Visible Surface Determination

CS4600 *Computer Graphics*

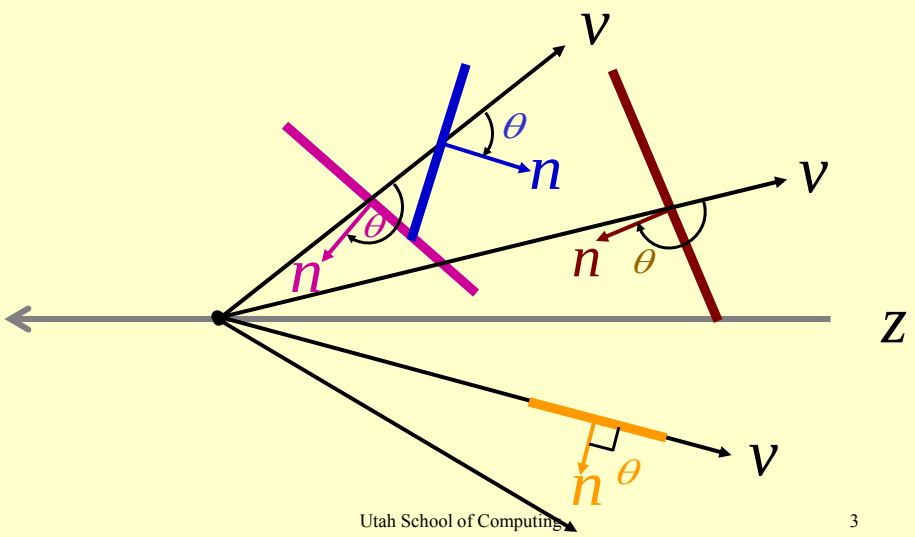
From Rich Riesenfeld

Fall 2015

Class of Algorithms

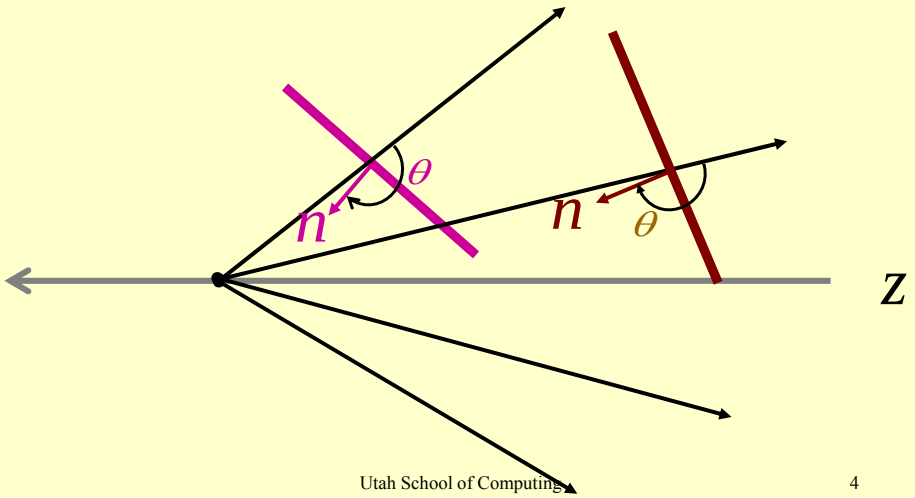
- Object (Model) Space Algorithms
  - Work in the model data space
- Image Space Algorithms
  - Work in the projected space
  - Most common VSD domain

Back Face Culling: Object Space



Utah School of Computing 3

Back Face Culling: Object Space

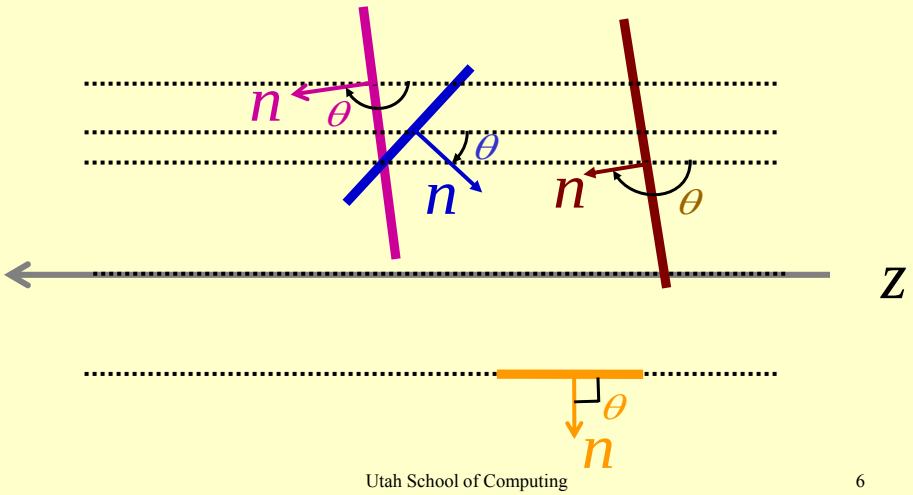


Utah School of Computing 4

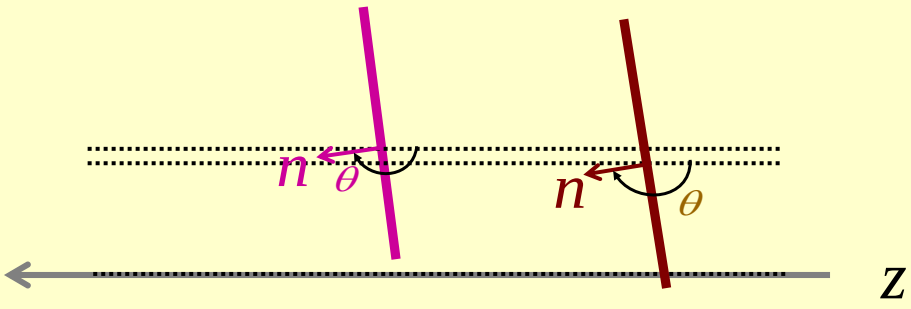
Back Face Culling Test

- For Object Space look at sign of $v \cdot n$
- For Image Space look at sign of n_z

Back Face Culling: Image Space

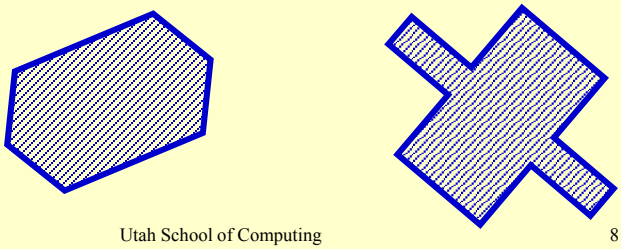


Back Face Culling: Image Space

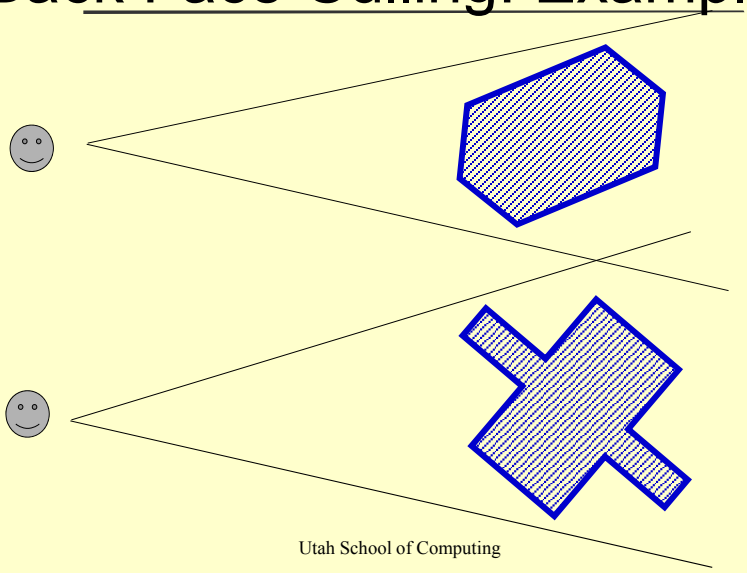


Back Face Culling

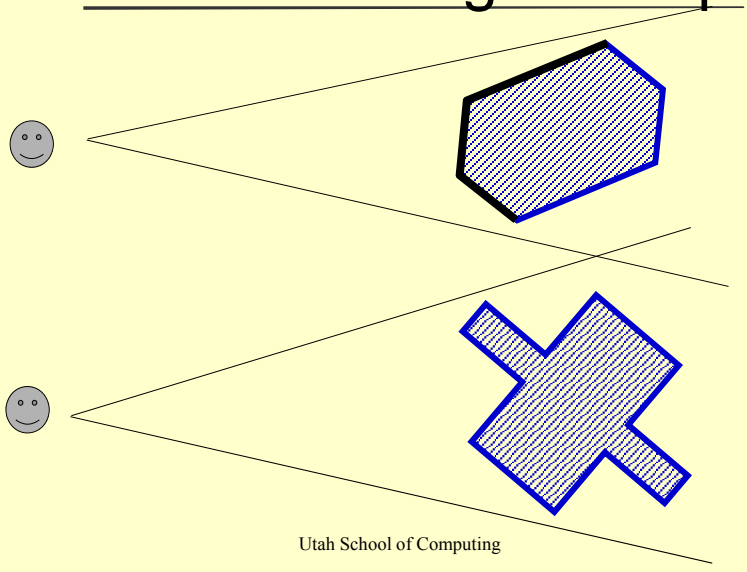
- Completes the job for convex polyhedral objects
- Nonconvex objects need additional processing beyond back face culling



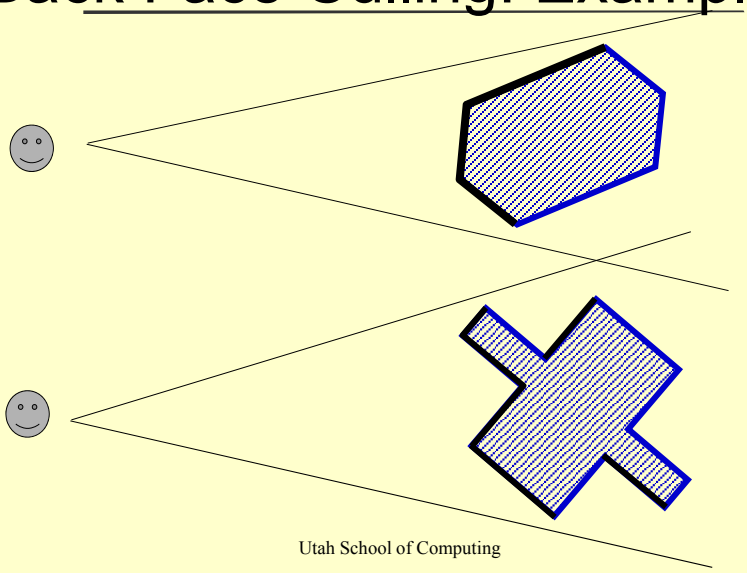
Back Face Culling: Examples



Back Face Culling: Examples

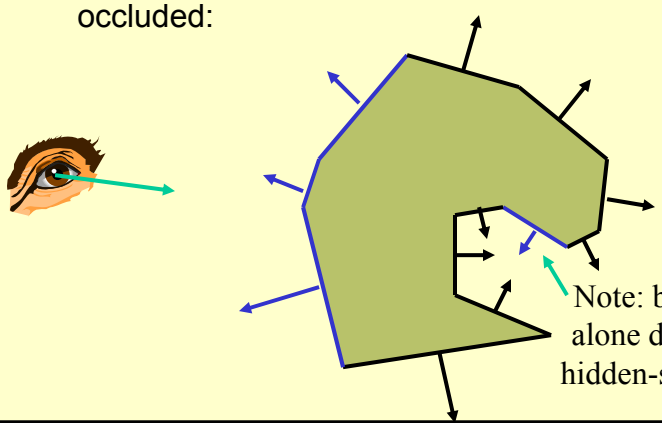


Back Face Culling: Examples



Back-Face Culling

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



Note: backface culling alone doesn't solve the hidden-surface problem!

Back-Face Culling

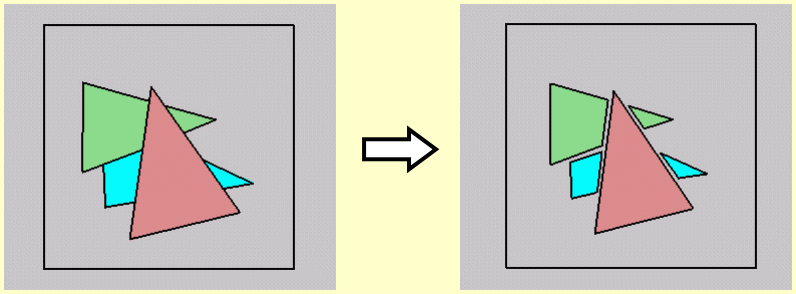
- Not rendering backfacing polygons improves performance
  - *By how much?*
    - *Reduces by about half the number of polygons to be considered for each pixel*

Silhouettes

- For Object Space $v \cdot n = 0$
- For Image Space $n_z = 0$

Occlusion

- For most interesting scenes, some polygons will overlap:



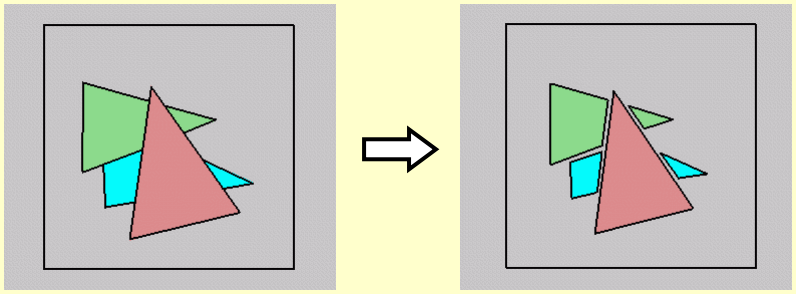
- To render the correct image, we need to determine which polygons *occlude* which

Painter's Algortihm

- How do painter's solve this?

Painter's Algorithm

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



  - Draw blue, then green, then orange
- *Will this work in the general case?*

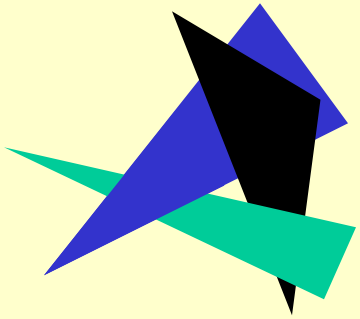
Painter's Algortihm

- How do painter's solve this?

- Sort the polygons in depth order
- Draw the polygons back-to-front
- QED

Painter's Algorithm: Problems

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



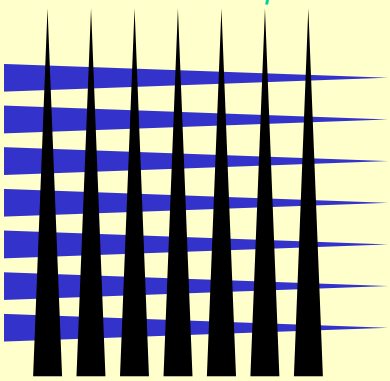
Analytic Visibility Algorithms

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

  - Now known as *analytic visibility algorithms*

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 two:
  - *Warnock's Algorithm*
  - *Binary Space-Partition (BSP) Trees*

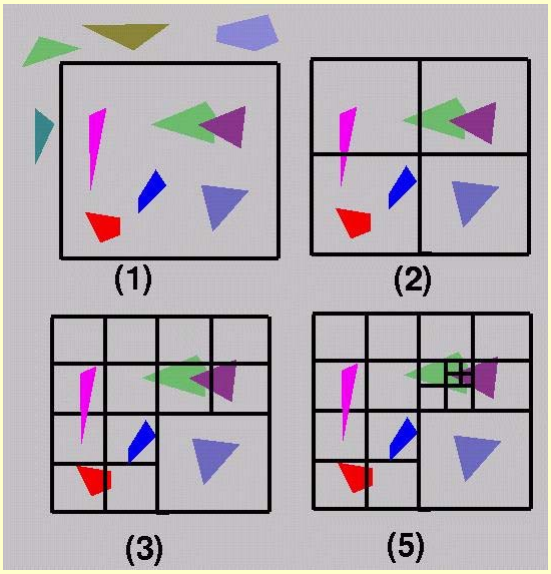
Warnock's Algorithm (1969)

PIXAR uses a similar scheme

- Elegant scheme based on a powerful general approach common in graphics: *if the situation is too complex,* ***subdivide***
  - Start with a *root viewport* and a list of all primitives
  - Then recursively:
    - Clip objects to viewport
    - If number of objects incident to viewport is zero or one, visibility is trivial
    - Otherwise, subdivide into smaller viewports, distribute primitives among them, and recurse

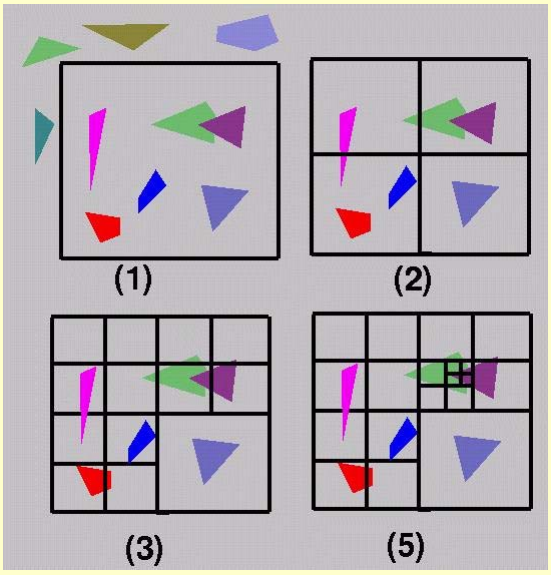
Warnock's Algorithm

- *What is the terminating condition?*
- *How to determine the correct visible surface in this case?*



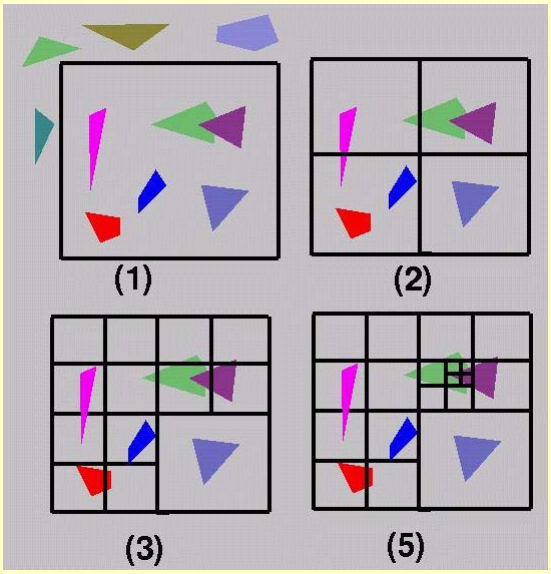
Warnock's Algorithm

- *What is the terminating condition?*
  - ***One polygon per cell***
- *How to determine the correct visible surface in this case?*



Warnock's Algorithm

- *What is the terminating condition?*
  - ***One polygon per cell***
- *How to determine the correct visible surface in this case?*
  - ***Cell = single pixel***



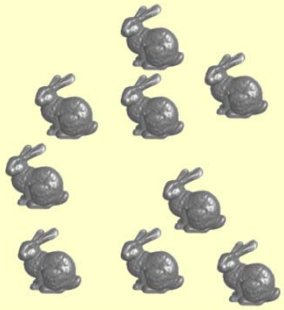
Warnock's Algorithm

- Pros:
  - Very elegant scheme
  - Extends to any primitive type
- Cons:
  - Hard to embed hierarchical schemes in hardware
  - Complex scenes usually have small polygons and high *depth complexity*
    - Thus most screen regions come down to the single-pixel case

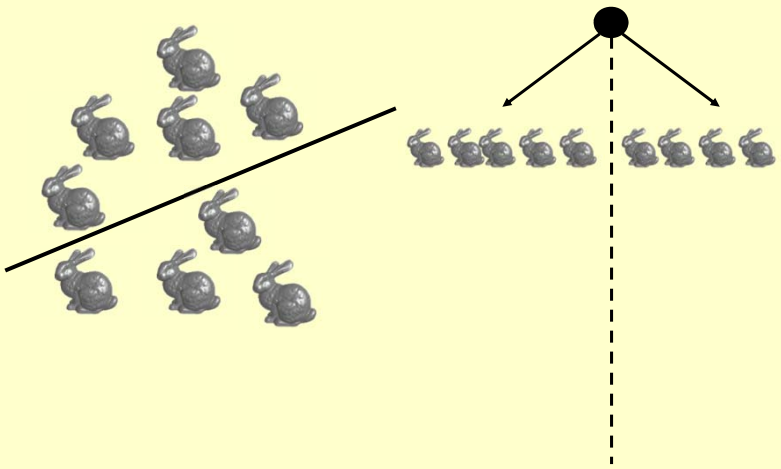
Binary Space Partition Trees (1979)

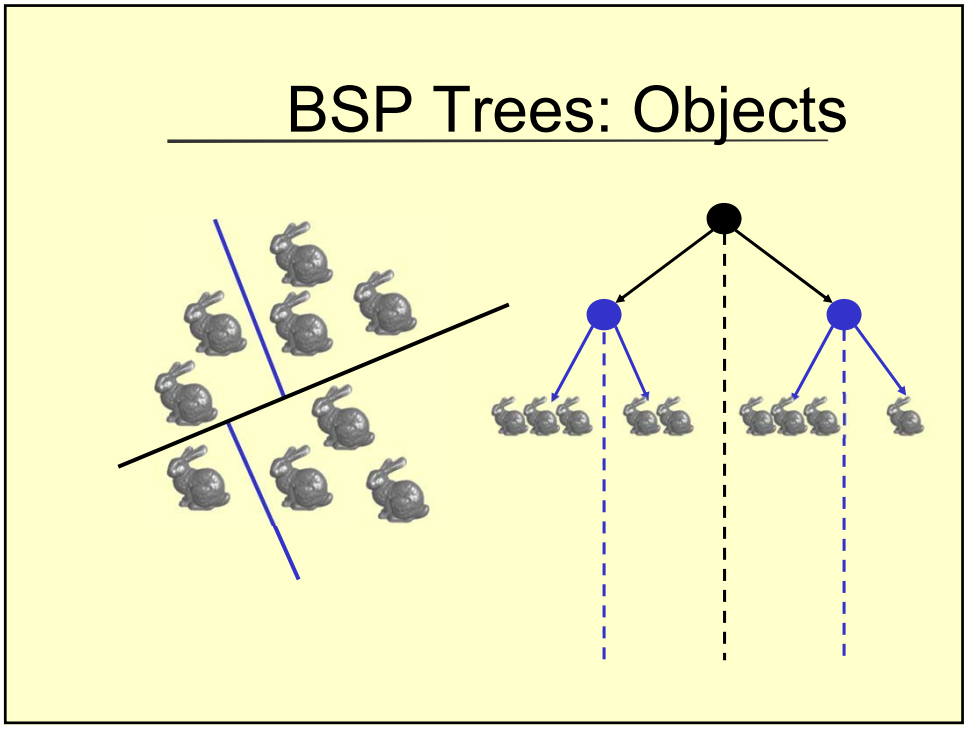
- BSP tree: organize all of space (hence *partition*) into a binary tree
  - *Preprocess*: overlay a binary tree on objects in the scene
  - *Runtime*: correctly traversing this tree enumerates objects from back to front
  - Idea: divide space recursively into half-spaces by choosing *splitting planes*
    - Splitting planes can be arbitrarily oriented
    - Notice: nodes are always convex

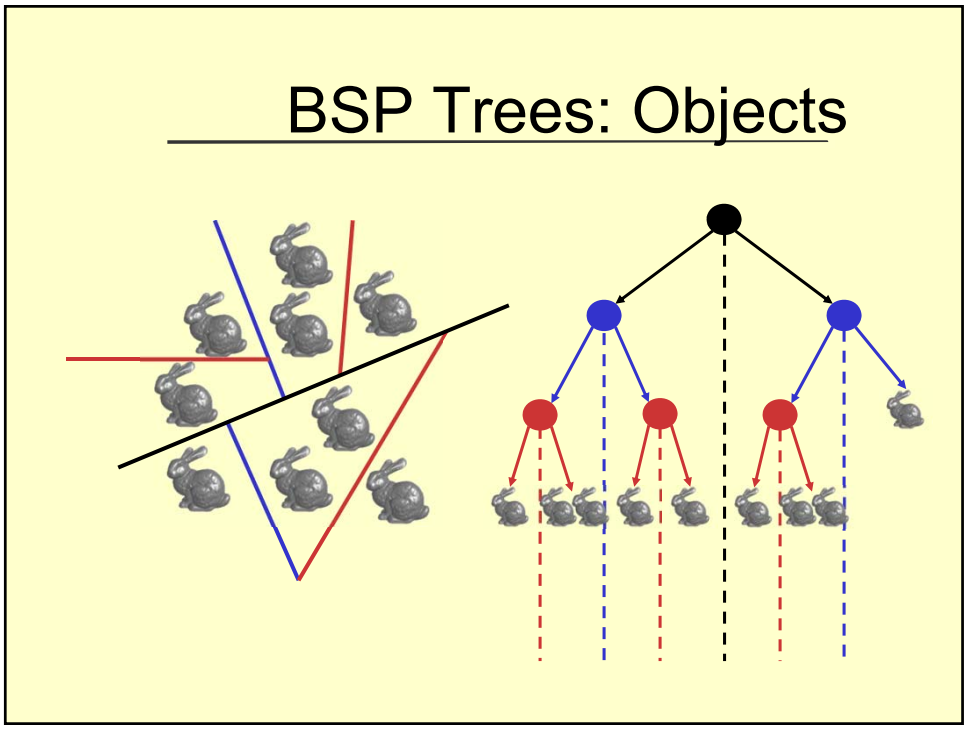
BSP Trees: Objects



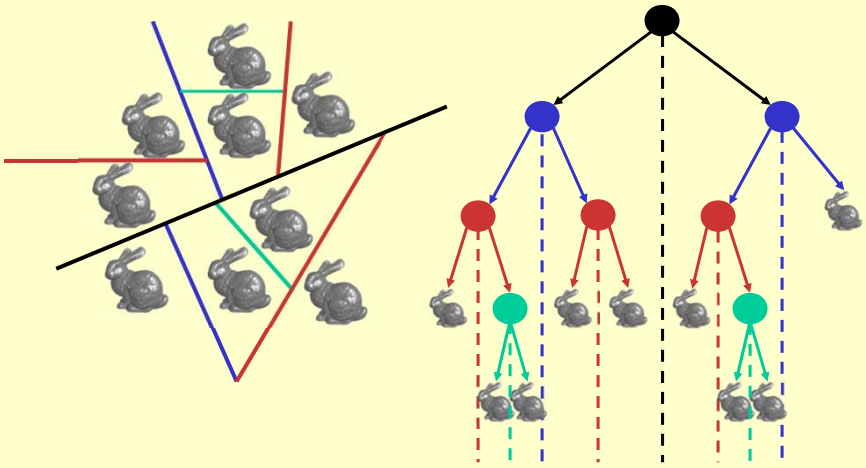
BSP Trees: Objects




BSP Trees: Objects


BSP Trees: Objects

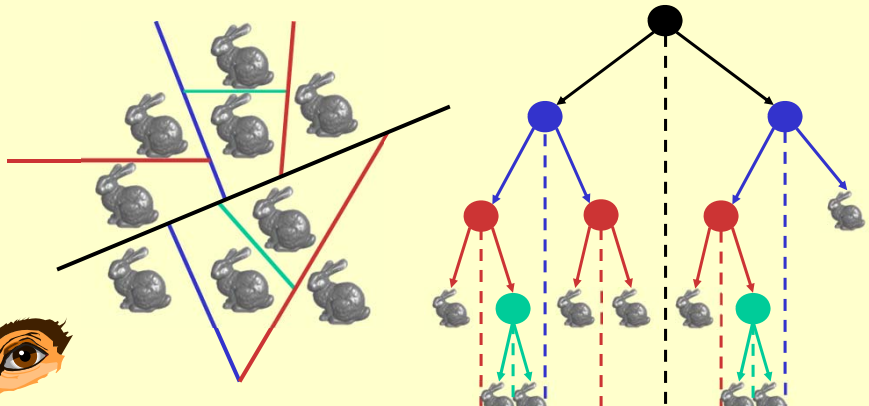
BSP Trees: Objects



Rendering BSP Trees

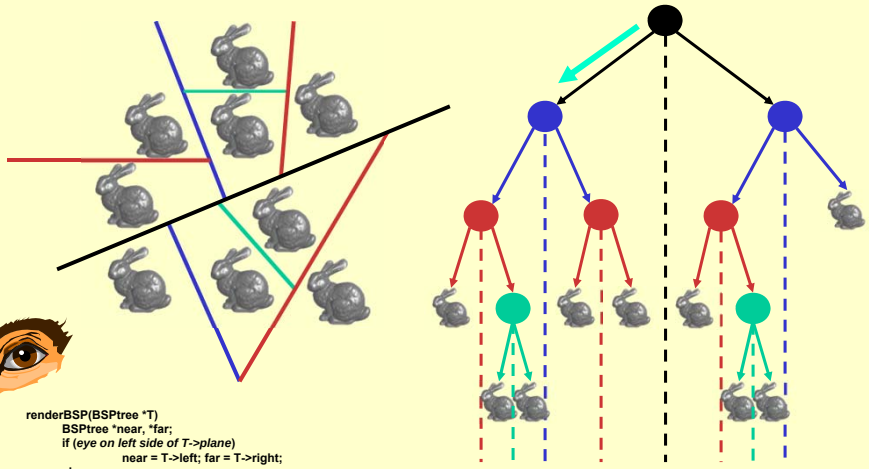
```
renderBSP(BSPtree *T)
  BSPtree *near, *far;
  if (T is a leaf node)
      renderObject(T)
  else {
      if (eye on left side of T->plane)
           near = T->left; far = T->right;
      else
           near = T->right; far = T->left;
      renderBSP(far);
      renderBSP(near);
  }
```

Rendering BSP Trees



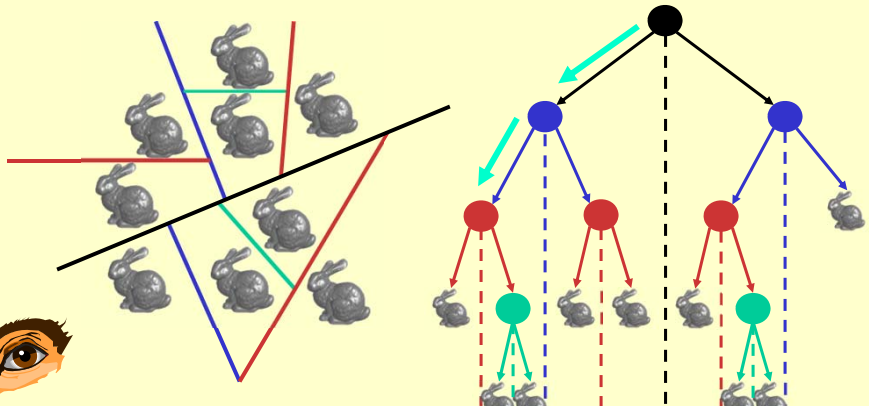
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



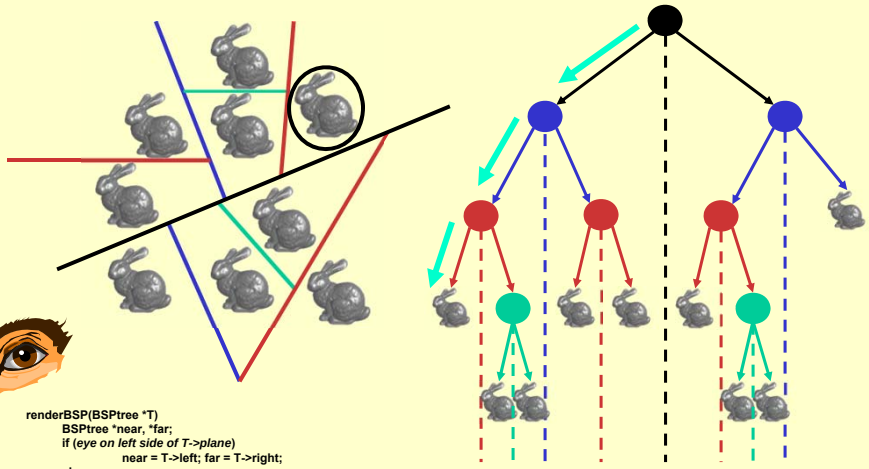
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



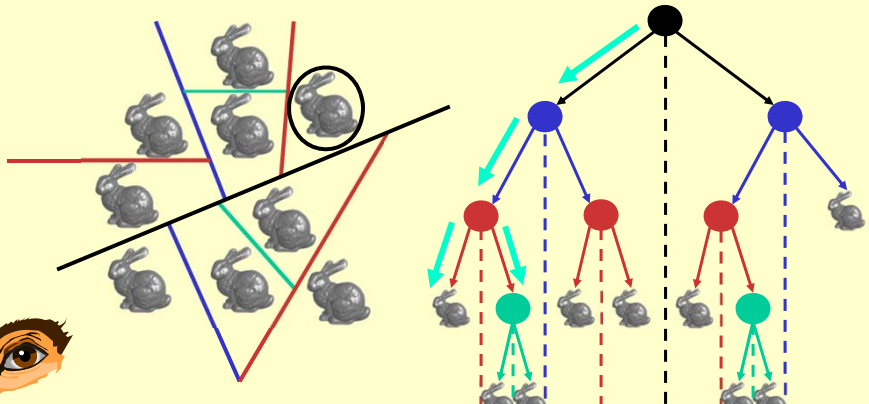
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



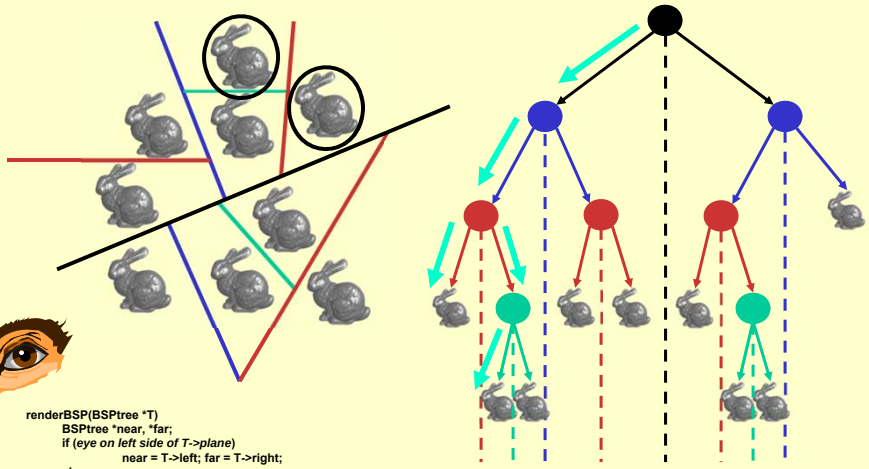
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



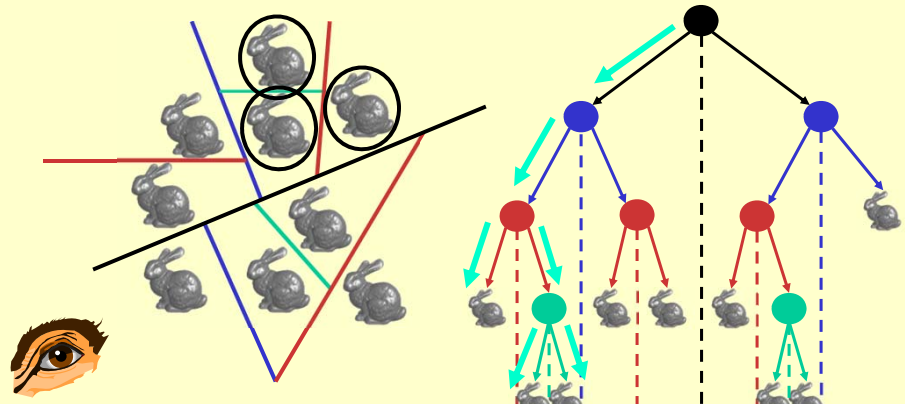
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



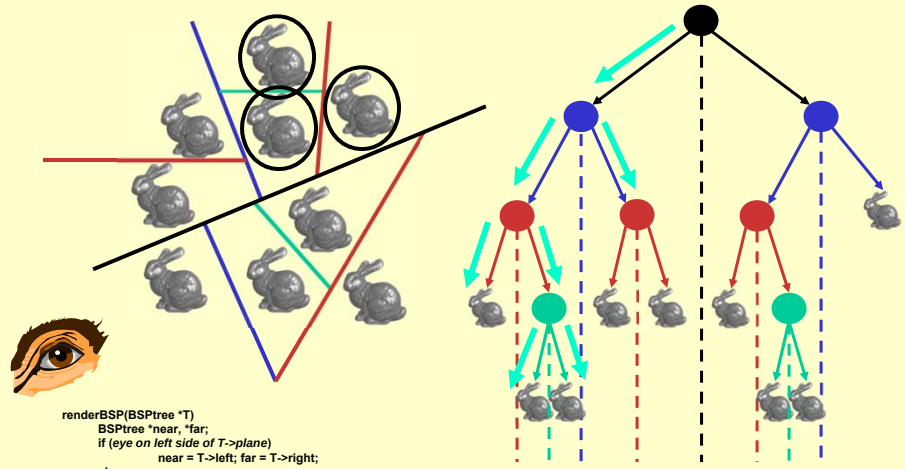
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



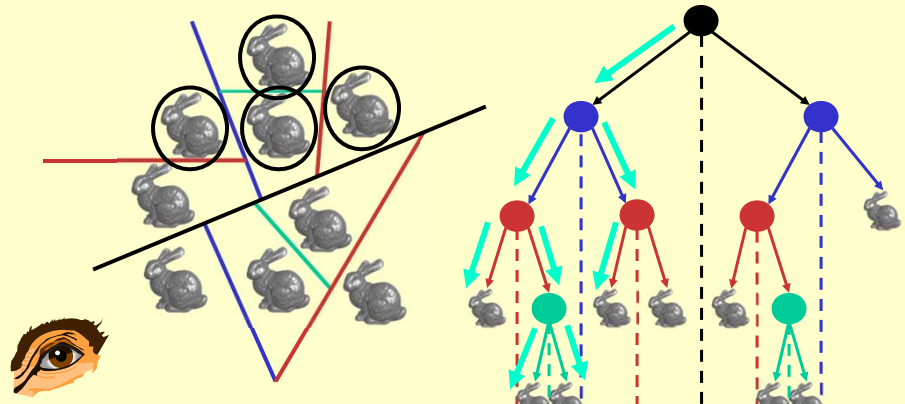
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



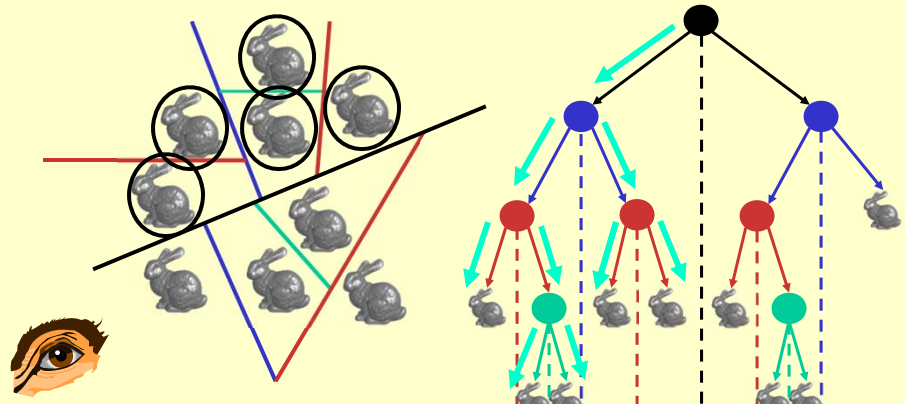
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



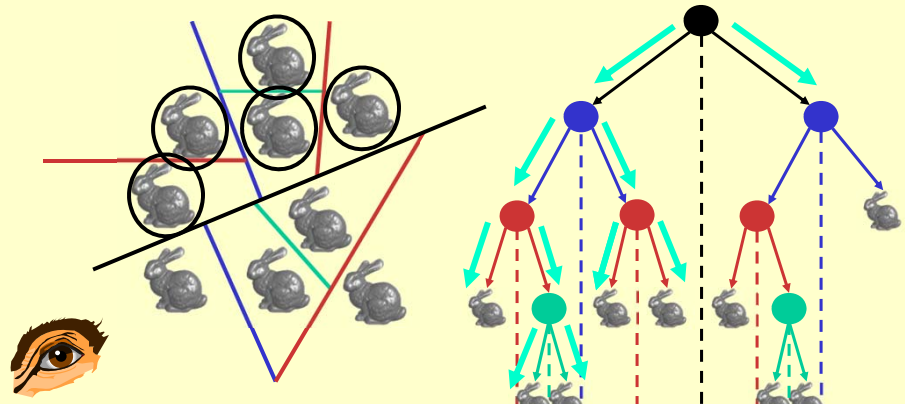
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



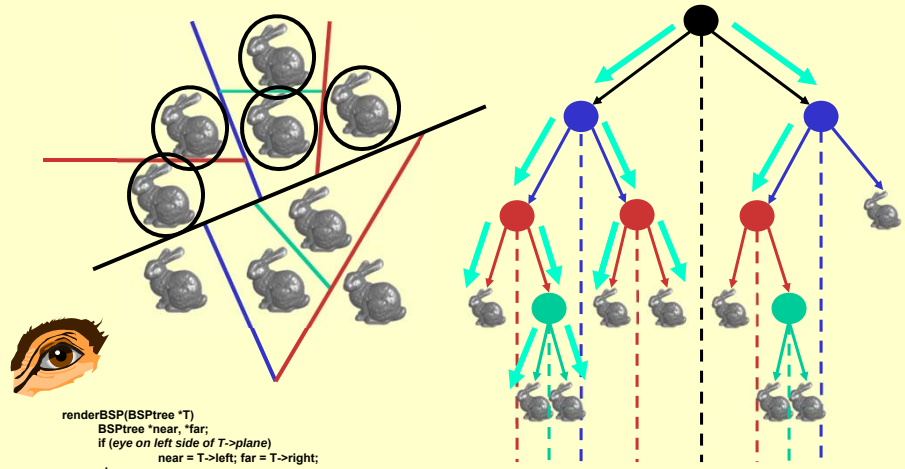

```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



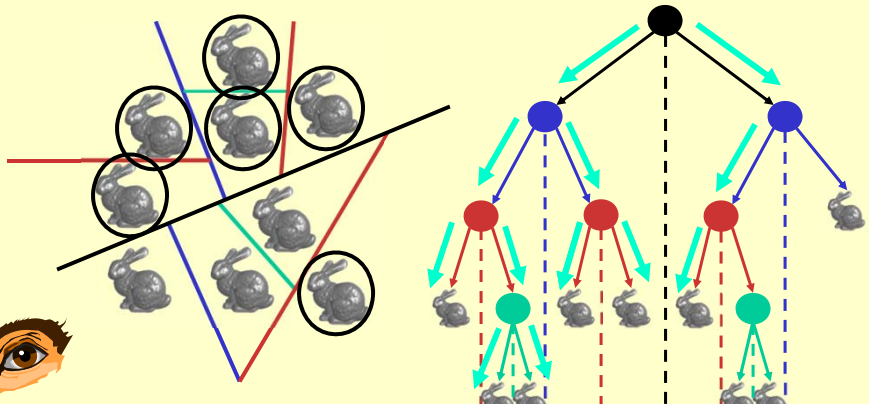
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



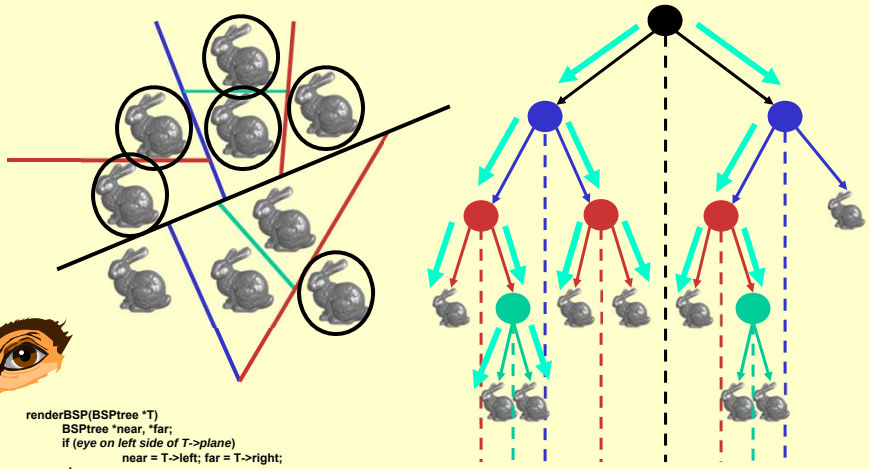
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



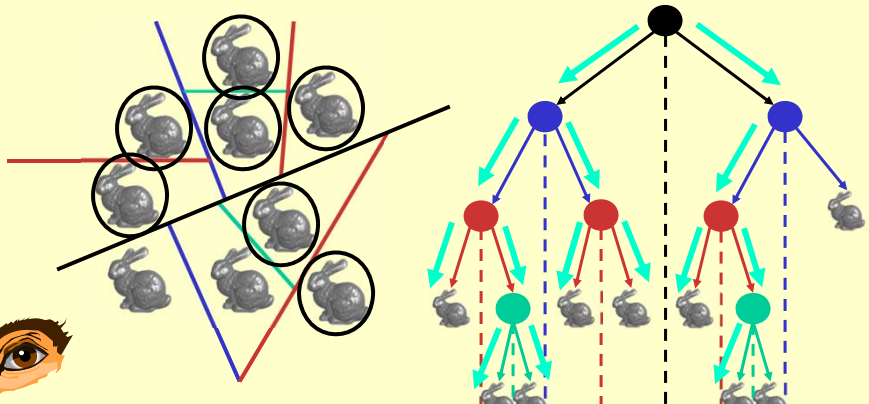
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



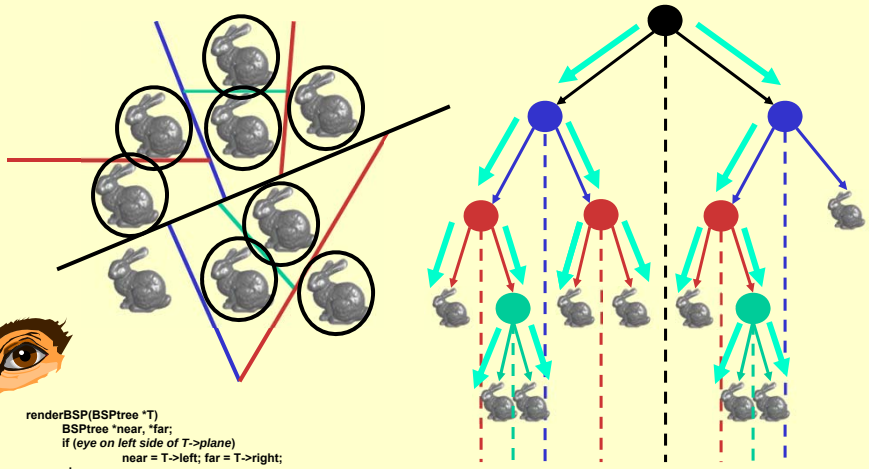
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



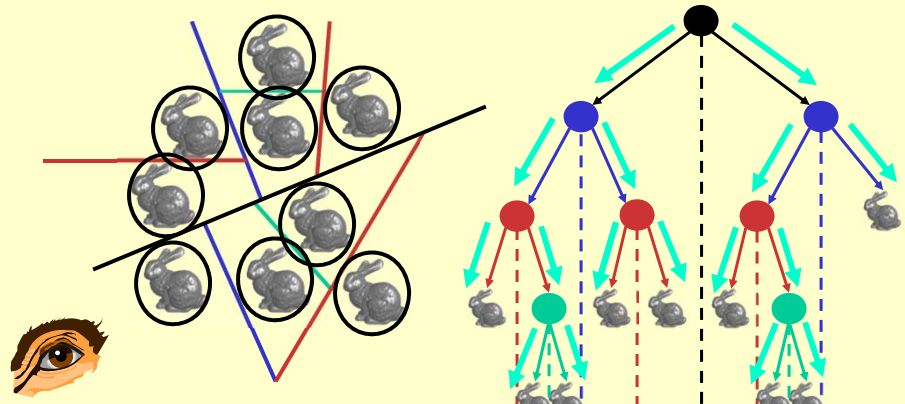
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



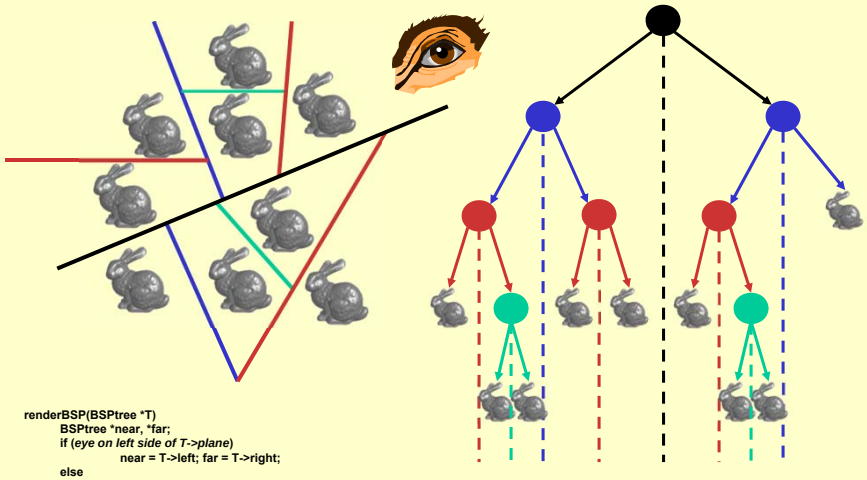
```
renderBSP(BSPtree *T)
        BSPtree *near, *far;
        if (eye on left side of T->plane)
                near = T->left; far = T->right;
        else
                near = T->right; far = T->left;
```

Rendering BSP Trees



```
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;
```

Rendering BSP Trees



```
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;
```

3D Polygons: BSP Tree Construction

- Split along the plane containing any polygon
- Classify all polygons into positive or negative half-space of the plane
  - If a polygon intersects plane, split it into two
- Recurse down the negative half-space
- Recurse down the positive half-space

Polygons: BSP Tree Traversal

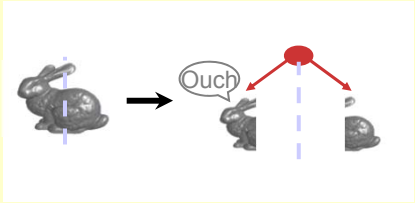
- Query: given a viewpoint, produce an ordered list of (possibly split) polygons from back to front:

```
BSPnode::Draw(Vec3 viewpt)
  Classify viewpt: in + or - half-space of node->plane?
  /* Call that the "near" half-space */
      farchild->draw(viewpt);
      render node->polygon; /* always on node->plane */
      nearchild->draw(viewpt);
```

- Intuitively: at each partition, draw the stuff on the farther side, then the polygon on the partition, then the stuff on the nearer side

Discussion: BSP Tree Cons

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



  - Worst case: can create up to $O(n^3)$ objects!

BSP Demo

- Nice demo:

  http://www.symbolcraft.com/graphics/bsp/

Summary: BSP Trees

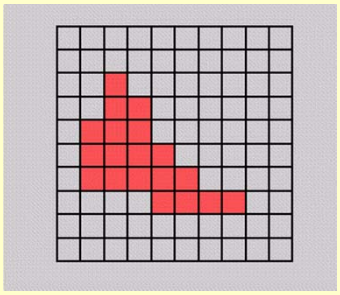
- Pros:
  - Simple, elegant scheme
  - Only writes to framebuffer (i.e., painters algorithm)
    - Once very popular for video games (but getting less so)
    - Widely used in ray-tacing
- Cons:
  - Computationally intense preprocess stage restricts algorithm to static scenes
  - Worst-case time to construct tree: $O(n^3)$
  - Splitting increases polygon count
    - Again, $O(n^3)$ worst case

The Z-Buffer Algorithm

- Both BSP trees and Warnock's algorithm were proposed when memory was expensive
  - Example: first 512x512 framebuffer > $50,000!
- Ed Catmull (mid-70s) proposed a radical new approach called *z-buffering*.
- The big idea: resolve visibility *independently at each pixel*

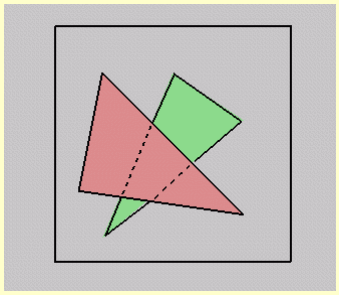
The Z-Buffer Algorithm

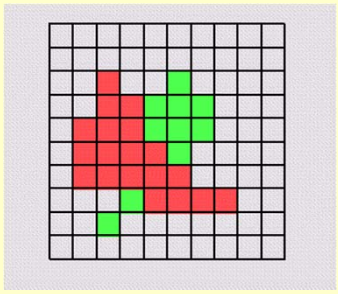
- We know how to rasterize polygons into an image discretized into pixels:



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 (Z in eye coordinates) through projection transform
  - Use canonical viewing volumes (Clip / NDC frames)
  - Can transform clip perspective volume into NDC parallel volume with:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & (n+f) & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

z-Buffer (Depth Buffer)

Conceptually:

$$\underset{(x,y)}{Sort} \; (\max z_{xy})$$

$$\underset{(x,y)}{Sort} \; (\min z_{xy})$$

The Z-Buffer Algorithm

- Augment 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 and store in pixel of Z-buffer
  - Suppress writing to a pixel if its Z value is more distant than the Z value already stored there

Interpolating Z

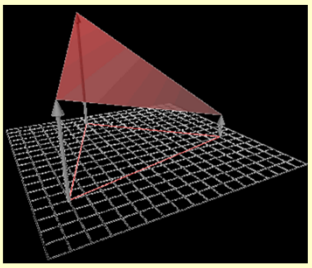
- **Edge equations: Z is just another planar parameter:**

  $z = (-D - Ax - By) / C$

  If walking across scanline by (Δx)

  $z_{new} = z - (A/C)(\Delta x)$

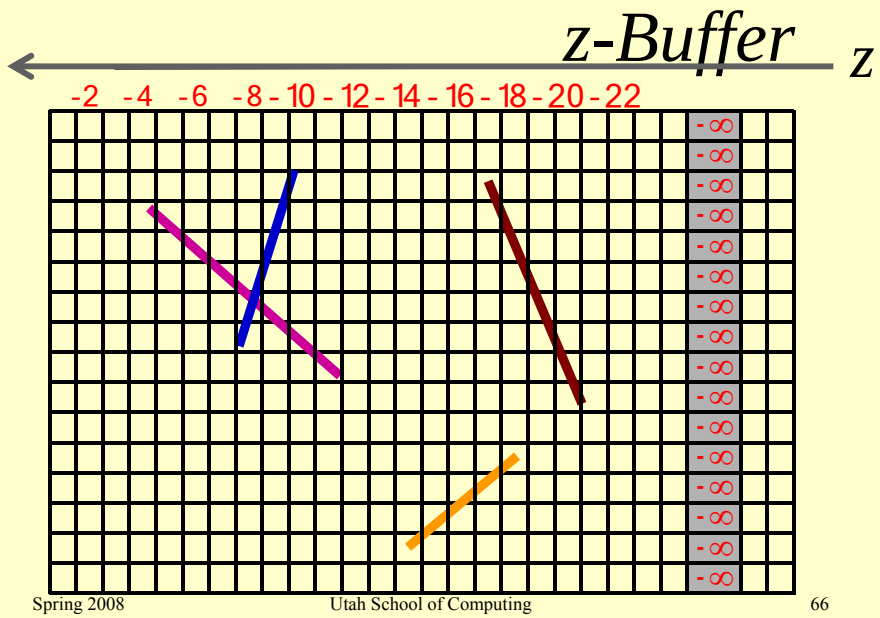
  - Like drawing lines?
  - Total cost:
    - 1 more parameter to increment in inner loop
    - 3x3 matrix multiply for setup

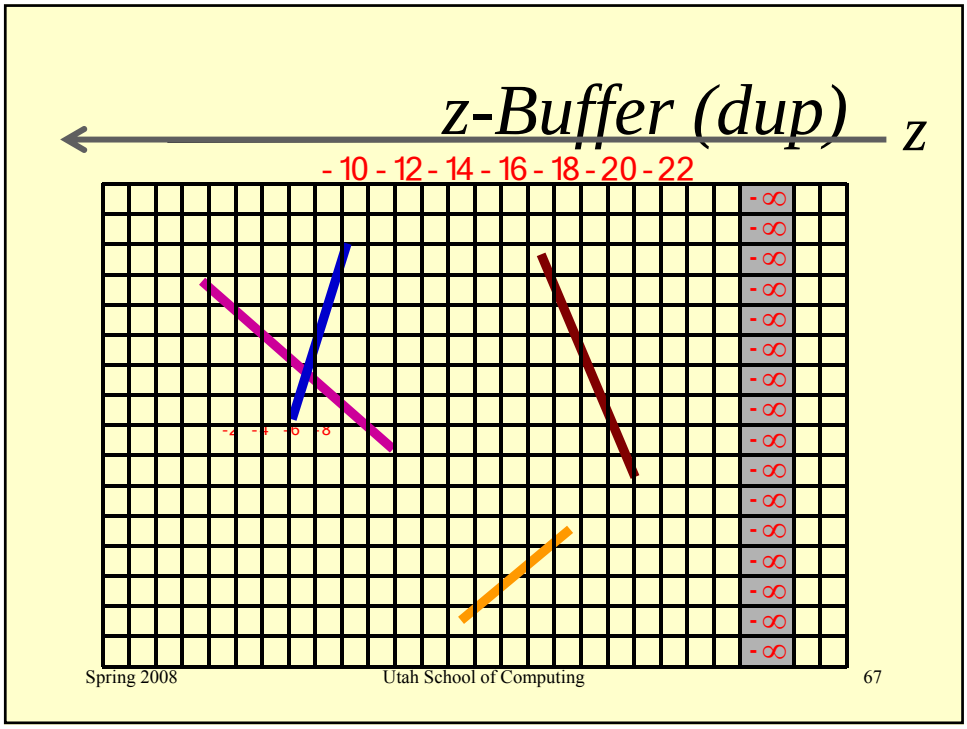


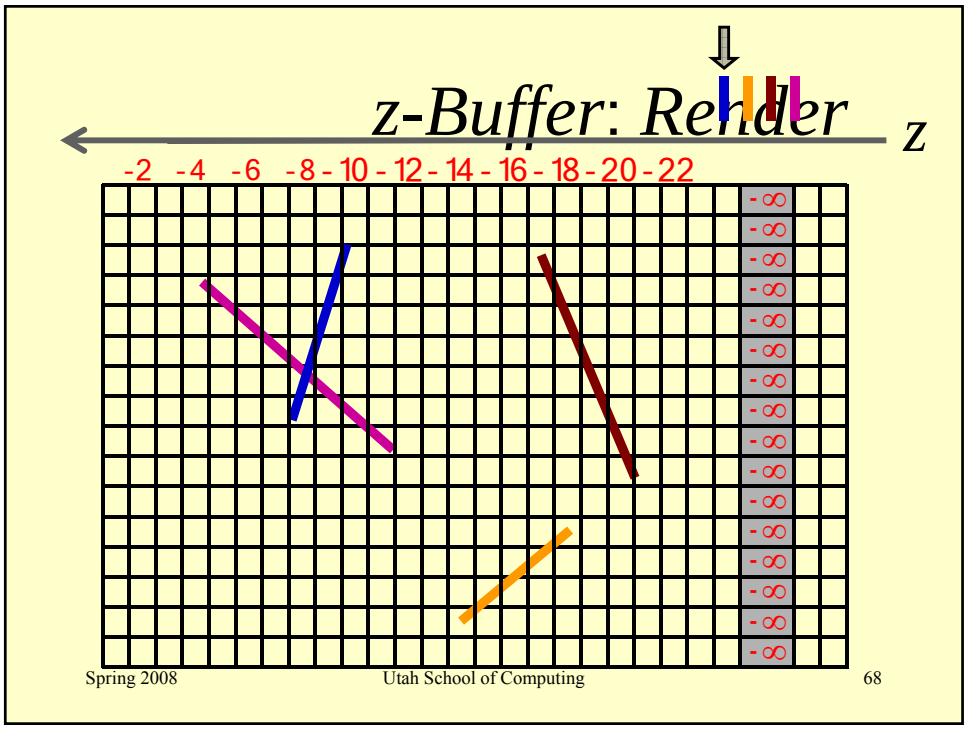
- **Edge walking: just interpolate Z along edges and across spans**

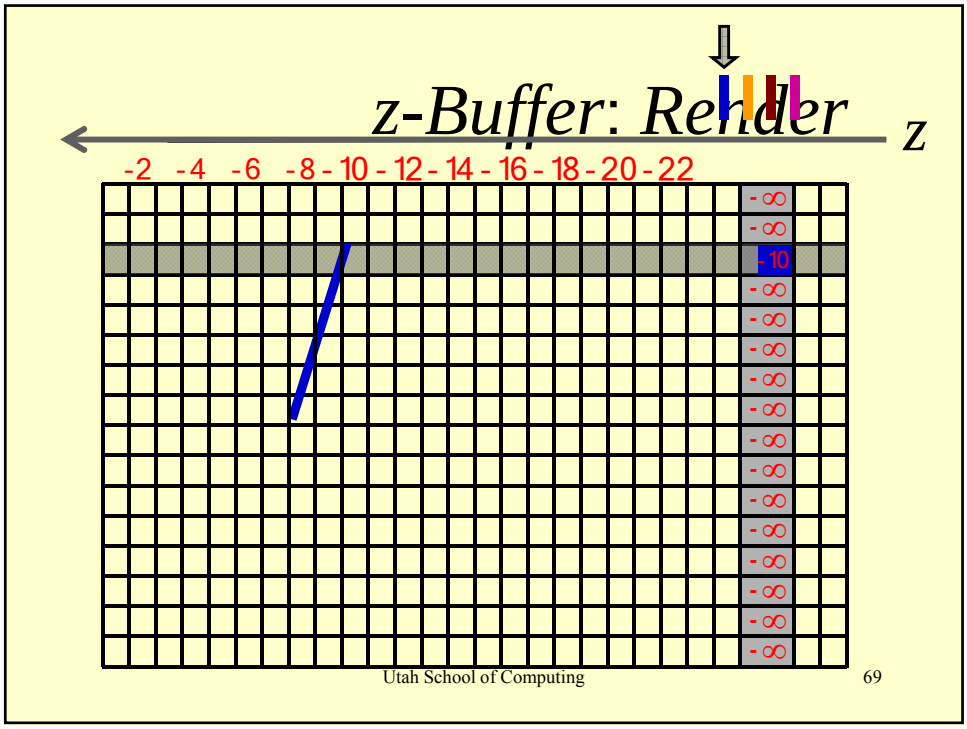
The Z-Buffer Algorithm

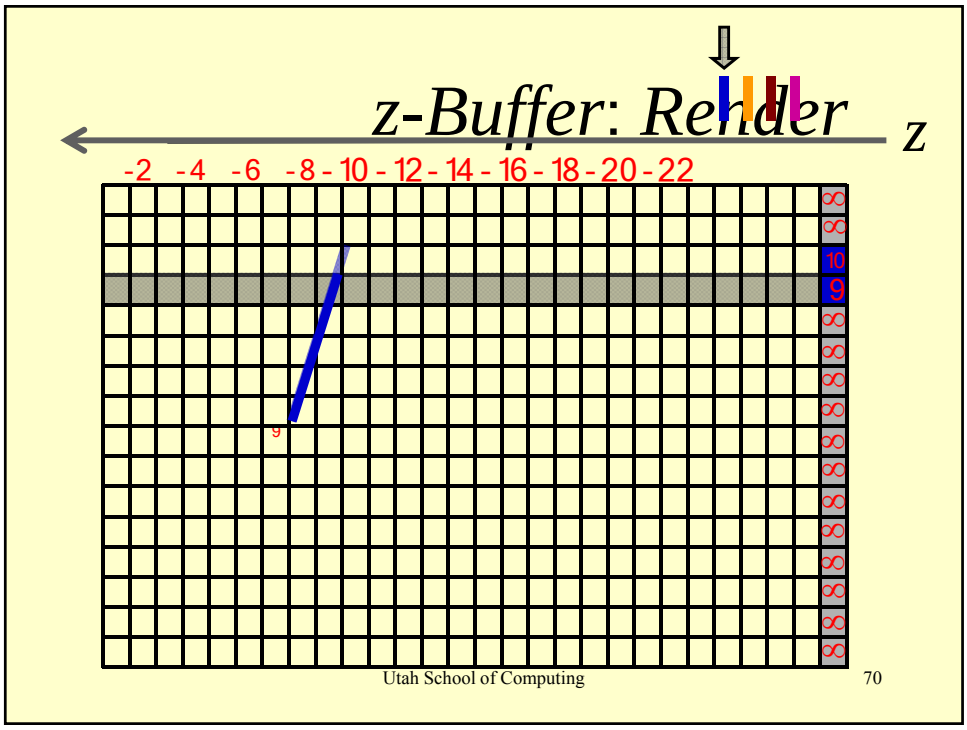
- *How much memory does the Z-buffer use?*
- *Does the image rendered depend on the drawing order?*
- *Does the time to render the image depend on the drawing order?*
- *How does Z-buffer load scale with visible polygons? With framebuffer resolution?*

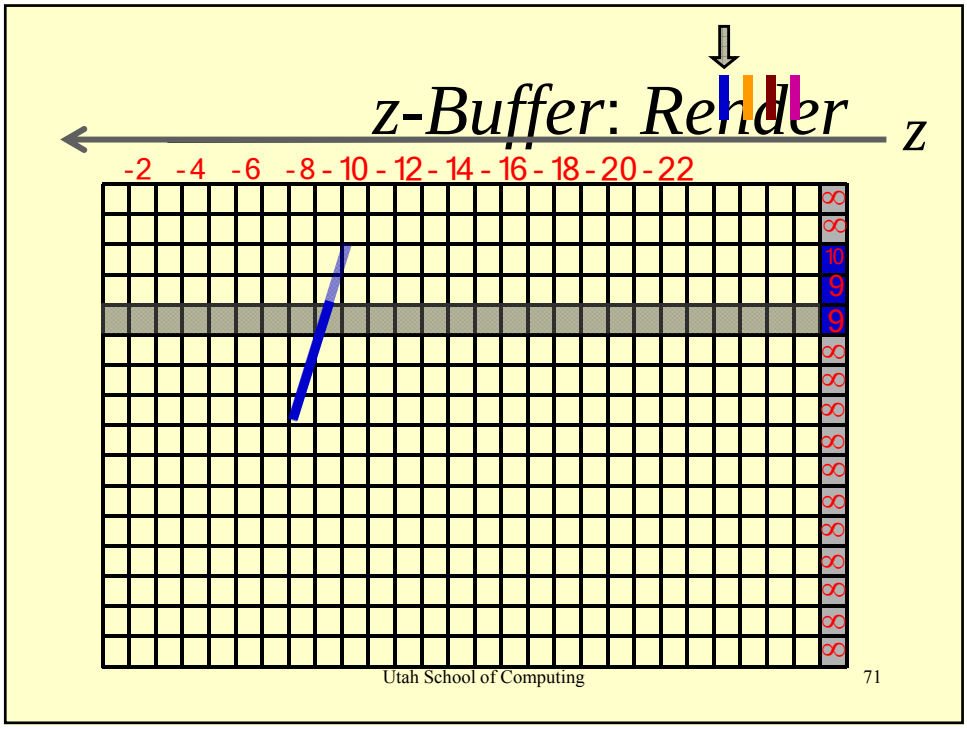


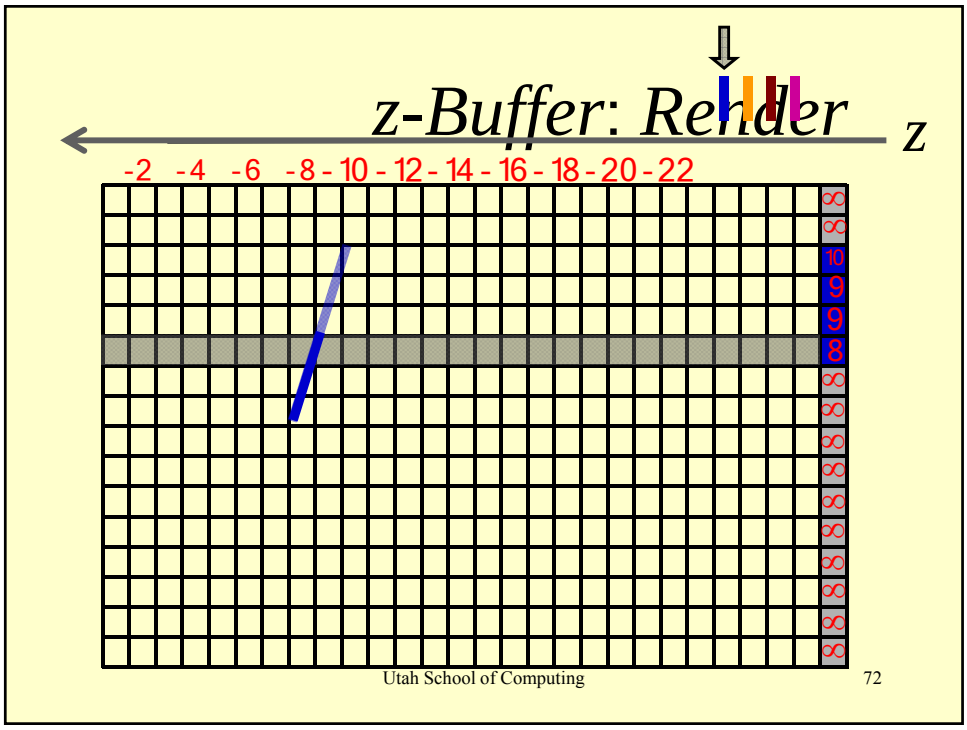

z-Buffer (dup)
z
- 10 - 12 - 14 - 16 - 18 - 20 - 22
- ∞
Spring 2008
Utah School of Computing
67

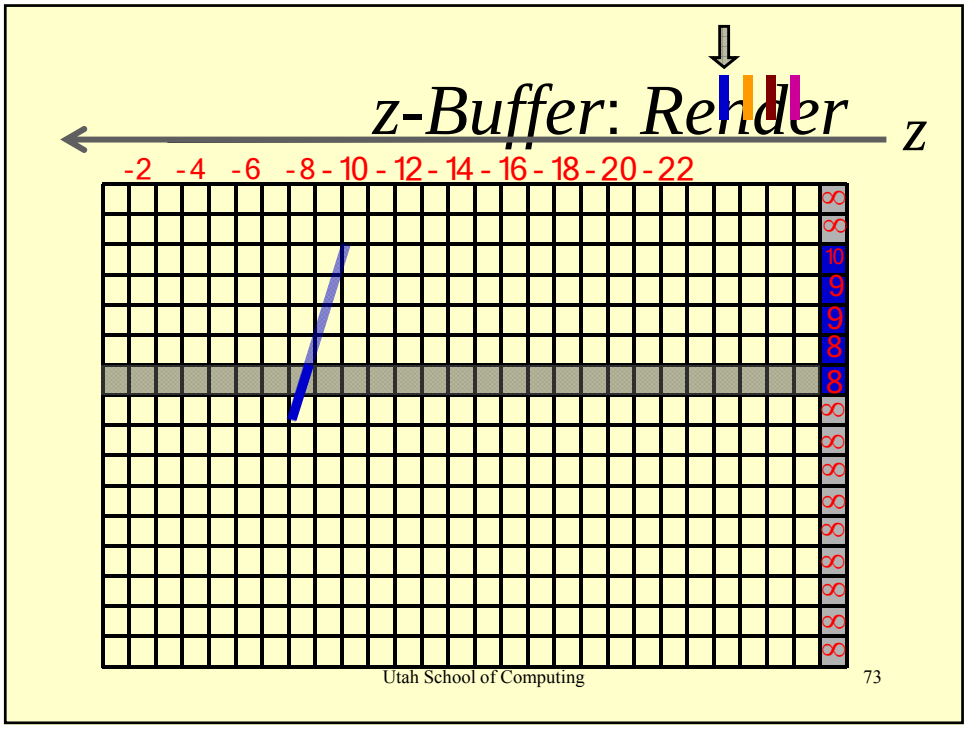

z-Buffer: Render
z
-2 -4 -6 -8 - 10 - 12 - 14 - 16 - 18 - 20 - 22
- ∞
Spring 2008
Utah School of Computing
68

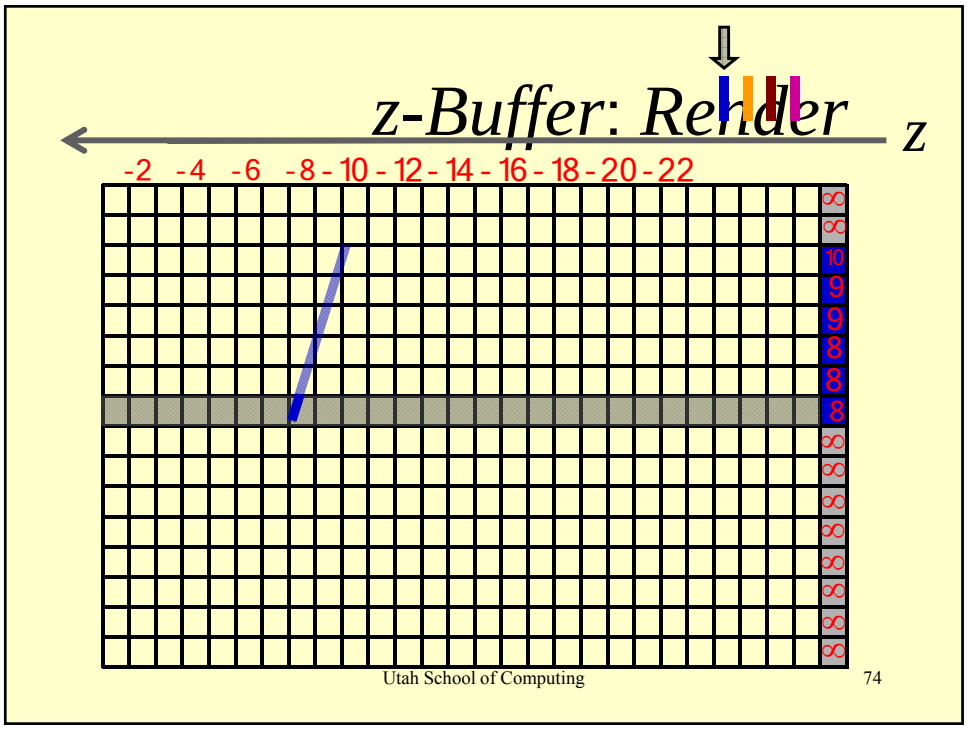

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
-∞
-∞
-10
-∞
-∞
-∞
-∞
-∞
-∞
-∞
-∞
-∞
-∞
-∞
-∞
Utah School of Computing
69

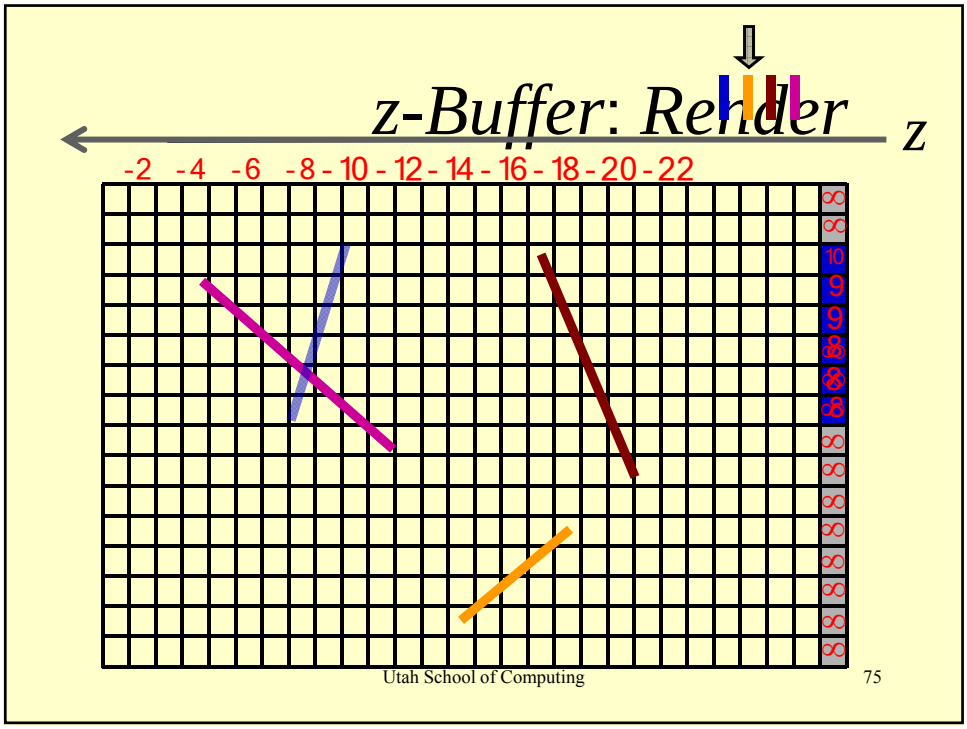

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞
∞
-10
-9
∞
∞
∞
∞
∞
∞
∞
∞
∞
∞
∞
-9
Utah School of Computing
70

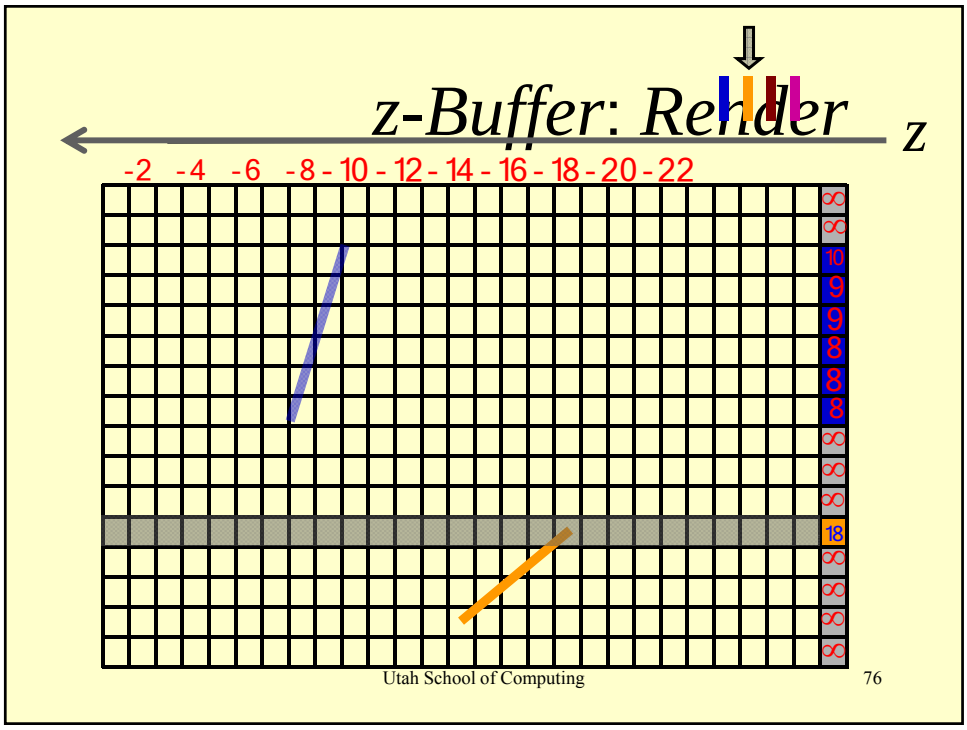

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
71

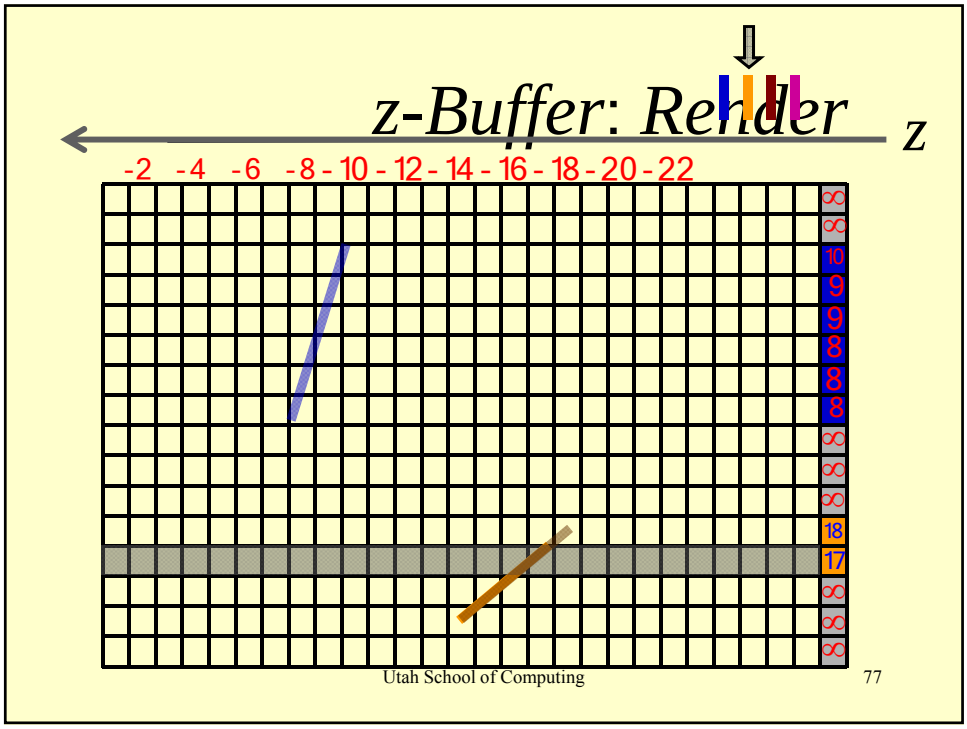

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
72

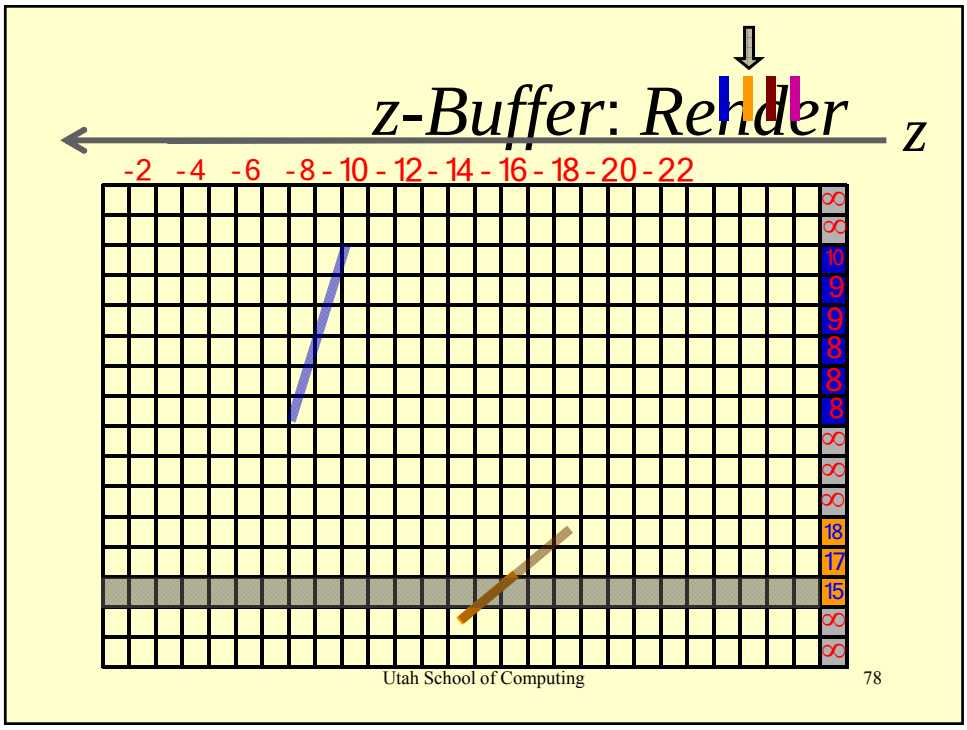

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 8 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
73

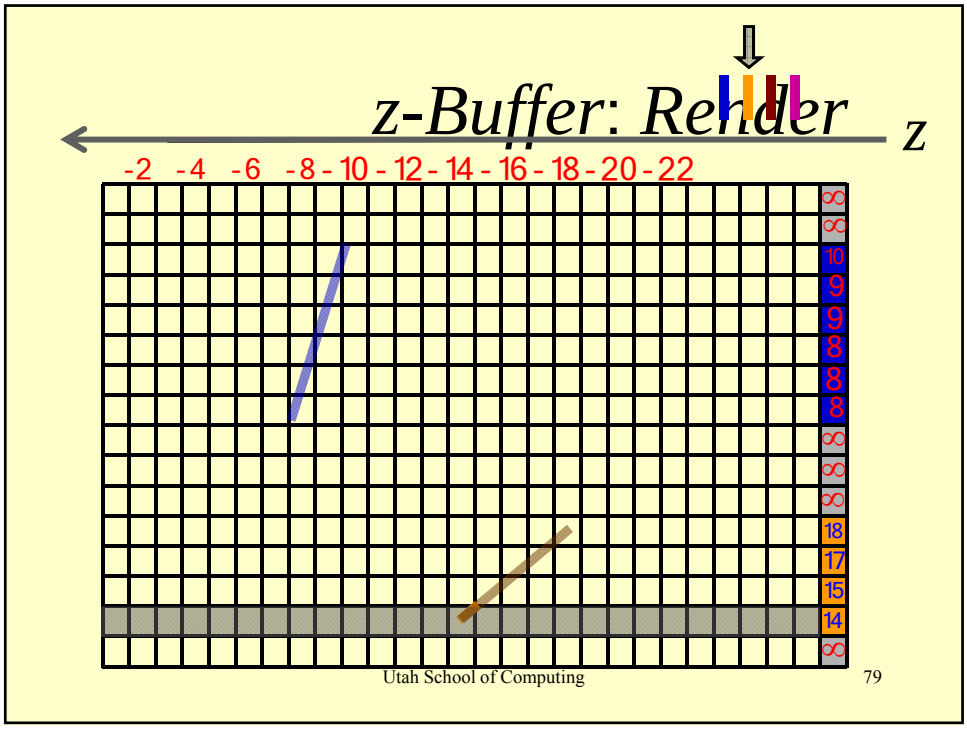

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
74

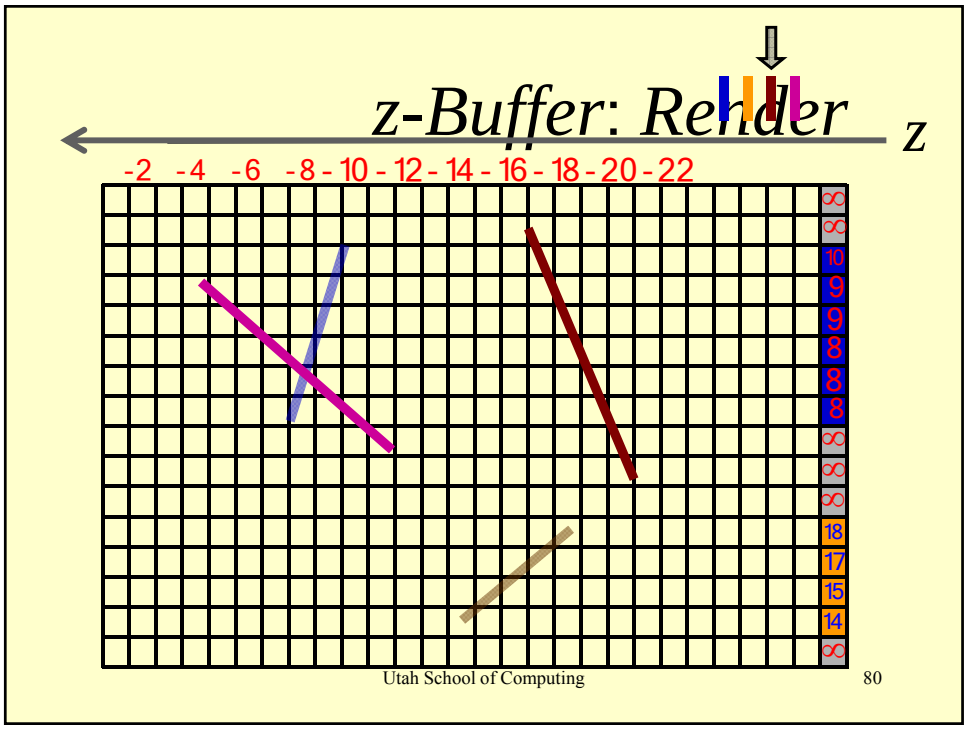

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
75

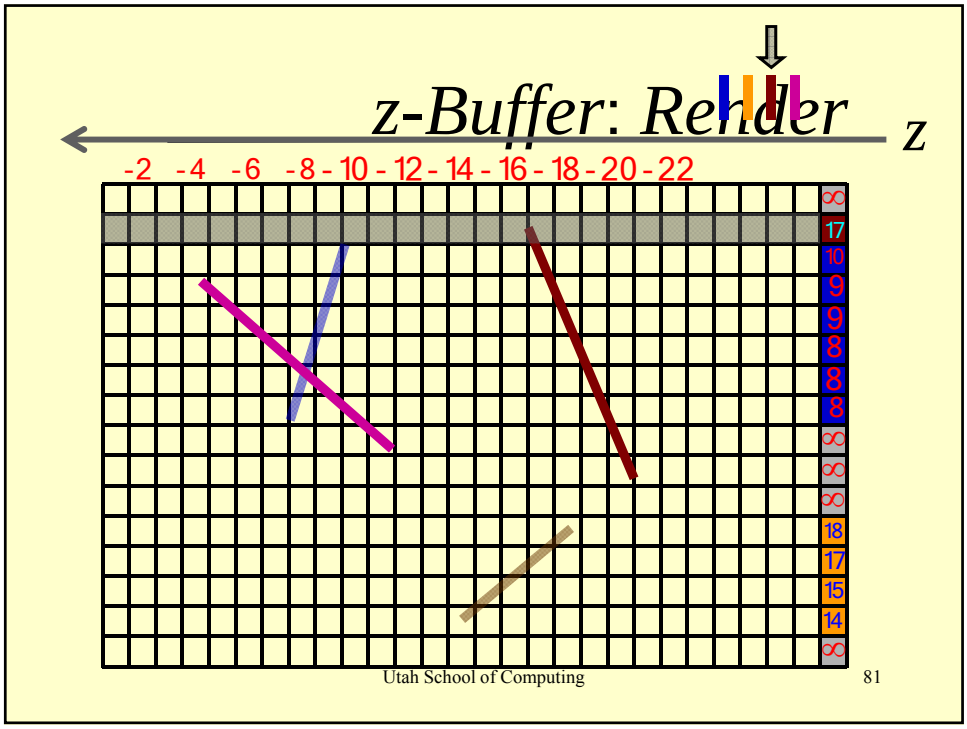

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 ∞ ∞ ∞
Utah School of Computing
76

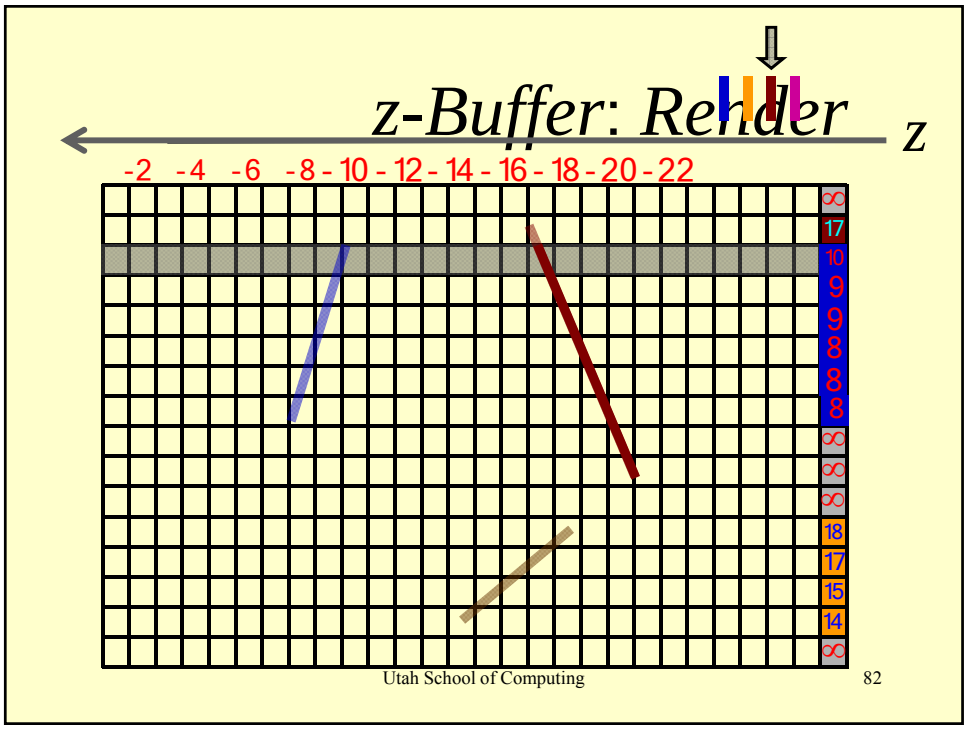

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 ∞ ∞ ∞
Utah School of Computing
77

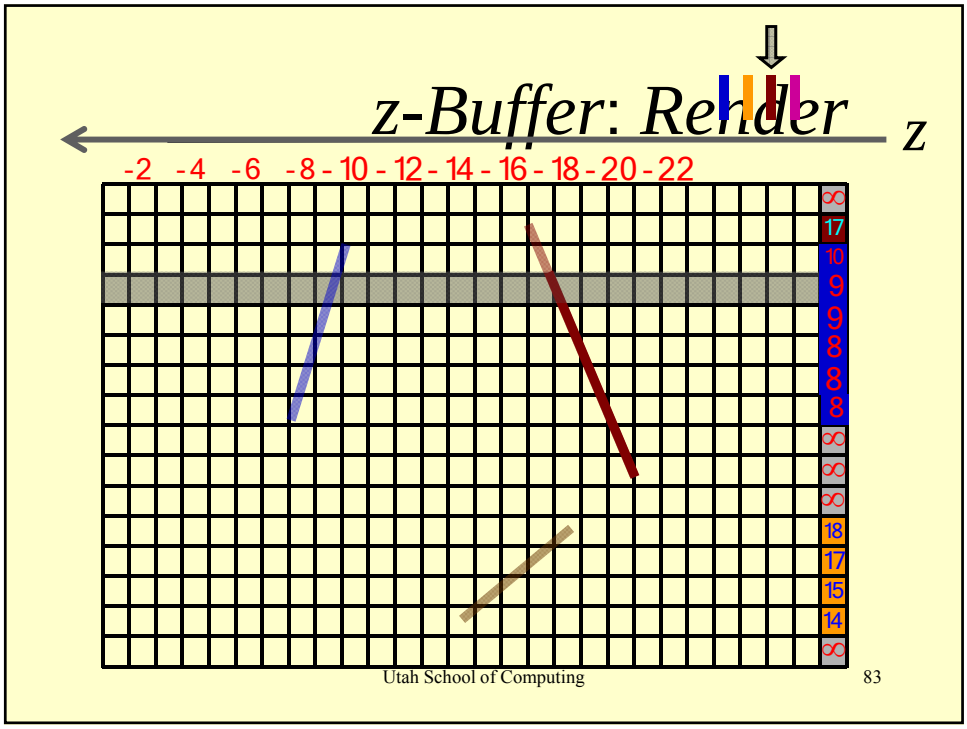

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 ∞ ∞
Utah School of Computing
78

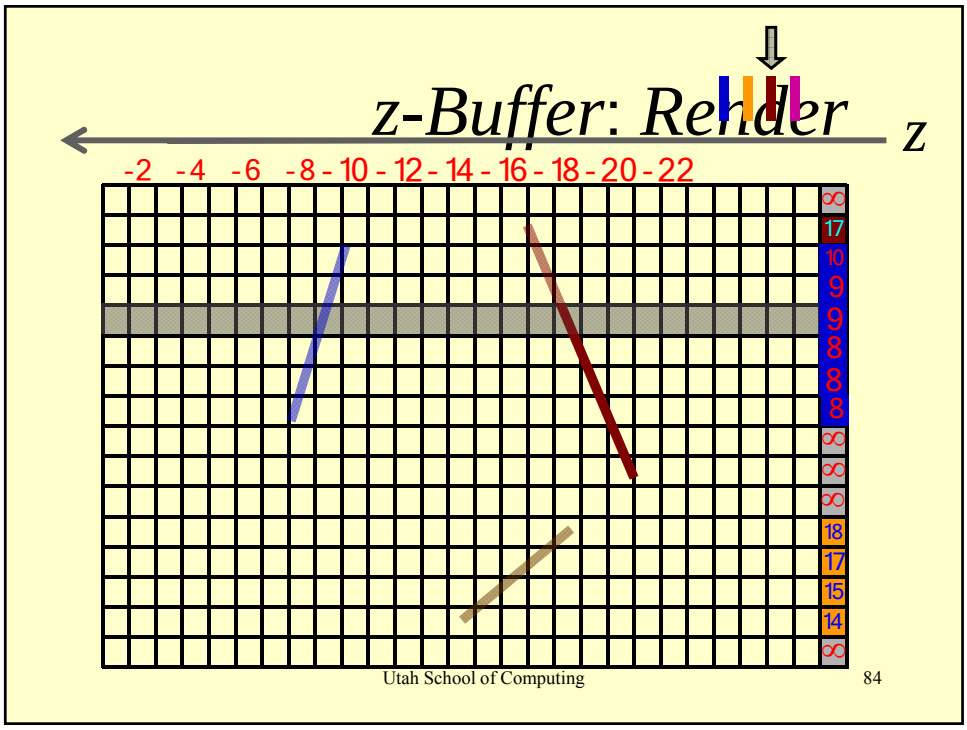

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
79

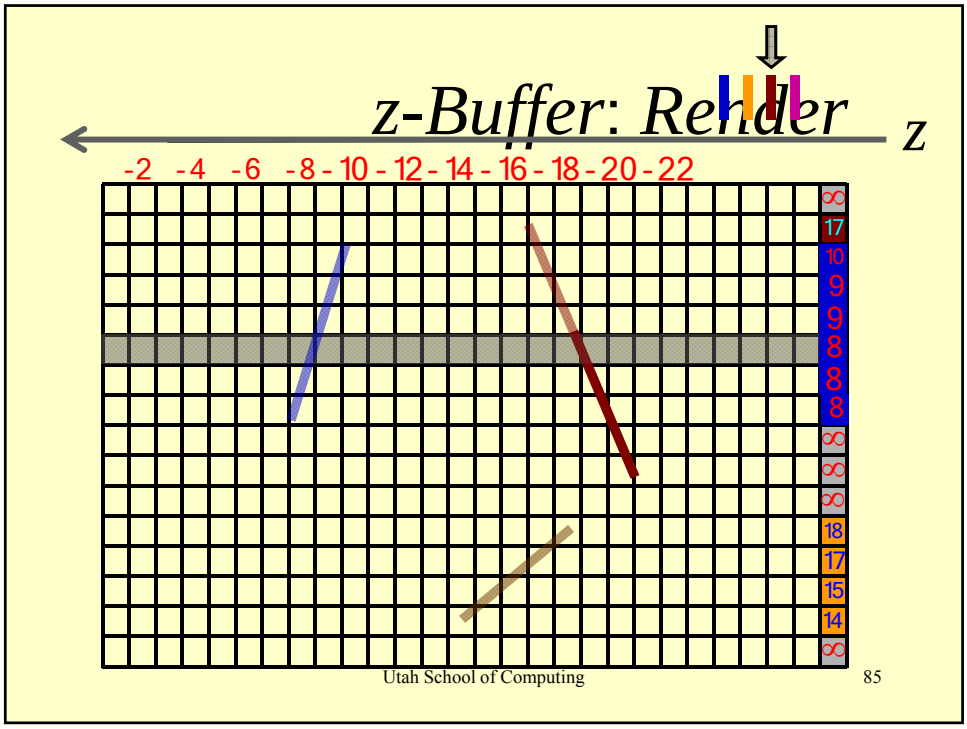

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
80

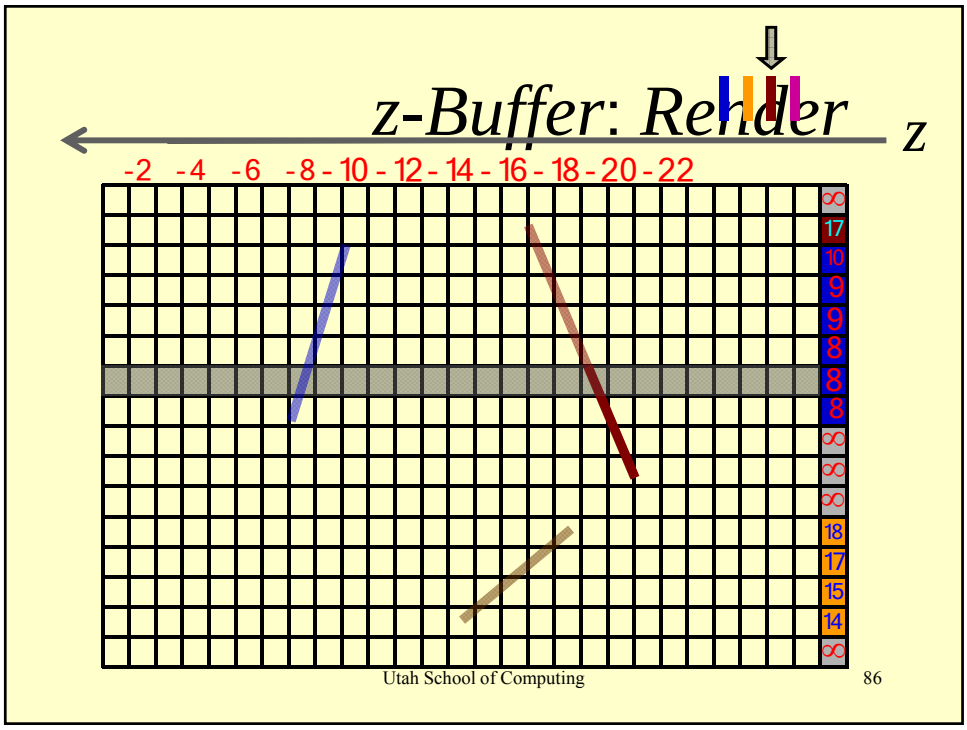

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
81

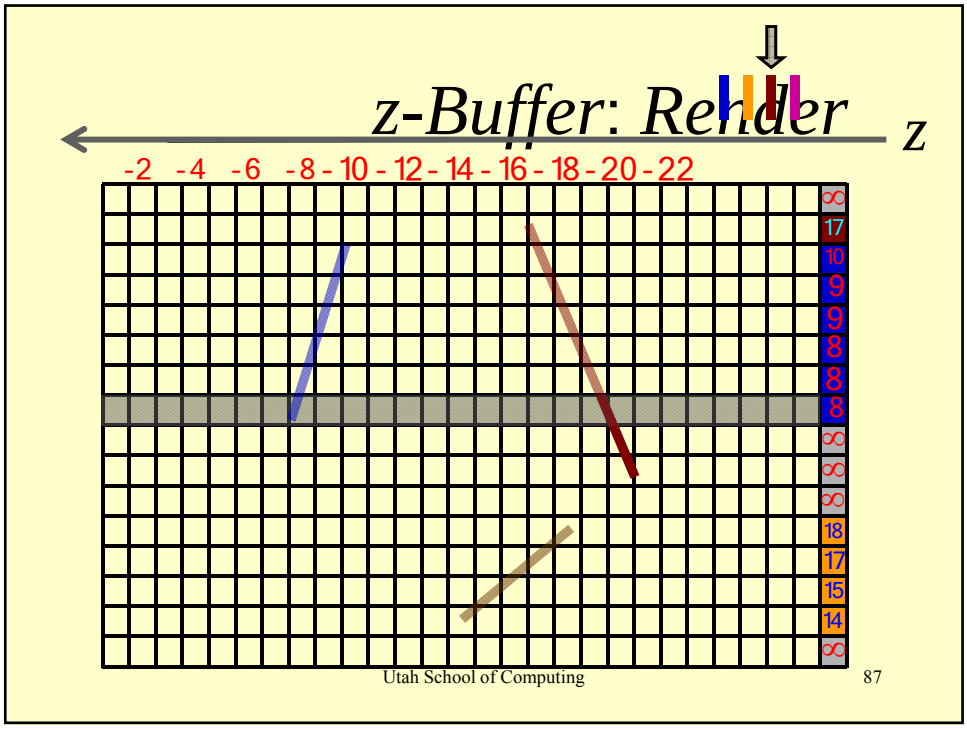

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
82

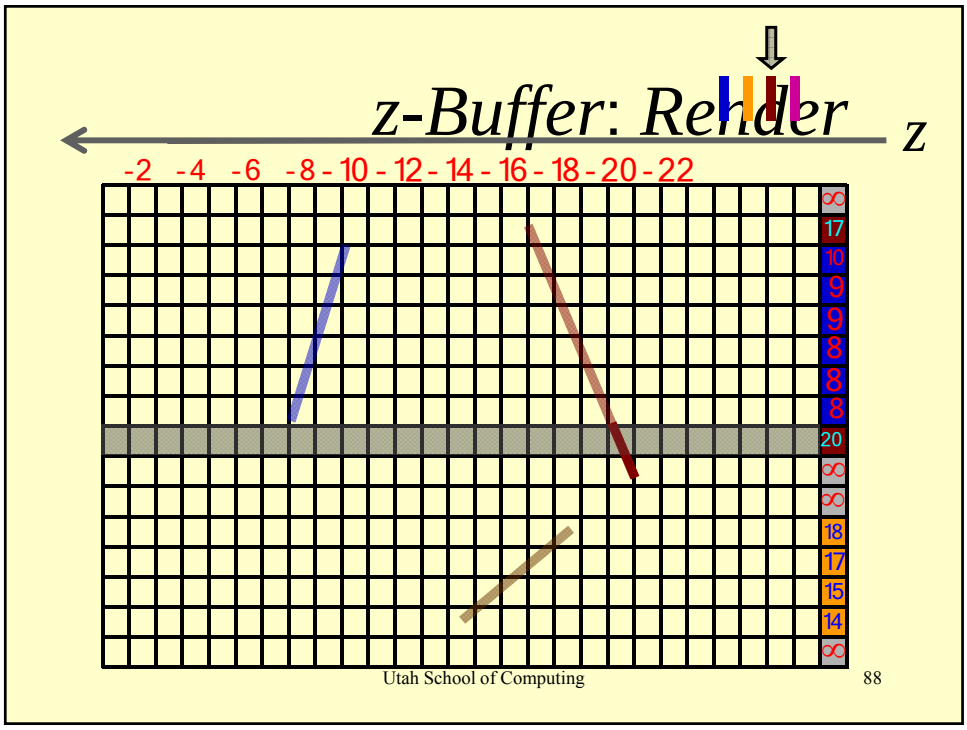

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
83

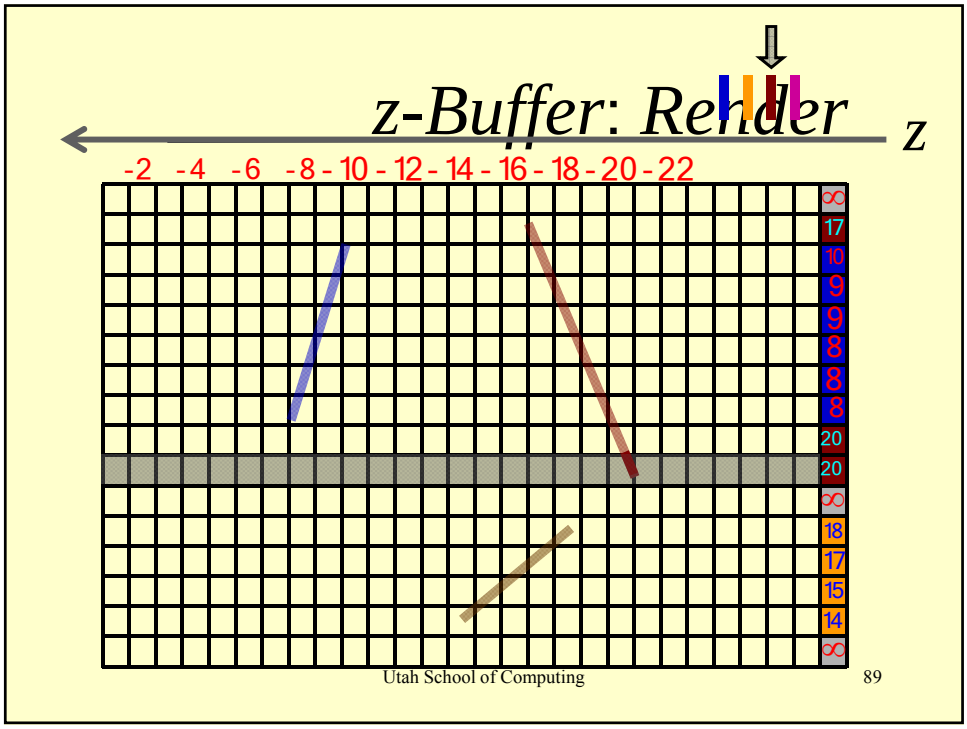

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
84

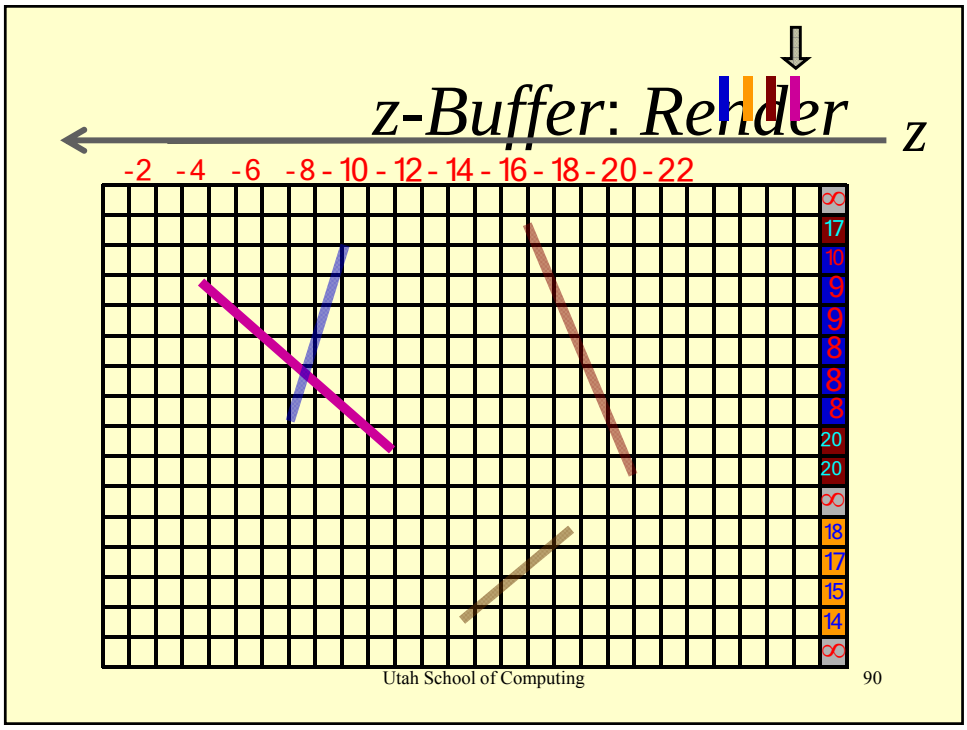

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
85

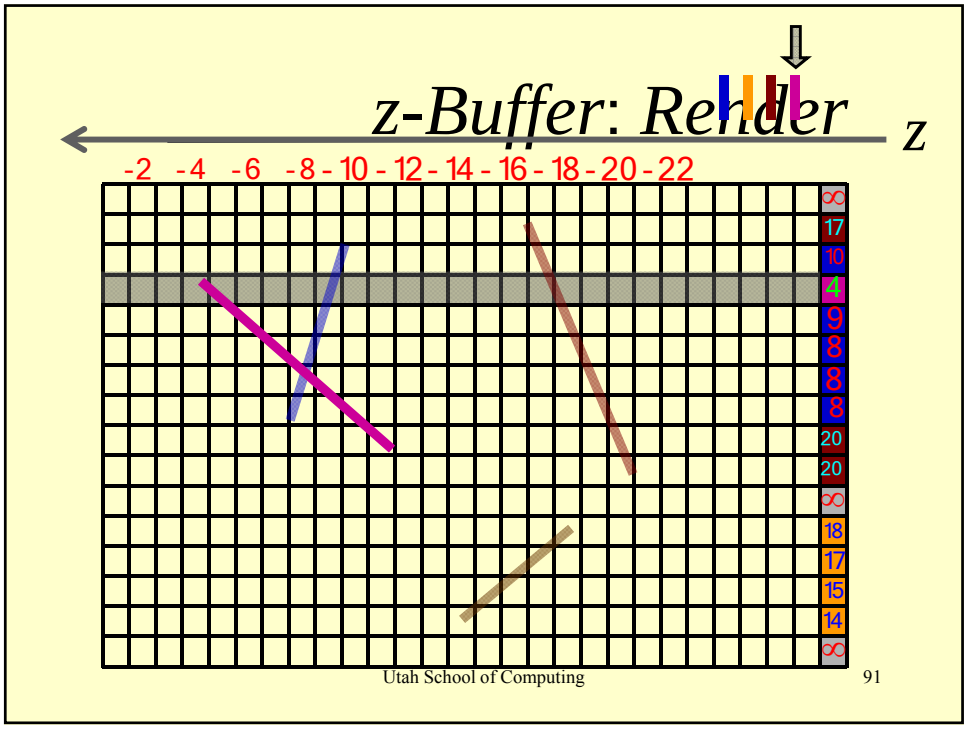

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
86

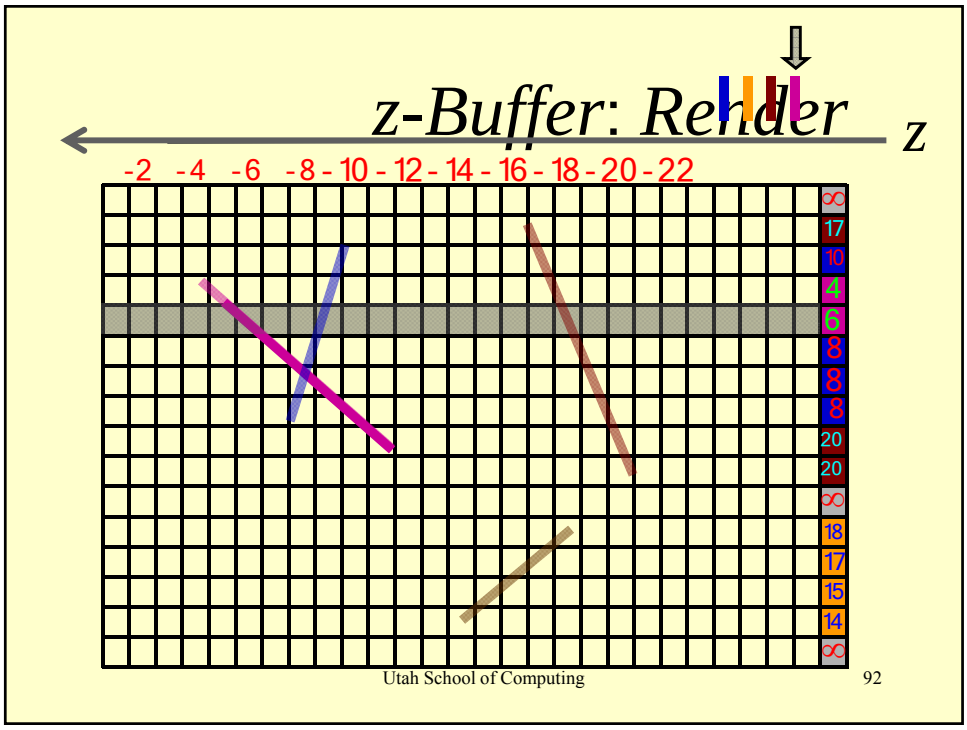

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
87

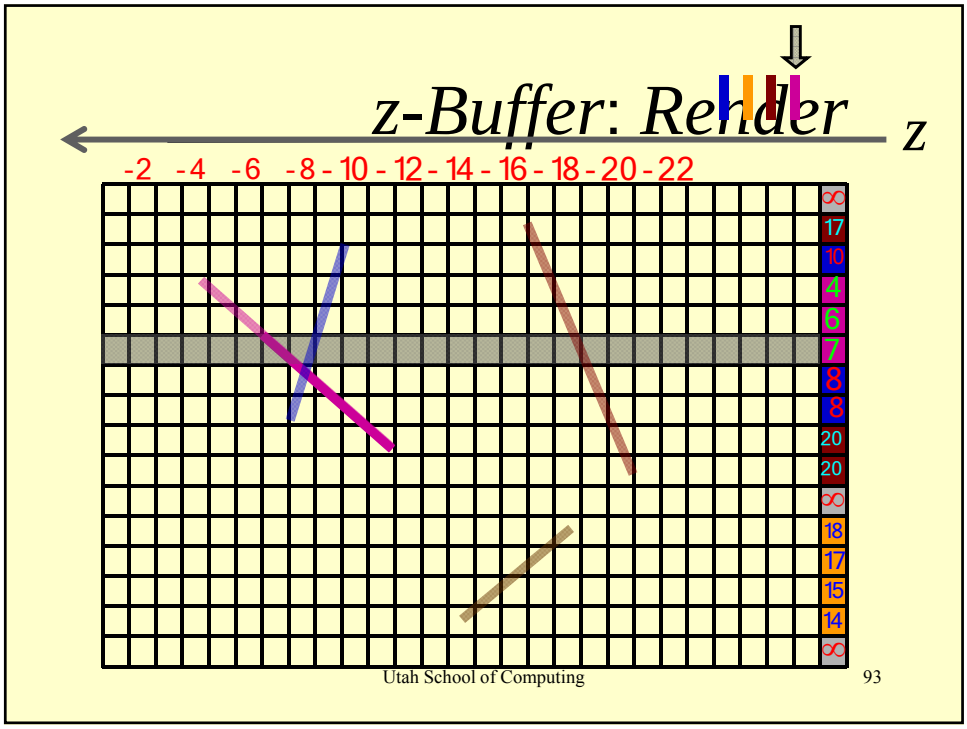

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 20 ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
88

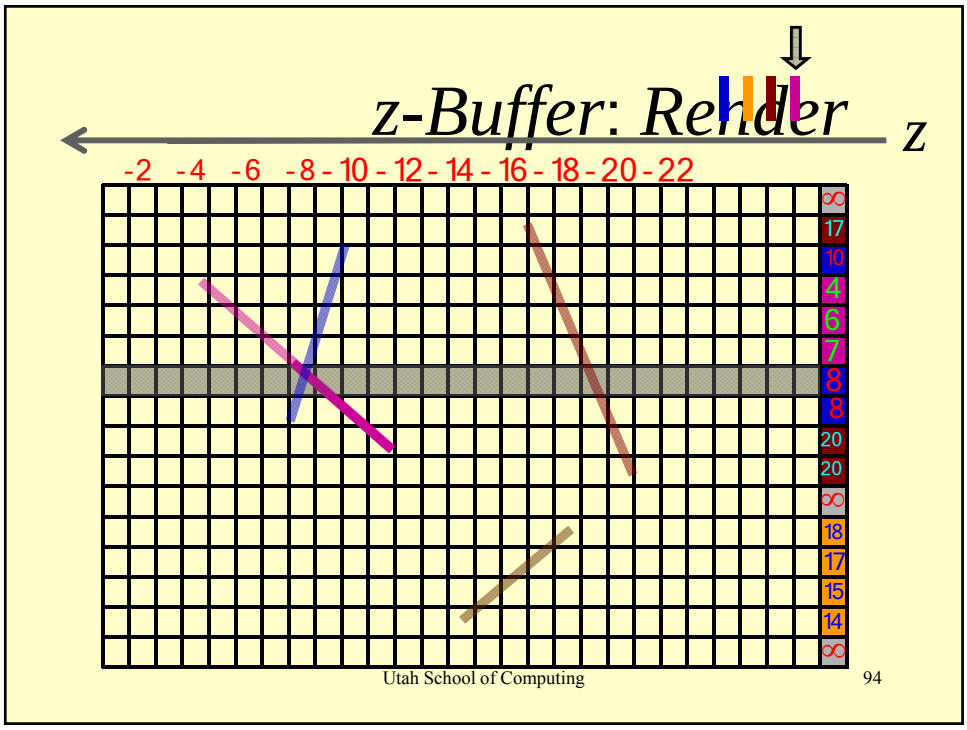

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
89

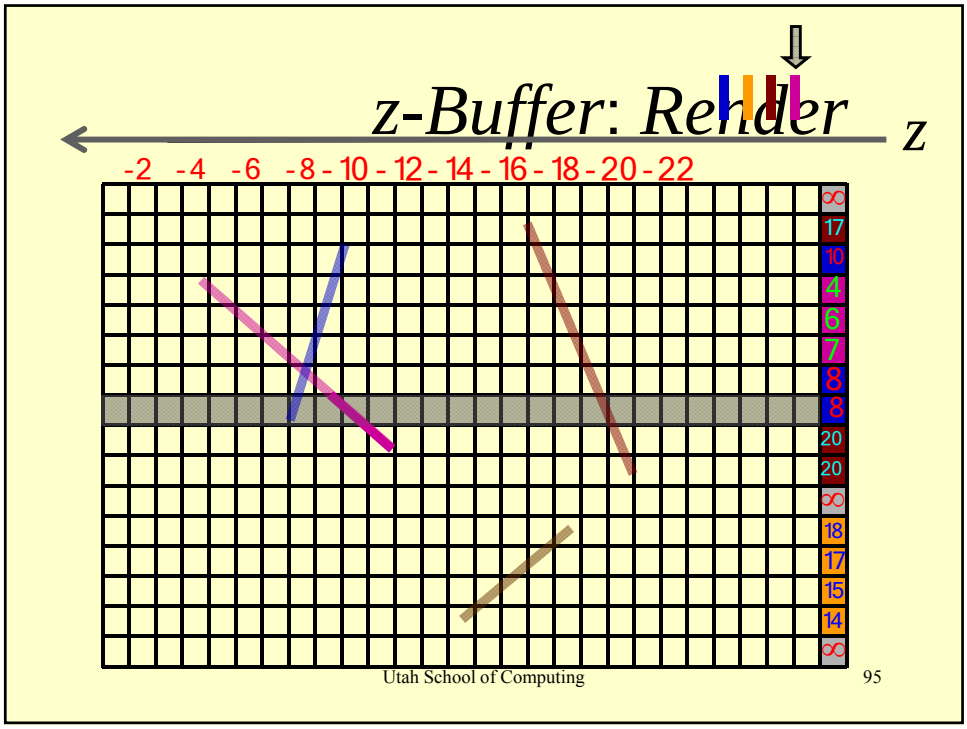

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 9 9 8 8 8 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
90

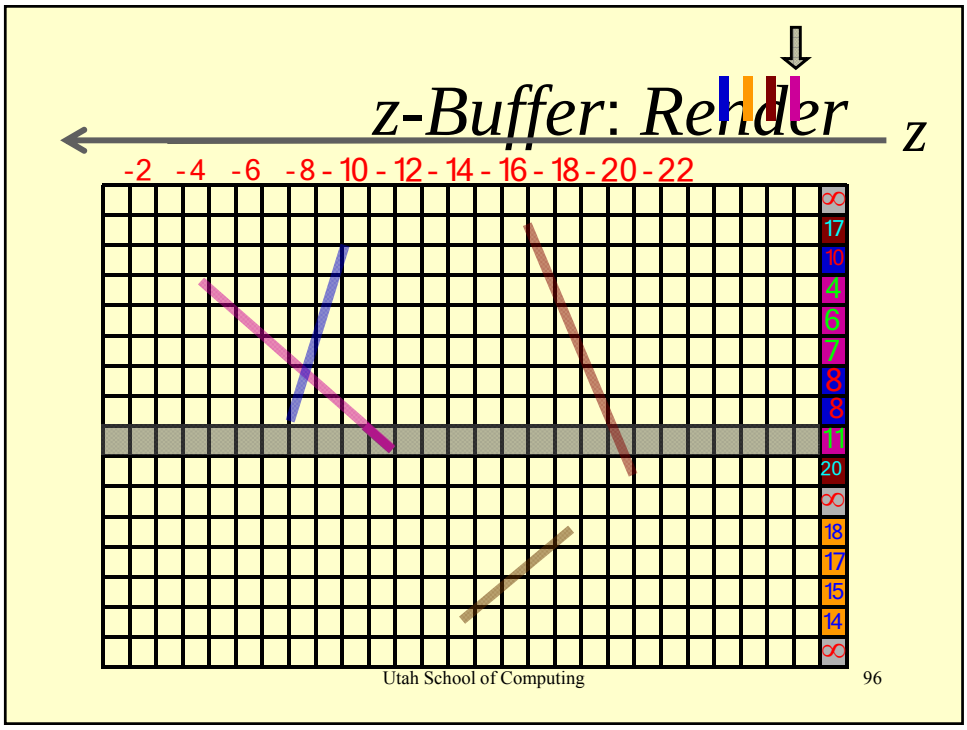

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 4 9 8 8 8 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
91

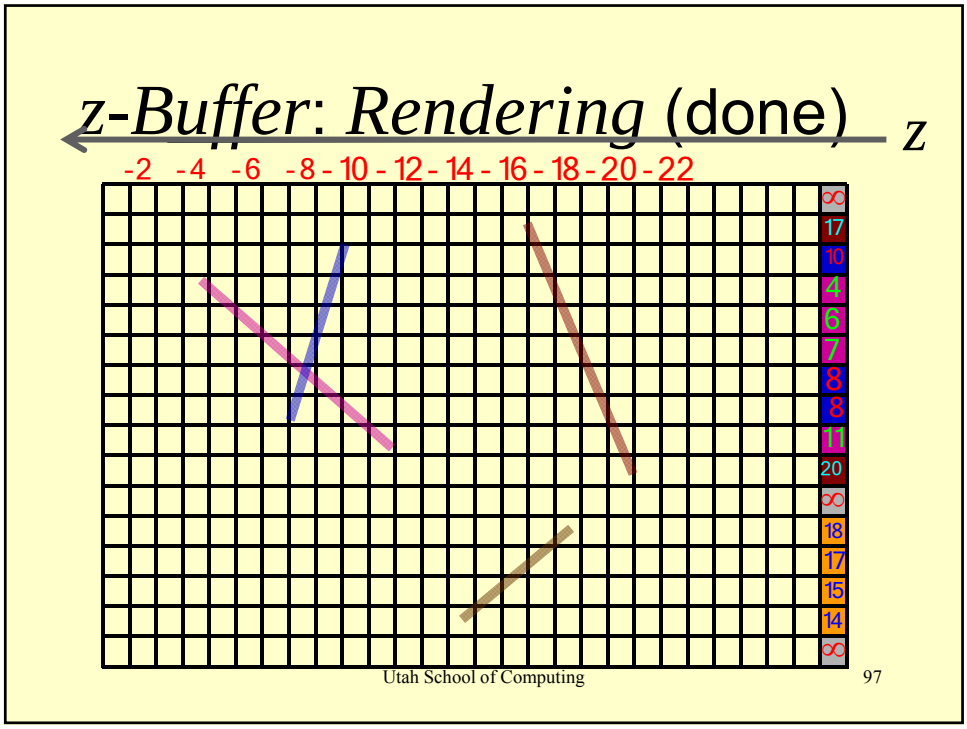

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 4 6 8 8 8 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
92

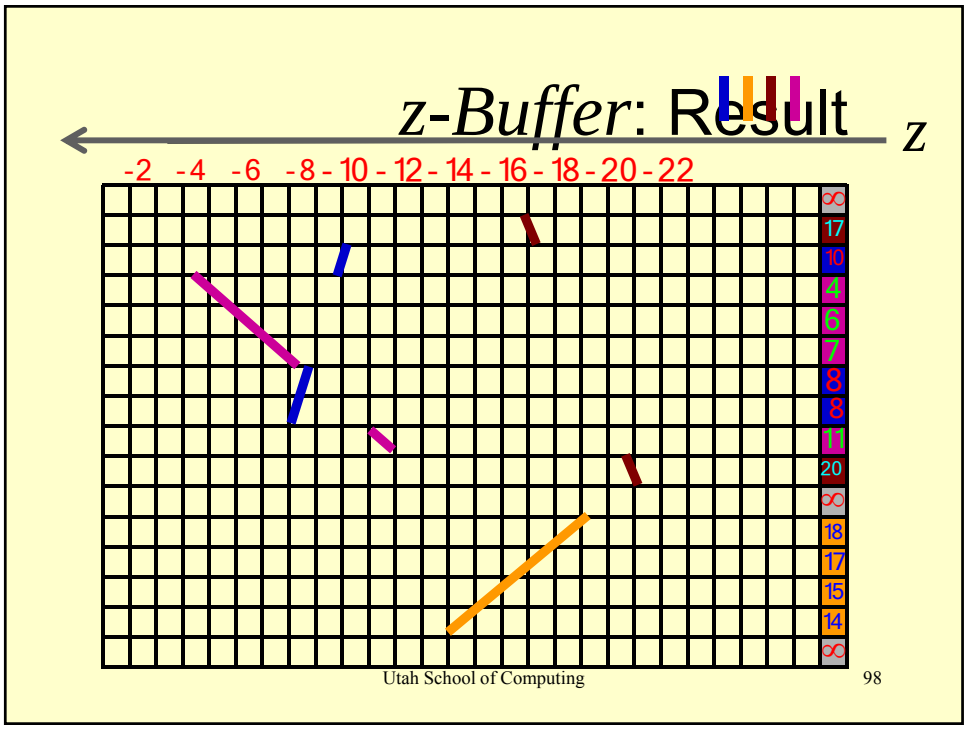

z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 4 6 7 8 8 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
93


z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 4 6 7 8 8 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
94


z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 4 6 7 8 8 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
95


z-Buffer: Render
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 4 6 7 8 8 11 20 ∞ 18 17 15 14 ∞
Utah School of Computing
96


z-Buffer: Rendering (done)
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 4 6 7 8 8 11 20 ∞ 18 17 15 14 ∞
Utah School of Computing
97


z-Buffer: Result
z
-2 -4 -6 -8 -10 -12 -14 -16 -18 -20 -22
∞ 17 10 4 6 7 8 8 11 20 ∞ 18 17 15 14 ∞
Utah School of Computing
98

z-Buffer: Pros

- Simple algorithm
- Easy to implement in hardware
- Complexity is order N, for polygons
- No polygon processing order required
- Easily handles polygon interpenetration

z-Buffer: Cons

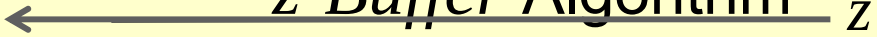
- Memory intensive
- Hard to do antialiasing
- Hard to simulate translucent polygons
- Precision issues
  - Scintillating = z-fighting
  - worse with perspective projection, why?

z-Buffer Algorithm

- Initialize buffer
  - Set background *intensity*, *color* <*r*,*g*,*b*>
  - Set *depth* to *max* (*min*) values

Utah School of Computing 101

z-Buffer Algorithm



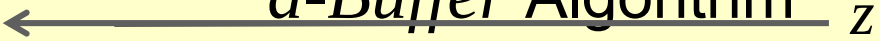
- As a *polygon P* is scan converted
  - Calculate depth $z(x,y)$ at each pixel (x,y) being processed
  - Compare $z(x,y)$ with $z\text{-}Buffer(x,y)$
  - Replace $z\text{-}Buffer(x,y)$ with $z(x,y)$ if closer to eye

Utah School of Computing 102

z-Buffer Algorithm

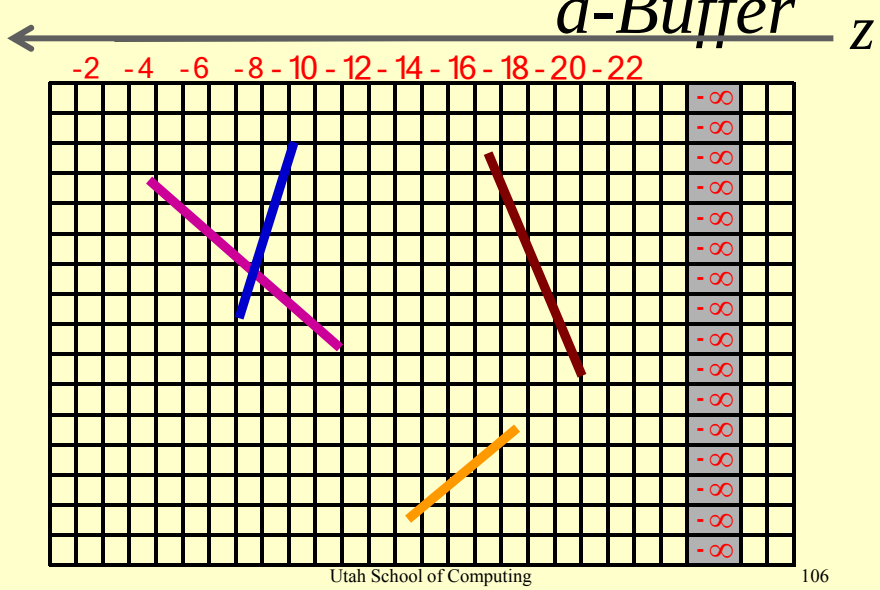
- *Convert* all *polygons*
- Correct image gets generated when *done*
- OpenGL: depth-buffer = z-Buffer

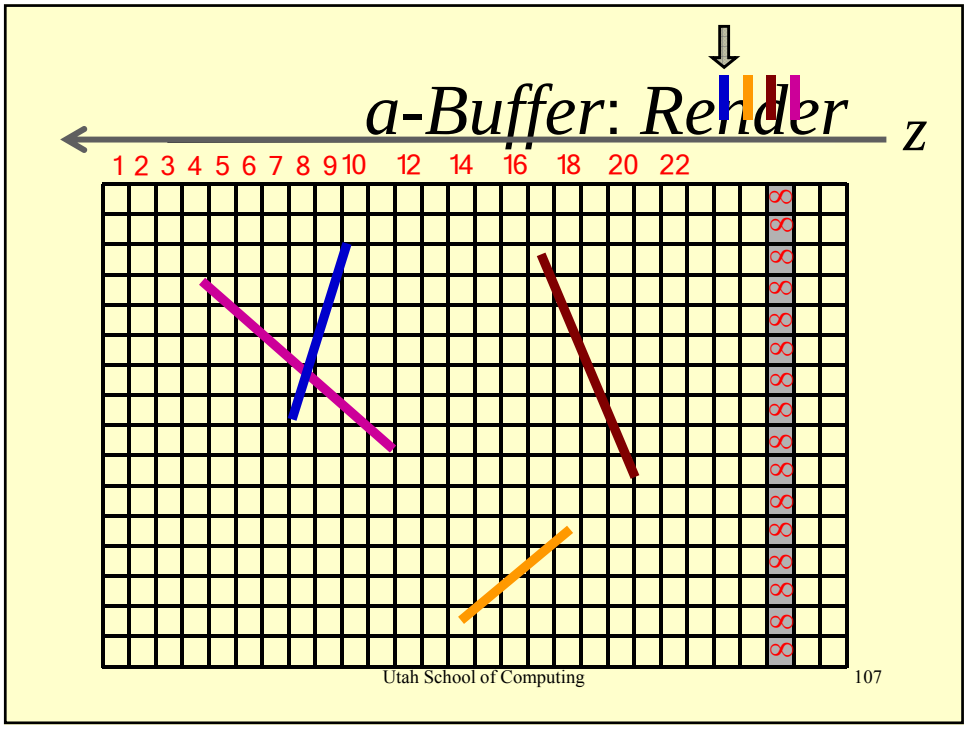
a-Buffer Algorithm

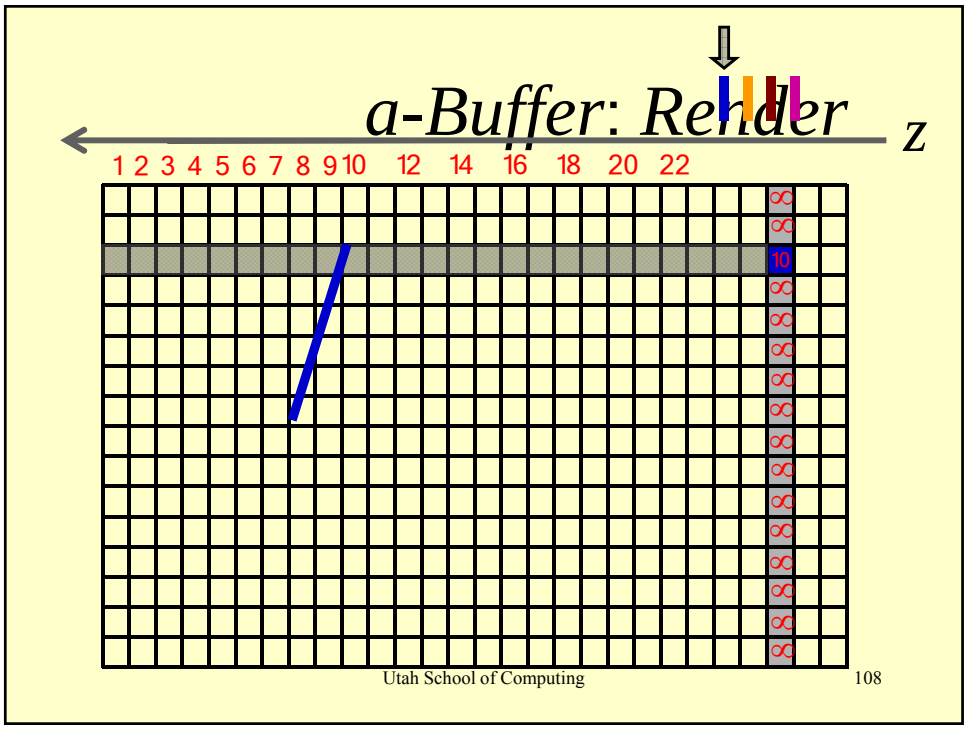


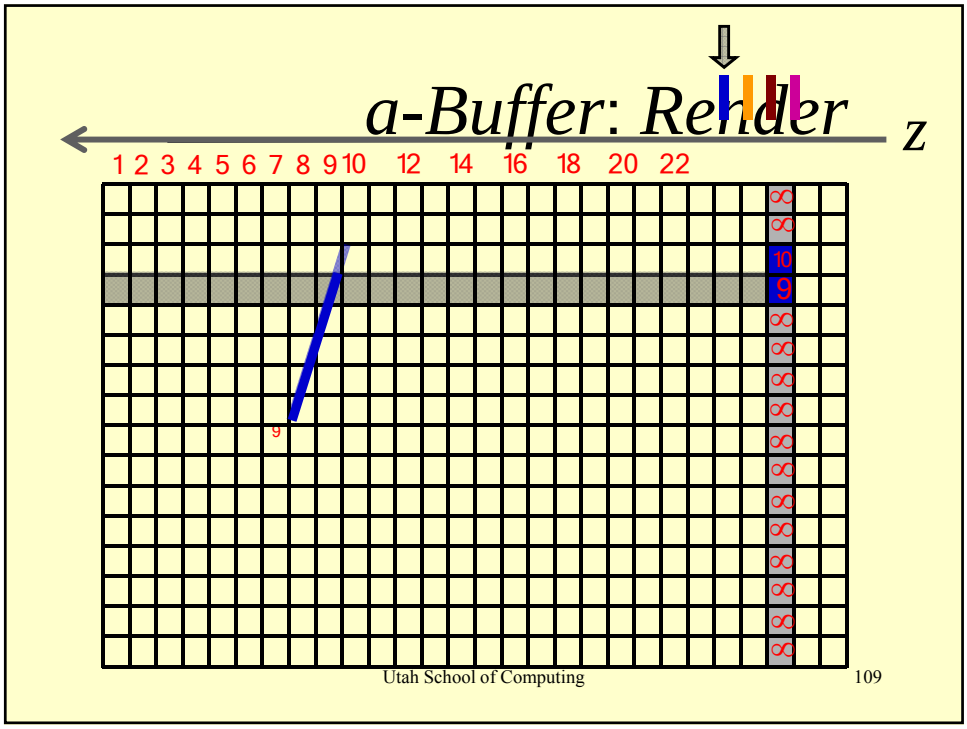
- Generates linked list for each pixel
- Memory of all contributions allows for proper handling of many advanced techniques
- Even *more* memory intensive
- Widely used for high quality rendering
- Widely used for deferred shading

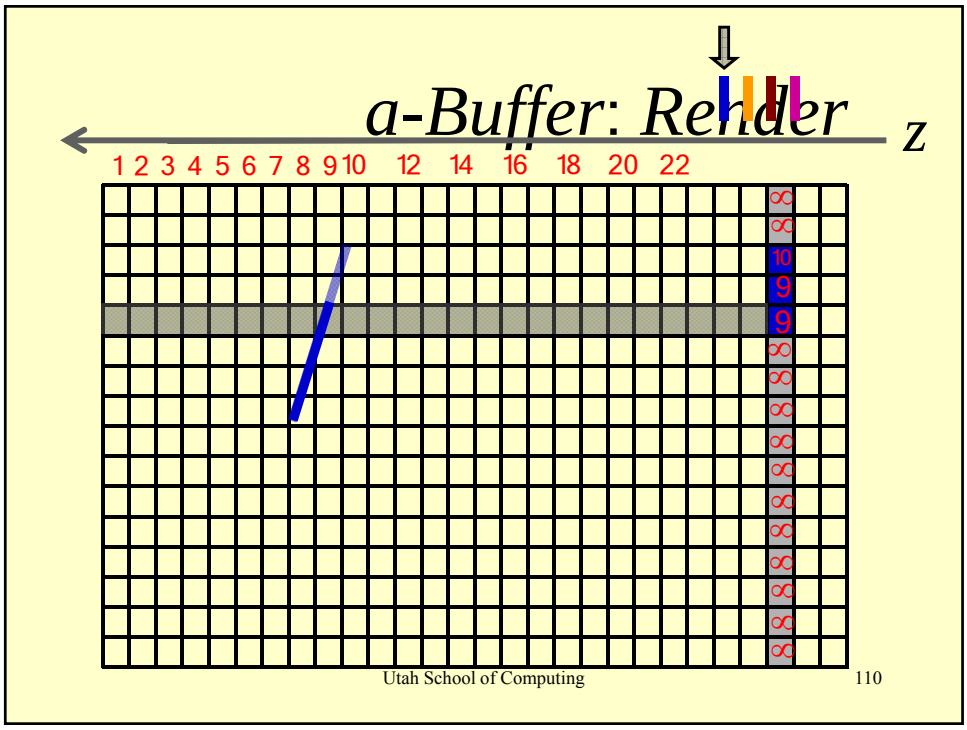
a-Buffer Algorithm: Example

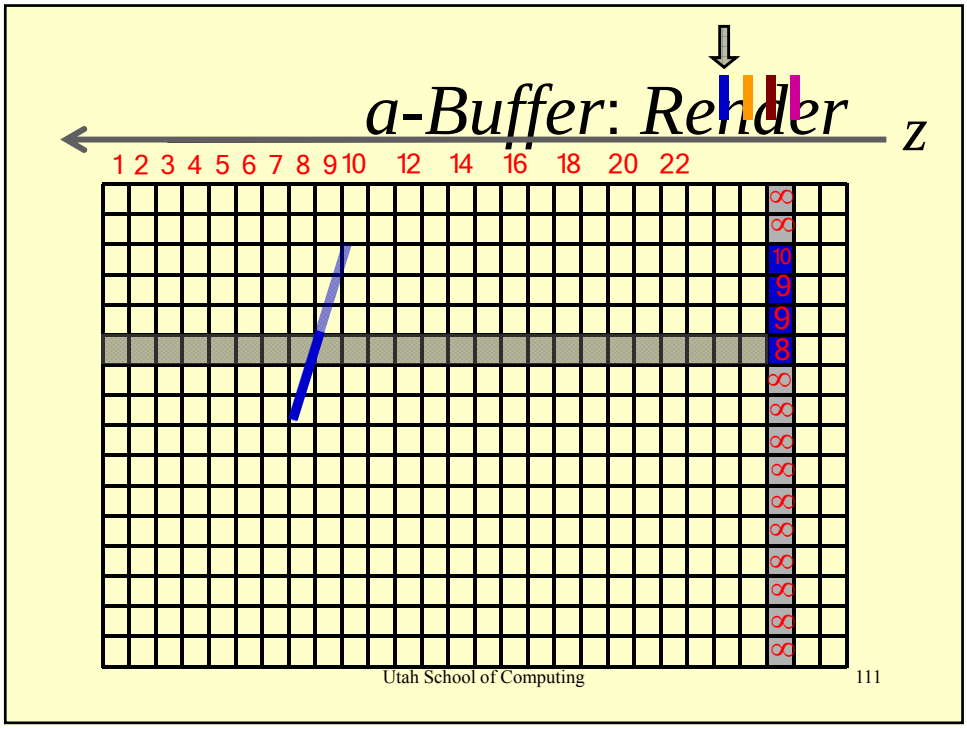


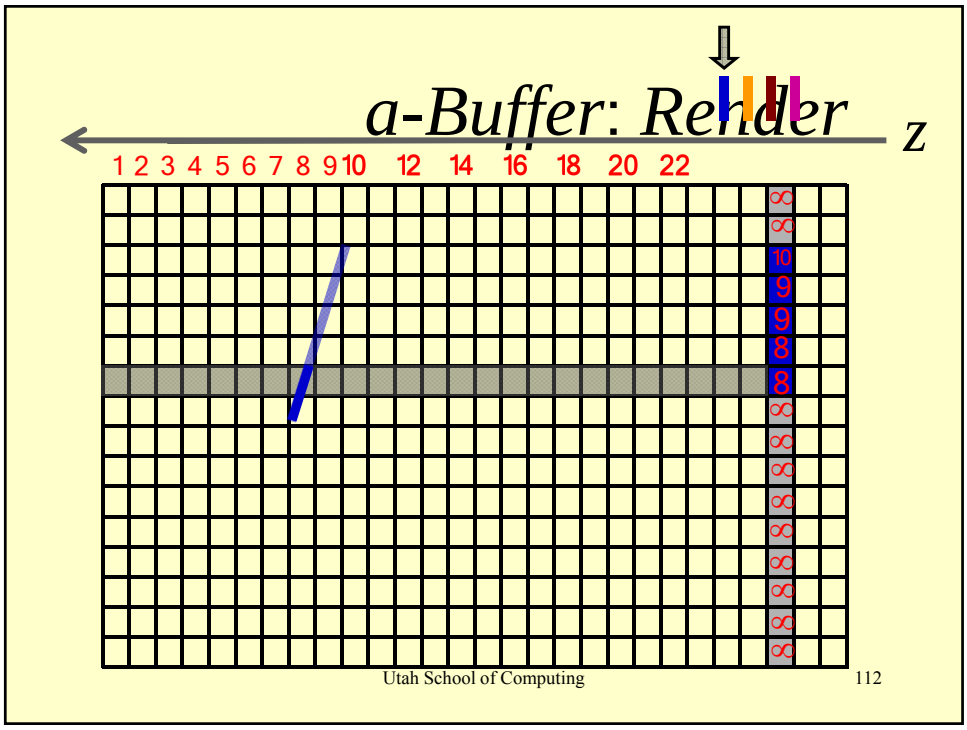

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞
Utah School of Computing
107

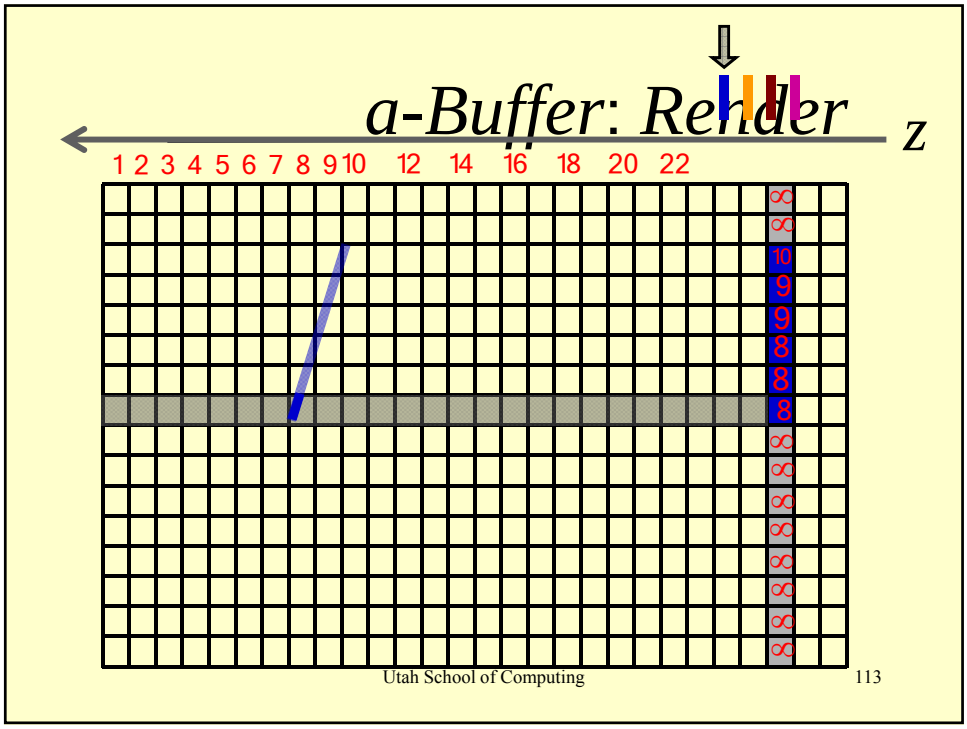

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
10
∞
Utah School of Computing
108

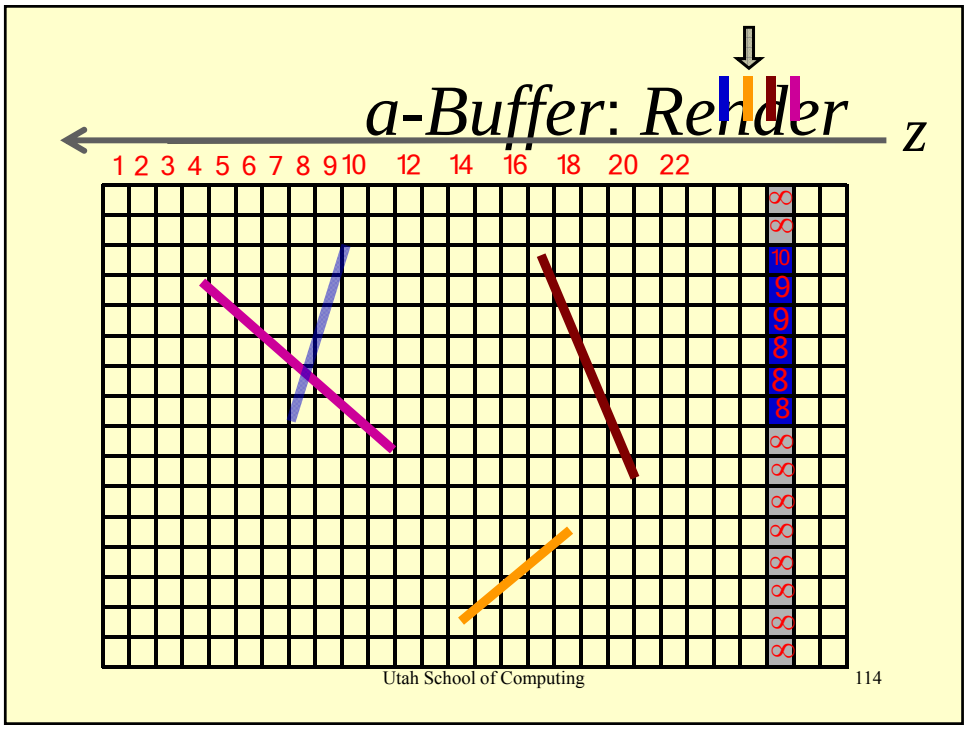

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞
∞
10
9
∞
∞
∞
∞
∞
∞
∞
∞
∞
∞
∞
∞
9
Utah School of Computing
109

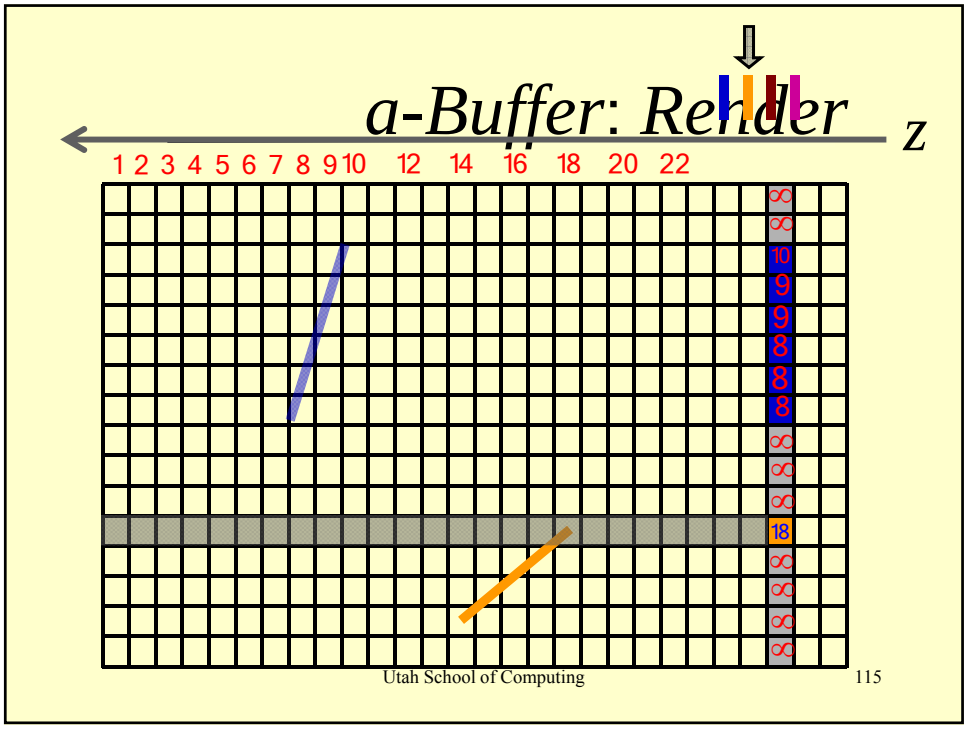

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞
∞
10
9
9
∞
∞
∞
∞
∞
∞
∞
∞
∞
∞
∞
∞
Utah School of Computing
110

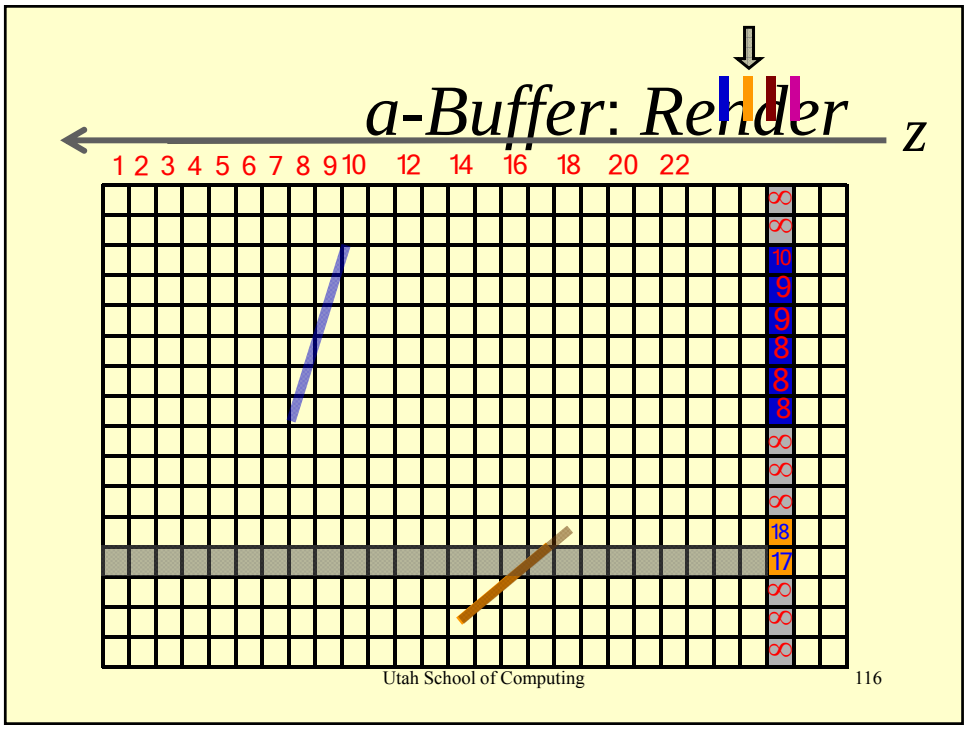

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
111

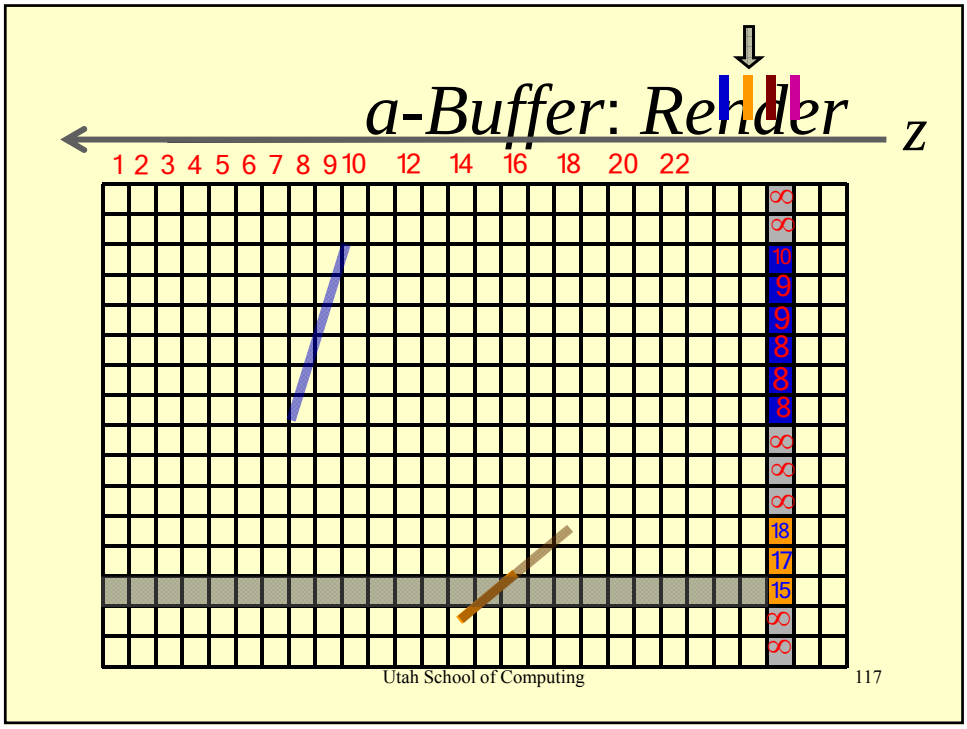

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 8 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
112

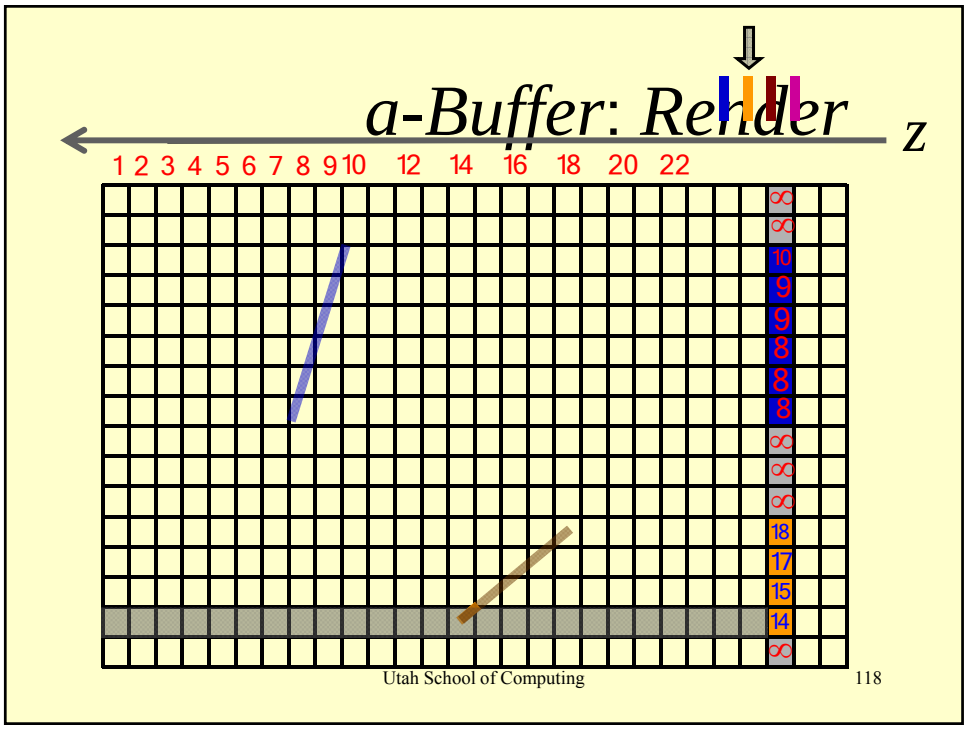

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
113

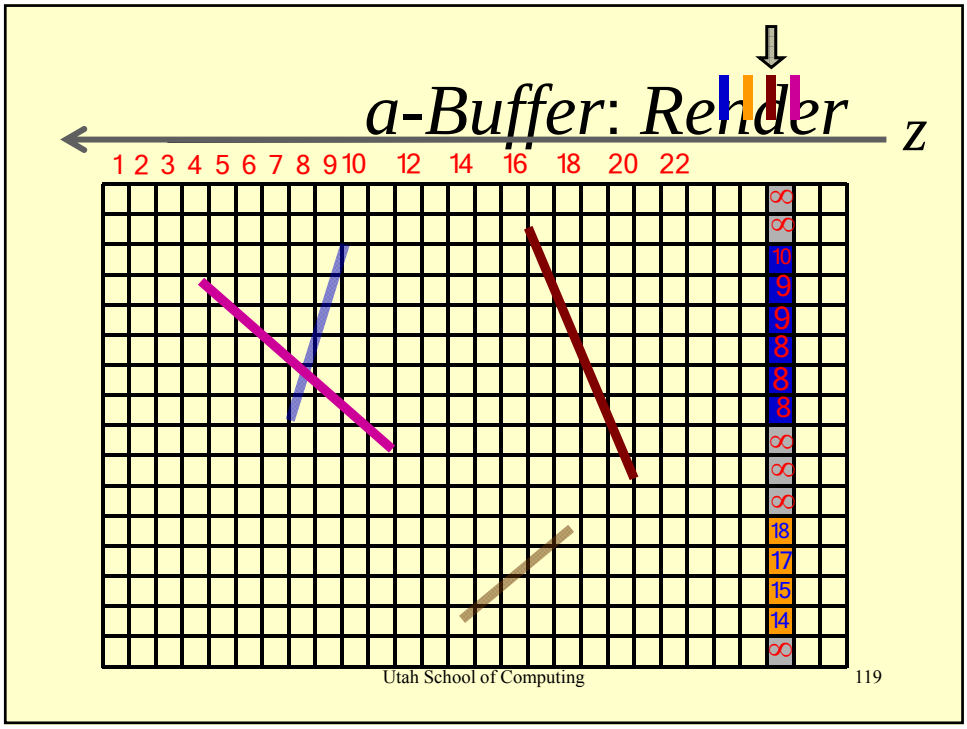

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
Utah School of Computing
114

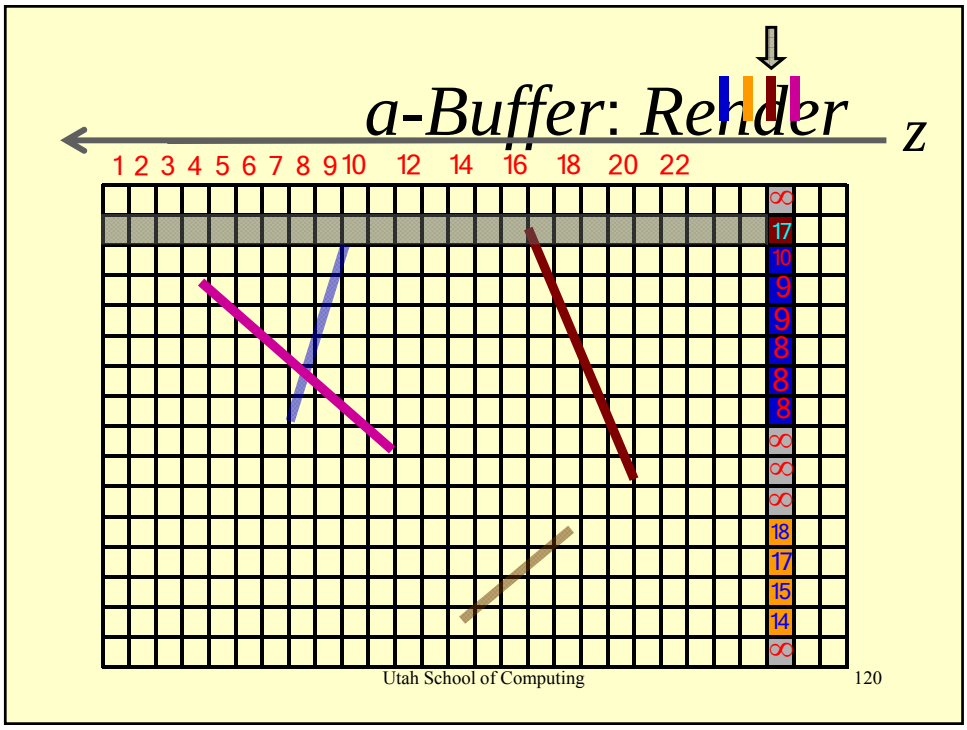

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 ∞ ∞ ∞ ∞
Utah School of Computing
115

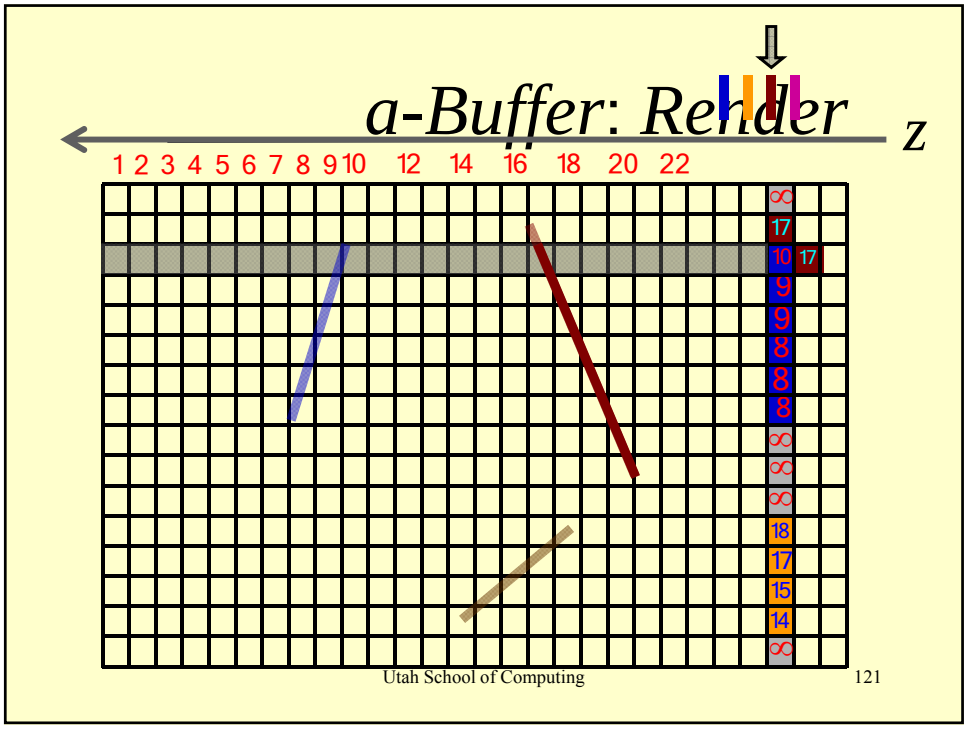

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 ∞ ∞ ∞
Utah School of Computing
116

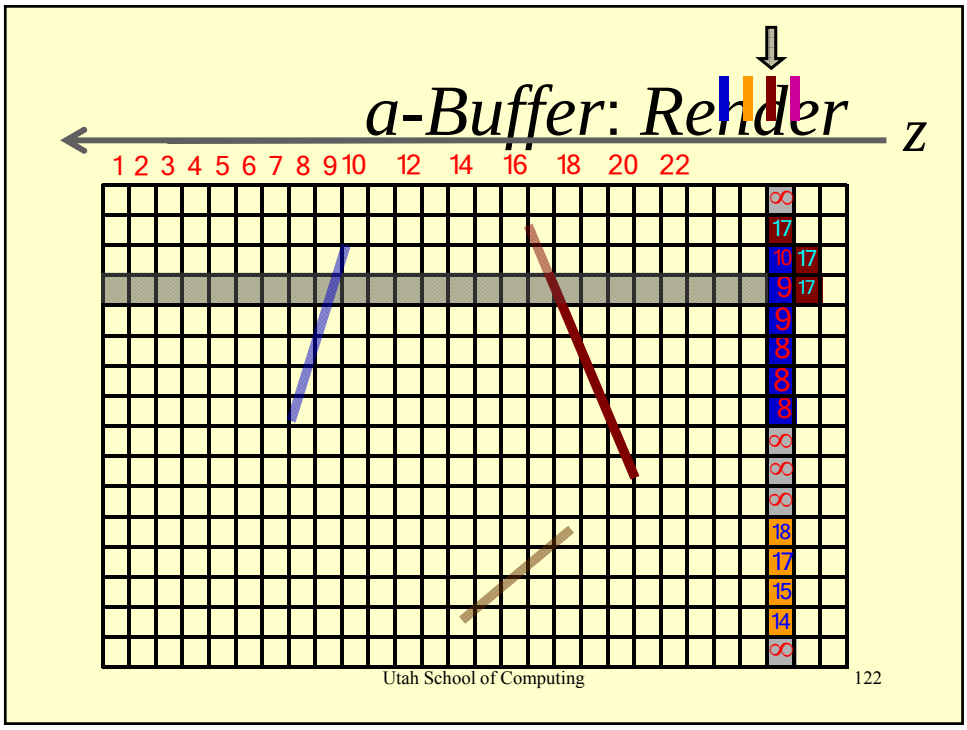

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 ∞ ∞
Utah School of Computing
117

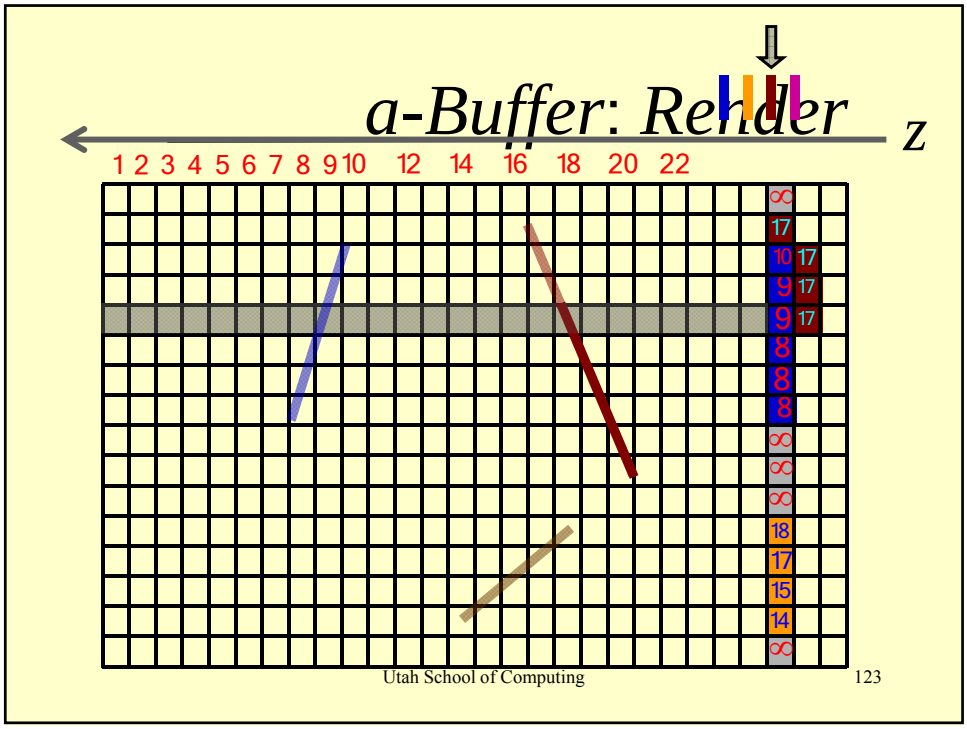

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
118

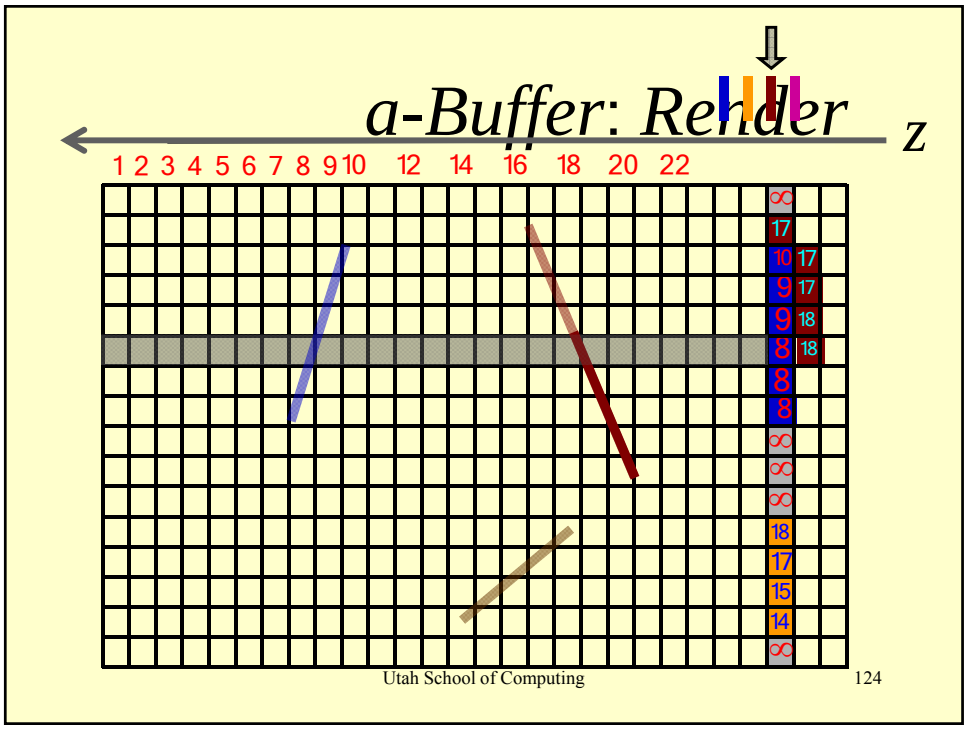

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ ∞ 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
119

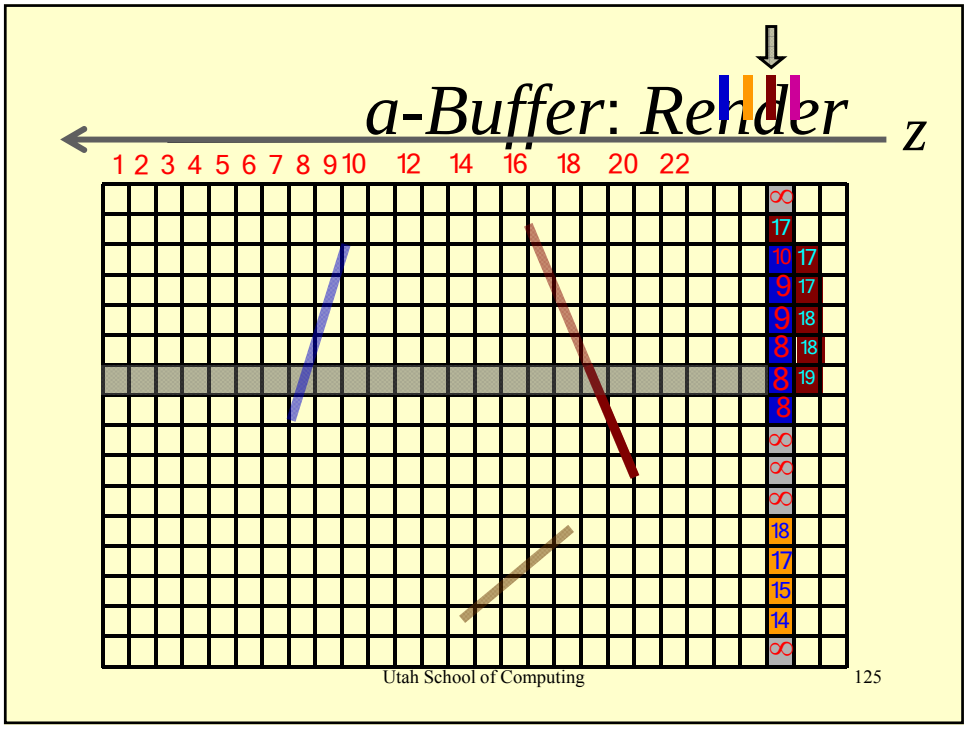

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
120

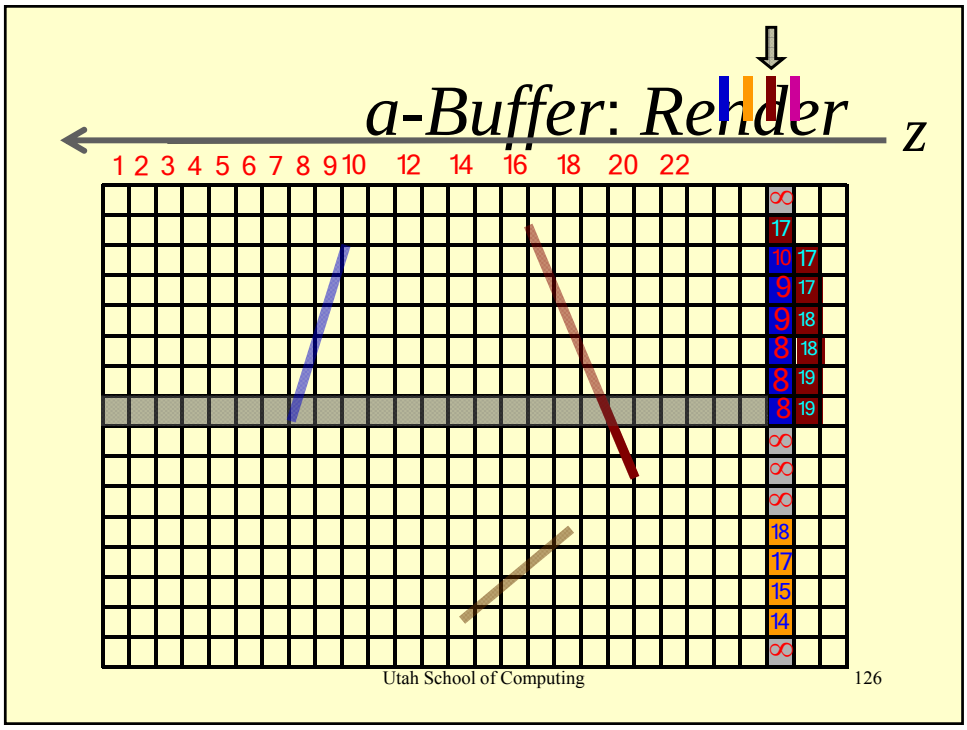

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
121

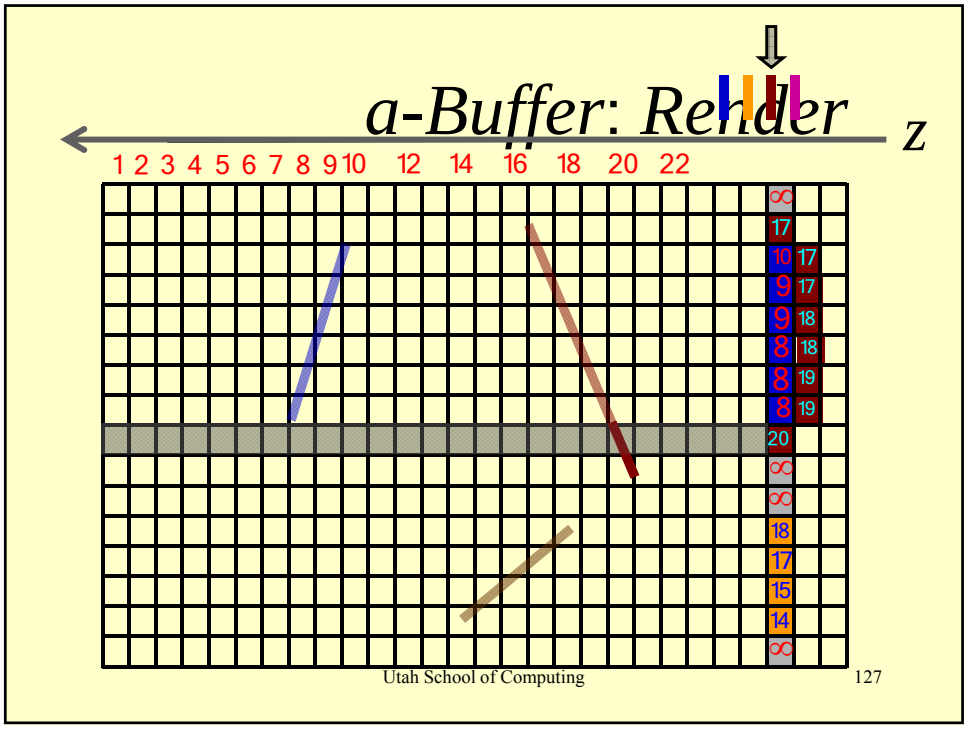

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 17 9 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
122

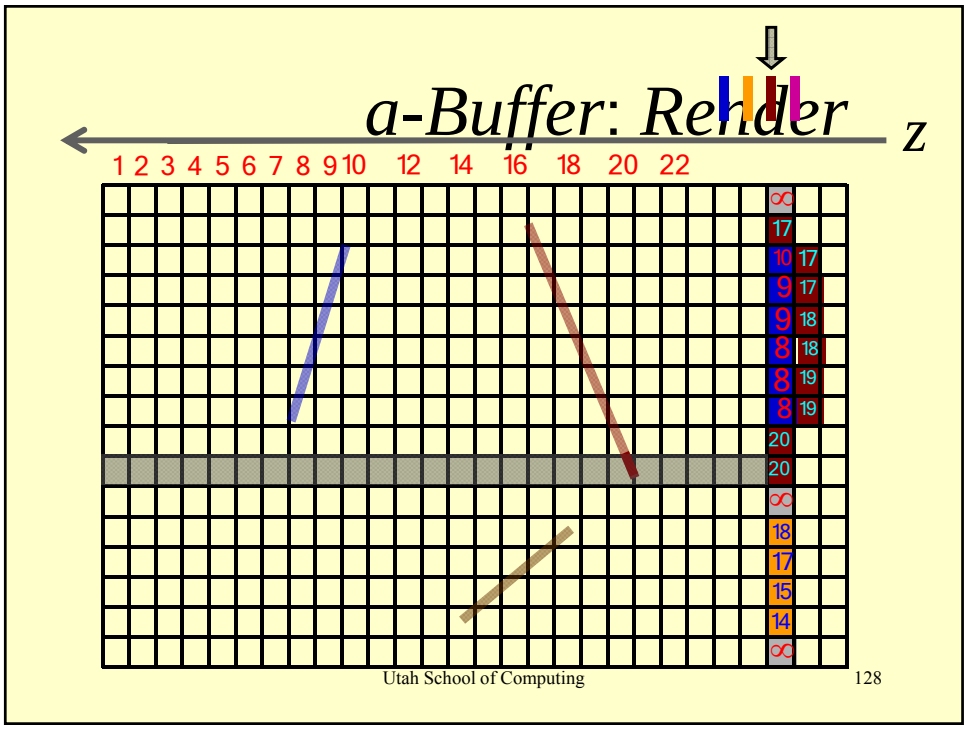

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 17 9 17 8 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
123

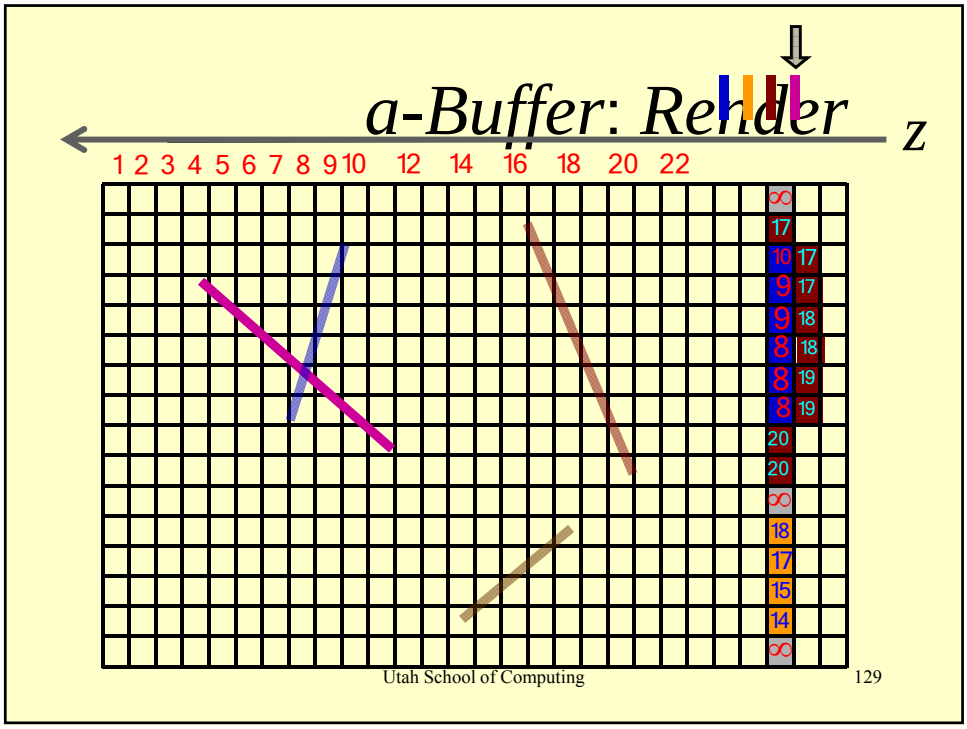

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 17 9 18 8 18 8 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
124

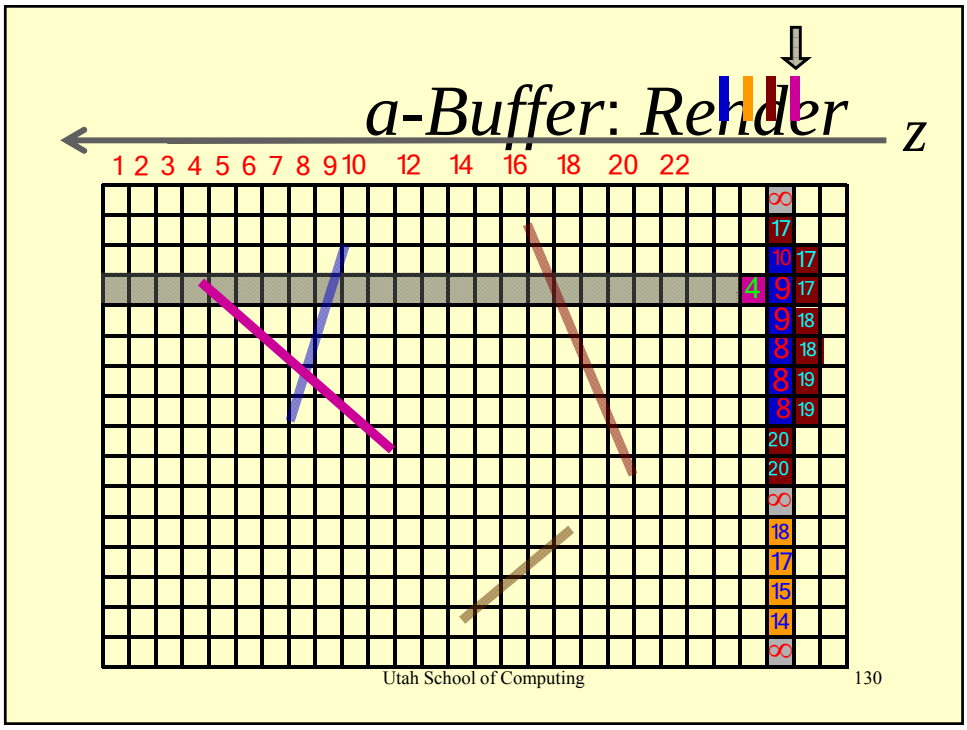

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 17 9 18 8 18 8 19 8 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
125

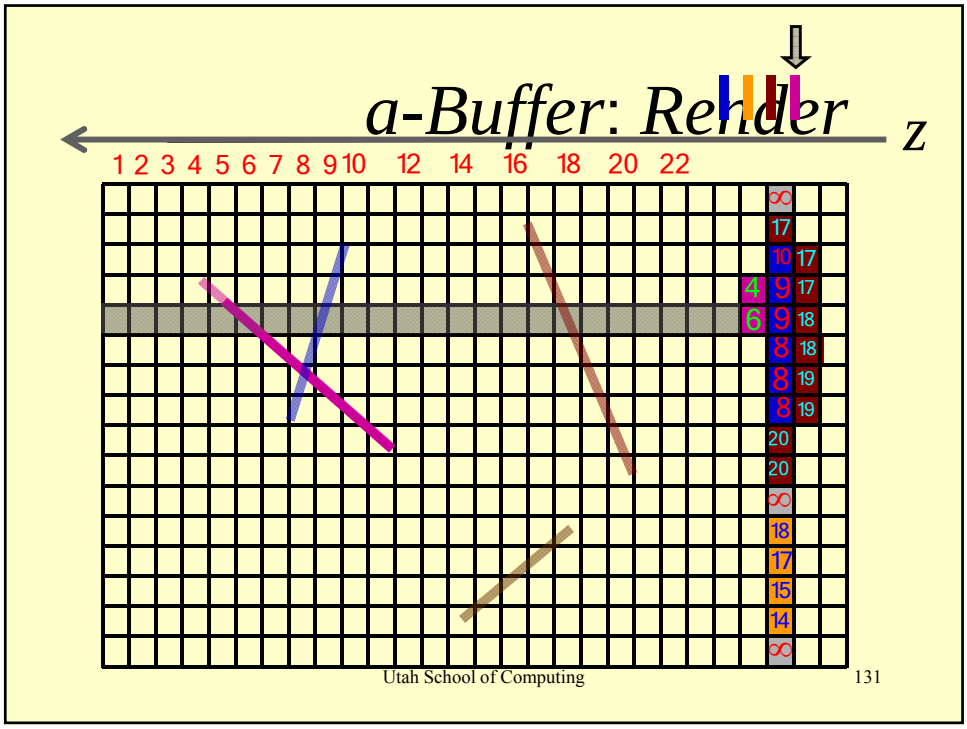

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 17 9 18 8 18 8 19 8 19 ∞ ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
126

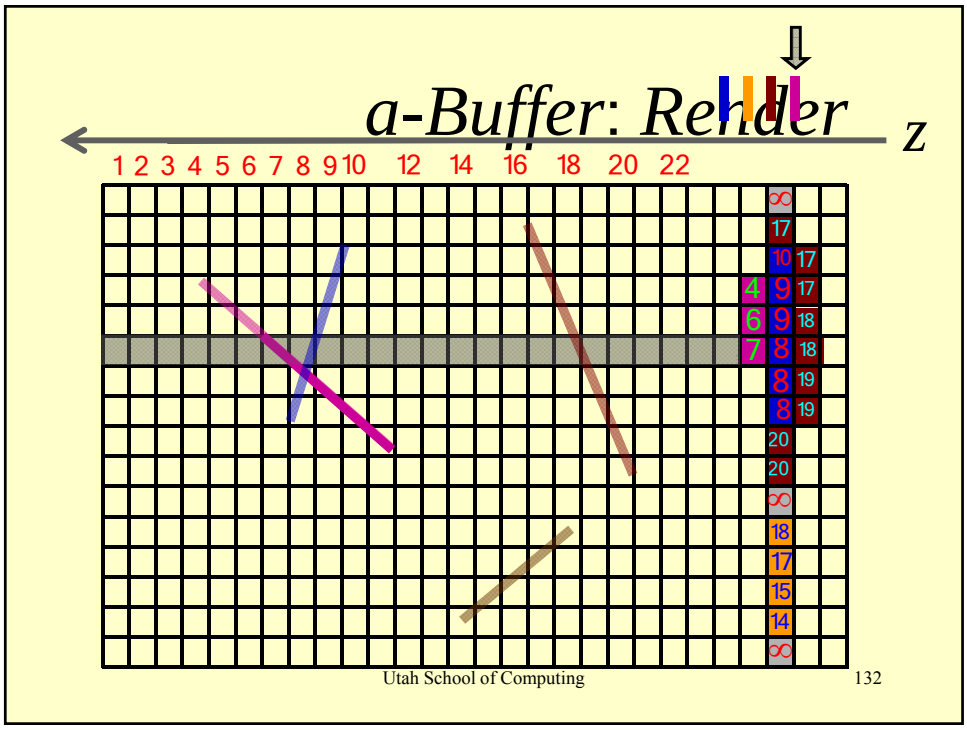

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 17 9 18 8 18 8 19 8 19 20 ∞ ∞ 18 17 15 14 ∞
Utah School of Computing
127

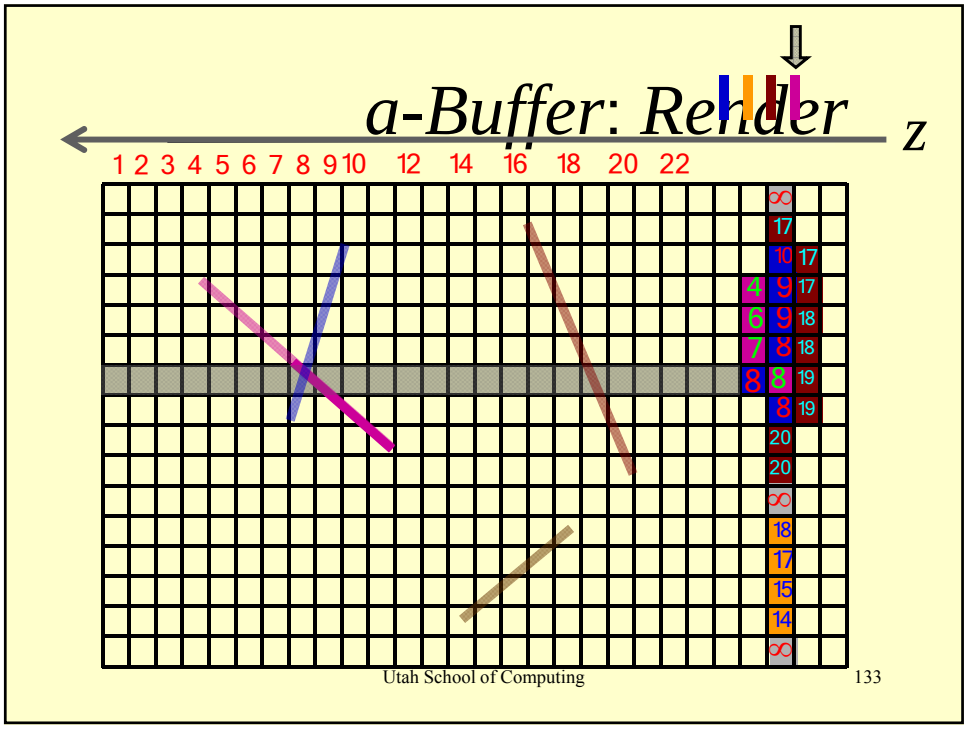

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 17 9 18 8 18 8 19 8 19 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
128

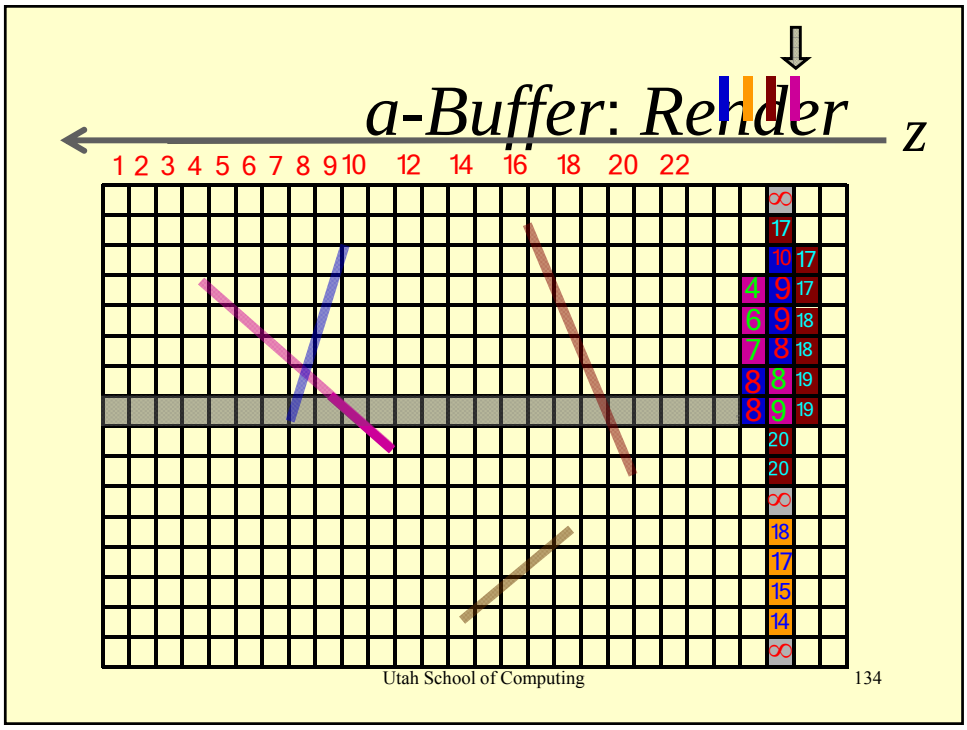

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 9 17 9 18 8 18 8 19 8 19 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
129

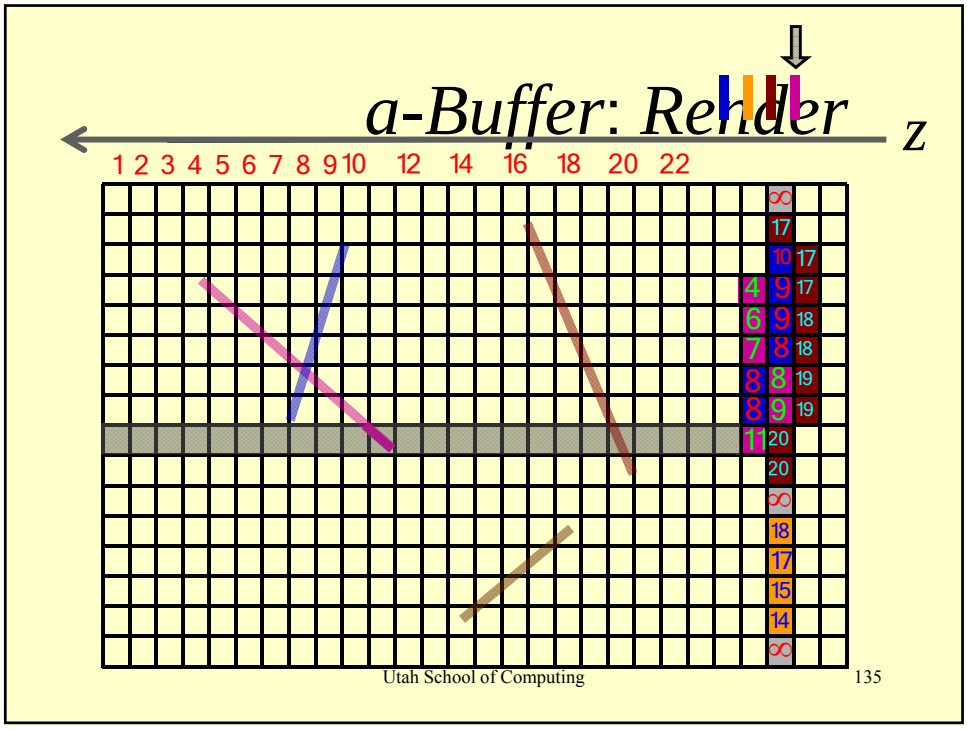

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 4 9 17 9 18 8 18 8 19 8 19 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
130

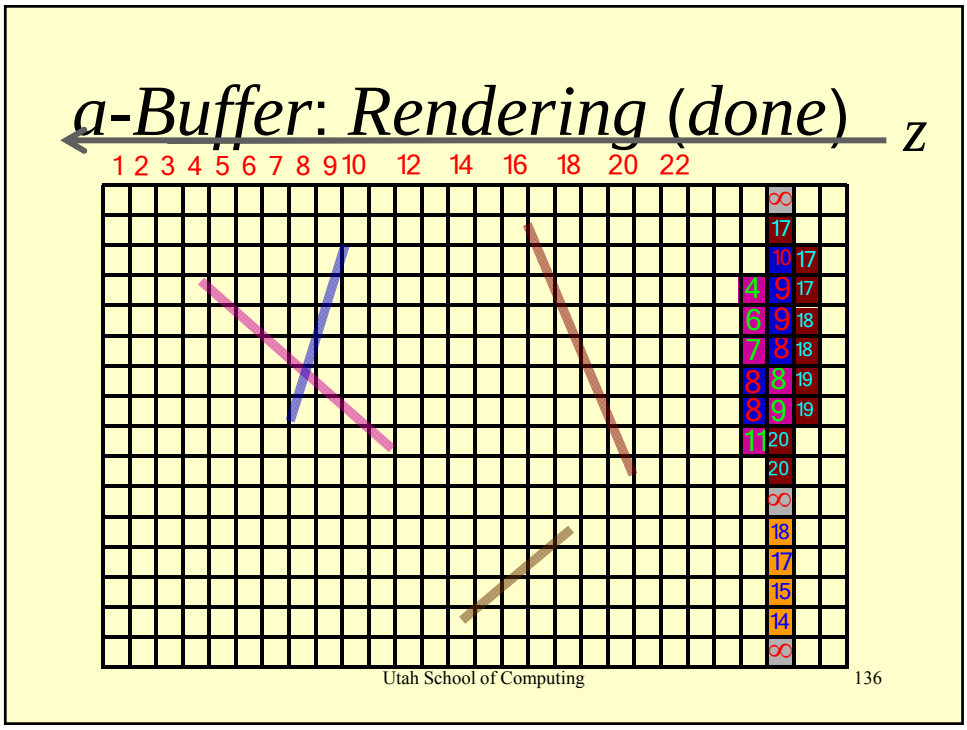

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 4 9 17 6 9 18 8 18 8 19 8 19 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
131

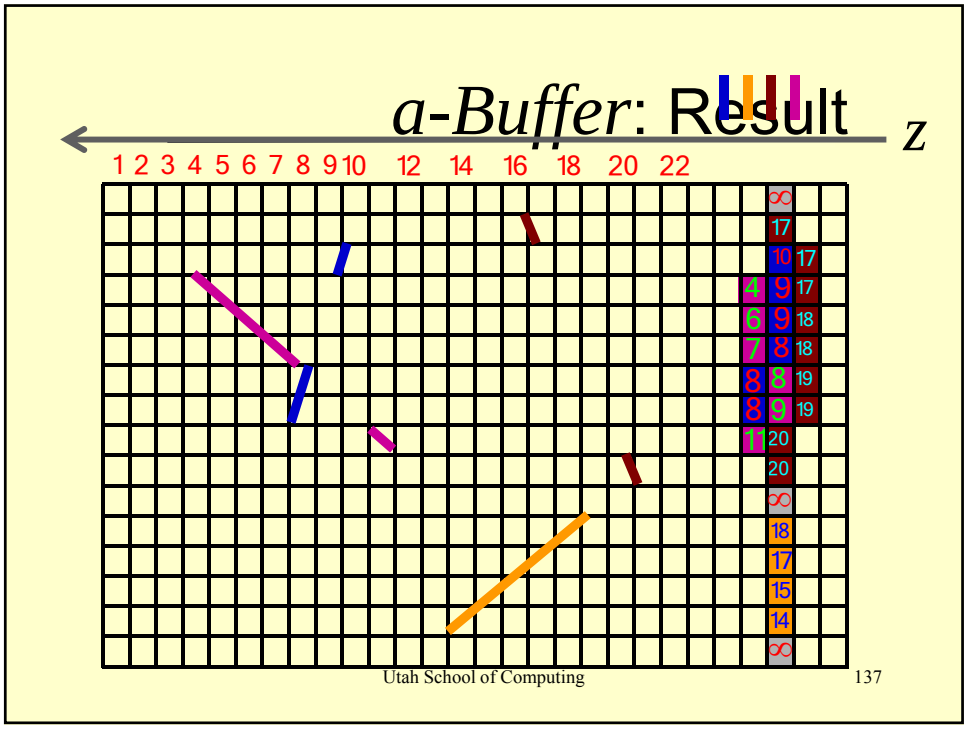

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 4 9 17 6 9 18 7 8 18 8 19 8 19 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
132

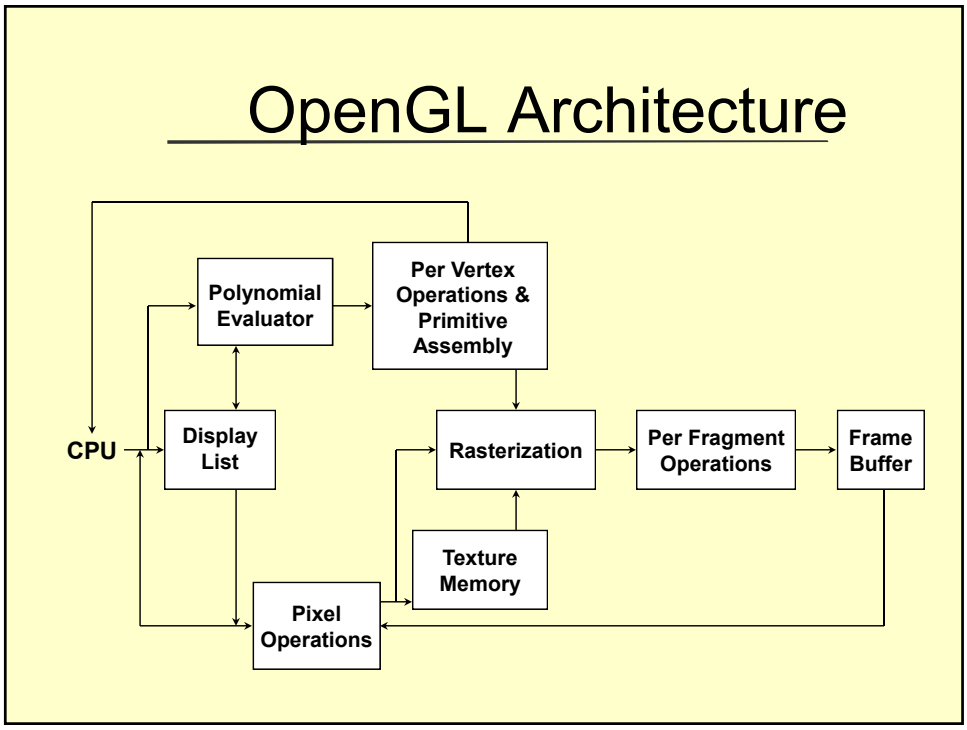

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 4 9 17 6 9 18 7 8 18 8 8 19 8 19 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
133


a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 4 9 17 6 9 18 7 8 18 8 8 19 8 9 19 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
134

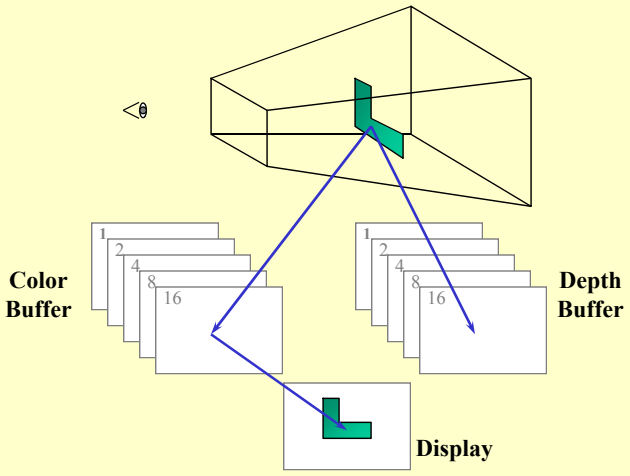

a-Buffer: Render
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 4 9 17 6 9 18 7 8 18 8 8 19 8 9 19 11 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
135


a-Buffer: Rendering (done)
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 4 9 17 6 9 18 7 8 18 8 8 19 8 9 19 11 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing
136


a-Buffer: Result
z
1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22
∞ 17 10 17 4 9 17 6 9 18 7 8 18 8 8 19 8 9 19 11 20 20 ∞ 18 17 15 14 ∞
Utah School of Computing 137


OpenGL Architecture
CPU
Display List
Polynomial Evaluator
Per Vertex Operations & Primitive Assembly
Rasterization
Per Fragment Operations
Frame Buffer
Texture Memory
Pixel Operations

Depth Buffering and Hidden Surface Removal



Depth Buffering Using WebGL

① Enable depth buffering (typically in init)

```
gl.enable(gl.DEPTH_TEST)
```

② Clear color and depth buffers

```
gl.clear(gl.COLOR_BUFFER_BIT |
  gl.DEPTH_BUFFER_BIT)
```

③ Render scene

④ Swap color buffers (you know how)

Cubev.html (Chapter 4)

```
window.onload = function init()
{
    canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

    gl.enable(gl.DEPTH_TEST);

    // shaders load/compile/link/use

    // set up vertex buffers and GUI

    render();
}
```

Cubev.html (Chapter 4)

```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    theta[axis] += 2.0;
    gl.uniform3fv(thetaLoc, theta);


    gl.drawElements( gl.TRIANGLES, numVertices, gl.UNSIGNED_BYTE, 0 );

    requestAnimFrame( render );
}
```

The End

Visible Surface Determination

143