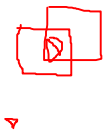
Hidden Surface Removal

1

Visibility

- Assumption: All polygons are **opaque**
- What polygons are visible with respect to your view frustum?
  - Outside: **Clipping**
    - Remove polygons outside of the view volume
    - For example, 3D Clipping
  - Inside: **Hidden Surface Removal**
    - Backface culling
      - Polygons facing away from the viewer
    - Occlusion
      - Polygons farther away are obscured by closer polygons
      - Full or partially occluded portions

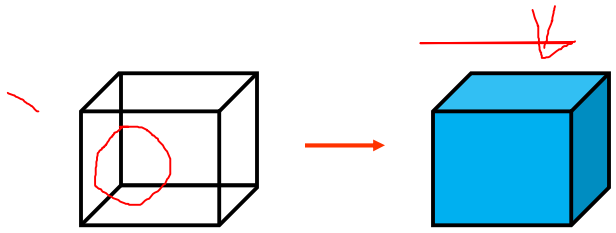


- Why should we remove these polygons?
  - Avoid unnecessary expensive operations on these polygons later

2

Visible-Surface Detection 1

Problem:
Given a scene and a projection,
what can we see?



3

Visible-Surface Detection 2

Terminology:

Visible-surface detection vs. *hidden-surface removal*

Hidden-line removal vs. *hidden-surface removal*

Many algorithms:

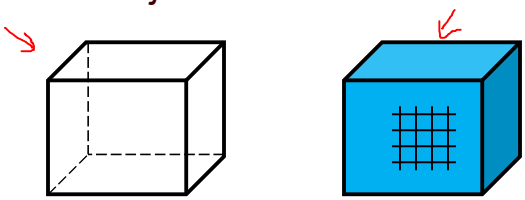
- Complexity scene
- Type of objects
- Hardware

4

Visible-Surface Detection 3

Two main types of algorithms:

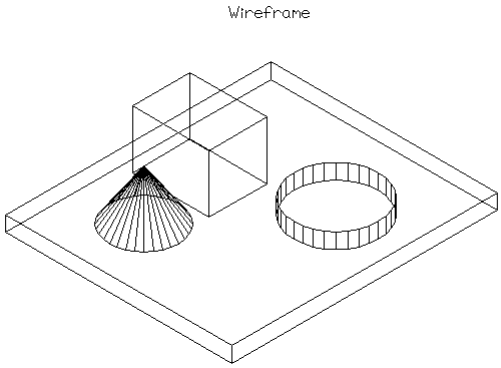
- *Object space: Determine which part of the object are visible*
- *Image space: Determine per pixel which point of an object is visible*



Object space *Image space*

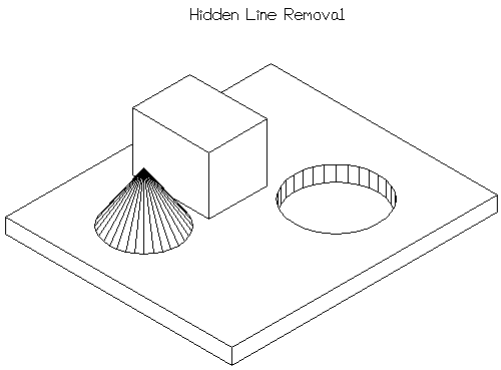
5

No Lines Removed



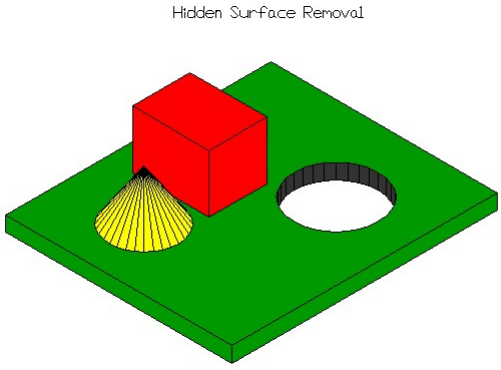
6

Hidden Lines Removed



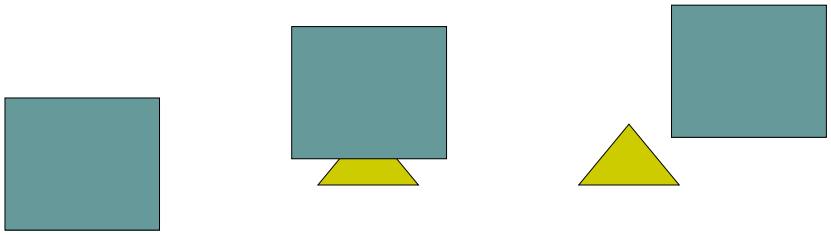
7

Hidden Surfaces Removed



8

Occlusion: Full, Partial, None



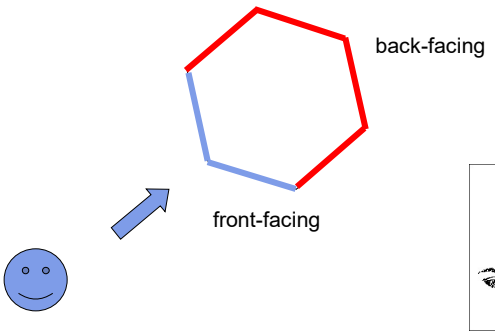
Full **Partial** **None**

- The rectangle is closer than the triangle
- Should appear in front of the triangle

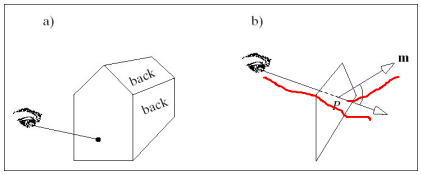
9

Backface Culling

- Avoid drawing polygons facing away from the viewer
  - Front-facing polygons occlude these polygons in a closed polyhedron
- Test if a polygon is front- or back-facing?



Ideas?

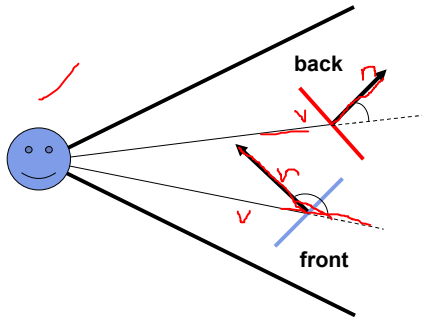


10

Detecting Back-face Polygons

- The polygon normal of a …
  - front-facing polygon points **towards** the viewer
  - back-facing polygon points **away** from the viewer

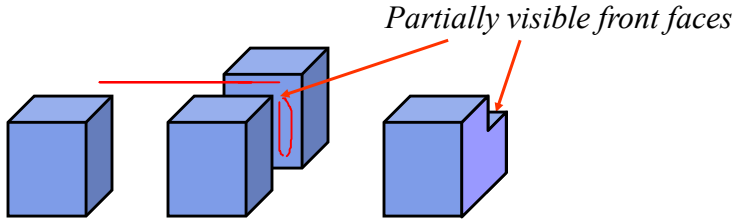
If (**n** • v) > 0 ⇒ "back-face"
If (**n** • v) ≤ 0 ⇒ "front-face"
v = view vector



11

Back-face Elimination

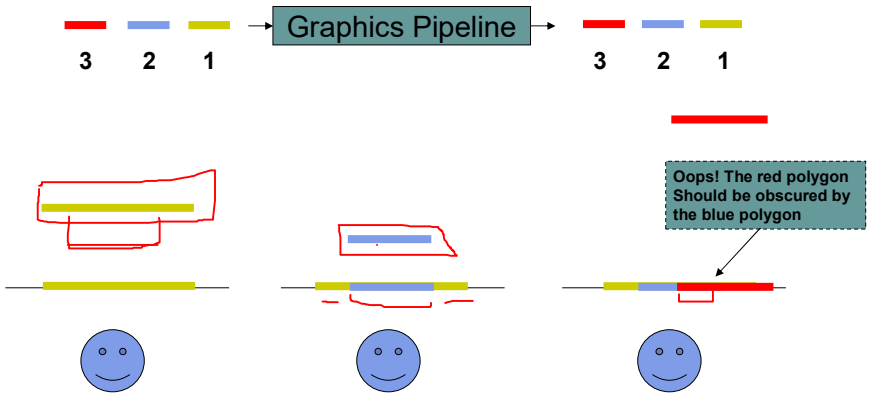
- Object-space method
- Works fine for convex polyhedra: ±50% removed
- Concave or overlapping polyhedra: require additional processing
- Interior of objects can not be viewed



12

Painter's Algorithm (1)

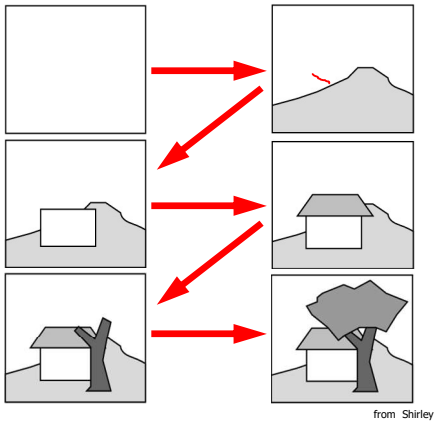
- Assumption: Later projected polygons overwrite earlier projected polygons



13

Painter's Algorithm (2)

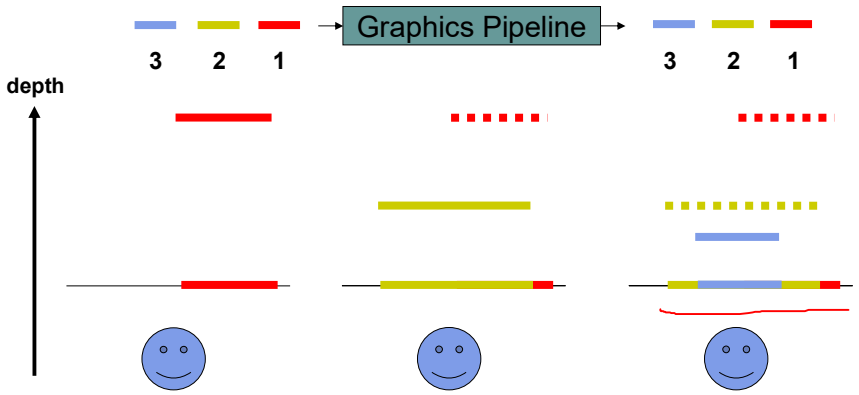
- Main Idea
  - A painter creates a picture by drawing background scene elemens before foreground ones
- Requirements
  - Draw polygons in back-to-front order
  - Need to **sort** the polygons by depth order to get a correct image



14

Painter's Algorithm (3)

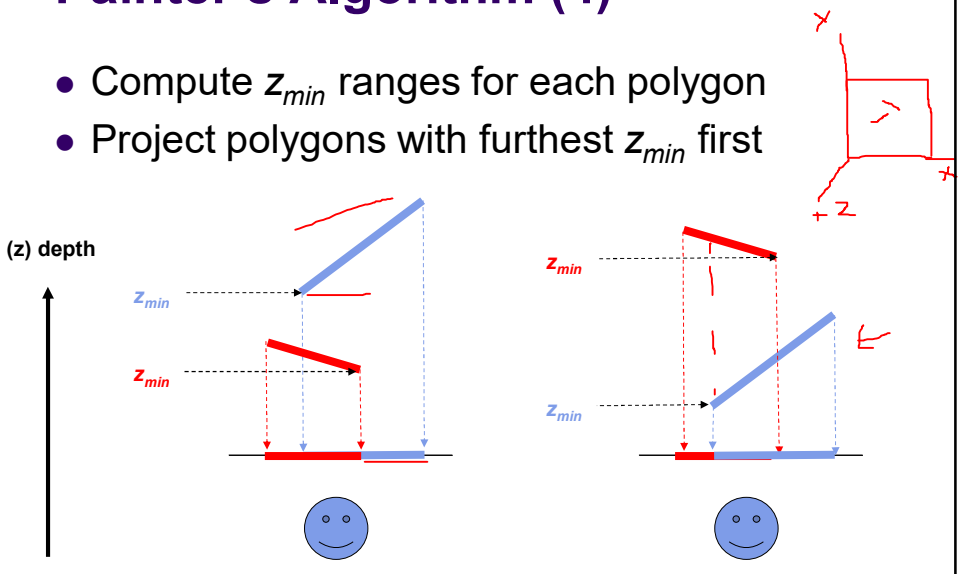
- Sort by the depth of each polygon



15

Painter's Algorithm (4)

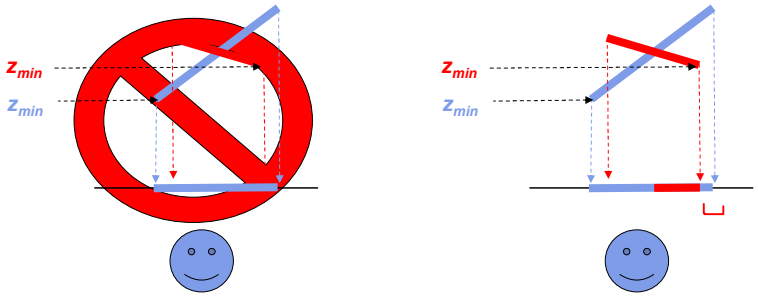
- Compute z_{min} ranges for each polygon
- Project polygons with furthest z_{min} first



16

Painter's Algorithm (5)

- Problem: Can you get a <u>total</u> sorting?

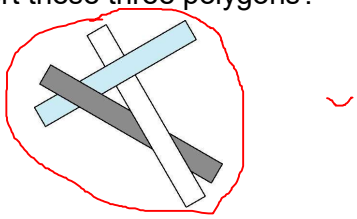


Correct?

17

Painter's Algorithm (6)

- Cyclic Overlap
  - How do we sort these three polygons?

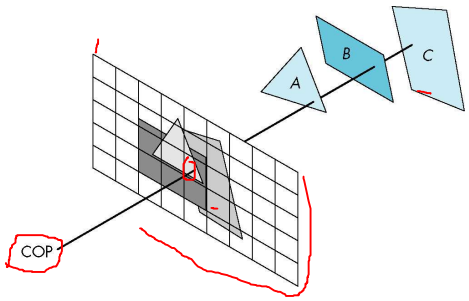


- Sorting is nontrivial
  - Split polygons in order to get a total ordering
  - Not easy to do in general

18

Visibility

- How do we ensure that closer polygons overwrite further ones in general?



19

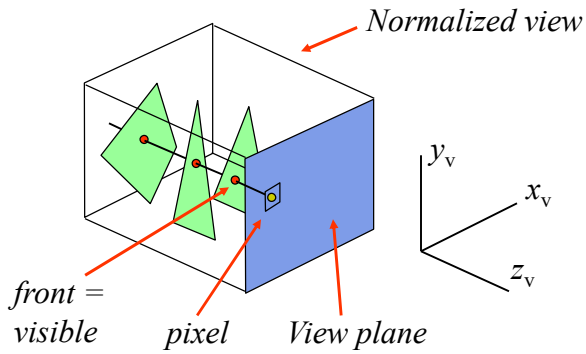
Z-Buffer

- Depth buffer (Z-Buffer)
  - A secondary image buffer that holds depth values
  - Same pixel resolution as the color buffer
  - Why is it called a **Z-Buffer**?
    - After eye space, depth is simply the *z*-coordinate
- Sorting is done at the pixel level
  - **Rule**: Only draw a polygon at a pixel if it is closer than a polygon that has already been drawn to this pixel

20

Z-Buffer Algorithm

- Image-space method
- Aka *z-buffer algorithm*



Algorithm:
Draw polygons,
Remember the color most in front.

21

Z-Buffer Algorithm

```
var zbuf: array[N,N] of real;          { z-buffer: 0=near,
      1=far }
    fbuf: array[N,N] of color;         { frame-buffer }

For all 1<= i, j <=N do
      zbuf[i,j] := 1.0; col[i,j] := BackgroundColour;
For all polygons do                    { scan conversion }
   For all covered pixels (i,j) do
      Calculate depth z;
      If z < zbuf[i,j] then { closer! }
         zbuf[i,j] := z;
         fbuf[i,j] := surfacecolor(i,j);
```

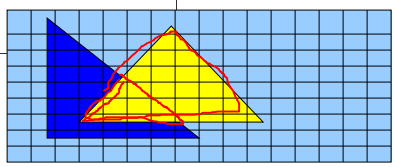
Sorting

22

Z-Buffer Algorithm

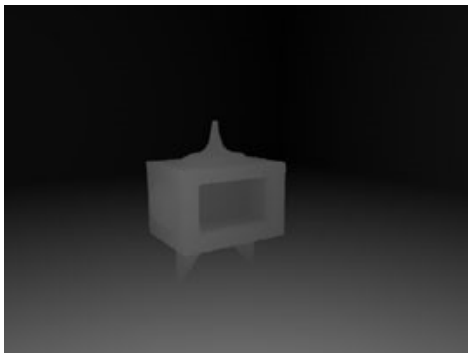
- Visibility testing is done during rasterization

```
for (each face F)
      for (each pixel (x,y) covering the face)
      {
         depth = depth of F at (x,y);
         if(depth < d[x][y])      //F is closest so far
          {
               c = color of F at (x, y);
               set the pixel color at (x, y) to c
               d[x][y] = depth; // update the depth buffer
          }
      }
```



23

Z-buffer: A Secondary Buffer



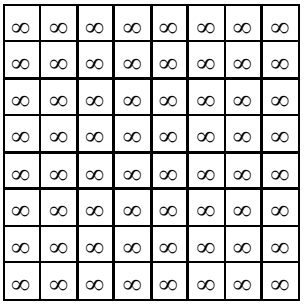
DAM Entertainment

Color buffer

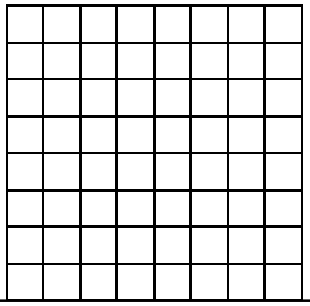
Depth buffer

24

Z-buffer - Example

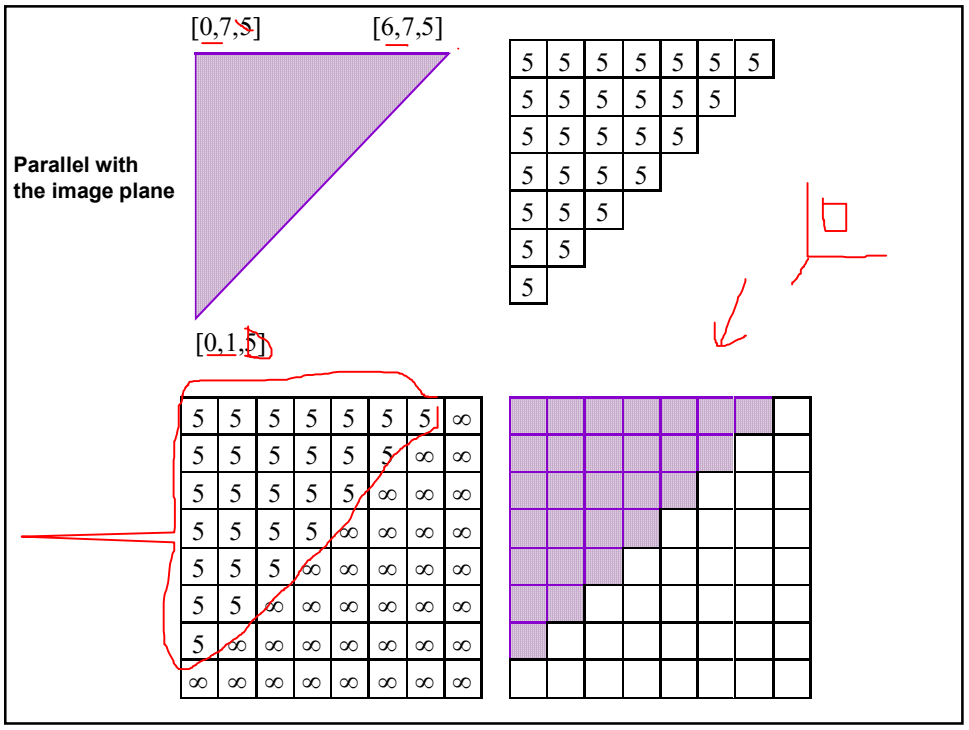


Z-buffer

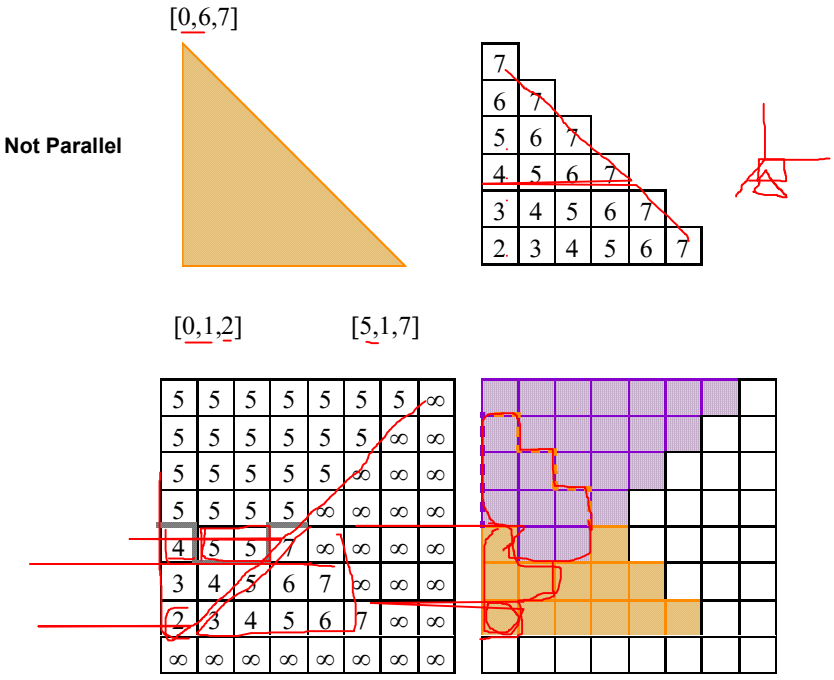


Screen

25



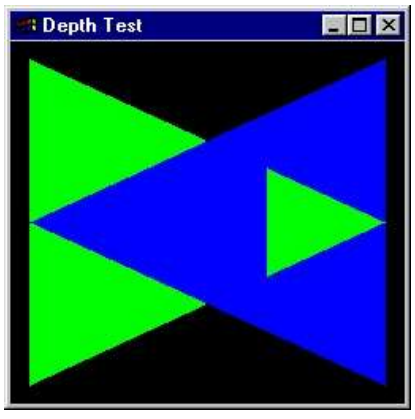
26



27

Z-Buffer Algorithm

- Algorithm easily handles this case



28

Z-buffering in OpenGL

- Create depth buffer by setting **`GLUT_DEPTH`** flag in **`glutInitDisplayMode()or the appropriate flag in the PIXELFORMATDESCRIPTOR.`**
- Enable per-pixel depth testing with **`glEnable(GL_DEPTH_TEST)`**
- Clear depth buffer by setting **`GL_DEPTH_BUFFER_BIT`** in **`glClear()`**

29