Hidden Surface Removal

The Z-Buffer Algorithm (mid-70's)
- BSP trees proposed when memory was expensive
- first 512x512 framebuffer was $>50,000!
- Ed Catmull proposed a radical new approach called z-buffering
- the big idea:
  - resolve visibility independently at each pixel

The Z-Buffer Algorithm
- we know how to rasterize polygons into an image discretized into pixels:

The Z-Buffer Algorithm
- augment color framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
- at frame beginning, initialize all pixel depths to $\infty$
- when rasterizing, interpolate depth (Z) across polygon
- check Z-buffer before storing pixel color in framebuffer and storing depth in Z-buffer
- don't write pixel if its Z value is more distant than the Z value already stored there

The Z-Buffer Algorithm
- barycentric coordinates
- interpolate Z like other planar parameters

Z-Buffer
- store (r,g,b,z) for each pixel
- typically 8+8+8+24 bits, can be more

Depth Test Precision
- therefore, depth-buffer essentially stores 1/z, rather than z!
- issue with integer depth buffers
  - high precision for near objects
  - low precision for far objects

Integer Depth Buffer
- reminder from viewing discussion
  - depth lies in the DCS z range [0,1]
  - format: multiply by $2^n-1$ then round to nearest int
  - where $n = \text{number of bits in depth buffer}$
  - 24 bit depth buffer $= 2^{24} = 16,777,216$ possible values
  - small numbers near, large numbers far

- consider VCS depth $z_{VCS} = (1+<N>)(a + b / z_{VCS})$
  - $N = \text{number of bits of Z precision}, 1<<N \text{ bitshift} = 2^n$
  - $a = z_{Far} / (z_{Far} - z_{Near})$
  - $b = z_{Far} * z_{Near} / (z_{Near} - z_{Far})$
  - $z_{VCS} = \text{distance from the eye to the object}$
Z-Buffer Algorithm Questions

• how much memory does the Z-buffer use?
• does the image rendered depend on the drawing order?
• does the time to render the image depend on the drawing order?
• how does Z-buffer load scale with visible polygons? with framebuffer resolution?

Z-Buffer Pros

• simple!!!
• easy to implement in hardware
• hardware support in all graphics cards today
• polygons can be processed in arbitrary order
• easily handles polygon interpenetration
• enables deferred shading
• rasterize shading parameters (e.g., surface normal) and only shade final visible fragments

Z-Buffer Cons

• poor for scenes with high depth complexity
• need to render all polygons, even if most are invisible

eye

shared edges are handled inconsistently
• ordering dependent

Three.js Intersection Support

http://soledadpenades.com/articles/three-js-tutorials/object-picking/

• projector = new THREE.Projector();
• mousevec = new THREE.Vector3();
• window.addEventListener('mousemove', onMouseMove, false);
• onMouseMove:
  • mouseVector.x = e.clientX * (e.clientX/containerWidth) - 1;
  • mouseVector.y = e.clientY * (e.clientY/containerHeight);
  // don't forget to flip Y from upper left origin!
  var raycaster = projector.pickingRay(mouseVector.clone(), camera);
  var intersects = raycaster.intersectObjects(geoms);

Bounding Extents

• keep track of axis-aligned bounding rectangles

Interactive Object Selection

• move cursor over object, click
• how to decide what is below?
• inverse of rendering pipeline flow
• from pixel back up to object: unprojecting
• ambiguity
• many 3D world objects map to same 2D point
• two common approaches
• ray intersection (three.js support)
• off-screen buffer color coding
• other approaches
• bounding extents
• deprecated: OpenGL selection region with hit list

Ray Intersection Picking

• computation in software within application
• map selection point to a ray
• intersect ray with all objects in scene.
• advantages
• flexible, straightforward
• supported by three.js
• disadvantages
• slow: work to do depends on total number and complexity of objects in scene

Offscreen Buffer Color Coding

• use offscreen buffer for picking
• create image as computational entity
• never displayed to user
• redraw all objects in offscreen buffer
• turn off lighting/shading calculations
• set unique color for each pickable object
• store in table
• read back pixel at cursor location
• check against table

Offscreen Buffer Picking

http://coffeesmudge.blogspot.ca/2013/08/implementing-picking-in-webgl.html

• create offscreen framebuffer
• like rendering into texture
• render each object with unique color in framebuffer (up to 16M with 24 bit integers)
• glReadPixels readback to find color under cursor
• look up object with that color
• color[0]*65536 + color[1]*256 + color[2]

OpenGL vs WebGL Picking

• very different world, don't get confused by old tutorials
• OpenGL
  • fast hardware support for select/hit
  • re-render small area around cursor
  • backbuffer color
  • straightforward but slow without hardware support
  • no standard library support for ray intersection
  • slow and laborious
• WebGL
  • good library support for intersection
  • best choice for most of you!
  • fast offscreen buffer hardware support
  • select/hit unsupported

Bounding Extents

• disadvantages
• low precision
• must keep track of object-rectangle relationship
• extensions
• do more sophisticated bound bookkeeping
• first level: box check
• second level: object check

Painter’s Algorithm

• simple: render the polygons from back to front, “painting over” previous polygons
• draw blue, then green, then orange
• will this work in the general case?

Painter's Algorithm

http://threejs.org/docs/index.html

• renderer.render(Scene, Camera);
• render all objects in scene.
• turn off lighting/shading calculations
• project ray (from camera) to find color under cursor
• draw blue, then green, then orange
• will this work in the general case?

Picking

• tutorials/object-picking/
• shared edges are handled inconsistently
• two common approaches
• onmouseMove
• hard to simulate translucent polygons
• requires fast memory
• Read-Modify-Write in inner loop
• hard to simulate translucent polygons
• we throw away color of polygons behind closest one
• works if polygons ordered back-to-front
• extra work throws away much of the speed advantage

Bounding Extents

• disadvantages
• low precision
• must keep track of object-rectangle relationship
• extensions
• do more sophisticated bound bookkeeping
• first level: box check
• second level: object check
Painter's Algorithm: Problems

- Intersecting polygons present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order.

Analytic Visibility Algorithms

- Early visibility algorithms computed the set of visible polygon fragments directly, then rendered the fragments to a display.

Analytic Visibility Algorithms

- What is the minimum worst-case cost of computing the fragments for a scene composed of n polygons?
- Answer: $O(n^2)$

Binary Space Partition Trees (1979)

- BSP Tree: partition space with binary tree of planes
  - Idea: divide space recursively into half-spaces by choosing splitting planes that separate objects in scene
  - Preprocessing: create binary tree of planes
  - Runtime: correctly traversing this tree enumerates objects from back to front

Creating BSP Trees: Objects

- No bunnies were harmed in previous example
- But what if a splitting plane passes through an object?
- Split the object; give half to each node

Splitting Objects

- For given viewpoint, decide which side is near and which is far
- Check which side of plane viewpoint is on independently for each tree vertex

Traversing BSP Trees

- Tree creation independent of viewpoint
- Preprocessing step
- Tree traversal uses viewpoint
- Runtime, happens for many different viewpoints
- Each plane divides world into near and far

Traversing BSP Trees

- Recursive algorithm
  - Recurse on far side
  - Draw object
  - Recurse on near side

Traversing BSP Trees

query: given a viewpoint, produce an ordered list of (possibly split) objects from back to front:

```
renderBSP(BSPtree *T)
BSPtree *near, *far;
if (eye on left side of T->plane)
  near = T->left; far = T->right;
else
  near = T->right; far = T->left;
renderBSP(far);
if (T is a leaf node)
  renderObject(T);
renderBSP(near);
```
BSP Trees: Viewpoint A

BSP Trees: Viewpoint A

BSP Trees: Viewpoint A

BSP Trees: Viewpoint A

BSP Trees: Viewpoint A

BSP Trees: Viewpoint A

BSP Trees: Viewpoint B

BSP Trees: Viewpoint B

BSP Tree Traversal: Polygons
• split along the plane defined by any polygon from scene
• classify all polygons into positive or negative half-space of the plane
  • if a polygon intersects plane, split polygon into two and classify them both
• recurse down the negative half-space
• recurse down the positive half-space

BSP Demo
• useful demo:
  http://symbolcraft.com/graphics/bsp

BSP Example
• order of insertion can affect half-plane extent

Summary: BSP Trees
• pros:
  • simple, elegant scheme
  • correct version of painter’s algorithm back-to-front rendering approach
  • was very popular for video games (but getting less so)
• cons:
  • slow to construct tree: \(O(n \log n)\) to split, sort
  • splitting increases polygon count: \(O(n^2)\) worst-case
  • computationally intense preprocessing stage restricts algorithm to static scenes
Hidden Surface Removal
- two kinds of visibility algorithms
  - object space methods
  - image space methods

Object Space Algorithms
- determine visibility on object or polygon level
  - using camera coordinates
  - resolution independent
  - explicitly compute visible portions of polygons
- early in pipeline
  - after clipping
  - requires depth-sorting
  - painter’s algorithm
  - BSP trees

Image Space Algorithms
- perform visibility test for in screen coordinates
  - limited to resolution of display
  - Z-buffer: check every pixel independently
  - performed late in rendering pipeline

Back-face Culling
- on the surface of a closed orientable manifold, polygons whose normals point away from the camera are always occluded:

Back-face Culling: VCS
- first idea: cull if $N_z < 0$
- sometimes misses polygons that should be culled

Back-face Culling: NDCS
- works to cull if $N_z > 0$

Invisible Primitives
- why might a polygon be invisible?
  - polygon outside the field of view / frustum
  - solved by clipping
  - polygon is back-facing
  - solved by backface culling
  - polygon is occluded by object(s) nearer the viewpoint
  - solved by hidden surface removal

Alpha and Premultiplication
- specify opacity with alpha channel $\alpha$
  - $\alpha=1$: opaque, $\alpha=0$: transparent
- how to express a pixel is half covered by a red object?
  - obvious way: store color independent from transparency (r,g,b,α)
  - intuition: $\alpha$ as transparent colored glass
    - 0% transparency can be represented with many different $\alpha$ values
  - pixel value is $(1,0,0,\alpha)$
  - update: easy to change opacity of image, very intuitive
  - downside: compositing calculations are more difficult - not associative
  - elegant way: premultiply by its own $\alpha$ (r,og,b,α)
    - intuition: $\alpha$ as accumulation
      - $\alpha$ specifies how much color object contributes to scene
      - $\alpha$ specifies how much object obscures whatever is behind it (coverage)
      - $\alpha=1$ means half the pixel is covered by the color, half completely transparent
      - $\alpha=0$ means object represents 100% transparency
      - pixel value is $(0,0,0,\alpha)$
      - update: compositing calculations easy (all additive blending for glowing)
      - downside: less intuitive

THE RENDERING PIPELINE
- Vertex Shader
  - Modelview transform
  - Per-vertex attributes
- Rasterization
  - Scan conversion
  - Interpolation
- Fragment Shader
  - Texturing...
  - Lighting/shading
- Per-Sample Operations
  - Depth test
  - Blending
- Framebuffer
Alpha and Simple Compositing

- F is foreground, B is background, F over B
- premultiply math: uniform for each component, simple, linear
  \[ R' = R_F \times A_F + (1 - A_F) \times R_B \]
  \[ G' = G_F \times A_F + (1 - A_F) \times G_B \]
  \[ B' = B_F \times A_F + (1 - A_F) \times B_B \]
  \[ A' = A_F \]
- associative: easy to chain together multiple operations
- non-premultiply math: trickier
  \[ R' = (R_F \times A_F + (1 - A_F) \times R_B \times A_B) / A' \]
  \[ G' = (G_F \times A_F + (1 - A_F) \times G_B \times A_B) / A' \]
  \[ B' = (B_F \times A_F + (1 - A_F) \times B_B \times A_B) / A' \]
- don't need divide if F or B is opaque. but still… oof!
- chaining difficult, must avoid double-counting with intermediate ops

Alpha and Complex Compositing

- foreground color A, background color B
- how might you combine multiple elements?
  - Compositing Digital Images, Porter and Duff, Siggraph '84
  - pre-multiplied alpha allows all cases to be handled simply
  - pre-multiply allows both conventional blend and additive blend
- blend white and clear equally (50% each)
  - white is (1, 1, 1, 1), clear is (0, 0, 0, 0), black is (0, 0, 0, 1)
  - premultiplied: multiply componentwise by 50% and just add together
  - (.5, .5, .5, .5) is indeed half-transparent white in premultiply format
  - 4-tuple would mean half-transparent grey in non-premultiply format
- technical academy award for Smith, Catmull, Porter, Duff
  - http://www.alvyray.com/Awards/AwardsAcademy96.htm
- for more: see nice writeup from Alvy Ray Smith

Alpha Examples

- blend white and clear equally (50% each)
- white is (1, 1, 1, 1), clear is (0, 0, 0, 0), black is (0, 0, 0, 1)
- premultiplied: multiply componentwise by 50% and just add together
- (.5, .5, .5, .5) is indeed half-transparent white in premultiply format
- 4-tuple would mean half-transparent grey in non-premultiply format
- alpha 0 and RGB nonzero: glowing/luminescent
- nice for particle systems!
- for more: see nice writeup from Alvy Ray Smith
  - technical academy award for Smith, Catmull, Porter, Duff
  - http://www.alvyray.com/Awards/AwardsAcademy96.htm