -1 to 1 in all dimensions: normalized view coordinates.

$$P_{ortho}(right, left, top, bottom, near, far) = \begin{bmatrix}
    \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\
    0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\
    0 & 0 & \frac{2}{far - near} & -\frac{far + near}{far - near} \\
    0 & 0 & 0 & 1
\end{bmatrix}$$

$$P_{ortho}(width, height, near, far) = \begin{bmatrix}
    \frac{2}{width} & 0 & 0 & 0 \\
    0 & \frac{2}{height} & 0 & 0 \\
    0 & 0 & \frac{2}{far - near} & \frac{far + near}{far - near} \\
    0 & 0 & 0 & 1
\end{bmatrix}$$
class Orthographic(Camera) {
    Vector3 Eye;
    Vector3 Target;
    float Width, Height, NearClip, FarClip;
    Matrix getProjection();
};
Viewport Transformation

Perspective View Volume

Orthographic View Volume

Perspective Projection

Orthographic Projection

Viewport Transformation
Viewport Transformation

- The final transformation!
  - Takes points from the -1...1 normalized view coordinates, and maps them to the range of pixels $x_0...x_1$, $y_0...y_1$ of the image file (image coordinates) or display (device coordinates).
  - Just a scale and translate:

$$D(x_0, x_1, y_0, y_1) = \begin{bmatrix}
\frac{(x_1 - x_0)}{2} & 0 & 0 & \frac{(x_0 + x_1)}{2} \\
0 & \frac{(y_1 - y_0)}{2} & 0 & \frac{(y_0 + y_1)}{2} \\
0 & 0 & 1/2 & 1/2 \\
0 & 0 & 0 & 1
\end{bmatrix}$$

- The depth value is usually mapped to a 32 bit fixed point value ranging from 0 (near) to 0xffffffff (far)

- (Actually, if the output is to a window in the window system, there’s another transformation, from the image coordinates in the window, to the location on screen where that window ends up. But that’s handled by the window system.)
The complete transform

Composing the modeling matrix $M$, the camera matrix $C$, the projection matrix $P$, and the viewport matrix $D$, we have, for some point $p$:

$$\text{pixel} = (D \ P \ C^{-1} \ M) \ p$$
Spaces in OpenGL

- OpenGL separates it into:
  - MODELVIEW=$C^{-1}M$
    - It’s up to you to compose the camera and model transforms. Typically start with the inverse camera transform, then push and pop model matrix values on top of it. `gluLookAt()` is a utility to do a look-at transformation.
  - PROJECTION=$P$
    - OpenGL provides utility routines to set the projection matrix:
      - `glFrustum()` lets you define the perspective view volume based on coordinates of the frustum
      - `glPerspective()` lets you specify the FOV, aspect, near clip and far clip distances
      - `glOrtho()` lets you specify a orthographic viewing transformation
  - Viewport=$D$
    - OpenGL provides a `glViewport()` routine to set the viewport
Outline for today

- Camera Space
- Projection
- View Volumes
- Culling
- Bounding Hierarchy
Culling

- Culling: doing a trivial reject test to avoid drawing objects that aren’t in the view volume.
  - OpenGL/GPU will do it
  - but you don’t want to waste time sending data that won’t be used

- Test if each object is inside or outside view volume
  - or on the border: clipping (in a few weeks)

- Too much work to test real object
  - Instead, introduce bounding volumes
Bounding Volume

- Simple shape that completely encloses an object
- Generally a box or sphere
- We’ll use spheres:
  - Easiest to work with
  - Though hard to get tight fits
Culling to frustum

- Defined by 6 planes
- Each plane divides space:
  - “outside”
  - “inside”
- Check each object against each plane:
  - entirely outside
  - entirely inside
  - intersecting
- If it’s completely “outside” any plane it’s outside the frustum
- If it’s completely “inside” all planes it’s inside the frustum
- Else it’s partly inside and partly out
Remember: Distance to Plane

- (from class 3)
- A plane is described by a point $p$ on the plane and a unit normal $\hat{n}$. Find the (perpendicular) distance from point $x$ to the plane
Remember: Distance to Plane

- The distance is the length of the projection of $\vec{x} - \vec{p}$ onto $\vec{n}$:

$$dist = (\vec{x} - \vec{p}) \cdot \vec{n}$$
The distance has a sign:
- positive on the side of the plane the normal points to
- negative on the opposite side
- (0 exactly on the plane)

This is true for all of space:
- Divides all of space into two infinite half-spaces

\[ \text{dist}(x) = (x - p) \cdot \vec{n} \]
Simplify: Distance to Plane

- Instead of considering \( \mathbf{x} \) and \( \mathbf{p} \) as points, use the vectors from the origin of whatever space we’re in. Then we have:

\[
\text{dist}(\bar{\mathbf{x}}) = (\bar{\mathbf{x}} - \bar{\mathbf{p}}) \cdot \mathbf{n}
\]

\[
= \bar{\mathbf{x}} \cdot \mathbf{n} - \bar{\mathbf{p}} \cdot \mathbf{n}
\]

\[
\text{dist}(\bar{\mathbf{x}}) = \bar{\mathbf{x}} \cdot \mathbf{n} - d
\]

where:

\[
d = \bar{\mathbf{p}} \cdot \mathbf{n}
\]

- The number \( d \) is independent of which \( \mathbf{p} \) we chose.
  - \( d \) is actually equal to the distance from the origin to the plane, along the normal.

- We can represent a plane with just \( d \) and \( \mathbf{n} \)
Frustum with signed planes

- Normal of each plane points outside
  - “outside” means positive distance
  - “inside” means negative distance
Test sphere and plane

- For sphere with radius $r$ and origin $\mathbf{x}$, test the distance to the origin, and see if it’s beyond the radius. Three cases:
  - $dist(\mathbf{x}) > r$
    - completely above
  - $dist(\mathbf{x}) < -r$
    - completely below
  - $-r < dist(\mathbf{x}) < r$
    - intersects
Summary of Sphere testing Algorithm

- Precompute the normal \( \mathbf{n} \) and value \( d \) for each of the six planes.
  - Remember, can get normal by taking cross-product of two edges of a triangle: any three corner points of a frustum face define a triangle.
  - Can use any of the corner points as \( \mathbf{p} \) in order to compute \( d \)
- Given a sphere with center \( \mathbf{x} \) and radius \( r \)
- For each plane:
  - if \( \text{dist}(\mathbf{x}) > r \): sphere is outside! (no need to continue loop)
  - add 1 to count if \( \text{dist}(\mathbf{x}) \leq -r \)
- If we made it through the loop, check the count:
  - if the count is 6, the sphere is completely inside
  - otherwise the sphere intersects the frustum
  - (can use a flag instead of a count)
Culling Orthographic View Volume

- The orthographic view volume is also defined by 6 planes. Use the same algorithm!
Outline for today

- Camera Space
- Projection
- View Volumes
- Culling
- *Bounding Hierarchy*
If an object is big and complex, it’s possible that only parts of it will be in view.

Or if we have groups of objects, it’s possible that entire groups will be out of view.

- Want to be able to cull the whole group quickly
- But if the group is partly in and partly out, want to be able to cull individual objects.
Hierarchical bounding volumes

- If you have a hierarchy of objects or parts, make a hierarchy of bounding volumes
  - The bounding volume of each node encloses the bounding volumes of all its children
- Start by testing the outermost bounding volume
  - If it’s entirely out, don’t draw the group at all
  - If it’s entirely in, draw the whole group
Hierarchical culling

- If the outermost bounding volume is partly inside and partly outside the view volume:
  - Test each child’s bounding volume individually
  - If the child is in, draw it; if it’s out cull it; if it’s partly in and partly out, recurse.
  - If recursion reaches a leaf node, draw it normally
Done.

- Project 4 will implement projection matrices and view volume culling. (goes out next Tuesday, after project 3 comes in--we’re now officially on track.)
- Next week: curves and curved surfaces.