University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2016

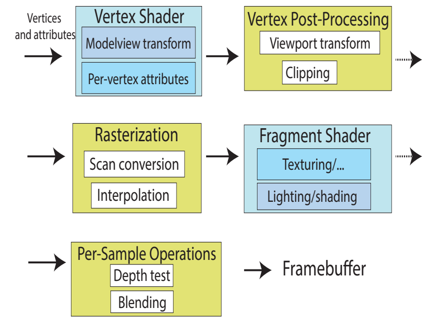
Tamara Munzner

Hidden Surfaces / Depth Test

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2016

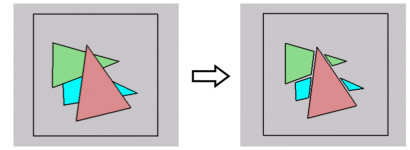
Hidden Surface Removal

THE RENDERING PIPELINE



Occlusion

- for most interesting scenes, some polygons overlap



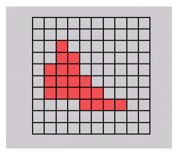
- to render the correct image, we need to determine which polygons occlude which

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

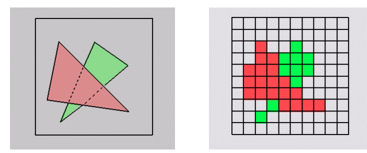
The Z-Buffer Algorithm

- we know how to rasterize polygons into an image discretized into pixels:



The Z-Buffer Algorithm

- what happens if multiple primitives occupy the same pixel on the screen?
  - which is allowed to paint the pixel?



The Z-Buffer Algorithm

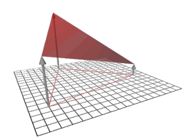
- idea: retain depth after projection transform
  - each vertex maintains z coordinate
    - relative to eye point
  - can do this with canonical viewing volumes

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 ∞
  - 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

Interpolating Z

- 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

```
for all i,j {
 Depth[i,j] = MAX_DEPTH
 Image[i,j] = BACKGROUND_COLOUR
}
for all polygons P {
  for all pixels in P {
    if (Z_pixel < Depth[i,j]) {
      Image[i,j] = C_pixel
      Depth[i,j] = Z_pixel
    }
  }
}
```


Depth Test Precision

- reminder: perspective transformation maps eye-space (VCS) z to NDC z

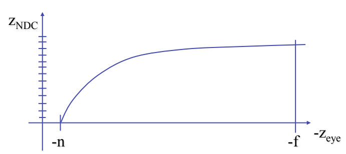
$$\begin{bmatrix} E & 0 & A & 0 \\ 0 & F & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} Ex+Az \\ Fy+Bz \\ Cz+D \\ -z \end{bmatrix} = \begin{bmatrix} -\left(\frac{Ex}{z}+A\right) \\ -\left(\frac{Fy}{z}+B\right) \\ -\left(C+\frac{D}{z}\right) \\ 1 \end{bmatrix}$$

- thus:

$$z_{NDC} = -\left(C+\frac{D}{\boxed{z_{VCS}}}\right)$$

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



Depth Test Precision

- low precision can lead to depth fighting for far objects
  - two different depths in eye space get mapped to same depth in framebuffer
  - which object "wins" depends on drawing order and scan-conversion
- gets worse for larger ratios $f{:}n$
  - *rule of thumb:* $f{:}n < 1000$ *for 24 bit depth buffer*
- with 16 bits cannot discern millimeter differences in objects at 1 km distance

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 = 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_{DCS} = (1<<N)*(a + b / z_{VCS})
  - N = number of bits of Z precision, 1<<N bitshift = 2^n
  - a = zFar / (zFar - zNear)
  - b = zFar * zNear / (zNear - zFar)
  - z_{VCS} = distance from the eye to the object

Z Buffer Calculator

- demo:
  - https://www.sjbaker.org/steve/omniv/love_your_z_buffer.html

Z-Buffer Algorithm Questions

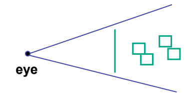
- how much memory does the Z-buffer use?
- does the image rendered depend on the drawing order?
- does the time to render the image depend on the drawing order?
- how does Z-buffer load scale with visible polygons? with framebuffer resolution?

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



- shared edges are handled inconsistently
  - *ordering dependent*

Z-Buffer Cons

- requires memory
  - (e.g. 1280x1024x32 bits)
- 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

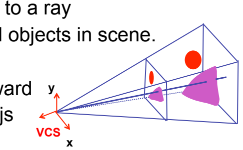
Picking

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



Three.js Intersection Support

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

- projector = new THREE.Projector();
- mousevector = new THREE.Vector3();
- window.addEventListener('mousemove', onMouseMove, false)
- onmouseMove:
  - mouseVector.x=2*(e.clientX/containerWidth)-1
  - mouseVector.y=1-2*(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>);

three.js Intersection

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

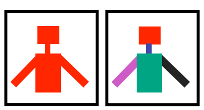
- intersectObjects function returns array
  - all ray intersections for children of root geometry
  - ordered by distance, nearest first
- intersection object contains
  - *distance* from camera
  - exact *point*
  - *face*
  - *object*

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 Color Coding

- advantages
  - conceptually simple
  - variable precision
  - hardware support
    - off-screen buffer creation/readback
- disadvantages
  - extra redraw delay (fixed overhead)
  - implementation complexity



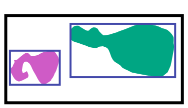
WebGL Offscreen Buffer Picking

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)
- gl.readPixels readback to find color under cursor
- look up object with that color
  - color[0]*65536 + color[1]*256 + color[2]

Bounding Extents

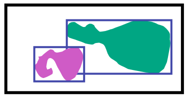
- keep track of axis-aligned bounding rectangles



- advantages
  - conceptually simple
  - easy to keep track of boxes in world space

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

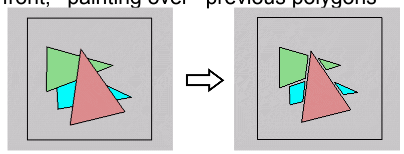


OpenGL vs WebGL Picking

- 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
    - straighforward 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

Painter's Algorithm

- simple: render the polygons from back to front, "painting over" previous polygons



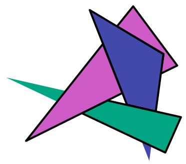
  - draw blue, then green, then orange
- will this work in the general case?

Painter's Algorithm: Problems

- *intersecting polygons* present a problem
- even non-intersecting polygons can form a cycle with no valid visibility order:

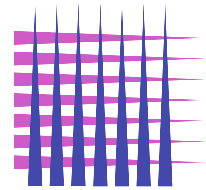
Analytic Visibility Algorithms

- 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: $O(n^2)$



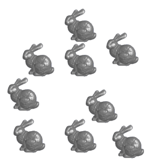
Analytic Visibility Algorithms

- so, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for hidden surface removal
- we'll talk about one:
  - *Binary Space Partition (BSP) Trees*

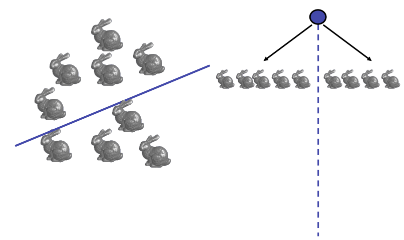
Binary Space Partition Trees (1979)

- 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

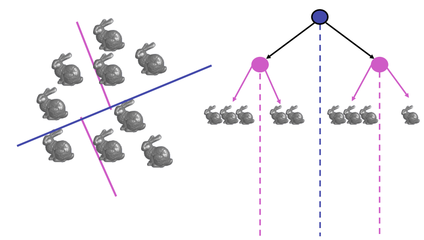
Creating BSP Trees: Objects



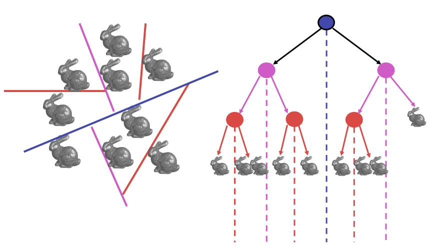
Creating BSP Trees: Objects



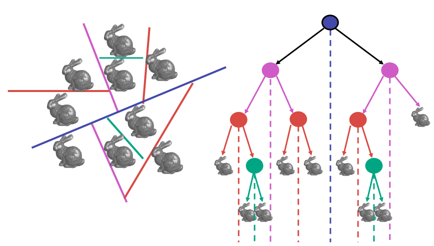
Creating BSP Trees: Objects



Creating BSP Trees: Objects

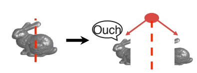


Creating BSP Trees: Objects



Splitting 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



Traversing BSP Trees

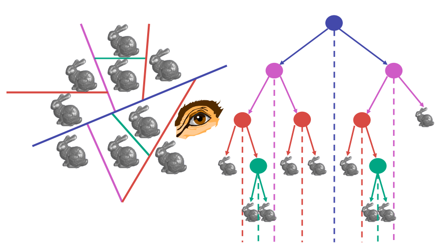
- tree creation independent of viewpoint
  - preprocessing step
- tree traversal uses viewpoint
  - runtime, happens for many different viewpoints
- each plane divides world into near and far
  - for given viewpoint, decide which side is near and which is far
    - check which side of plane viewpoint is on independently for each tree vertex
    - tree traversal differs depending on viewpoint!
  - recursive algorithm
    - recurse on far side
    - draw object
    - recurse on near side

Traversing BSP Trees

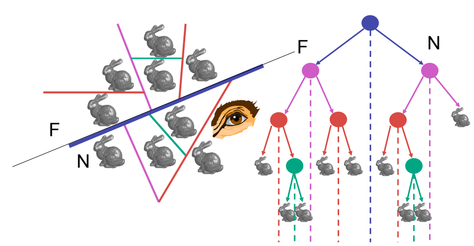
query: given a viewpoint, produce an ordered list of (possibly split) objects from back to front:

```
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);
```

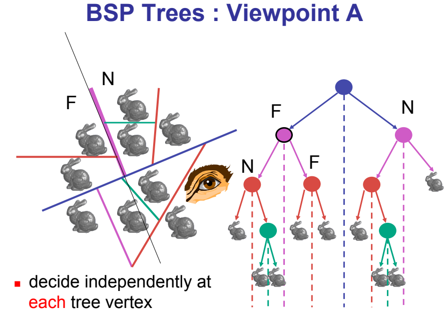

BSP Trees : Viewpoint A



BSP Trees : Viewpoint A

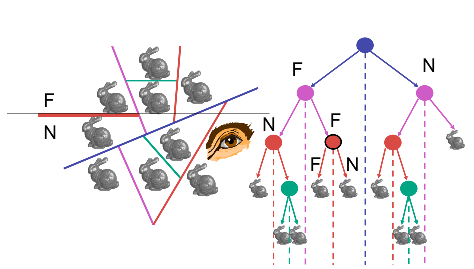


BSP Trees : Viewpoint A

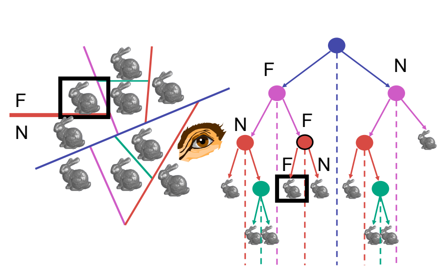


- decide independently at each tree vertex
- not just left or right child!

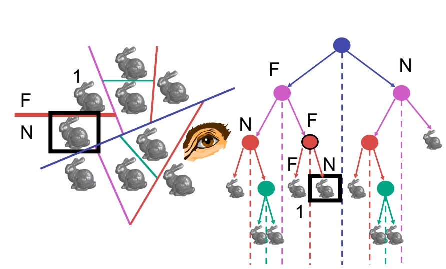
BSP Trees : Viewpoint A



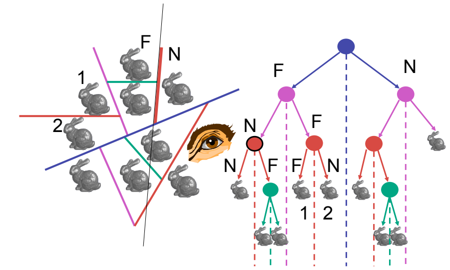
BSP Trees : Viewpoint A



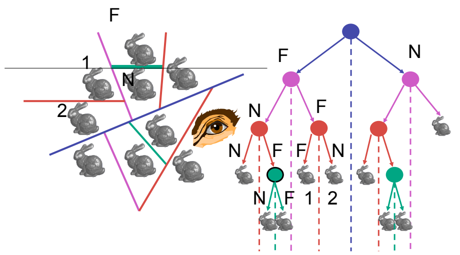
BSP Trees : Viewpoint A



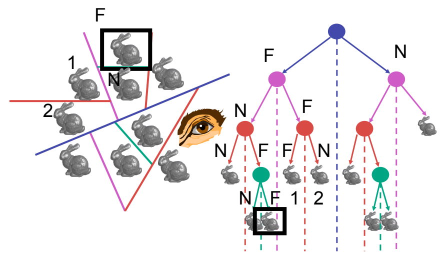
BSP Trees : Viewpoint A



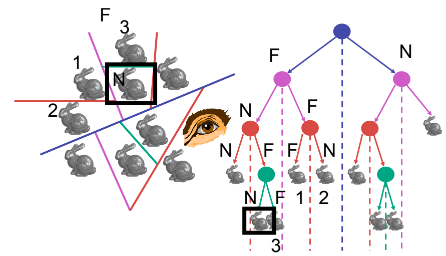
BSP Trees : Viewpoint A



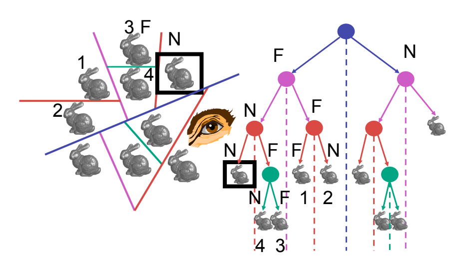
BSP Trees : Viewpoint A



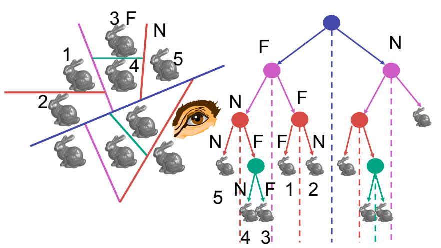
BSP Trees : Viewpoint A



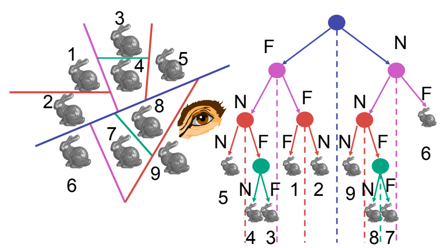
BSP Trees : Viewpoint A



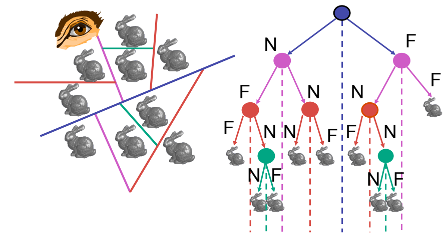
BSP Trees : Viewpoint A



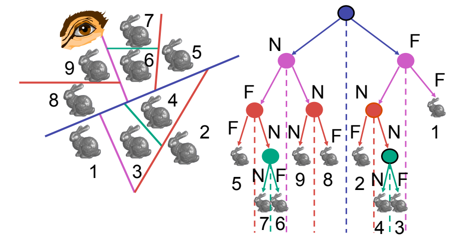
BSP Trees : Viewpoint A



BSP Trees : Viewpoint B



BSP Trees : Viewpoint B

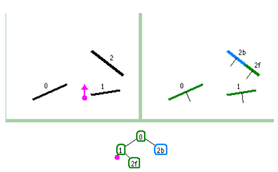


BSP Tree Traversal: Polygons

- 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

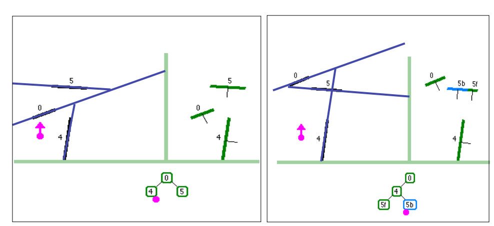
BSP Demo

- useful demo:
  http://symbolcraft.com/graphics/bsp



BSP Example

- order of insertion can affect half-plane extent

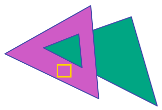


Summary: BSP Trees

- 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



65

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

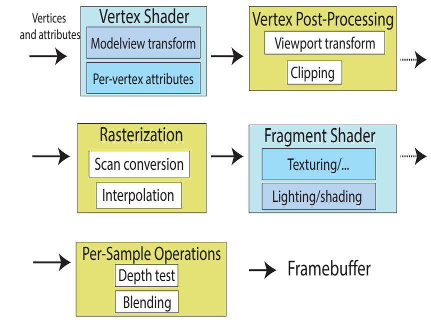
66

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

67

THE RENDERING PIPELINE

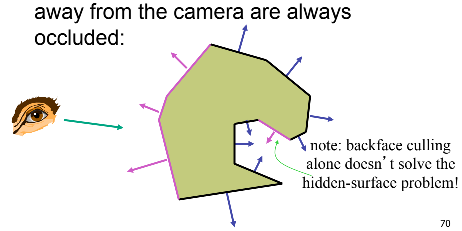


Backface Culling

69

Back-Face Culling

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



70

Back-Face Culling

- not rendering backfacing polygons improves performance
  - by how much?
    - reduces by about half the number of polygons to be considered for each pixel
  - optimization when appropriate

71

Back-Face Culling

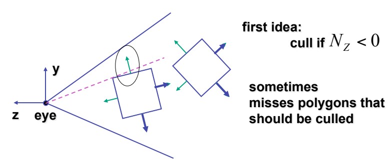
- most objects in scene are typically "solid"
- rigorously: orientable closed manifolds
  - orientable: must have two distinct sides
    - cannot self-intersect
    - a sphere is orientable since has two sides, 'inside' and 'outside'.
    - a Mobius strip or a Klein bottle is not orientable
  - closed: cannot "walk" from one side to the other
    - sphere is closed manifold
    - plane is not

Back-Face Culling

- examples of non-manifold objects:
  - a single polygon
  - a terrain or height field
  - polyhedron w/ missing face
  - anything with cracks or holes in boundary
  - one-polygon thick lampshade

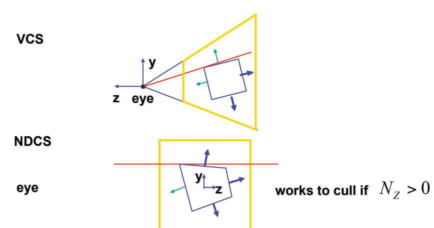
73

Back-face Culling: VCS



74

Back-face Culling: NDCS



75

Invisible Primitives

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

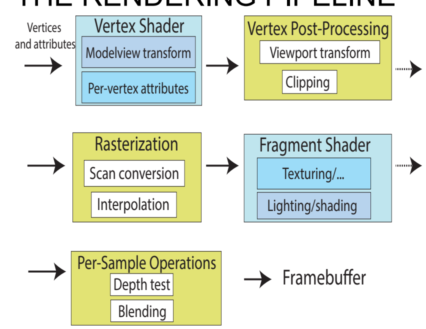
76

77

Blending

78

THE RENDERING PIPELINE



Alpha and Premultiplication

- specify opacity with alpha channel α
  - α=1: opaque, α=.5: translucent, α=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
      - 100% transparency can be represented with many different RGB values
    - pixel value is (1,0,0,.5)
    - upside: easy to change opacity of image, very intuitive
    - downside: compositing calculations are more difficult - not associative
  - elegant way: premultiply by α so store (αr, αg, αb,α)
    - intuition: alpha as screen/mesh
      - RGB specifies how much color object contributes to scene
      - alpha specifies how much object obscures whatever is behind it (coverage)
      - alpha of .5 means half the pixel is covered by the color, half completely transparent
      - only one 4-tuple represents 100% transparency: (0,0,0,0)
    - pixel value is (.5, 0, 0, .5)
    - upside: compositing calculations easy (& additive blending for glowing!)
    - downside: less intuitive

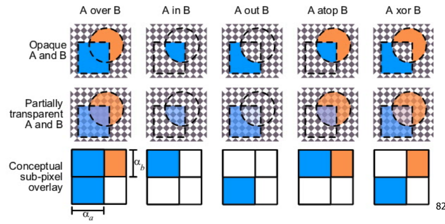
80

Alpha and Simple Compositing

- F is foreground, B is background, F over B
- premultiply math: uniform for each component, simple, linear
  - $R' = R_F+(1-A_F)*R_B$
  - $G' = G_F+(1-A_F)*G_B$
  - $B' = B_F+(1-A_F)*B_B$
  - $A' = A_F+(1-A_F)*A_B$
  - associative: easy to chain together multiple operations
- non-premultiply math: trickier
  - $R' = (R_F*A_F + (1-A_F)*R_B*A_B)/A'$
  - $G' = (G_F*A_F + (1-A_F)*G_B*A_B)/A'$
  - $B' = (B_F*A_F + (1-A_F)*B_B*A_B)/A'$
  - $A' = A_F+(1-A_F)*A_B$
  - 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



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
- premultiply allows both conventional blend and additive blend
  - 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