Welcome!
Fastest Ray Tracer:
Today's Agenda:

- Depth Sorting
- Clipping
- Visibility
Perspective

Scenegraph

- camera
  - $T_{\text{camera}}$
  - world
    - $T_{\text{car}1}$
    - $T_{\text{plane}1}$
    - $T_{\text{car}2}$
    - $T_{\text{plane}2}$
      - car
        - wheel
        - wheel
        - zone
        - wheel
        - turret
        - dude
      - plane
        - wheel
        - wheel
        - wheel
        - turret
        - dude
      - car
        - wheel
        - wheel
        - wheel
        - turret
        - dude
      - plane
        - wheel
        - wheel
        - wheel
        - turret
        - dude
      - buggy
        - wheel
        - wheel
        - wheel
        - dude
The world according to the camera:

Camera space
Perspective

Transformation Pipeline

World space $\Rightarrow$ camera space $\Rightarrow$ orthographic view $\Rightarrow$ canonical view

$I \times M_{\text{camera}} \times M_{\text{ortho}} \times M_{\text{canonical}}$

These can be collapsed into a single $4 \times 4$ matrix.
Today’s Agenda:

- Depth Sorting
- Clipping
- Visibility
Depth Sorting

Rendering – Functional overview

1. Transform: translating / rotating meshes

2. Project: calculating 2D screen positions

3. Rasterize: determining affected pixels

4. Shade: calculate color per affected pixel
3. Rasterize: *determining affected pixels*

Questions:

- What is the screen space position of the fragment?
- Is that position actually on-screen?
- Is the fragment the nearest fragment for the affected pixel?

How do we efficiently determine *visibility* of a pixel?
Part of the tree is off-screen
Too far away to draw
Tree requires little detail
City obscured by tree
Torso closer than ground
Tree between ground & sun
Old-skool depth sorting: Painter’s Algorithm

- Sort polygons by depth
- Based on polygon center
- Render depth-first

Advantage:

- Doesn’t require z-buffer

Problems:

- Cost of sorting
- Doesn’t handle all cases
- Overdraw
Depth Sorting

Overdraw:

Inefficiency caused by drawing multiple times to the same pixel.
Depth Sorting

Overdraw:

Inefficiency caused by drawing multiple times to the same pixel.
Depth Sorting

Overdraw:

Inefficiency caused by drawing multiple times to the same pixel.
Depth Sorting

Correct order: BSP
Depth Sorting

Correct order: BSP

![Diagram of BSP tree with nodes labeled as correct order: root, front, back.]

```swift
// code snippet...
```

INFOGR – Lecture 12 – “Visibility”
Depth Sorting

Correct order: BSP

```
x = inside / 3
if x = nt / in, do
  if N = 1 (nt = 0): 
    2021 = 1
    n = 1
    E = diffuse,
    n = true;

  if n = r / int
    if E = diffuse,
      n = true;

  if n = int
    if E = diffuse,
      n = true;

(WADDEPTH)
```

Survive = SurviveProbability(diffuse,
exist = doing it properly, chances = 4
if radiance = SampleLight('Brand, I, A, Alignment, 
x = radiance.y = radiance.z) > 0) && (cost < M
```
INFOGR – Lecture 12 – “Visibility”

Depth Sorting

Correct order: BSP

root

front

back

---

Survive = Survivability; diffuse,
extension = doing it properly, (shading)
Radiance = SampleLight
reflect = E = dot(R, R) / (pdf)

Depth Sorting

Correct order: BSP
Depth Sorting

Correct order: BSP
Depth Sorting

Correct order: BSP

Sorting by BSP traversal:
1. Render far side of plane
2. Render near side of plane
Depth Sorting

Draw order using a BSP:

- Guaranteed to be correct (hard cases result in polygon splits)
- No sorting required, just a tree traversal

But:

- Requires construction of BSP: not suitable for dynamic objects
- Does not eliminate overdraw
Z-buffer

A z-buffer stores, per screen pixel, a depth value. The depth of each fragment is checked against this value:

- If the fragment is further away, it is discarded
- Otherwise, it is drawn, and the z-buffer is updated.

The z-buffer requires:

- An additional buffer
- Initialization of the buffer to \( z_{\text{max}} \)
- Interpolation of \( z \) over the triangle
- A z-buffer read and compare, and possibly a write.
Depth Sorting

Z-buffer

What is the best representation for depth in a z-buffer?

1. Interpolated z (convenient, intuitive);
2. 1/z (or: $n + f - \frac{f_n}{z}$) (more accurate nearby);
3. $(\text{int})(\frac{2^{31}-1}{z})$;
4. $(\text{uint})(\frac{2^{32}-1}{-z})$;
5. $(\text{uint})(\frac{2^{32}-1}{-z + 1})$.

Note: we use $z_{\text{int}} = \frac{(2^{32}-1)}{-z+1}$.

This way, any $z < 0$ will be in the range $z_{\text{adjusted}} = -z_{\text{original}} + 1 = 1..\infty$, therefore $1/z_{\text{adjusted}}$ will be in the range $0..1$, and thus the integer value we will store uses the full range of $0..2^{32} - 1$.

Here, $z_{\text{int}} = 0$ represents $z_{\text{original}} = 0$, and $z_{\text{int}} = 2^{32} - 1$ represents $z_{\text{original}} = -\infty$.

Even more details:
https://developer.nvidia.com/content/depth-precision-visualized
http://outerra.blogspot.nl/2012/11/maximizing-depth-buffer-range-and.html
Depth Sorting

Z-buffer optimization

In the ideal case, the nearest fragment for a pixel is drawn first:

- This causes all subsequent fragments for the pixel to be discarded;
- This minimizes the number of writes to the frame buffer and z-buffer.

The ideal case can be approached by using Painter's to 'pre-sort'.
‘Z-fighting’:

Occurs when two polygons have almost identical z-values.

Floating point inaccuracies during interpolation will cause unpleasant patterns in the image.
Part of the tree is off-screen

Stuff that is too far to draw

Tree requires little detail

City obscured by tree

√ Torso closer than ground

Tree between ground & sun
Today’s Agenda:

- Depth Sorting
- Clipping
- Visibility
Clipping

Many triangles are partially off-screen. This is handled by *clipping* them.

Sutherland-Hodgeman clipping:

Clip triangle against 1 plane at a time;

Emit *n-gon* (0, 3 or 4 vertices).
Clipping

Sutherland-Hodgeman

Input: list of vertices

Algorithm:

Per edge with vertices \( v_0 \) and \( v_1 \):
- If \( v_0 \) and \( v_1 \) are ‘in’, emit \( v_1 \)
- If \( v_0 \) is ‘in’, but \( v_1 \) is ‘out’, emit C
- If \( v_0 \) is ‘out’, but \( v_1 \) is ‘in’, emit C and \( v_1 \)

where C is the intersection point of the edge and the plane.

Output: list of vertices, defining a convex n-gon.

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex 0</td>
<td>Vertex 1</td>
</tr>
<tr>
<td>Vertex 1</td>
<td>Intersection 1</td>
</tr>
<tr>
<td>Vertex 2</td>
<td>Intersection 2</td>
</tr>
<tr>
<td>Vertex 0</td>
<td></td>
</tr>
</tbody>
</table>
Sutherland-Hodgeman

Calculating the intersections with plane $ax + by + cz + d = 0$:

$$\text{dist}_v = v \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} + d$$

$$f = \frac{|\text{dist}_{v0}|}{|\text{dist}_{v0}| + |\text{dist}_{v1}|}$$

$$I = v_0 + f(v_1 - v_0)$$

After clipping, the input n-gon may have at most 1 extra vertex. We may have to triangulate it:

$0, 1, 2, 3, 4 \Rightarrow 0, 1, 2 + 0, 2, 3 + 0, 3, 4$. 

0,1,2,3,4 ➔ 0, 1, 2 + 0, 2, 3 + 0, 3, 4.
Guard bands

To reduce the number of polygons that need clipping, some hardware uses **guard bands**: an invisible band of pixels outside the screen.

- Polygons outside the screen are discarded, even if they touch the guard band;
- Polygons partially inside, partially in the guard band are drawn without clipping;
- Polygons partially inside the screen, partially outside the guard band are clipped.
Sutherland-Hodgeman

Clipping can be done against arbitrary planes.
Today's Agenda:

- Depth Sorting
- Clipping
- Visibility
Stuff that is too far to draw

- Part of the tree is off-screen
- Torso closer than ground
- City obscured by tree

Tree requires little detail

- Tree between ground & sun
Visibility

Only rendering what’s visible:

“Performance should be determined by visible geometry, not overall world size.”

- Do not render geometry outside the view frustum
- Better: do not process geometry outside frustum
- Do not render occluded geometry
- Do not render anything more detailed than strictly necessary
Visibility

Culling

Observation:
50% of the faces of a cube are not visible.

On average, this is true for all meshes.

Culling ‘backfaces’:

Triangle: \( ax + by + cz + d = 0 \)
Camera: \((x, y, z)\)
Visible: fill in camera position in plane equation.

\( ax + by + cz + d > 0 \): visible.

Cost: 1 dot product per triangle.
Visibility

Culling

Observation:
If the *bounding sphere* of a mesh is outside the view frustum, the mesh is not visible.

But also:
If the *bounding sphere* of a mesh intersects the view frustum, the mesh may be not visible.

View frustum culling is typically a *conservative test*: we sacrifice accuracy for efficiency.

**Cost:** 1 dot product per mesh.
Visibility

Culling

Observation:
If the bounding sphere over a group of bounding spheres is outside the view frustum, a group of meshes is invisible.

We can store a bounding volume hierarchy in the scene graph:

- Leaf nodes store the bounds of the meshes they represent;
- Interior nodes store the bounds over their child nodes.

Cost: 1 dot product per scene graph subtree.
Visibility

Culling

Observation:
If a grid cell is outside the view frustum, the contents of that grid cell are not visible.

Cost: 0 for out-of-range grid cells.
Visibility

Indoor visibility: Portals

Observation: if a window is invisible, the room it links to is invisible.
Visibility

Visibility determination

Coarse:
- Grid-based (typically outdoor)
- Portals (typically indoor)

Finer:
- Frustum culling
- Occlusion culling

Finest:
- Backface culling
- Clipping
- Z-buffer
Today's Agenda:

- Depth Sorting
- Clipping
- Visibility
INFOGR – Computer Graphics

Jacco Bikker & Debabrata Panja - April-July 2017

END OF lecture 12: “Visibility”

Next lecture: “Postprocessing”