Computer Graphics

Prof. Feng Liu

Fall 2015

http://www.cs.pdx.edu/~fliu/courses/cs447/

11/10/2015
Last time

- Rasterization
Today

- Hidden Surface Removal
Where We Stand

☐ At this point we know how to:
  ■ Convert points from local to window coordinates
  ■ Clip polygons and lines to the view volume
  ■ Determine which pixels are covered by any given line or polygon
  ■ Anti-alias

☐ Next thing:
  ■ Determine which polygon is in front
Visibility

- Given a set of polygons, which is visible at each pixel? (in front, etc.). Also called hidden surface removal

- Very large number of different algorithms known. Two main classes:
  - Object precision: computations that operate on primitives
  - Image precision: computations at the pixel level

- All the spaces in the viewing pipeline maintain depth, so we can work in any space
  - World, View and Canonical Screen spaces might be used
  - Depth can be updated on a per-pixel basis as we scan convert polygons or lines
  - Actually, run Bresenham-like algorithm on $z$ and $w$ before perspective divide
Painters Algorithm

- **Algorithm:**
  - Choose an order for the polygons based on some choice (e.g. depth of a point on the polygon)
  - Render the polygons in that order, deepest one first
- **This renders nearer polygons over further**
- **Difficulty:**
  - Doesn’t work in this form for most geometries - need at least better ways of determining ordering
The Z-buffer (1) (Image Precision)

- For each pixel on screen, have at least two buffers
  - Color buffer stores the current color of each pixel
    - The thing to ultimately display
  - Z-buffer stores at each pixel the depth of the nearest thing seen so far
    - Also called the depth buffer

- Initialize this buffer to a value corresponding to the furthest point

- As a polygon is filled in, compute the depth value of each pixel that is to be filled
  - if depth < z-buffer depth, fill in pixel color and new depth
  - else disregard
The Z-buffer (2)

- **Advantages:**
  - Simple and now ubiquitous in hardware
  - A z-buffer is part of what makes a graphics card “3D”
  - Computing the required depth values is simple

- **Disadvantages:**
  - Over-renders - rasterizes polygons even if they are not visible
  - Depth quantization errors can be annoying
  - Can’t easily do transparency or filter-based anti-aliasing (Requires keeping information about partially covered polygons)
OpenGL Depth Buffer

- OpenGL defines a depth buffer as its visibility algorithm
- The enable depth testing: `glEnable(GL_DEPTH_TEST)`
- To clear the depth buffer: `glClear(GL_DEPTH_BUFFER_BIT)`
  - To clear color and depth:
    - `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`
- The number of bits used for the depth values can be specified (windowing system dependent, and hardware may impose limits based on available memory)
- The comparison function can be specified: `glDepthFunc(...)`
  - Sometimes want to draw furthest thing, or equal to depth in buffer
The A-buffer (Image Precision)

- Handles transparent surfaces and filter anti-aliasing
- At each pixel, maintain a pointer to a list of polygons sorted by depth, and a sub-pixel coverage mask for each polygon
  - Coverage mask: Matrix of bits saying which parts of the pixel are covered
- Algorithm: Drawing pass (do not directly display the result)
  - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
  - if polygon is transparent or only partially covers pixel, insert into list, but don’t remove farther polygons
The A-buffer (2)

- Algorithm: Rendering pass
  - At each pixel, traverse buffer using polygon colors and coverage masks to composite:

- Advantage:
  - Can do more than Z-buffer
  - Coverage mask idea can be used in other visibility algorithms

- Disadvantages:
  - Not in hardware, and slow in software
  - Still at heart a z-buffer: Over-rendering and depth quantization problems

- But, used in high quality rendering tools
BSP-Trees (Object Precision)

- Construct a binary space partition tree
  - Tree gives a rendering order
  - A list-priority algorithm

- Tree splits 3D world with planes
  - The world is broken into convex cells
  - Each cell is the intersection of all the half-spaces of splitting planes on tree path to the cell

- Also used to model the shape of objects, and in other visibility algorithms
BSP-Tree Example
Building BSP-Trees

- Choose polygon (arbitrary)
- Split its cell using plane on which polygon lies
  - May have to chop polygons in two (Clipping!)
- Continue until each cell contains only one polygon fragment
- Splitting planes could be chosen in other ways, but there is no efficient optimal algorithm for building BSP trees
  - Optimal means minimum number of polygon fragments in a balanced tree
Building Example

- We will build a BSP tree, in 2D, for a 3 room building
  - Ignoring doors
- Splitting edge order is shown
  - “Back” side of edge is side with the number
Building Example (Done)
Using a BSP-Tree

- Observation: Things on the opposite side of a splitting plane from the viewpoint cannot obscure things on the same side as the viewpoint.

- A statement about rays - a ray must hit something on this side of the split plane before it hits the split plane and before it hits anything on the back side.

- NOT a statement about distance - things on the far side of the plane can be closer than things on the near side.
  - Gives a relative ordering of the polygons, not absolute in terms of depth or any other quantity.
Observation: Things on the opposite side of a splitting plane from the viewpoint cannot obscure things on the same side as the viewpoint.

Rendering algorithm is recursive descent of the BSP Tree.

At each node (for back to front rendering):
- Recurse down the side of the sub-tree that does not contain the viewpoint.
- Test viewpoint against the split plane to decide which tree.
- Draw the polygon in the splitting plane.
- Paint over whatever has already been drawn.
- Recurse down the side of the tree containing the viewpoint.
Back-to-front rendering order is 3a, 4a, 6, 1, 4b, 5a, 2, 3b, 5b
BSP-Tree Rendering (2)

- Advantages:
  - One tree works for any viewing point
  - Filter anti-aliasing and transparency work
    - Have back to front ordering for compositing

- Disadvantages:
  - Can be many small pieces of polygon
  - Over-rendering
Exact Visibility

- An *exact visibility* algorithm tells you what is visible and only what is visible
  - No over-rendering
- Difficult to achieve efficiently in practice
  - Small detail objects in an environment make it particularly difficult
- But, in mazes and other simple environments, exact visibility is extremely efficient
Cells and Portals

- Assume the world can be broken into *cells*
  - Simple shapes
  - Rooms in a building, for instance
- Define *portals* to be the transparent boundaries between cells
  - Doorways between rooms, windows, etc
- In a world like this, can determine exactly which parts of which rooms are visible
  - Then render visible rooms plus contents
Cell-Portal Example (1)
Cell and Portal Visibility

- Start in the cell containing the viewer, with the full viewing frustum
- Render the walls of that room and its contents
- Recursively clip the viewing frustum to each portal out of the cell, and call the algorithm on the cell beyond the portal
Cell-Portal Example (2)
Cell-Portal Example (3)
Cell-Portal Example (4)
Cell-Portal Example (5)
Cell-Portal Example (6)
Next Time

☐ Lighting

☐