Visibility
Week 9, Fri 31 Oct 2003

News
• extra office hours
  – Thu 5:30-6:30
  – Friday 11-1:30, 4:30-5:30
  – Mon 10:30-12:30, 1-3
  – (normal lab hours: Thu 12-1, Fri 10-11)
• don’t use graphics remotely!
  – or else console person can’t use graphics
  – reboot if you have this problem
• this week’s labs:
  – picking, texturing details

Representing 3 Rotational DOFs
• 3x3 Matrix (9 DOFs)
  – Rows of matrix define orthogonal axes
• Euler Angles (3 DOFs)
  – Rot x + Rot y + Rot z
• Axis-angle (4 DOFs)
  – Axis of rotation + Rotation amount
• Quaternion (4 DOFs)
  – 4 dimensional complex numbers

Rotation Matrices Won’t Interpolate
• interpolate linearly from +90 to -90 in y

  \[
  \begin{bmatrix}
  0 & 0 & 1 \\
  0 & 1 & 0 \\
  -1 & 0 & 0 \\
  \end{bmatrix}
  \begin{bmatrix}
  0 & 0 & -1 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  \end{bmatrix}
  \]

• halfway through component interpolation

  \[
  \begin{bmatrix}
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 0 \\
  \end{bmatrix}
  \]

  – problem 1: not a rotation matrix anymore!
  • not orthonormal, x flattened out

Euler Angles Have Gimbal Lock
• keep rotation angle for each axis
• problem 2: gimbal lock
  – occurs when two axes are aligned
• second and third rotations have effect of transforming earlier rotations
  – if Rot y = 90 degrees, Rot z == -Rot x
**Gimbal Lock**

http://www.anticz.com/eularqua.htm

---

**Axis-angle Won’t Concatenate**

A counter-clockwise (right-handed) rotation $\theta$ about the axis specified by the unit vector $\mathbf{n} = (n_x, n_y, n_z)$

- problem 3
  - no easy way to determine how to concatenate

---

**Quaternions**

- quaternion is a 4-D unit vector $\mathbf{q} = [x \ y \ z \ w]$
  - lies on the unit hypersphere $x^2 + y^2 + z^2 + w^2 = 1$
- for rotation about (unit) axis $\mathbf{v}$ by angle $\theta$
  - vector part = $(\sin \theta/2) \mathbf{v} = [x \ y \ z]$
  - scalar part = $(\cos \theta/2) = w$
- rotation matrix
  
  $\begin{bmatrix}
  1-2y^2-2z^2 & 2xy+2zw & 2xz-2wy \\
  2xy-2zw & 1-2x^2-2z^2 & 2yz+2wx \\
  2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \\
  \end{bmatrix}$

- quaternion multiplication $\mathbf{q}_1 \ast \mathbf{q}_2 = [\mathbf{v}_1, w_1] \ast [\mathbf{v}_2, w_2] = [(w_1\mathbf{v}_2+\mathbf{v}_2\mathbf{v}_1+\mathbf{v}_1 \times \mathbf{v}_2), w_1w_2-\mathbf{v}_1 \cdot \mathbf{v}_2]$

---

**Rotation Methods Summary**

- 3x3 matrices
  - good: simple, bad: drifting, can’t interpolate
- Euler angles
  - good: can interpolate, no drift
  - bad: gimbal lock
- axis-angle
  - good: no gimbal lock, can interpolate
  - bad: can’t concatenate
- quaternions
  - good: solve all problems, bad: complex

---

**Rendering Pipeline**

- modeling transformations
- viewing transformations
- projection transformations
- clipping
- scan conversion
- lighting
- shading
- we now know everything about how to draw a polygon on the screen, except visible surface determination
Invisible Primitives

• why might a polygon be invisible?
  – polygon outside the field of view / frustum
  – polygon is backfacing
  – polygon is occluded by object(s) nearer the viewpoint

• for efficiency reasons, we want to avoid spending work on polygons outside field of view or backfacing
• for efficiency and correctness reasons, we need to know when polygons are occluded

View Frustum Clipping

• remove polygons entirely outside frustum
  – note that this includes polygons “behind” eye (actually behind near plane)
• pass through polygons entirely inside frustum
• modify remaining polygons to include only portions intersecting view frustum

Back-Face Culling

• most objects in scene are typically “solid”
• rigorously: orientable closed manifolds
  – orientable: must have two distinct sides
    • cannot self-intersect
    • a sphere is orientable since has two sides, ‘inside’ and ‘outside’.
    • a Mobius strip or a Klein bottle is not orientable
  – closed: cannot “walk” from one side to the other
    • sphere is closed manifold
    • plane is not

Back-Face Culling

• most objects in scene are typically “solid”
• rigorously: orientable closed manifolds
  – manifold: local neighborhood of all points isomorphic to disc
  – boundary partitions space into interior & exterior

Manifold

• examples of manifold objects:
  – sphere
  – torus
  – well-formed CAD part

Back-Face Culling

• examples of non-manifold objects:
  – a single polygon
  – a terrain or height field
  – polyhedron w/ missing face
  – anything with cracks or holes in boundary
  – one-polygon thick lampshade
Back-Face Culling
- on the surface of a closed manifold, polygons whose normals point away from the camera are always occluded:

note: backface culling alone doesn’t solve the hidden-surface problem!

Back-Face Culling
- not rendering backfacing polygons improves performance
  - by how much?
  - reduces by about half the number of polygons to be considered for each pixel

Back-face Culling: VCS
- first idea: cull if $N_z < 0$
  - works, but sometimes misses polygons that should be culled
- better idea: cull if eye is below polygon plane

Back-face Culling: NDCS
- works to cull if $N_z > 0$

Occlusion
- for most interesting scenes, some polygons overlap
  - to render the correct image, we need to determine which polygons occlude which

Painter’s Algorithm
- simple: render the polygons from back to front, “painting over” previous polygons
  - draw blue, then green, then orange
  - will this work in the general case?
Painter’s Algorithm: Problems

- intersecting polygons present a problem
- even non-intersecting polygons can form a cycle with no valid visibility order:

Analytic Visibility Algorithms

- early visibility algorithms computed the set of visible polygon fragments directly, then rendered the fragments to a display:

Analytic Visibility Algorithms

- what is the minimum worst-case cost of computing the fragments for a scene composed of \( n \) polygons?
- answer: \( O(n^2) \)

Binary Space Partition Trees (1979)

- BSP tree: organize all of space (hence partition) into a binary tree
  - preprocess: overlay a binary tree on objects in the scene
  - runtime: correctly traversing this tree enumerates objects from back to front
  - idea: divide space recursively into half-spaces by choosing splitting planes
    - splitting planes can be arbitrarily oriented

BSP Trees: Objects
BSP Trees: Objects

Rendering BSP Trees

```c
renderBSP (BSPtree *T) {
    BSPtree *near, *far;
    if (eye on left side of T->plane)
        near = T->left; far = T->right;
    else
        near = T->right; far = T->left;
    renderBSP (far);
    if (T is a leaf node)
        renderObject (T);
    renderBSP (near);
}
```
Polygons: BSP Tree Construction

- split along the plane defined by any polygon from scene
- classify all polygons into positive or negative half-space of the plane
  - if a polygon intersects plane, split polygon into two and classify them both
- recurse down the negative half-space
- recurse down the positive half-space

Polygons: BSP Tree Traversal

- query: given a viewpoint, produce an ordered list of (possibly split) polygons from back to front:

  ```c
  BSPnode::Draw(Yec3 viewpt)
    Classify viewpt: in + or - half-space of node->plane? // Call that the "near" half-space */
    farchild->draw(viewpt);
    render node->polygon; /* always on node->plane */
    nearchild->draw(viewpt);
  ```

- intuitively: at each partition, draw the stuff on the farther side, then the polygon on the partition, then the stuff on the nearer side

Discussion: BSP Tree Cons

- no bunnies were harmed in my example
- but what if a splitting plane passes through an object?
  - split the object: give half to each node

Summary: BSP Trees

- pros:
  - simple, elegant scheme
  - only writes to framebuffer (no reads to see if current polygon is in front of previously rendered polygon, i.e., painters algorithm)
    - thus very popular for video games (but getting less so)
- cons:
  - computationally intense preprocess stage
  - restricts algorithm to static scenes
  - slow time to construct tree: O(n log n) to
Warnock's Algorithm (1969)

- elegant scheme based on a powerful general approach common in graphics: if the situation is too complex, **subdivide**
  - start with a root viewport and a list of all primitives (polygons)
  - then recursively:
    - clip objects to viewport
    - if number of objects incident to viewport is zero or one, visibility is trivial
    - otherwise, subdivide into smaller viewports, distribute primitives among them, and recurse

Warnock's Algorithm

- pros:
  - very elegant scheme
  - extends to any primitive type
- cons:
  - hard to embed hierarchical schemes in hardware
  - complex scenes usually have small polygons and high depth complexity
    - thus most screen regions come down to the single-pixel case

The Z-Buffer Algorithm

- both BSP trees and Warnock's algorithm were proposed when memory was expensive
  - example: first 512x512 framebuffer > $50,000!
- Ed Catmull (mid-70s) proposed a radical new approach called z-buffering.
  - the big idea: resolve visibility independently at each pixel

The Z-Buffer Algorithm

- we know how to rasterize polygons into an image discretized into pixels:
  - what happens if multiple primitives occupy the same pixel on the screen? Which is allowed to paint the pixel?
The Z-Buffer Algorithm

- idea: retain depth (Z in eye coordinates) through projection transform
  - use canonical viewing volumes
  - each vertex has Z coordinate (relative to eye point) intact

Interpolating Z

- edge equations: Z just another planar parameter:
  - \( z = (D - Ax - By) / C \)
  - if walking across scanline by (Dx)
    \( z_{\text{new}} = z_{\text{old}} - (A/C)(Dx) \)
  - total cost:
    - 1 more parameter to increment in inner loop
    - 3x3 matrix multiply for setup
  - edge walking: just interpolate Z along edges and across spans

Z-buffer

- store \((r,g,b,z)\) for each pixel
  - typically 8+8+8+24 bits, can be more
    ```
    for all i, j {
      Depth[i,j] = MAX_DEPTH
      Image[i,j] = BACKGROUND_COLOUR
    }
    for all polygons P {
      for all pixels in P {
        if (z_pixel < Depth[i,j]) {
          Image[i,j] = C_pixel
          Depth[i,j] = z_pixel
        }
      }
    }
    ```

Depth Test Precision

- reminder: projective transformation maps eye-space \( z \) to generic \( z \)-range (NDC)
- simple example:
  \[
  T \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\
  \]
- thus:
  \[
  z_{\text{NDC}} = \frac{a \cdot z_{\text{eye}} + b}{z_{\text{eye}}} = a + \frac{b}{z_{\text{eye}}} \\
  \]

Depth Test Precision

- therefore, depth-buffer essentially stores \( 1/z \), rather than \( z \)!
- this yields precision problems with integer depth buffers:
**Depth Test Precision**
- precision of depth buffer is bad for far objects
- depth fighting: two different depths in eye space get mapped to same depth in framebuffer
  - which object “wins” depends on drawing order and scan-conversion
- gets worse for larger ratios \( f/n \)
  - rule of thumb: \( f/n < 1000 \) for 24 bit depth buffer

**Z-buffer**
- hardware support in graphics cards
- poor for high-depth-complexity scenes
  - need to render all polygons, even if most are invisible

- “jaggies”: pixel staircase along edges

**The A-Buffer**
- antialiased, area-averaged accumulation buffer
  - \( z \)-buffer: one visible surface per pixel
  - \( A \)-buffer: linked list of surfaces
  - data for each surface includes
    - RGB, \( Z \), area-coverage percentage, ...

**The Z-Buffer Algorithm**
- how much memory does the \( Z \)-buffer use?
- does the image rendered depend on the drawing order?
- does the time to render the image depend on the drawing order?
- how does \( Z \)-buffer load scale with visible polygons? with framebuffer resolution?

**Z-Buffer Pros**
- simple!!
- easy to implement in hardware
- polygons can be processed in arbitrary order
- easily handles polygon interpenetration
- enables deferred shading
  - rasterize shading parameters (e.g., surface normal) and only shade final visible fragments

**Z-Buffer Cons**
- lots of memory (e.g., 1280x1024x32 bits)
  - with 16 bits cannot discern millimeter differences in objects at 1 km distance
- Read-Modify-Write in inner loop requires fast memory
- hard to do analytic antialiasing
  - we don’t know which polygon to map pixel back to
- shared edges are handled inconsistently
  - ordering dependent
- hard to simulate translucent polygons
  - we throw away color of polygons behind closest one
**Visibility**

- **object space algorithms**
  - explicitly compute visible portions of polygons
  - painter's algorithm: depth-sorting, BSP trees

- **image space algorithms**
  - operate on pixels or scan-lines
  - visibility resolved to the precision of the display
  - Z-buffer

**Hidden Surface Removal**

- 2 classes of methods
  - **image-space algorithms**
    - perform visibility test for every pixel independently
    - limited to resolution of display
    - performed late in rendering pipeline
  - **object-space algorithms**
    - determine visibility on a polygon level
    - resolution independent
    - early in rendering pipeline (after clipping)
    - expensive