Visibility

Week 7, Wed Feb 23

http://www.ubc.ca/~cs314/Vjan2005
Project 2 Clarification

- you don’t have to support relative and absolute camera motion simultaneously
  - OK to reset the view when you switch between modes
- use ‘m’ to toggle between modes
Reminder: Project Handin

- due 6pm Thursday
- when handing after the deadline, handin has this unfriendly warning message
  - Checking that handin was successful ...
    /cs/csbox/user FAILED to find user a1b2. Your files DO NOT appear to be handed in successfully
  - Do you want to cancel?
- don’t panic
  - go ahead and complete the handin, do not cancel!
  - your submission will be put in the LATE directory
Review: Bilinear Interpolation

- Interpolate quantity along $L$ and $R$ edges, as a function of $y$
- Then interpolate quantity as a function of $x$
Review: Barycentric Coordinates

A weighted combination of vertices

\[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]

\[ \alpha + \beta + \gamma = 1 \]

\[ 0 \leq \alpha, \beta, \gamma \leq 1 \]

\[ a_1 = \frac{c_1}{c_1 + c_2} \cdot \frac{b_1}{b_1 + b_2} \]

\[ a_2 = \frac{c_2}{c_1 + c_2} \cdot \frac{d_2}{d_1 + d_2} + \frac{c_1}{c_1 + c_2} \cdot \frac{b_2}{b_1 + b_2} \]

\[ a_3 = \frac{c_2}{c_1 + c_2} \cdot \frac{d_1}{d_1 + d_2} \]
Review: Clipping

analytically calculating the portions of primitives within the viewport
Review: Clipping Lines To Viewport

- combining trivial accepts/rejects
  - trivially accept lines with both endpoints inside all edges of the viewport
  - trivially reject lines with both endpoints outside the same edge of the viewport
- otherwise, reduce to trivial cases by splitting into two segments
Review: Cohen-Sutherland Line Clipping

outcodes

4 flags encoding position of a point relative to top, bottom, left, and right boundary

\[ \text{OC}(p_1) == 0 \land \text{OC}(p_2) == 0 \] trivial accept

\[ (\text{OC}(p_1) \land \text{OC}(p_2)) != 0 \] trivial reject

\[ \begin{array}{ccc}
0010 & 0000 & 0001 \\
0110 & 0100 & 0101 \\
1010 & 1000 & 1001 \\
\end{array} \]

\( y = y_{\text{max}} \)

\( y = y_{\text{min}} \)
Review: Polygon Clipping

- not just clipping all boundary lines
- may have to introduce new line segments
Review: Sutherland-Hodgeman Clipping

- for each viewport edge
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped

- for each polygon vertex
  - decide what to do based on 4 possibilities
    - is vertex inside or outside?
    - is previous vertex inside or outside?
Review: Sutherland-Hodgeman Clipping

- An edge from \( p[i-1] \) to \( p[i] \) has four cases.
- Decide what to add to output vertex list.

Diagram:

- Inside to outside:
  - \( p[i-1] \): \( p[i] \) output
  - Inside to inside:
  - Outside to outside:
  - Outside to inside:
  - Inside to outside:
    - \( p[i] \) output
  - No output
  - \( i \) output
  - \( p[i-1] \) output
  - \( p[i] \) output
Visibility
FCG Chapter 7
Rendering Pipeline

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer
Covered So Far

- 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
  - solved by clipping
- polygon is backfacing
  - solved by backface culling
- polygon is occluded by object(s) nearer the viewpoint
  - solved by hidden surface removal

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
Backface Culling
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
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 $N_Z < 0$

sometimes
misses polygons that
should be culled

better idea:

cull if eye is below polygon plane
Back-face Culling: NDCS

works to cull if $N_Z > 0$
Hidden Surface Removal
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

\[ \text{what is the minimum worst-case cost of computing the fragments for a scene composed of } n \text{ polygons?} \]

\[ \text{answer: } O(n^2) \]
so, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for hidden surface removal
we’ll talk about two:

- *Binary Space-Partition (BSP) Trees*
  - this time
- *Warnock’s Algorithm*
  - next time
Binary Space Partition Trees (1979)

BSP Tree: partition space with binary tree of planes

- idea: divide space recursively into half-spaces by choosing splitting planes that separate objects in scene
- preprocessing: create binary tree of planes
- runtime: correctly traversing this tree enumerates objects from back to front
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Splitting Objects

no bunnies were harmed in previous example
but what if a splitting plane passes through an object?

split the object; give half to each node
Traversing BSP Trees

- tree creation independent of viewpoint
  - preprocessing step
- tree traversal uses viewpoint
  - runtime, happens for many different viewpoints
- each plane divides world into near and far
  - for given viewpoint, decide which side is near and which is far
    - check which side of plane viewpoint is on
- recursive algorithm
  - recurse on far side
  - draw object
  - recurse on near side
Traversing BSP Trees

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

```
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);
```
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees : Viewpoint A
BSP Trees: Viewpoint B
BSP Trees : Viewpoint B
BSP Trees: Viewpoint B
BSP Trees: Viewpoint B
BSP Trees: Viewpoint B
BSP Trees: Viewpoint B
BSP Trees : Viewpoint B
BSP Trees: Viewpoint B
BSP Tree Traversal: Polygons

- 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.
BSP Demo

useful demo:

http://symbolcraft.com/graphics/bsp
Summary: BSP Trees

pros:
- simple, elegant scheme
- correct version of painter’s algorithm back-to-front rendering approach
- was very popular for video games (but getting less so)

cons:
- slow to construct tree: $O(n \log n)$ to split, sort
- splitting increases polygon count: $O(n^2)$ worst-case
- computationally intense preprocessing stage restricts algorithm to static scenes