Clipping II, Hidden Surfaces I

Week 8, Fri Mar 12

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2010
News

- midterms returned, solutions out
- unscaled average 52, scaled average 62
P1 Hall of Fame: Honorable Mentions

Pierre Jondeau

Shawn Luo

David Roodnick
P1 Hall of Fame: Winner

Sung-Hoo Kim
Correction: Blinn-Phong Model

• variation with better physical interpretation
  • Jim Blinn, 1977
  \[
  I_{out}(x) = I_{in}(x) (k_s (h \cdot n)^{n_{shiny}}); \text{with } h = (l + v)/2
  \]

• \(h\): halfway vector
  • \(h\) must also be explicitly normalized: \(h / |h|\)
  • highlight occurs when \(h\) near \(n\)
Review: Ray Tracing

• issues:
  • generation of rays
  • intersection of rays with geometric primitives
  • geometric transformations
  • lighting and shading
  • efficient data structures so we don’t have to test intersection with every object
Review: Radiosity

- capture indirect diffuse-diffuse light exchange
- model light transport as flow with conservation of energy until convergence
  - view-independent, calculate for whole scene then browse from any viewpoint
- divide surfaces into small patches
- loop: check for light exchange between all pairs
  - form factor: orientation of one patch wrt other patch (n x n matrix)
Review: Subsurface Scattering

- light enters and leaves at different locations on the surface
  - bounces around inside
- technical Academy Award, 2003
  - Jensen, Marschner, Hanrahan
Review: Non-Photorealistic Rendering

• simulate look of hand-drawn sketches or paintings, using digital models

www.red3d.com/cwr/npr/
Review: Non-Photorealistic Shading

- cool-to-warm shading: \( k_w = \frac{1 + \mathbf{n} \cdot \mathbf{l}}{2}, c = k_w c_w + (1 - k_w) c_c \)
- draw silhouettes: if \((\mathbf{e} \cdot \mathbf{n}_0)(\mathbf{e} \cdot \mathbf{n}_1) \leq 0\), \(\mathbf{e}=\)edge-eye vector
- draw creases: if \((\mathbf{n}_0 \cdot \mathbf{n}_1) \leq \text{threshold}\)

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
Cohen-Sutherland Line Clipping

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

- $OC(p1) = 0010$
- $OC(p2) = 0000$
- $OC(p3) = 1001$
Cohen-Sutherland Line Clipping

- assign outcode to each vertex of line to test
  - line segment: \((p_1, p_2)\)
- trivial cases
  - \(\text{OC}(p_1) == 0 \land \text{OC}(p_2) == 0\)
    - both points inside window, thus line segment completely visible (trivial accept)
  - \((\text{OC}(p_1) \land \text{OC}(p_2)) \neq 0\)
    - there is (at least) one boundary for which both points are outside (same flag set in both outcodes)
    - thus line segment completely outside window (trivial reject)
Cohen-Sutherland Line Clipping

- if line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded
- pick an edge that the line crosses (*how?*)
- intersect line with edge (*how?*)
- discard portion on wrong side of edge and assign outcode to new vertex
- apply trivial accept/reject tests; repeat if necessary
Cohen-Sutherland Line Clipping

• if line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded

• pick an edge that the line crosses
  • check against edges in same order each time
    • for example: top, bottom, right, left
Cohen-Sutherland Line Clipping

• intersect line with edge
Cohen-Sutherland Line Clipping

- discard portion on wrong side of edge and assign outcode to new vertex

- apply trivial accept/reject tests and repeat if necessary
Viewport Intersection Code

- \((x_1, y_1), (x_2, y_2)\) intersect vertical edge at \(x_{\text{right}}\)
  - \(y_{\text{intersect}} = y_1 + m(x_{\text{right}} - x_1)\)
  - \(m = (y_2 - y_1)/(x_2 - x_1)\)

- \((x_1, y_1), (x_2, y_2)\) intersect horiz edge at \(y_{\text{bottom}}\)
  - \(x_{\text{intersect}} = x_1 + (y_{\text{bottom}} - y_1)/m\)
  - \(m = (y_2 - y_1)/(x_2 - x_1)\)
Cohen-Sutherland Discussion

- key concepts
  - use opcodes to quickly eliminate/include lines
    - best algorithm when trivial accepts/rejects are common
  - must compute viewport clipping of remaining lines
    - non-trivial clipping cost
    - redundant clipping of some lines
- basic idea, more efficient algorithms exist
Line Clipping in 3D

• approach
  • clip against parallelepiped in NDC
    • after perspective transform
  • means that clipping volume always the same
    • xmin=ymin= -1, xmax=ymax= 1 in OpenGL

• boundary lines become boundary planes
  • but outcodes still work the same way
  • additional front and back clipping plane
    • zmin = -1, zmax = 1 in OpenGL
Polygon Clipping

• objective
  • 2D: clip polygon against rectangular window
    • or general convex polygons
    • extensions for non-convex or general polygons
  • 3D: clip polygon against parallelepiped
Polygon Clipping

- not just clipping all boundary lines
- may have to introduce new line segments
Why Is Clipping Hard?

• what happens to a triangle during clipping?
  • some possible outcomes:
    triangle to triangle
    triangle to quad
    triangle to 5-gon

• how many sides can result from a triangle?
  • seven
Why Is Clipping Hard?

- a really tough case:

concave polygon to multiple polygons
Polygon Clipping

• classes of polygons
  • triangles
  • convex
  • concave
  • holes and self-intersection
Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Clipping

• basic idea:
  • consider each edge of the viewport individually
  • clip the polygon against the edge equation
  • after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Clipping

• basic idea:
  • consider each edge of the viewport individually
  • clip the polygon against the edge equation
  • after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Clipping

• basic idea:
  • consider each edge of the viewport individually
  • clip the polygon against the edge equation
  • after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped
Sutherland-Hodgeman Algorithm

- input/output for whole algorithm
  - input: list of polygon vertices in order
  - output: list of clipped polygon vertices consisting of old vertices (maybe) and new vertices (maybe)
- input/output for each step
  - input: list of vertices
  - output: list of vertices, possibly with changes
- basic routine
  - go around polygon one vertex at a time
  - decide what to do based on 4 possibilities
    - is vertex inside or outside?
    - is previous vertex inside or outside?
Clipping Against One Edge

- \( p[i] \) inside: 2 cases

<table>
<thead>
<tr>
<th>Inside</th>
<th>Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p[i-1] )</td>
<td>( p[i] )</td>
</tr>
</tbody>
</table>
| output: \( p[i] \) | output: \( p, p[i] \)
Clipping Against One Edge

- $p[i]$ outside: 2 cases

```
<table>
<thead>
<tr>
<th>inside</th>
<th>outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p[i-1]$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p[i]$</td>
<td>$p[i]$</td>
</tr>
</tbody>
</table>

output: $p$
```

```
<table>
<thead>
<tr>
<th>inside</th>
<th>outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p[i]$</td>
<td>$p[i-1]$</td>
</tr>
</tbody>
</table>

output: nothing
```
Clipping Against One Edge

clipPolygonToEdge( p[n], edge ) {
    for( i = 0 ; i< n ; i++ ) {
        if( p[i] inside edge ) {
            if( p[i-1] inside edge ) output p[i];  // p[-1]= p[n-1]
            else {
                p= intersect( p[i-1], p[i], edge ); output p, p[i];
            }
        } else {                                     // p[i] is outside edge
            if( p[i-1] inside edge ) {
                p= intersect(p[i-1], p[i], edge ); output p;
            }
        }
    }
}
Sutherland-Hodgeman Example
Sutherland-Hodgeman Discussion

• similar to Cohen/Sutherland line clipping
  • inside/outside tests: outcodes
  • intersection of line segment with edge: window-edge coordinates
• clipping against individual edges independent
  • great for hardware (pipelining)
  • all vertices required in memory at same time
    • not so good, but unavoidable
    • another reason for using triangles only in hardware rendering
Hidden Surface Removal
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: $O(n^2)$
Analytic Visibility Algorithms

• 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 one:
  • *Binary Space Partition (BSP) Trees*
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

Ouch
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 independently for each tree vertex
    - tree traversal differs depending on viewpoint!
- 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

- decide independently at each tree vertex
- not just left or right child!
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 A
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
Clarification: BSP Demo

• order of insertion can affect half-plane extent
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