Announcements
Movie from Assignment 1
Grades out soon

3D Viewing & Clipping

Where do geometries come from?
• Pin-hole camera
• Perspective projection
• Viewing transformation
• Clipping lines & polygons

Watt 5.2 and 6.1
COMPUTER GRAPHICS
15-462

Where do geometries come from?
• Build them with 3D modelers
• Digitize or scan them
• Results of simulation/physically based modeling
• Combinations:
  – Edit a digitized model
  – Simplify a scanned model
  – “Evolve” a model
• Often, need multiple models at different complexity

Getting Geometry on the Screen
Given geometry in the world coordinate system, how do we get it to the display?
• Transform to camera coordinate system
• Transform (warp) into canonical view volume
• Clip
• Project to display coordinates
• (Rasterize)

Viewing and Projection
• Our eyes collapse 3-D world to 2-D retinal image (brain then has to reconstruct 3D)
• In CG, this process occurs by projection
• Projection has two parts:
  – Viewing transformations: camera position and direction
  – Perspective/orthographic transformation: reduces 3-D to 2-D
• Use homogeneous transformations
• As you learned in Assignment 1, camera can be animated by changing these transformations—the root of the hierarchy

Pinhole Optics
• Stand at point P, and look through the hole - anything within the cone is visible, and nothing else is
• Reduces the hole to a point - the cone becomes a ray
• Pin hole is the focal point, eye point or center of projection.
**Perspective Projection of a Point**

- **View plane or image plane** - a plane behind the pinhole on which the image is formed
  - A point $I$ sees anything on the line (ray) through the pinhole $F$
  - A point $W$ projects along the ray through $F$ to appear at $I$ (intersection of WF with image plane)

**Problems with Pinholes**

- Correct optics requires infinitely small pinhole
  - No light gets through
  - Diffraction
- Solution: Lens with finite aperture

**Image Formation**

- Projecting a shape
  - Project each point onto the image plane
  - Lines are projected by projecting end points only

**Orthographic Projection**

- When the focal point is at infinity, the rays are parallel and orthogonal to the image plane
- Good model for telephoto lens. No perspective effects.
- When $xy$-plane is the image plane, $(x,y,z) \rightarrow (x,y,0)$

**A Simple Perspective Camera**

- Canonical case:
  - Camera looks along the $z$-axis
  - Focal point is the origin
  - Image plane is parallel to the $xy$-plane at distance $d$
  - $\text{Lens Law}: \frac{1}{v} = \frac{1}{u} + \frac{1}{f}$

**Similar Triangles**

- $\nu_{up}$: a vector that is pointing straight up in the image; usually want world “up” direction
- Diagram shows $y$-coordinate, $x$-coordinate is similar
- Using similar triangles:
  - $\text{point } [x,y,z]$ projects to $(d\nu_{x}, d\nu_{y}, d)$
**A Perspective Projection Matrix**

- Projection using homogeneous coordinates:
  - transform \([x, y, z]\) to \([d/zx, d/zy, d]\)

\[
\begin{bmatrix}
0 & 0 & 0 & x \\
0 & 0 & 0 & y \\
0 & 0 & 0 & z \\
0 & 0 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
dx \\
dy \\
dz \\
1
\end{bmatrix}
\]

- Divide by 4th coordinate (the "w" coordinate)

- 2-D image point:
  - discard third coordinate
  - apply viewport transformation to obtain physical pixel coordinates

**Wait, there’s more!**

Perspective transformation can also

- map rectangle in the image plane to the viewport
- specify near and far clipping planes
  - instead of mapping \(z\) to \(d\), transform \(z\) between \(z_{\text{near}}\) and \(z_{\text{far}}\) on to a fixed range
  - used for z-buffer hidden surface removal
- specify field-of-view (fov) angle

**The View Volume**

- Pyramid in space defined by focal point and window in the image plane (assume window mapped to viewport)
- Defines visible region of space
- Pyramid edges are clipping planes
- Frustum = truncated pyramid with near and far clipping planes
  - Why near plane? Prevent points behind the camera being seen
  - Why far plane? Allows \(z\) to be scaled to a limited fixed-point value (z-buffering)

**But wait...**

- What if we want the camera somewhere other than the canonical location?
- Alternative #1: derive a general projection matrix. (hard)
- Alternative #2: transform the world so that the camera is in canonical position and orientation (much simpler)
- These transformations are viewing transformations
- They can be specified in many ways - some more sensible than others (beware of Foley, Angel and Watt are ok)

**Camera Control Values**

- All we need is a single translation and angle-axis rotation (orientation), but...
- Good animation requires good camera control—we need better control knobs
- Translation knob - move to the \(\text{lookfrom}\) point
- Orientation can be specified in several ways:
  - specify camera rotations
  - specify a \(\text{lookat}\) point (solve for camera rotations)

**A Popular View Specification Approach**

- Focal length, image size/shape and clipping planes are in the perspective transformation
- In addition:
  - \(\text{lookfrom}:\) where the focal point (camera) is
  - \(\text{lookat}:\) the world point to be centered in the image
- Also specify camera orientation about the \(\text{lookat-lookfrom}\) axis
**Implementation**

Implementing the lookat/lookfrom/vup viewing scheme

1. Translate by -lookfrom, bring focal point to origin
2. Rotate lookat-lookfrom to the z-axis with matrix R:
   - rotation axis: \( a = (v \times z)/|v \times z| \)
   - rotation angle: \( \cos \theta = v \cdot z \) and \( \sin \theta = |v \times z| \)
   \( \text{glRotate}(\theta, a_x, a_y, a_z) \)
3. Rotate about z-axis to get vup parallel to the y-axis

**The Whole Picture**

- **LOOKFROM**: Where the camera is
- **LOOKAT**: A point that should be centered in the image
- **VUP**: A vector that will be pointing straight up in the image
- **FOV**: Field-of-view angle
- **d**: Focal length
- **WORLD COORDINATES**

**Virtual Trackballs**

- Imagine world contained in crystal ball, rotates about center
- Spin the ball (and the world) with the mouse
- Given old and new mouse positions
  - project screen points onto the sphere surface
  - rotation axis is normal to plane of points and sphere center
- There are other methods to map screen coordinates to rotations

**Clipping**

- There is something missing between projection and viewing.
- Before projecting, we need to eliminate the portion of scene that is outside the viewing frustum.
- Need to clip objects to the frustum (truncated pyramid)

**Normalizing the Viewing Frustum**

- Solution: transform frustum to a cube before clipping
- Converts perspective frustum to orthographic frustum
- This is yet another homogeneous transform!
The Normalized Frustum

- OpenGL uses \(-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 1\)
- But it doesn’t really matter… we can clip against any such cube.
  - Or, we can translate normalizing transformations by applying the appropriate transforms.
- Must clip in homogeneous coordinates:
  \(w > 0: -w \leq x \leq w, -w \leq y \leq w, -w \leq z \leq w\)
  \(w < 0: -w \geq x \geq w, -w \geq y \geq w, -w \geq z \geq w\)

But wait! Divide by zero?

- But doesn’t projection require dividing by the \(z\) coordinate? If \(-1 \leq z \leq 1\), won’t we get divide by 0?
  - Ah, but it’s really the \(w\) coordinate we divide by, and it’s positive definite!
    - The original perspective transformation puts a vertex’s \(z\) value in \(w\)
    - Since \(hither <= z <= yon\) for vertices that don’t get clipped, \(w\) is positive definite (modulo sign convention for \(hither\) and \(yon\))
  - Hence, no worries on that front. All the \(z=0\) vertices will get clipped before we divide out the homogeneous coordinate.

Clipping to a Cube

- Determine which parts of the scene lie within cube
- We will consider the 2D version: clip to rectangle
  - This has its own uses (viewport clipping)
- Two approaches:
  - clip during scan conversion (rasterization) - check per pixel or end-point
  - clip before scan conversion
  - We will cover
    - clip to rectangular viewport before scan conversion

Line Clipping

- Modify endpoints of lines to lie in rectangle
- How to define “interior” of rectangle?
  - Convenient definition: intersection of 4 half-planes
    - Nice way to decompose the problem
    - Generalizes easily to 3D (intersection of 6 half-planes)

Line Clipping

- Modify endpoints of lines to lie in rectangle
  - Convenient definition: intersection of 4 half-planes
    - Nice way to decompose the problem
    - Generalizes easily to 3D (intersection of 6 half-planes)

Cohen-Sutherland Algorithm

- Uses outcodes to encode the half-plane tests results
  - bit 1: \(y>y_{\text{max}}\)
  - bit 2: \(y<y_{\text{min}}\)
  - bit 3: \(x>x_{\text{max}}\)
  - bit 4: \(x<x_{\text{min}}\)

- Rules:
  - Trivial accept: outcode(end1) and outcode(end2) both zero
  - Trivial reject: outcode(end1) & (bitwise and) outcode(end2) nonzero
  - Else subdivide

More algebraically “nuanced” special cases:
- One inside: find intersection and clip
- Both outside: either clip or reject (tacky case)
Cohen-Sutherland Algorithm

- Uses outcodes to encode the half-plane tests results

\[
\begin{array}{cccc}
0101 & 0100 & 0110 & 0010 \\
\text{bit 1: } y > \text{ymax} & \text{bit 2: } y < \text{ymin} & \text{bit 3: } x > \text{xmax} & \text{bit 4: } x < \text{xmin} \\
0001 & 0000 & 0011 & 1010 \\
\text{Trivial accept: outcode(end1) and outcode(end2) both zero} & \text{Trivial reject: outcode(end1) & (bitwise and) outcode(end2) nonzero} & \text{else subdivision}
\end{array}
\]

Cohen-Sutherland Algorithm: Subdivision

- If neither trivial accept nor reject:
  - Pick an outside endpoint (with nonzero outcode)
  - Pick an edge that is crossed (nonzero bit of outcode)
  - Find line’s intersection with that edge
  - Replace outside endpoint with intersection point
  - Repeat until trivial accept or reject

Sutherland-Hodgman Polygon Clipping Algorithm

- Subproblem:
  - clip a polygon (vertex list) against a single clip plane
  - output the vertex list(s) for the resulting clipped polygon(s)

- Clip against all four planes
  - generalizes to 3D (6 planes)
  - generalizes to any convex clip polygon/polyhedron

Sutherland-Hodgman Polygon Clipping Algorithm (Cont.)

To clip vertex list against one half-plane:
- if first vertex is inside - output it
- loop through list testing inside/outside transition - output depends on transition:

  - in-to-in: output vertex
  - in-to-out: no output
  - out-to-in: output intersection

Polyon Clipping

- Convert a polygon into one or more polygons that form the intersection of the original with the clip window

Cleaning Up

- Post-processing is required when clipping creates multiple polygons
- As external vertices are clipped away, one is left with edges running along the boundary of the clip region.
- Sometimes those edges dead-end, hitting a vertex on the boundary and doubling back.
  - Need to prune back those edges
- Sometimes the edges form infinitely-thin bridges between polygons.
  - Need to cut those polygons apart