Painter's & Z-Buffer Algorithms

and Polygon Rendering

OUTLINE:

- Painter's Algorithm
- Z-Buffer Algorithm
- Polygon Rendering
- Comparison of Visibility Algorithms

Painter's Algorithm

sort objects by depth, splitting if necessary to handle intersections
loop on objects (drawing from back to front)
    loop on y within y range of this object
        loop on x within x range of this scan line of this object
            image[x,y] = shade(x,y)

This is back-to-front Painter's; it's also possible to do a front-to-back version, but then you need a Boolean variable at each pixel to record if a pixel has been covered or not.

basic operations:

- find y range of an object
- find x range of a given scanline of an object
- find intersection point (world space position & normal) of a given object with ray through pixel (x,y).
- compare depth of two objects, determine if A is in front of B, or B is in front of A, if they don't overlap in xy, or if they intersect
- split one object by another object

Strength: the inner loops are dirt simple. **Drawback**: sorting is a nuisance. This method gets clumsy for intersecting surfaces, because of need to split them.

Postscript uses Painter's algorithm.

Z-buffer Algorithm

```
loop on y
    loop on x
        zbuf[x,y] = infinity
loop on objects
    loop on y within y range of this object
        loop on x within x range of this scan line of this object
            if z(x,y) < zbuf[x,y]              compute z of this object at this pixel & test
                zbuf[x,y] = z(x,y)             update z-buffer
                image[x,y] = shade(x,y)        update image (typically RGB)
```

basic operations:

find y range of an object

find x range of a given scanline of an object

find intersection point (world space position & normal, screen space depth) of a given object with ray through pixel (x,y).

Most popular geometric primitive: polygons. Also possible: quadrics, parametric surfaces.

Set clipping planes carefully or you get poor z-buffer resolution.

Polygon Shading Methods

Shading Styles:

- **faceted shading**: color constant within polygon
  - *shade each polygon - least expensive*
- **Gouraud shading**: interpolate color
  - create vertex normals, *shade each vertex*
  - linearly interpolate color along left & right edges, along scanline
- **Phong shading**: interpolate normal vector
  - create vertex normals
  - linearly interpolate normal along edges, along scanline
  - at each pixel: normalize normal, *shade each pixel - most expensive*

Gouraud and Phong shading are examples of smooth shading. They use faceted, polygonal geometry but shade as if surface were curved (a trick). They are most commonly used in painter's or z-buffer algs, but they can also be used with ray casters or other visibility algorithms

In scan conversion:

- z can be computed incrementally (it's linear in x,y: z = ax+by+c, so horizontally, increment z by a).
  - Why is z linear in x,y? Because perspective transformations preserve planes.
- Gouraud shading can be done by incremental interpolation of color
- texture mapping can be done similarly, as we'll see later
- x loop can be done in integer arithmetic, and/or in hardware

Polygon Rendering Inputs

INPUTS:

- polygons
  - vertex coordinates in object space
  - material (color, diffuse & specular reflectance, etc.)
- transformations from object space(s) to world space
- camera transformation (world space to screen space)
- set of lights

If we're doing smooth shading (Gouraud or Phong) then we might also have vertex normals or pre-computed vertex colors.

If we're doing texture mapping then we might also have texture coordinates (u,v) at each vertex.

Steps for Polygon Scan Conversion (Z-buffer or Painter's)

if Painter's algorithm, sort polygons back-to-front
if Z-buffer, initialize z-buffer
for each polygon
    transform vertices to world space
    if doing faceted shading, shade polygon center
    if doing Gouraud shading, shade polygon vertices
    if doing backface removal, test if world normal points away from view dir.
    transform vertices to homogeneous screen space
    clip polygon in homogeneous screen space
    if polygon visible
        do homogeneous division on vertices to compute projected screen space
        scan convert polygon
            if doing Gouraud shading, interpolate colors across polygon
            if doing Phong shading, interpolate normal across polygon, shade each pixel

Z-buffered Polygons with Gouraud Shading

z, r, g, and b are linear functions of x,y (each of the form Ax+By+C)

```
loop on objects
    object setup: transform, clip, shade vertices, compute dz, dr, dg, db
    loop on y within y range of this object
        scanline setup: compute z,r,g,b at beginning of scan line
        loop on x within x range of this scan line of this object
            if z < zbuf[x,y]
                zbuf[x,y] = z
                image[x,y] = (r,g,b)
            z += dz
            r += dr
            g += dg
            b += db
```

This is done in hardware or firmware (microcode) on graphics workstations such as Silicon Graphics' or Hewlett Packard's.

Comparison of Visibility Algorithms

ray casting:
- memory: used for object database
- implementation: easy, but to make it fast you need spatial data structures
- speed: slow if many objects: cost is O((#pixels)×(#objects))
- generality: excellent, can even do CSG (constructive solid geometry), shadows, transp.

painter's:
- memory: used for image buffer
- implementation: moderate, requires scan conversion; hard if sorting & splitting needed
- speed: fast *if objects can be sorted a priori*, otherwise sorting many objs. can be costly
- generality: splitting of intersecting objects & sorting make it clumsy for general 3-D rendering

z-buffer:
- memory: used for image buffer & z-buffer
- implementation: moderate, requires scan conversion. It can be put in hardware.
- speed: fast, unless depth complexity is high (redundant shading)
- generality: very good

others (scanline, object space): tend to be hard to implement, and very hard to generalize to non-polygon models