Texturing, Clipping
Week 9, Mon 27 Oct 2003
Reading

• Chapter 7.1-7.10: texturing
• Chapter 8.3-8.7: clipping

• bump mapping extra reading
Texture Mapping

- texture map is an image, two-dimensional array of color values (texels)
- texels are specified by texture’s (u,v) space
- at each screen pixel, texel can be used to substitute a polygon’s surface property (color)
- we must map (u,v) space to polygon’s (s, t) space
Example Texture Map

Texture

Object

Mapped Texture

\( \text{glVertex3d} (s, s, s) \)
\( \text{glTexCoord2d}(1,1); \)

\( \text{glVertex3d} (-s, -s, -s) \)
\( \text{glTexCoord2d}(1,1); \)
Texture Coordinate Transforms

```markdown
glVertex3d (s, s, s)  
glTexCoord2d(5, 5);

(5, 0) (5, 5)  
Texture  +  
(5, 0)  
Object  =  
Mapped Texture
```

```markdown
glVertex3d (s, s, s)  
glTexCoord2d(1, 1);

(1, 0) (1, 1)  
Texture  +  
(1, 0)  
Object  =  
Mapped Texture
```
Texture Mapping

(s_0, t_0) -> (s_1, t_1) -> (s_2, t_2)
Texture Mapping and Filtering

• ideal algorithm:
  – given texture map as regular grid of texels, reconstruct continuous texture function using low pass filtering
  – map this continuous texture onto 3D surface
  – project surface onto image plane using model/view and perspective transformation
  – low-pass filter resulting continuous function according to desired image resolution (avoid aliasing)
  – sample filtered continuous image at pixel positions
Texture Mapping and Filtering

• in practice: 2 cases

texture magnification

interpolation

- Texel
- Pixel

texture minification

averaging
Texture Magnification

• synopsis
  – texture appears magnified on screen
  – only need to low-pass filter in texture space
    • that already removes frequencies higher than the Nyquist limit for the final image resolution
  – what filter to use?
    • nearest neighbor: just choose color of closest texel for every pixel
      – worst of all possible choices!
    • linear interpolation: interpolate from the closest samples (2 in 1D texture, 4 for 2D, 8 for 3D)
Texture Minification

• synopsis
  – texture appears reduced in size on screen
  – only need to low-pass filter in image space
    • will also remove all the high frequencies in texture space
  – same filter as magnification case?
    • problem: a lot of texels could fall within the support of the low-pass filter for a single image
      – e.g. when an object is very far away so that it maps to a single pixel in the final image
      – too expensive: have to evaluate filter function at an unbounded number of places and average results!
Texture Minification Filters

• solution: precomputation
  – MIP-Mapping (Multum In Parvo)
    • “many things in a small place”
    • store not one texture image, but whole pyramid
    • resolution from level to level varies by factor of two
      (original resolution … 1x1)
    • every level is correctly filtered for its resolution
Environment Mapping

- used to model an object that reflects surrounding textures to the eye
  - polished sphere reflects walls and ceiling textures
  - cyborg in Terminator 2 reflects flaming destruction
- texture is distorted fish-eye view of environment
- spherical texture mapping creates texture coordinates that correctly index into this texture map
Sphere Mapping
Blinn/Newell Latitude Mapping
Cube Mapping
Cube Mapping – Greene ‘86
Cube Mapping – Greene ‘86

• direction of reflection vector \( r \) selects the face of the cube to be indexed
  – co-ordinate with largest magnitude
    • e.g., the vector \((-0.2, 0.5, -0.84)\) selects the –Z face!

  – remaining two coordinates (normalized by the 3\(^{rd}\) coordinate) selects the pixel from the face.
    • e.g., \((-0.2, 0.5)\) gets mapped to \((0.38, 0.80)\).

• difficulty in interpolating across faces!

• OpenGL support \texttt{GL\_CUBE\_MAP}
Bump Mapping

- image encodes normal change
  - see book, extra reading for full derivation
Embossing

• at transitions
  – rotate point’s surface normal by $\theta$ or $-\theta$
Displacement Mapping

• bump mapped normals are inconsistent with actual geometry.
  – problems: shadows, silhouettes
• displacement mapping actually affects the surface geometry
Next Topic: Clipping

• we’ve been assuming that all primitives (lines, triangles, polygons) lie entirely within the *viewport*
  – in general, this assumption will not hold:
Clipping

• analytically calculating the portions of primitives within the viewport
Why Clip?

• bad idea to rasterize outside of framebuffer bounds
• also, don’t waste time scan converting pixels outside window
  – could be billions of pixels for very close objects!
Line Clipping

• 2D
  – determine portion of line inside an axis-aligned rectangle (screen or window)

• 3D
  – determine portion of line inside axis-aligned parallelepiped (viewing frustum in NDC)
  – simple extension to the 2D algorithms
Clipping

• naïve approach to clipping lines:
  for each line segment
    for each edge of viewport
      find intersection point
      pick “nearest” point
    if anything is left, draw it

• what do we mean by “nearest”?
• how can we optimize this?
Trivial Accepts

• big optimization: trivial accept/rejects
• Q: how can we quickly determine whether a line segment is entirely inside the viewport?
• A: test both endpoints.
Trivial Rejects

• Q: how can we know a line is outside viewport?

• A: if both endpoints on wrong side of same edge, can trivially reject line
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

<table>
<thead>
<tr>
<th></th>
<th>OC(p1)</th>
<th>OC(p2)</th>
<th>OC(p3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td></td>
<td></td>
<td>1001</td>
</tr>
</tbody>
</table>

- 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
  – \(\text{OC}(p_1) == 0 \&\& \text{OC}(p_2) == 0\)
    • both points inside window, thus line segment completely visible (trivial accept)
  – \((\text{OC}(p_1) \& \text{OC}(p_2)) != 0\)
    • 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 (how?)
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 with vertical edge at \(x_{\text{right}}\)
  \[
  y_{\text{intersect}} = y_1 + m(x_{\text{right}} - x_1), \quad m = (y_2 - y_1)/(x_2 - x_1)
  \]

- \((x_1, y_1), (x_2, y_2)\) intersect with horizontal edge at \(y_{\text{bottom}}\)
  \[
  x_{\text{intersect}} = x_1 + (y_{\text{bottom}} - y_1)/m, \quad m = (y_2 - y_1)/(x_2 - x_1)
  \]
Cohen-Sutherland Review

- 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
- more efficient algorithms exist
Line Clipping in 3D

● approach:
  – clip against parallelepiped in NDC
    • after perspective transform
  – means that the clipping volume always the same
    • \( x_{\text{min}} = y_{\text{min}} = -1, \ x_{\text{max}} = y_{\text{max}} = 1 \) in OpenGL

  – boundary lines become boundary planes
    • but outcodes still work the same way
    • additional front and back clipping plane
      \( z_{\text{min}} = -1, \ z_{\text{max}} = 1 \) 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?
• possible outcomes:

  triangle ⇒ triangle

  triangle ⇒ quad

  triangle ⇒ 5-gon

• how many sides can a clipped triangle have?
How Many Sides?

• seven…
Why Is Clipping Hard?

• a really tough case:
Why Is Clipping Hard?

• a really tough case:

concave polygon $\Rightarrow$ 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

![Polygon Clipping Illustration]
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 algorithm:
  – input: list of polygon vertices in order
  – output: list of clipped polygon vertices consisting of old vertices (maybe) and new vertices (maybe)

• note: this is exactly what we expect from the clipping operation against each edge
Sutherland-Hodgeman Clipping

• Sutherland-Hodgeman basic routine:
  – go around polygon one vertex at a time
  – current vertex has position \( p \)
  – previous vertex had position \( s \), and it has been added to the output if appropriate
Polygon Clipping

• clipping against one edge:

\[ \text{clipPolygonToEdge}( p[n], \text{edge} ) \{ \]

\[ \text{for}( i = 0 ; i < n ; i++ ) \{ \]

\[ \text{if}( p[i] \text{ inside edge} ) \{ \]

\[ \text{if}( p[i-1] \text{ inside edge } ) // p[-1]= p[n-1] \]

\[ \text{output } p[i]; \]

\[ \text{else } \{ \]

\[ p = \text{intersect}( p[i-1], p[i], \text{edge} ); \]

\[ \text{output } p, p[i]; \]

\[ \} \]

\[ \} \text{ else…} \]
Polygon Clipping

• clipping against one edge (cont)
  – p[i] inside: 2 cases

\[ \begin{align*}
\text{inside} & \quad \text{outside} \\
\bullet \ p[i-1] & \quad \bullet \ p[i] \\
\bullet \ p[i] & \quad \bullet \ p[i]
\end{align*} \]

Output: \( p[i] \)

\[ \begin{align*}
\text{inside} & \quad \text{outside} \\
\bullet \ p[i] & \quad \bullet \ p[i-1] \\
\bullet \ p & \quad \bullet \ p[i]
\end{align*} \]

Output: \( p, p[i] \)
Polygon Clipping

• clipping against one edge (cont)

... else { // p[i] is outside edge
    if( p[i-1] inside edge ) {
        p = intersect(p[i-1], p[I], edge );
        output p;
    }
} // end of algorithm
Polygon Clipping

- clipping against one edge (cont)
  - p[i] outside: 2 cases

Output: p

Output: nothing
Polygon Clipping

• example
Polygon Clipping

• Sutherland/Hodgeman Algorithm
  – inside/outside tests: outcodes
  – intersection of line segment with edge: window-edge coordinates
  – similar to Cohen/Sutherland algorithm for line clipping
Sutherland/Hodgeman Discussion

- clipping against individual edges independent
  - great for hardware (pipelining)
  - all vertices required in memory at the same time
    - not so good, but unavoidable
    - another reason for using triangles only in hardware rendering
Sutherland/Hodgeman Discussion

• for rendering pipeline:
  – re-triangulate resulting polygon
    (can be done for every individual clipping edge)