Now Playing:

15 Step
Radiohead
from *In Rainbows*
Released October 10, 2007

Texture Mapping and BSP Trees

Rick Skarbez, Instructor
COMP 575
October 11, 2007

Assignment 1 Back

- Mostly did very well:
  - Note that the "null" scale factor (that is, the scale factor that doesn't change a dimension) is 1
    - So, to scale in x and y (but not z)
      glScale(2.0, 3.0, 1.0)
    - NOT
      glScale(2.0, 3.0, 0.0)
- Doing the P-matrix problem on the board

Announcements

- Programming Assignment 2 (3D graphics in OpenGL) is out
  - Due Thursday, October 25 by 11:59pm
- Programming Assignment 3 (Rasterization) is out
  - Due Thursday, November 1 by 11:59pm

Last Time

- Concluded our discussion of line-antialiasing
  - Ratio method
- Presented some methods for polygon rasterization
  - Scan-line drawing
  - Flood Fill

Last Time

- Discussed hidden surface removal
  - Backface Culling
  - Depth Culling
    - Z-Buffering
    - Painter's Algorithms

Today

- Discussing Binary Space Partition (BSP) Trees
- Texture Mapping in Theory and Practice

Polygon Drawing

- After clipping, we know that the entire polygon is inside the viewing region
  - Makes the problem easier
- Need to determine which pixels are inside the polygon, and color those
  - Find edges, and fill in between them
  - Edges - Connected line segments
    - How to fill?

Scan-Line Polygons

- Algorithm:
  1. Mark local minima and maxima
  2. Mark all distinct y values on edges
  3. For each scan line:
     1. Create pairs of edge pixels (going from left to right)
     2. Fill in between pairs

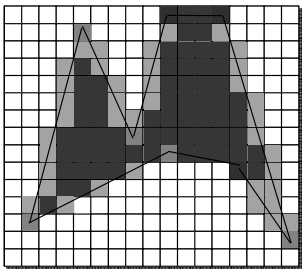
Scan-Line Polygon Example

Polygon Vertices

Maxima / Minima

Edge Pixels

Scan Line Fill

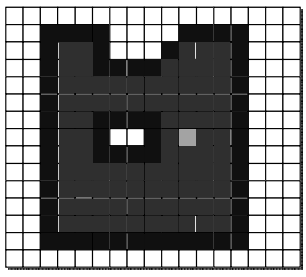


Flood Fill

- Algorithm:
  1. Draw all edges into some buffer
  2. Choose some “seed” position inside the area to be filled
  3. As long as you can
     1. “Flood out” from seed or colored pixels
        - 4-Fill, 8-Fill

Flood Fill Example

- 4-fill:



Edge Pixels

Seed

Filled Pixels

Backface Culling

- Where?
  - Object space
- When?
  - After transformation but before clipping
- What?
  - If **normal** • **toViewer** < 0, discard face
    - That is, if the polygon face is facing away from the viewer, throw it out

Backface Culling

- So what does this buy us?
  - Up to 50% fewer polygons to clip/rasterize
- Is this all we have to do?
  - No.
    - Can still have 2 (or more) front faces that map to the same screen pixel
    - Which actually gets drawn?

Depth Culling

- Can happen here (fragment processing)
  - z-buffering
- Can happen before rasterization
  - Painter's algorithm

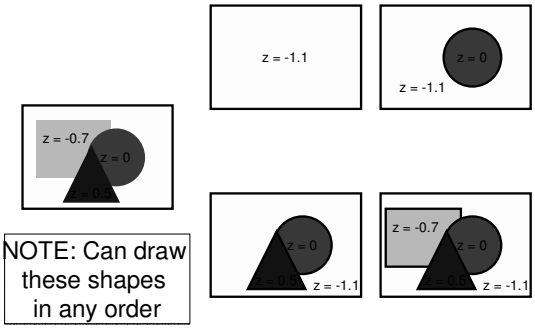
Z-Buffering

- Where?
  - Fragment space
- When?
  - Immediately after rasterization
- How?
  - Basically, remember how far away polygons are, and only keep the ones that are in front

Z-Buffering

- Need to maintain z for all fragments
  - Why we project to a volume instead of a plane
- Maintain a separate depth buffer, the same size and resolution of the color buffer
  - Initialize this buffer to z=-1.1 (all z is in [-1, 1])
- As each fragment comes down the pipe, test fragment.z > depth[s][t]
  - If true, the fragment is in front of whatever was there before, so set color[s][t]=frag.color and depth[s][t]=frag.z

Z-Buffering Example

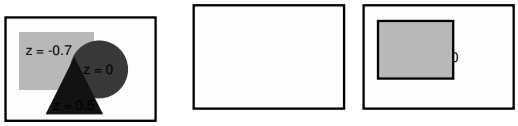


NOTE: Can draw these shapes in any order

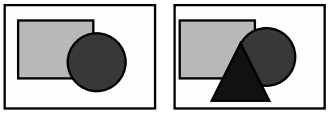
Painter's Algorithm

- Really a class of algorithms
  - Somehow sort the objects by distance from the viewer
  - Draw objects in order from farthest to nearest
    - The *entire* object
  - Nearer objects will "overwrite" farther ones

Painter's Example



Sort by depth:
Green rect
Red circle
Blue tri



Spatial Data Structures

- When we talked about using a painter's algorithm for rendering, we talked about needing to sort the scene geometry
- The algorithm we presented is simple and works
  - However, it is only valid for a single viewpoint
  - We can do better

BSP Trees

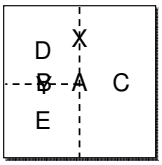
Fuchs, Kedem, & Naylor; SIGGRAPH 1980

- Based on the concept of binary space partitioning
  - A plane divides space into two half-spaces; all objects can be classified as being on one side or the other
- A preprocessing step builds a BSP tree that can be used for any viewpoint
  - However, assumes that the geometry does not change

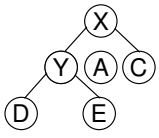
BSP Tree Illustration

- Let's see an example (in 2D, not 3D)
  - Here, a line divides the plane into two half-planes

Space



BSP Tree



Line/Plane Definitions

- We want to define the partitioning lines/planes in a way such that it is easy to test which side an object is on
- Recall the implicit plane definition

  $f(x,y,z) = ax + by + cz + d = 0$

- Implicit surface definitions have *exactly* this property
  - For all points on the plane, $f(\mathbf{p}) = 0$
  - For all points on one side, $f(\mathbf{p}) > 0$
  - For all points on the other side, $f(\mathbf{p}) < 0$

The Desired Output

- A set of implicit planes that partition the space in such a way that every object in one subspace is entirely on one side of every object in every other subspace

The Algorithm

- So, here's how you do it (at a very high level):
  1. Choose a partition plane
  2. Partition the set of polygons with respect to this plane
     - Now have 2 sets of polygons
  3. Recurse on each of the new sets

Choosing the Plane

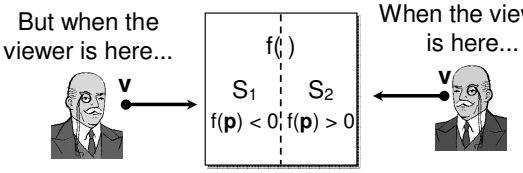
- There are many ways to choose
  - Best depends on the application
- One way is just to choose a polygon from the input set to define a plane
  - Randomly?
  - Attempt to balance the number of polys on either side?
- In general, preferred characteristics are a more balanced tree and/or fewer polygon splits

Partitioning the Polygons

- Test each vertex of the poly against the plane
  - If all negative, place in negative subtree
  - If all positive, place in positive subtree
  - If some positive and some negative, need to split the polygon into smaller polygons that are entirely on one side of the plane

Why Does a BSP Work for Any Viewpoint?

- Consider a very simple example:



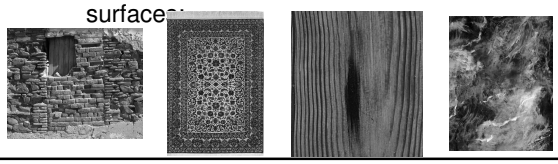
f(**v**) ≈ 0, so objects with f(**p**) ≈ 0 must be nearer to the viewer
So, draw region S_2, then region S_1

BSP Tree Review

- Use implicit planes to carve up space (and the geometry in it) into distinct subspaces
- One BSP tree can be used for *any* viewpoint
- Can be used to implement a painter's algorithm
  - Or to speed up a raytracer...
    - We'll be seeing this again
- Any questions?

The World is a Complicated Place

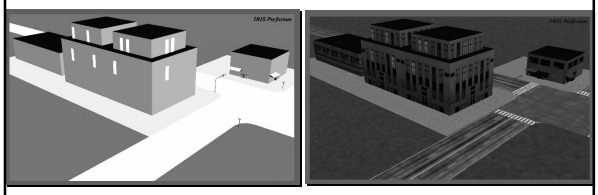
- So far, we know how to render Gouraud shaded polygons
  - Not too shabby
- Sadly, the world doesn't cooperate by only having simple smooth-shaded surfaces



The World is a Complicated Place

- Now, maybe we could simulate these materials/objects with simple primitives
  - Thousands?
  - Millions?
  - ...This may not be such a good idea
- Luckily, we can take advantage of a technique called texture mapping

Texturing Example



Before Texturing After Texturing

Texture Mapping

- Texture mapping allows us to render surfaces with very complex appearance
- How it works:
  - Store the appearance as a function or image
    - Take a picture
  - Map it onto a surface made up of simple polygons
    - Paste the picture on an object

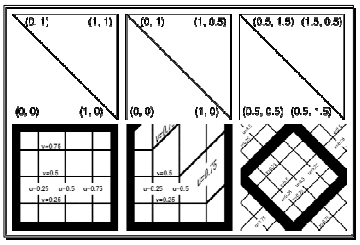
Texture Mapping

- So, assume we have an image, stored in a buffer in our code
  - unsigned char img[256][256][3], for example
- How do we map that image onto a triangle?
  - HINT: Done after rasterization
    - Why?
  - Image → World?
  - World → Image?

How to Map

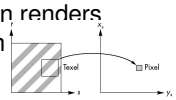
- We generally map the world (that is, the geometry) into the image
- Image is defined in (u, v) coordinate frame
  - $u, v \in [0, 1]$
- Each vertex in your geometry is associated with a texture coordinate (u_v, v_v)
  - What to do at interior points?
    - Interpolate u and v (using barycentric coordinates)

Mapping Example

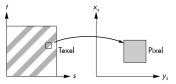


Sampling Issues

- So we can define the mapping, and it works fine
  - *As long as the size of the rendered image is approximately the same size as the texture source*
- What if the textured polygon renders much smaller in the final im than the original texture?



- How about much bigger?



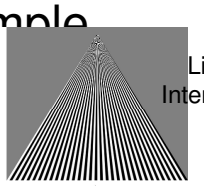
Mip-mapping to the Rescue

- Mip-mapping is a technique that creates multiple resolutions of an image
  - *i.e.* Takes a 512x512 image and filters it to create 256x256, 128x128, 64x64, ..., 1x1 versions of it
- Then, when you're looking up your texture coordinates, it uses the most appropriate mip-map level
  - Or, more likely, interpolates between the two closest

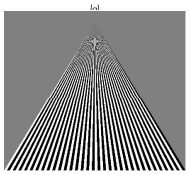
Mip-mapping Example

Direct Texture Mapping

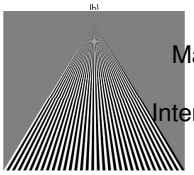
Linear Interpolation



Mip-Mapping

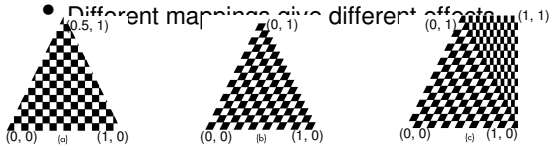


Mip-Mapping + Interpolation



Assigning Texture Coordinates

- As we've seen, if you have a square texture, and are mapping it to 2 triangles forming a square, this is very straight-forward
  - This is not usually the case
- Different mappings give different effects



Next Time

- More mapping
  - Finish up Texture Mapping
  - Bump Maps
  - Displacement Maps
- Discussion of programmable graphics hardware
- Discussion of class project