Computer Graphics (CS 543)
Lecture 9b: Sphere Maps, Viewport Transformation & Hidden Surface Removal

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Sphere Environment Map

- Cube can be replaced by a sphere (sphere map)
Sphere Mapping

- Original environmental mapping technique
- Proposed by Blinn and Newell
- Map longitude and latitude to texture coordinates
- OpenGL supports sphere mapping
- Requires a circular texture map equivalent to an image taken with a fisheye lens
Sphere Map

- A sphere map is basically a photograph of a reflective sphere in an environment

Paul DeBevec, www.debevec.org
Sphere map

- example

Sphere map (texture)

Sphere map applied on torus
Capturing a Sphere Map
Viewport Transformation
Viewport Transformation

- After projection, clipping, do viewport transformation
- Clipping eliminates lines outside view volume, truncates lines partially in-out
- More on clipping later

User implements in Vertex shader
Manufacturer implements In hardware
Viewport Transformation

- Maps CVV \((x, y)\) -> screen \((x, y)\) coordinates

```cpp
glViewport(x, y, width, height)
```
Viewport Transformation: What of z?

- Also maps z (pseudo-depth) from [-1,1] to [0,1]
- [0,1] pseudo-depth stored in depth buffer,
  - Used for Depth testing (Hidden Surface Removal)
Hidden Surface Removal
Rasterization

- Rasterization Determines what pixels to color to draw a shape
  - Generates set of fragments
- Implemented by graphics hardware
- Rasterization algorithms for primitives (e.g., lines, circles, triangles, polygons)

Rasterization: Determine Pixels (fragments) each primitive covers
Hidden surface Removal

- Drawing polygonal faces on screen consumes CPU cycles
- User cannot see every surface in scene
- To save time, draw only surfaces we see
- Methods to eliminate surfaces we cannot see?

1. Occluded surfaces: hidden surface removal (visibility)
2. Back faces: back face culling
Hidden surface Removal

- Surfaces we cannot see and elimination methods:
  - 3. Faces outside view volume: viewing frustrum culling

Classes of HSR techniques:
- **Object space techniques**: applied before rasterization
- **Image space techniques**: applied after vertices have been rasterized
Visibility (hidden surface removal)

- Overlapping opaque polygons
- **Correct visibility?** Draw only the closest polygon
  - (remove occluded/hidden surfaces)
Image Space Approach

- Start from pixel, work backwards into the scene
- Through each pixel, \((nm\text{ for an } n \times m\text{ frame buffer})\) find closest of \(k\) polygons
- Complexity \(O(nmk)\)
- Examples:
  - Ray tracing
  - z-buffer : OpenGL
OpenGL - Image Space Approach

- Paint pixel with color of closest object

```plaintext
for (each pixel in image) {
    determine the object closest to the pixel
    draw the pixel using the object’s color
}
```
Z buffer Illustration

Correct Final image

Top View

eye

Z = 0.5
Z = 0.3
Z buffer Illustration

**Step 1:** Initialize the depth buffer

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Largest possible z values is 1.0
Z buffer Illustration

Step 2: Draw blue polygon
(actually order does not affect final result)

<table>
<thead>
<tr>
<th>1.0</th>
<th>1.0</th>
<th>1.0</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

1. Determine group of pixels corresponding to blue polygon
2. Figure out z value of blue polygon for each covered pixel (0.5)
3. For each covered pixel, z = 0.5 is less than 1.0
   1. Smallest z so far = 0.5, color = blue
Step 3: Draw the yellow polygon

1. Determine group of pixels corresponding to yellow polygon
2. Figure out z value of yellow polygon for each covered pixel (0.3)
3. For each covered pixel, z = 0.3 becomes minimum, color = yellow

z-buffer drawback: wastes resources drawing and redrawing faces
OpenGL HSR Commands

- 3 main commands to do HSR

- `glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB)` instructs OpenGL to create depth buffer

- `glEnable(GL_DEPTH_TEST)` enables depth testing

- `glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` initializes depth buffer every time we draw a new picture
Z-buffer Algorithm

- Initialize every pixel’s z value to 1.0
- Rasterize every polygon
- For each pixel in polygon, find its z value (interpolate)
- Track smallest z value so far at each pixel
- As we rasterize polygon, for each pixel in polygon
  - If polygon’s z at this pixel < current min z through pixel
  - Paint pixel with polygon’s color

Find depth (z) of every polygon at each pixel
Z (depth) Buffer Algorithm

Depth of polygon being rasterized at pixel \((x, y)\)

Largest depth seen so far
Through pixel \((x, y)\)

For each polygon {

for each pixel \((x,y)\) in polygon area {

if \((z\_polygon\_pixel(x,y) < depth\_buffer(x,y))\) {

\[ depth\_buffer(x,y) = z\_polygon\_pixel(x,y); \]

\[ color\_buffer(x,y) = \text{polygon color at (x,y)} \]

}

}

Note: know depths at vertices. Interpolate for interior \(z\_polygon\_pixel(x, y)\) depths
Combined z-buffer and Gouraud Shading

(Hill Book, 2nd edition, pg 438)

- Can combine shading and HSR through scan line algorithm

```java
for(int y = ybott; y <= ytop; y++) // for each scan line
{
    for(each polygon){
        find xleft and xright
        find dleft, dright, and dinc
        find colorleft, colorright, and colorinc
        for(int x = xleft, c = colorleft, d = dleft; x <= xright;
            x++, c+= colorinc, d+= dinc)
            if(d < d[x][y])
            {
                put c into the pixel at (x, y)
                d[x][y] = d; // update closest depth
            }
    }
}
```
Perspective Transformation: Z-Buffer Depth Compression

- **Pseudodepth calculation**: Recall we chose parameters \((a\) and \(b)\) to map \(z\) from range \([\text{near, far}]\) to \text{pseudodepth} range\([-1,1]\)

  \[
  \begin{bmatrix}
  \frac{2N}{x_{\text{max}} - x_{\text{min}}} & 0 & \frac{\text{right} + \text{left}}{\text{top} + \text{bottom}} & 0 \\
  0 & \frac{2N}{\text{top} - \text{bottom}} & 0 & 0 \\
  0 & 0 & \frac{\text{right} - \text{left}}{\text{top} - \text{bottom}} & 0 \\
  0 & 0 & 0 & -\frac{F + N}{F - N}
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix}
  \]

  These values map \(z\) values of original view volume to \([-1, 1]\) range.
Z-Buffer Depth Compression

- This mapping is almost linear close to eye
- Non-linear further from eye, approaches asymptote
- Also limited number of bits
- Thus, two z values close to far plane may map to same pseudodepth: **Errors!!**

![Diagram showing mapping of z values]

\[
a = -\frac{F+N}{F-N}
\]

\[
b = -\frac{-2FN}{F-N}
\]
Painter’s HSR Algorithm

- Render polygons farthest to nearest
- Similar to painter layers oil paint

Viewer sees B behind A

Render B then A
Depth Sort

- Requires sorting polygons (based on depth)
  - $O(n \log n)$ complexity to sort $n$ polygon depths
  - Not every polygon is clearly in front or behind other polygons

Polygons sorted by distance from COP
Easy Cases

- Case a: A lies behind all polygons
- Case b: Polygons overlap in z but **not** in x or y
Hard Cases

Overlap in (x,y) and z ranges

- cyclic overlap
- penetration
Back Face Culling

- **Back faces:** faces of opaque object that are “pointing away” from viewer
- **Back face culling:** do not draw back faces (saves resources)

How to detect back faces?
Back Face Culling

- Goal: Test if a face F is a backface
- How? Form vectors
  - View vector, V
  - Normal N to face F

Backface test: F is backface if $N \cdot V < 0$ why??
Back Face Culling: Draw mesh front faces

```c
void drawFrontFaces()
{
    for(int f = 0; f < numFaces; f++)
    {
        if(isBackFace(f, ....) continue; // if N.V < 0
        glDrawArrays(GL_POLYGON, 0, N);
    }
}
```
View-Frustum Culling

- **Goal:** Remove objects outside view frustum
- Done by 3D clipping algorithm (e.g. Liang-Barsky)
Ray Tracing

- Ray tracing is another image space method.
- Ray tracing: Cast a ray from eye through each pixel into world.
- Ray tracing algorithm figures out: what object seen in direction through given pixel?

Overview later
References