

Visible Surface Determination

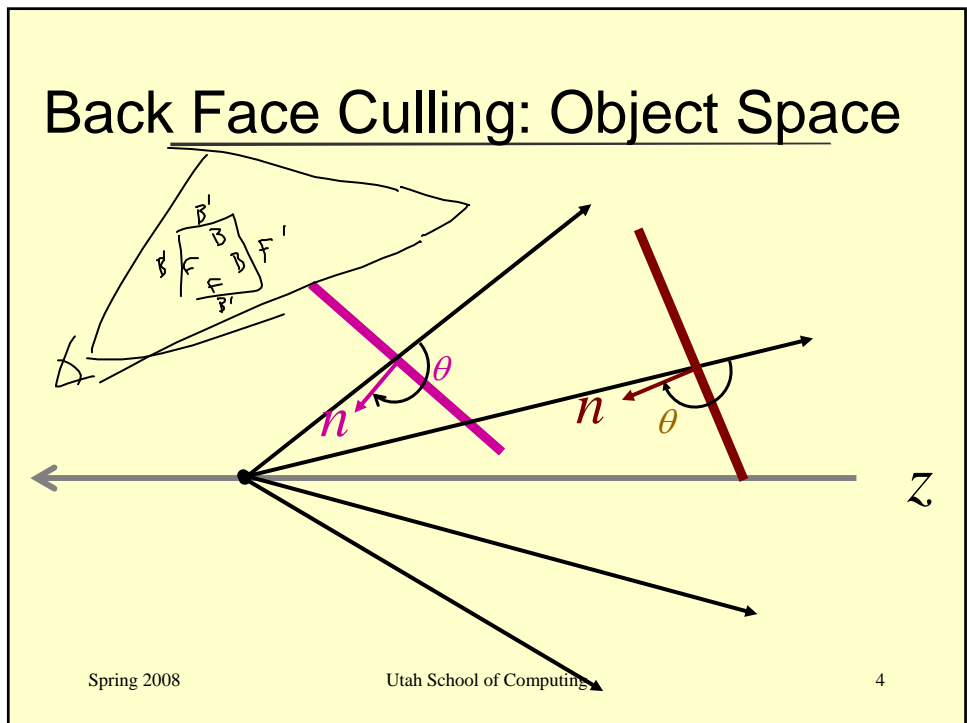
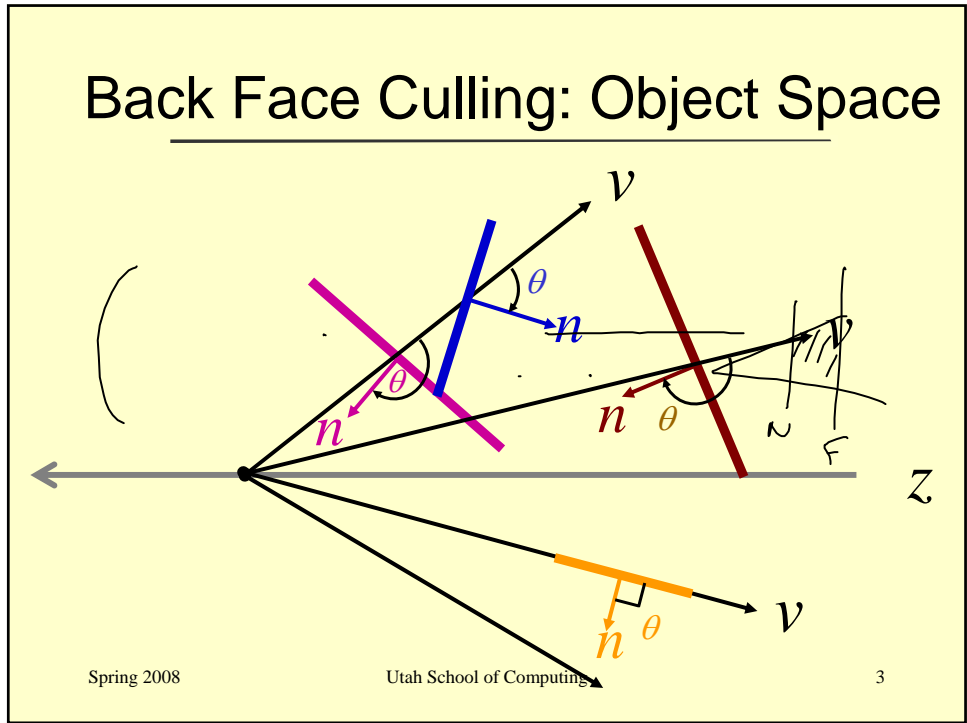
CS5600 Computer Graphics
From Rich Riesenfeld
Spring 2009

Lecture Set 11

Class of Algorithms

- Object (Model) Space Algorithms
 - Work in the ^{World} model data space
- Image Space Algorithms
 - Work in the projected space ✓
 - Most common VSD domain

Spring 2008 Utah School of Computing



Back Face Culling Test

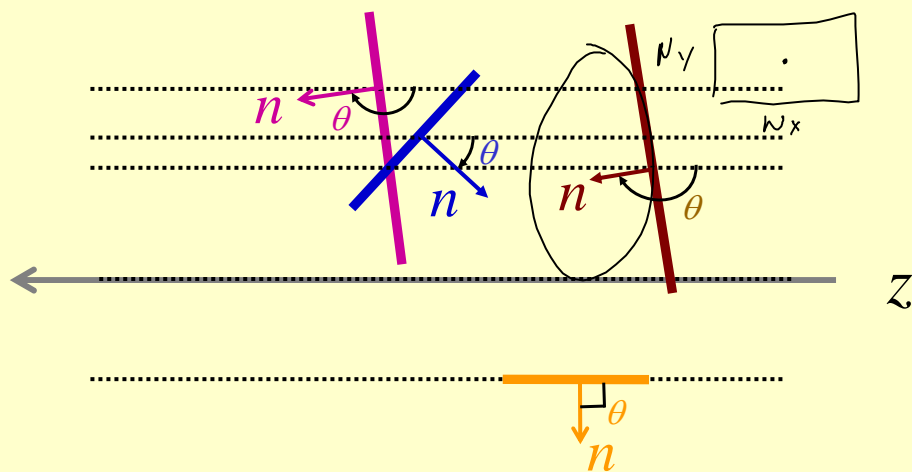
- For Object Space look at sign of $v \cdot n$
- For Image Space look at sign of n_z

Spring 2008

Utah School of Computing

5

Back Face Culling: Image Space

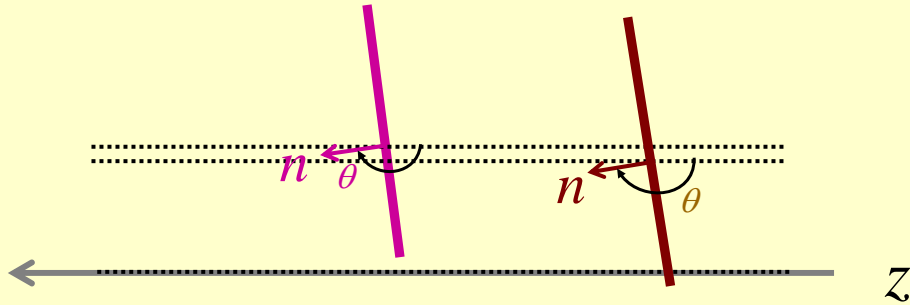


Spring 2008

Utah School of Computing

6

Back Face Culling: Image Space



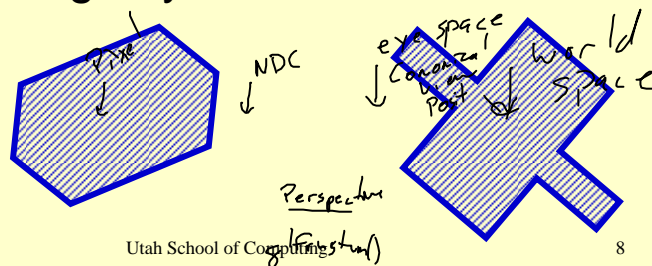
Spring 2008

Utah School of Computing

7

Back Face Culling

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



Spring 2008

Utah School of Computing

8

Back Face Culling: Examples

The diagram shows two viewing frustums originating from smiley faces on the left. The top frustum is directed at a blue hatched hexagonal object. The bottom frustum is directed at a blue hatched cross-shaped object. Handwritten annotations include 'from' with an arrow pointing to the left, 'at' with an arrow pointing to the objects, and 'up?' with an arrow pointing to the top object. To the right of the objects are two rectangular boxes with 'D' labels and arrows, possibly representing depth or distance calculations. An 'X' is drawn over the top object, indicating it is culled because its back faces are visible. The bottom object is not culled as its front faces are visible.

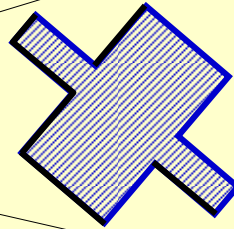
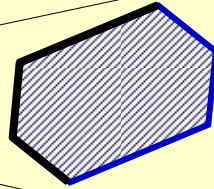
Spring 2006 Utah School of Computing 9

Back Face Culling: Examples

This diagram shows the same two viewing frustums as the previous slide. The top frustum is directed at the blue hatched hexagonal object, and the bottom frustum is directed at the blue hatched cross-shaped object. In this view, the top object is rendered with only its front faces, and the bottom object is rendered with only its front faces. The back faces of both objects are not visible, demonstrating the result of back face culling.

Utah School of Computing 10

Back Face Culling: Examples

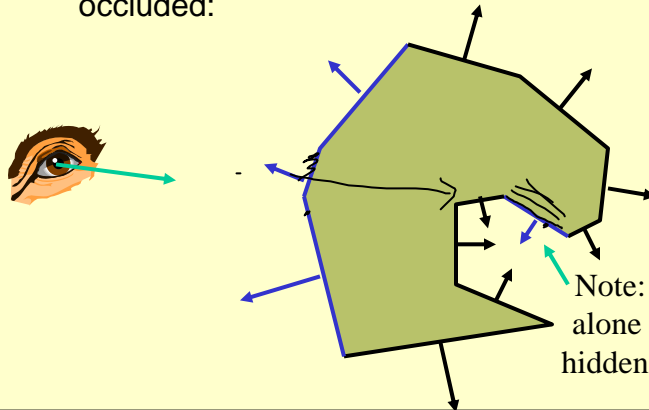


Utah School of Computing

11

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

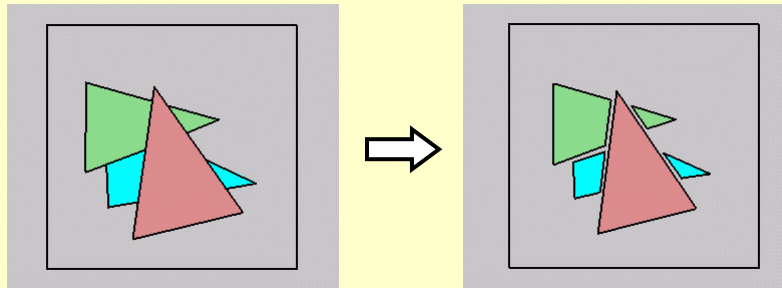
- Not rendering backfacing polygons improves performance
 - *By how much?*
 - *Reduces by about half the number of polygons to be considered for each pixel*

Silhouettes

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

Occlusion

- For most interesting scenes, some polygons will overlap:



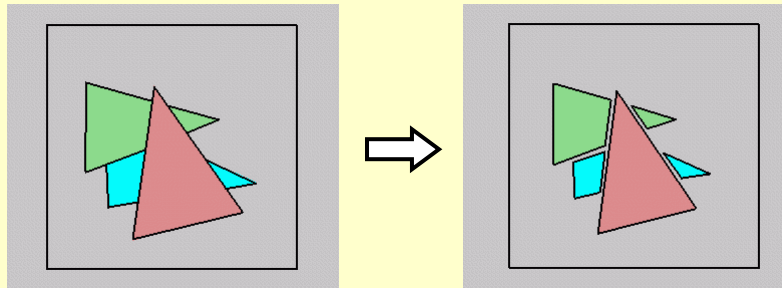
- To render the correct image, we need to determine which polygons occlude which

Painter's Algorithm

- How do painter's solve this?

Painter's Algorithm

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



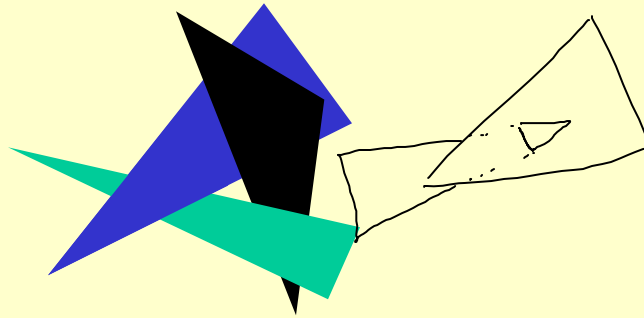
- Draw blue, then green, then orange
- *Will this work in the general case?*

Painter's Algorithm

- 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:

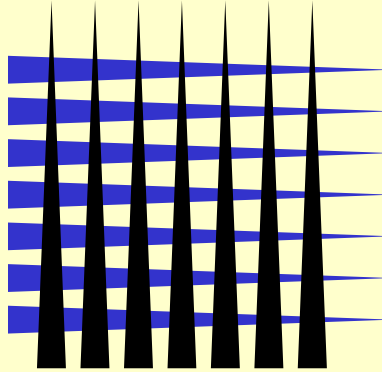


– Now known as *analytic visibility algorithms*

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

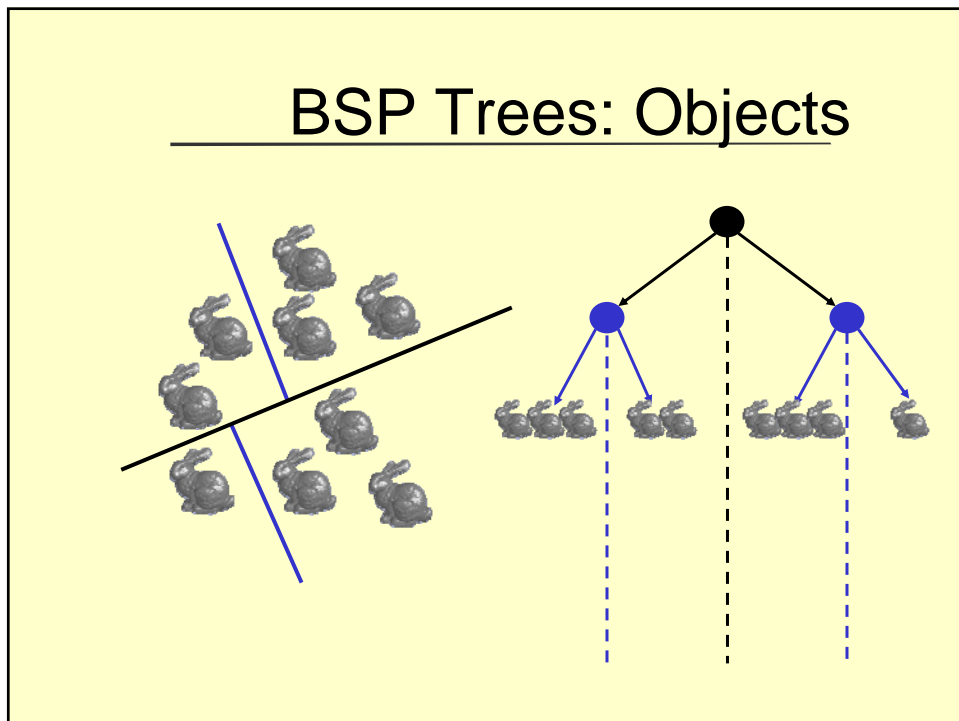
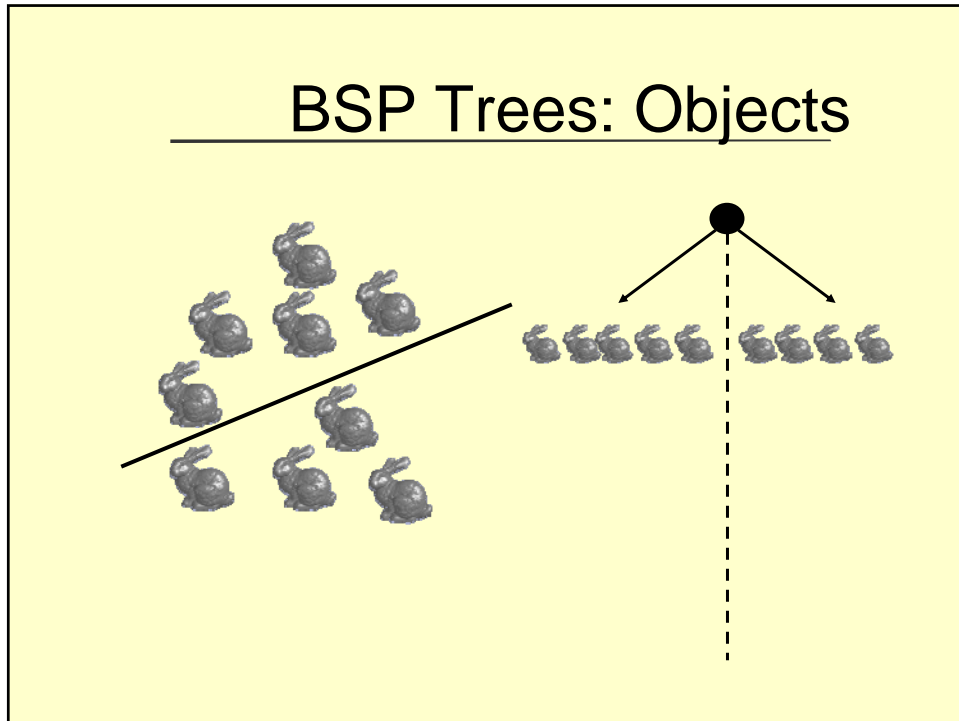
- So, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for *hidden surface removal*
- We'll talk about two:
 - *Binary Space-Partition (BSP) Trees*
 - *Warnock's Algorithm*

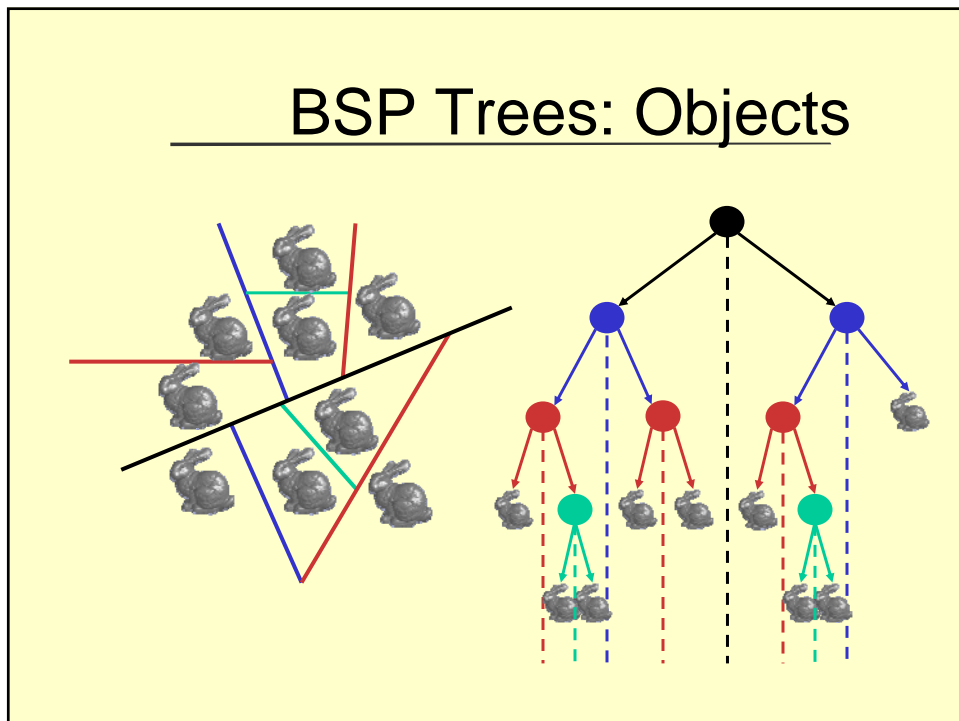
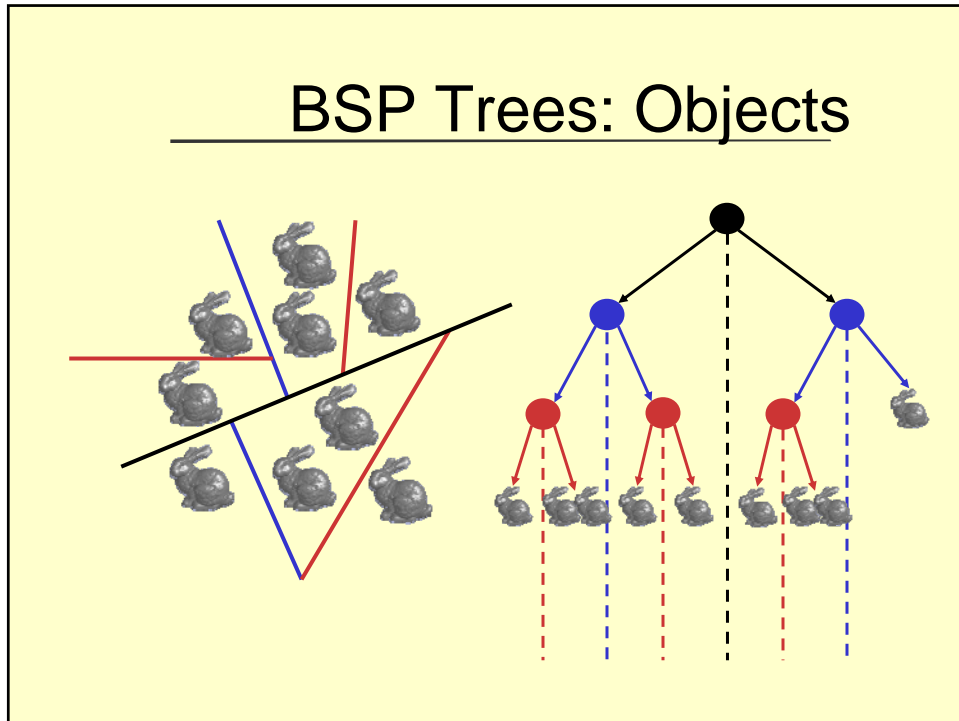
Binary Space Partition Trees (1979)

- 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







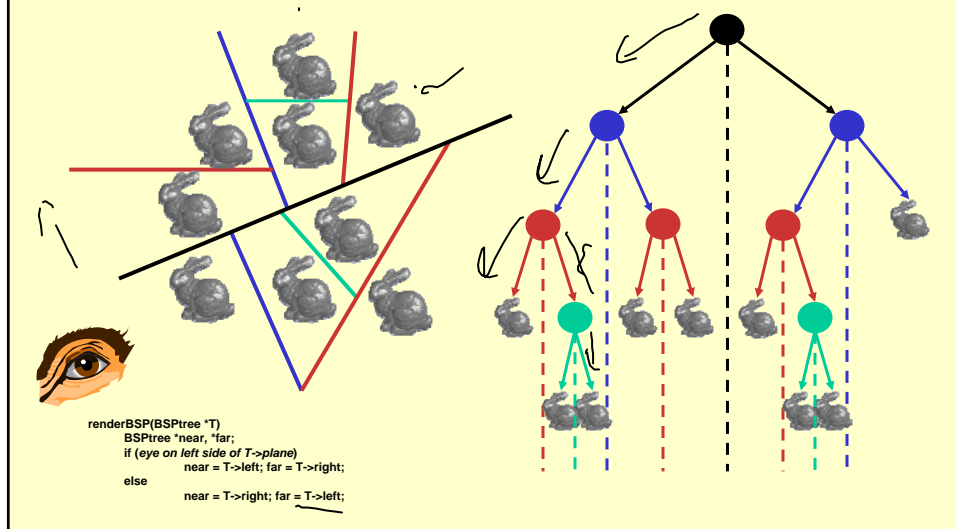
Rendering BSP Trees

```

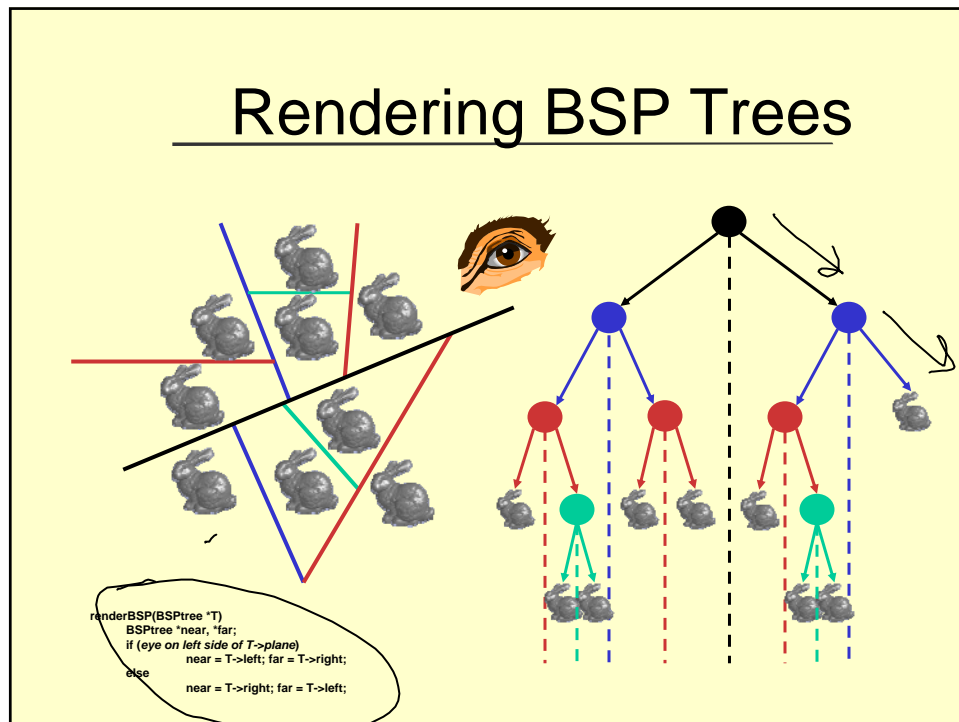
renderBSP(BSPtree *T)
  BSPtree *near, *far;
  if (eye on left side of T->plane)
    near = T->left; far = T->right;
  else
    near = T->right; far = T->left;
  renderBSP(far);
  if (T is a leaf node)
    renderObject(T)
  renderBSP(near);

```

Rendering BSP Trees



Rendering BSP Trees



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:

```

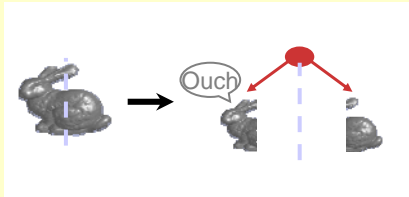
BSPnode::Draw(Vec3 viewpt)
  Classify viewpt: in + or - half-space of node->plane?
  /* Call that the "near" half-space */
  farchild->draw(viewpt);
  render node->polygon; /* always on node->plane */
  nearchild->draw(viewpt);

```

- 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)
 - Thus very popular for video games (but getting less so)
- 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

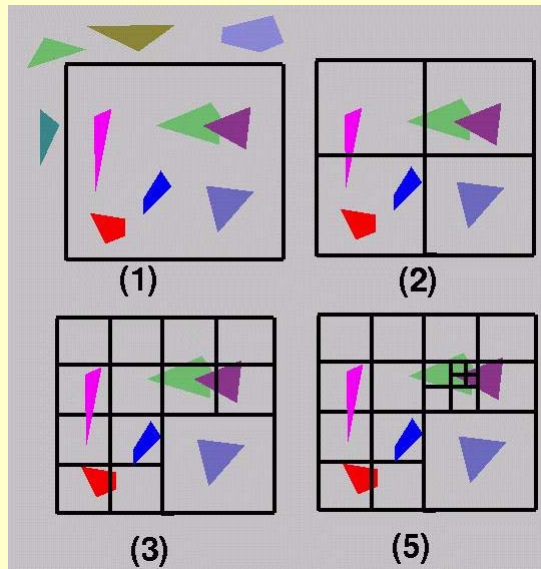
Start lecture here

Warnock's Algorithm (1969)

- 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

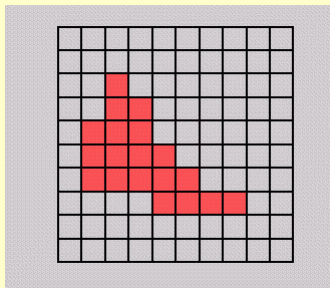
- **Pros:**
 - Very elegant scheme
 - Extends to any primitive type
- **Cons:**
 - Hard to embed hierarchical schemes in hardware
 - Complex scenes usually have small polygons and high *depth complexity*
 - Thus most screen regions come down to the single-pixel case

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*

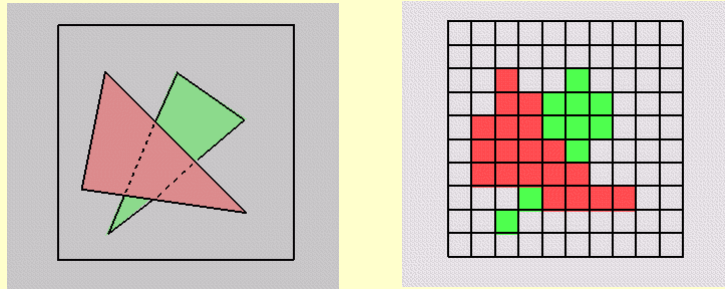
The Z-Buffer Algorithm

- 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
 - Can transform canonical perspective volume into canonical parallel volume with:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

z-Buffer (Depth Buffer)

Conceptually:

$$\text{Sort}_{(x,y)} (\max z_{xy})$$
$$\text{Sort}_{(x,y)} (\min z_{xy})$$

Spring 2008

Utah School of Computing

45

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

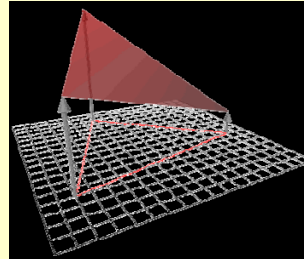
- **Edge equations: Z is just another planar parameter:**

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

If walking across scanline by (Δx)

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

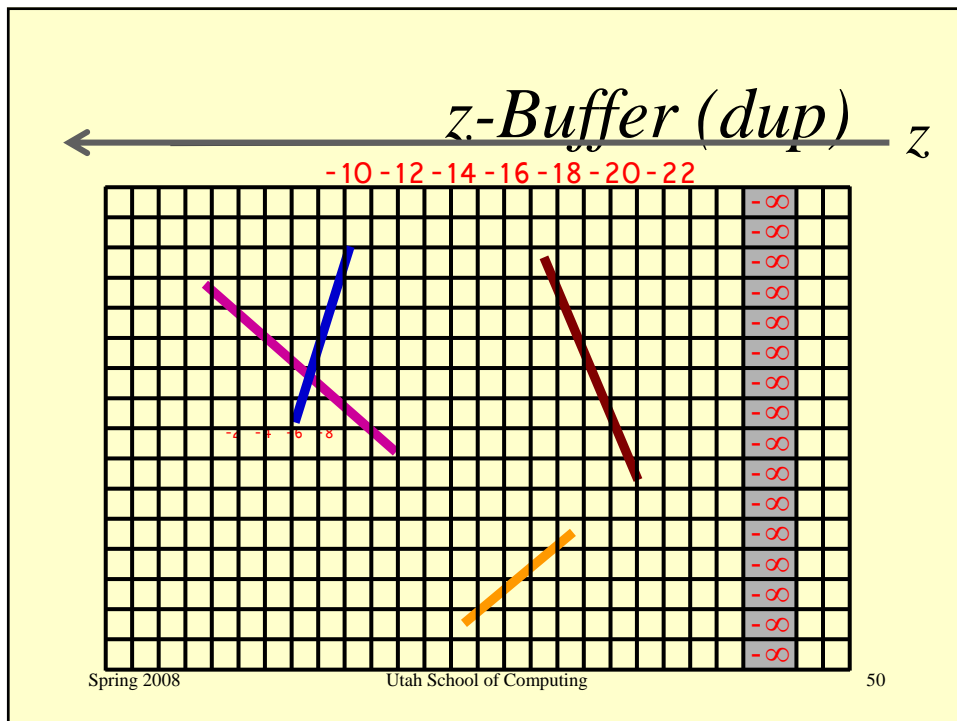
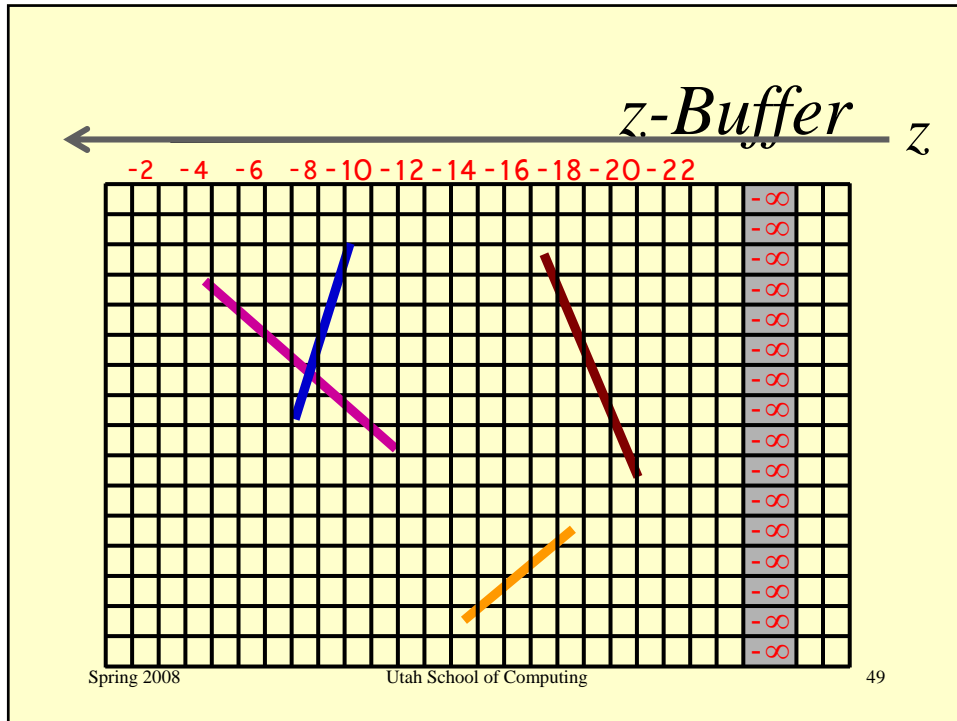
- Look familiar?
- Total cost:
 - 1 more parameter to increment in inner loop
 - 3x3 matrix multiply for setup

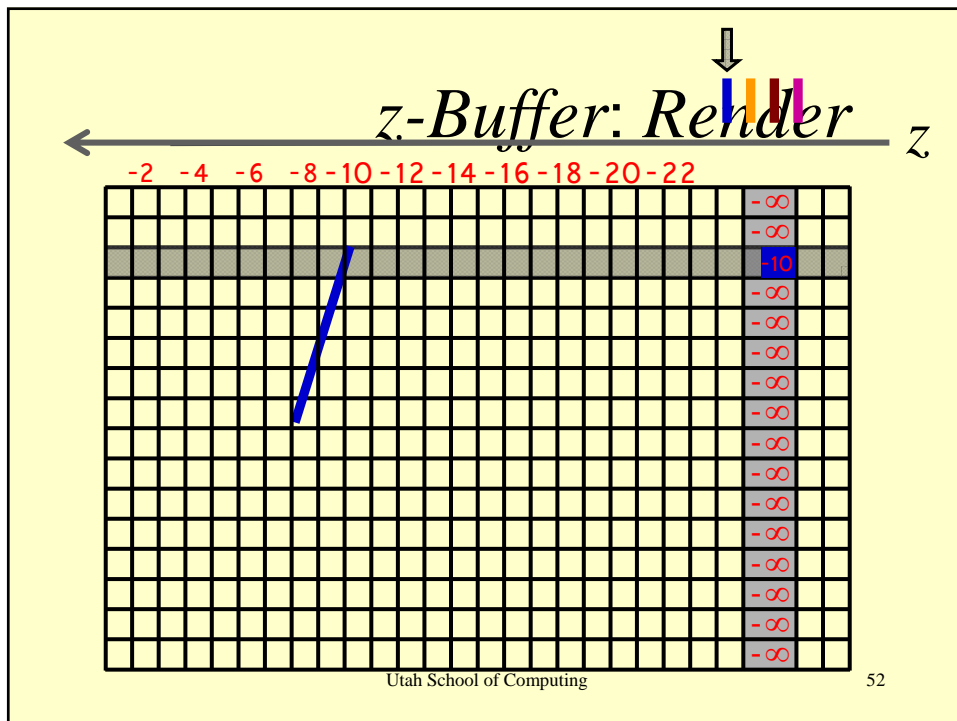
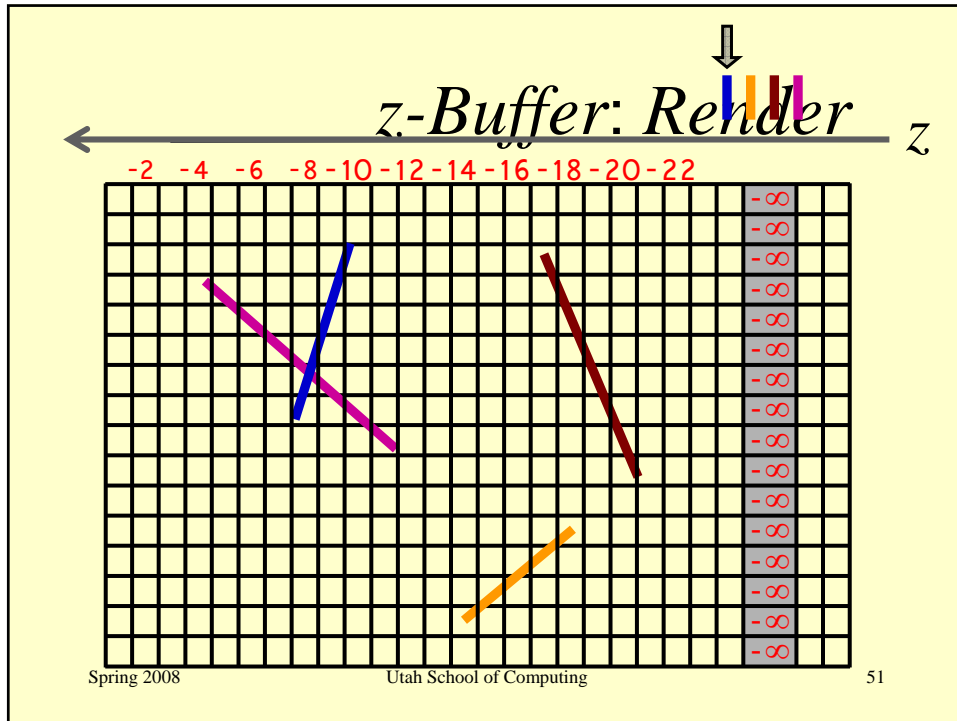


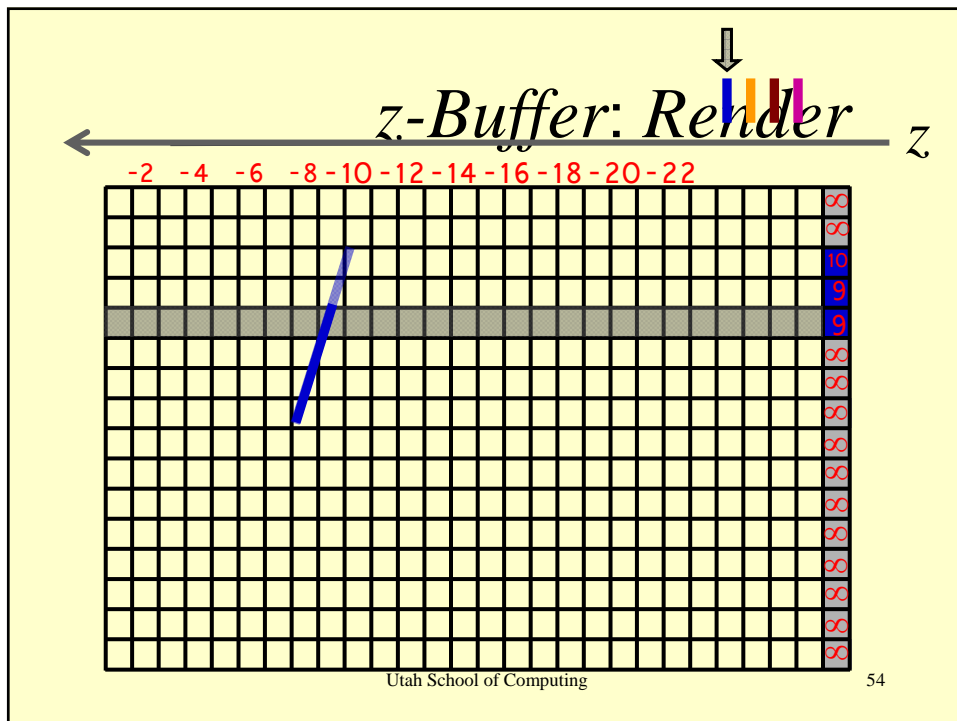
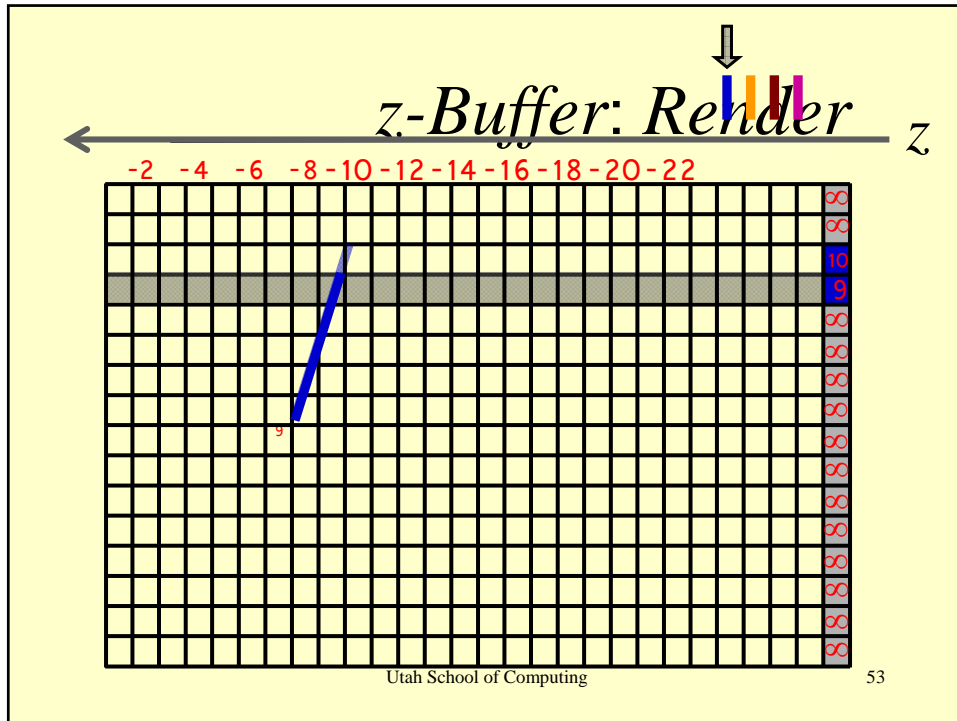
- **Edge walking: just interpolate Z along edges and across spans**

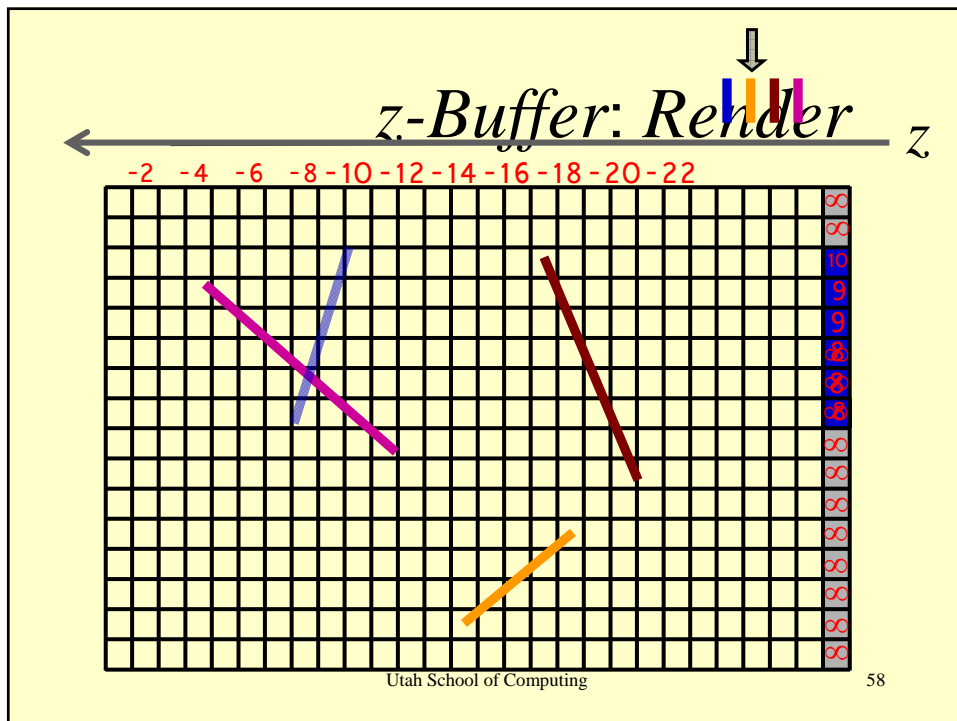
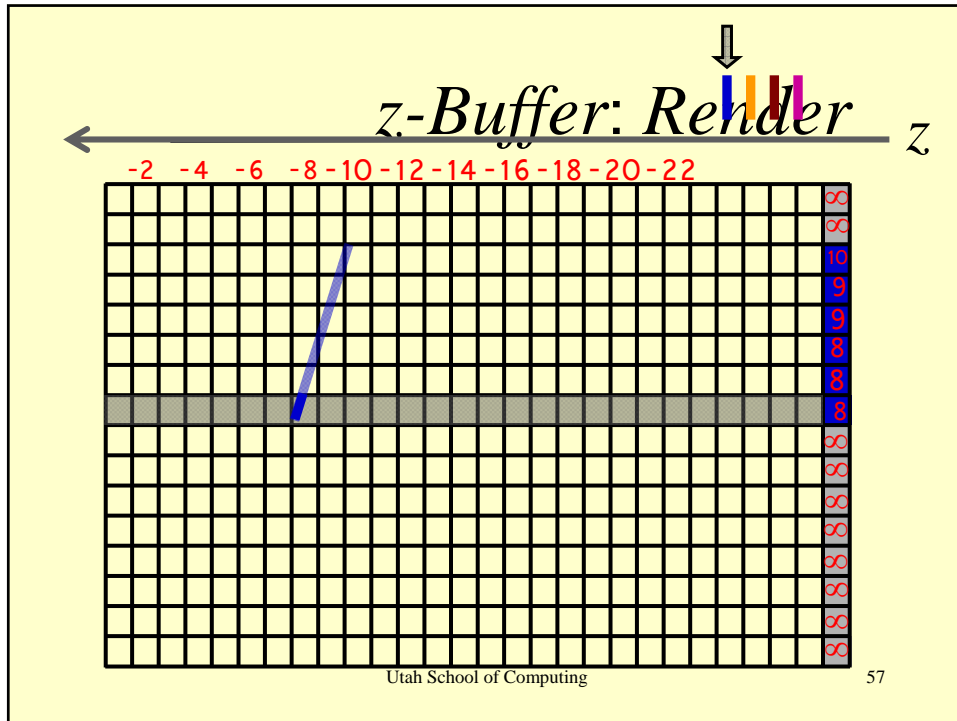
The Z-Buffer Algorithm

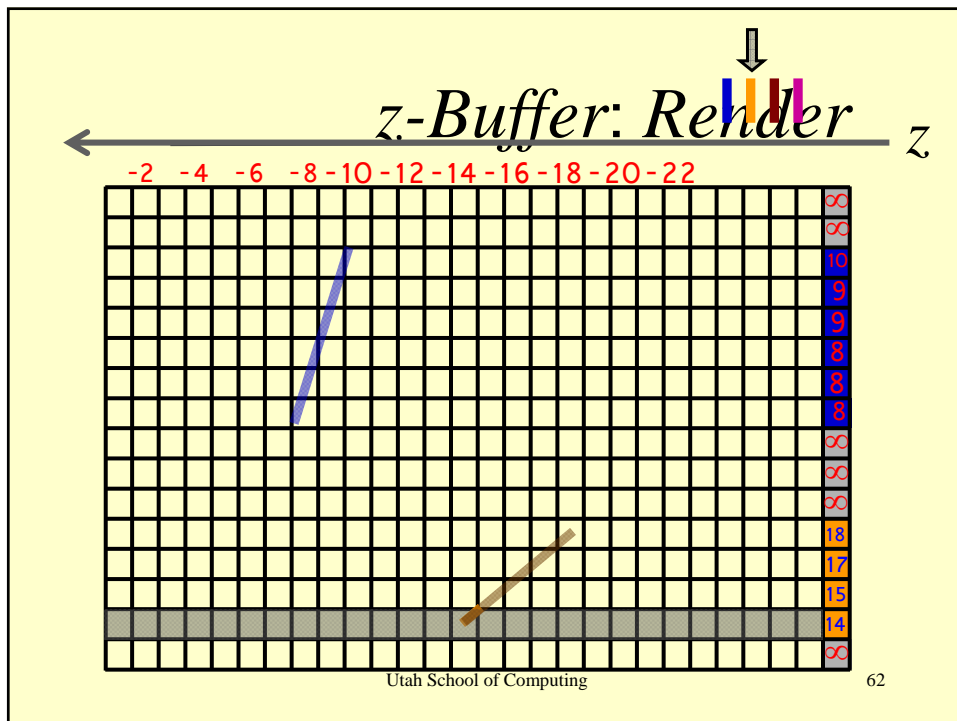
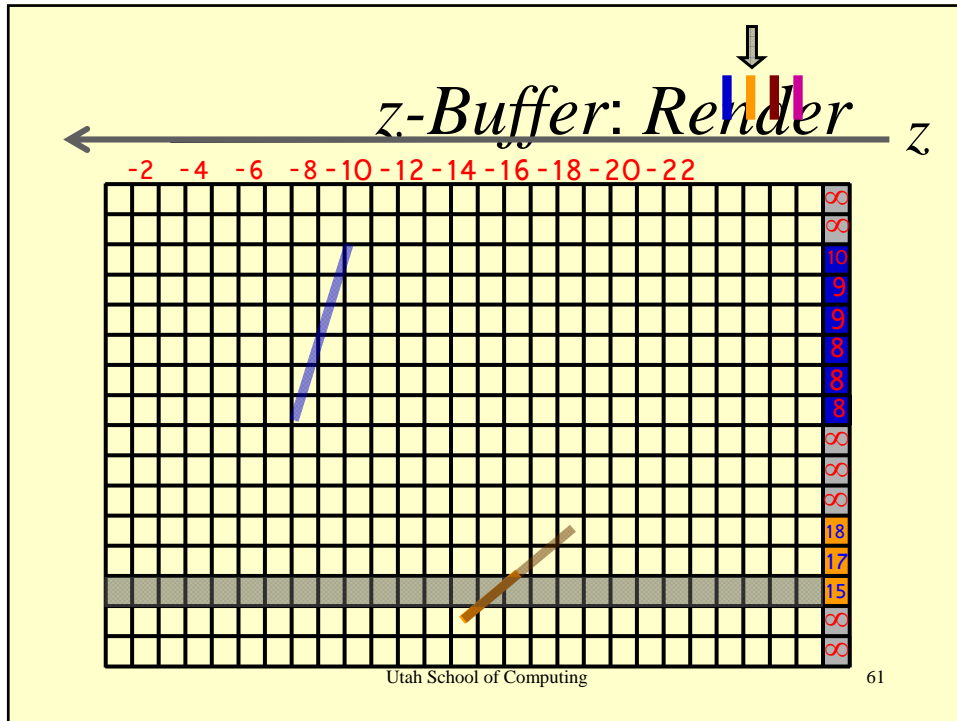
- *How much memory does the Z-buffer use?*
- *Does the image rendered depend on the drawing order?*
- *Does the time to render the image depend on the drawing order?*
- *How does Z-buffer load scale with visible polygons? With framebuffer resolution?*

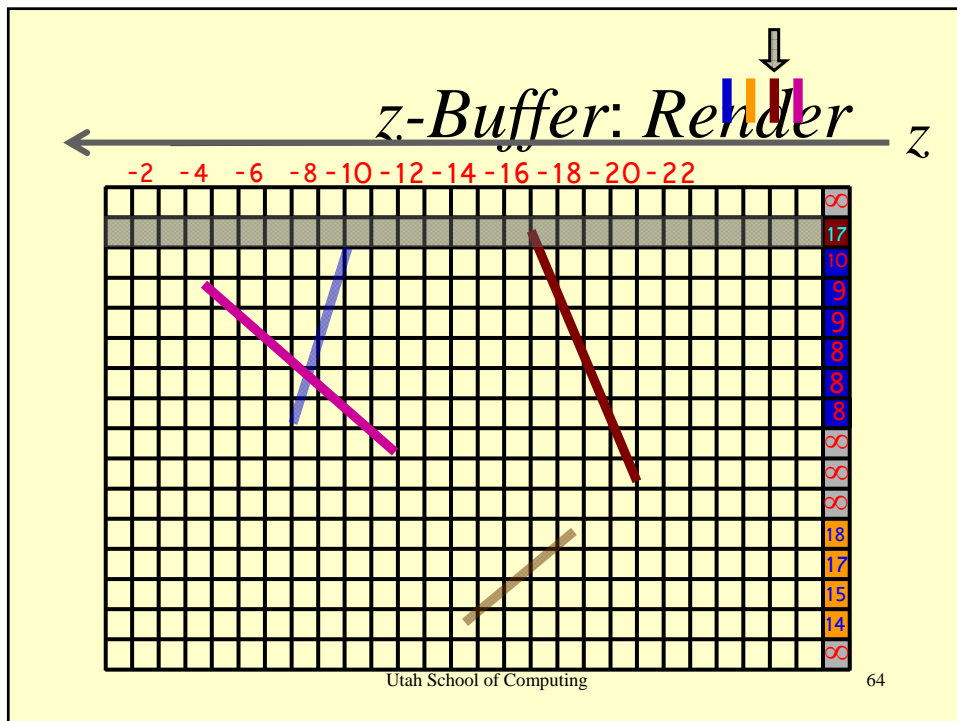
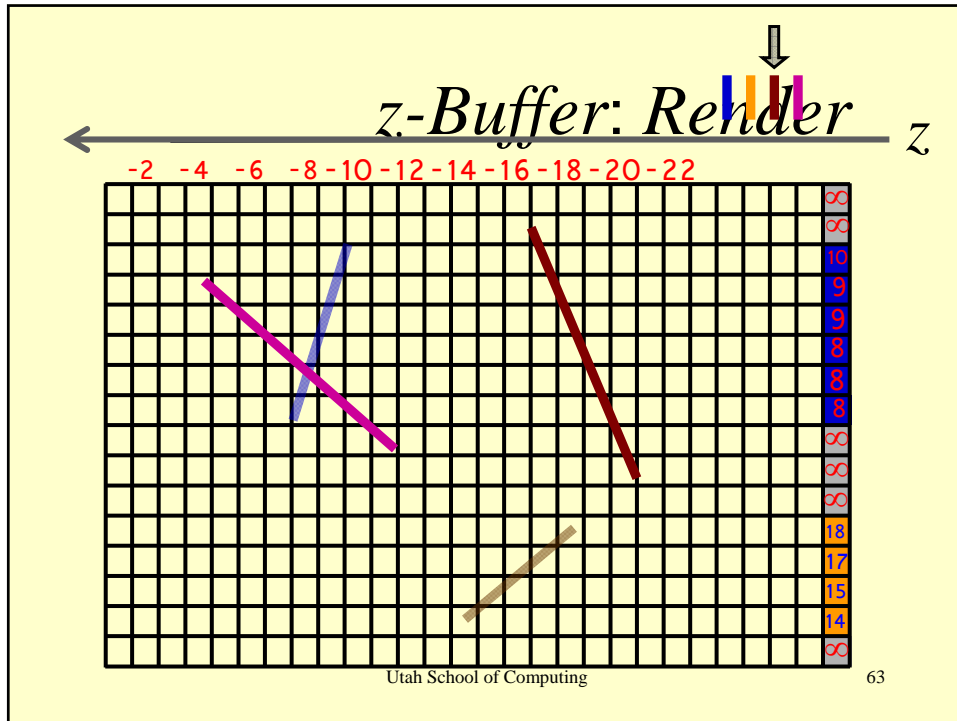


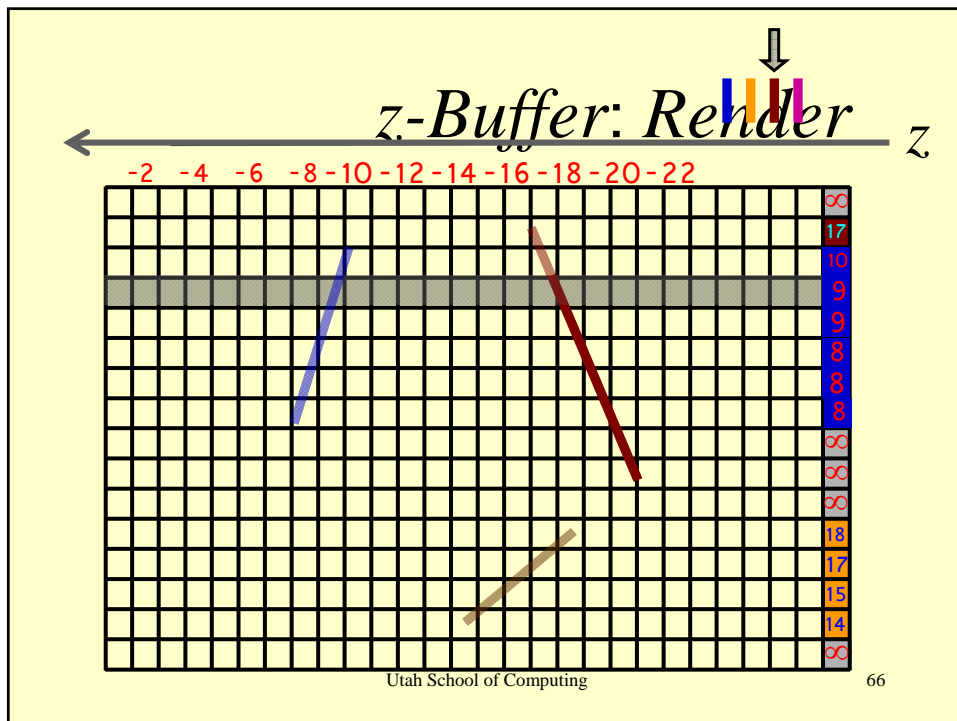
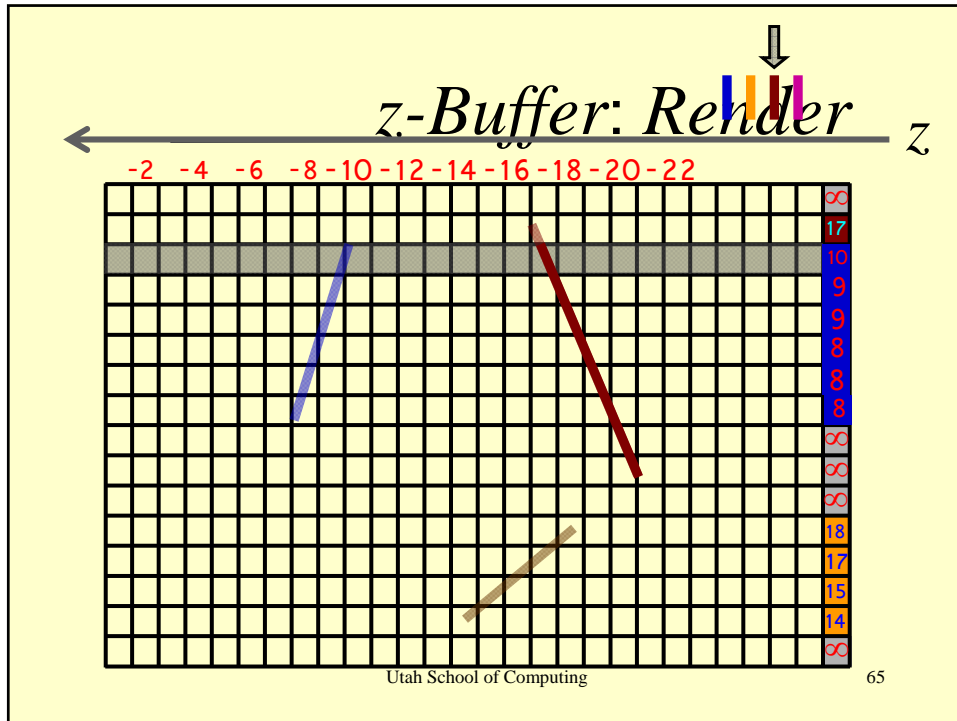


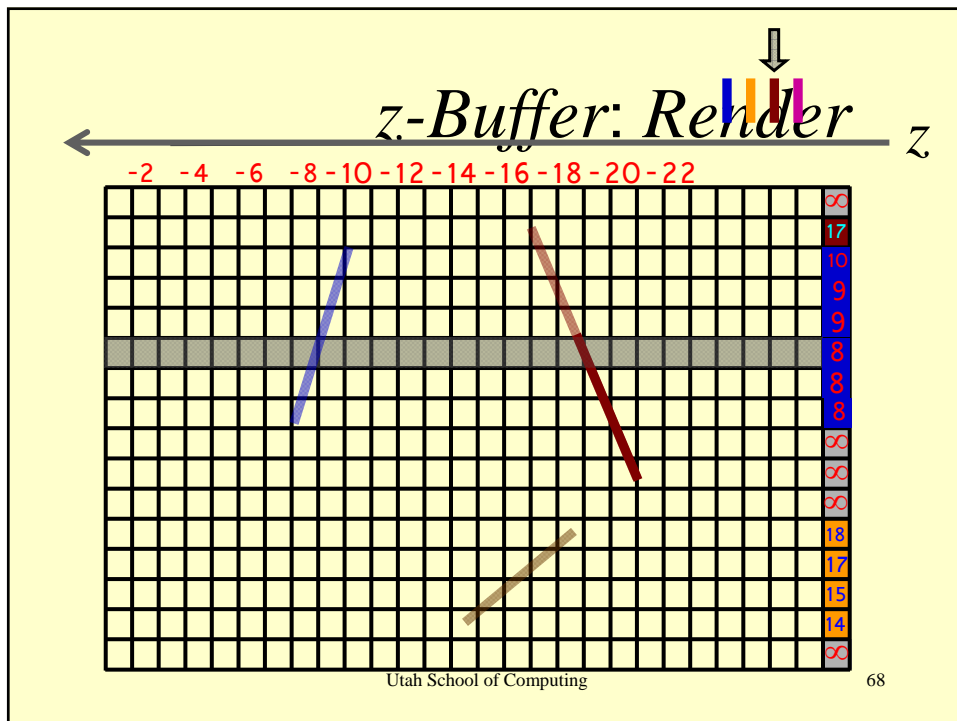
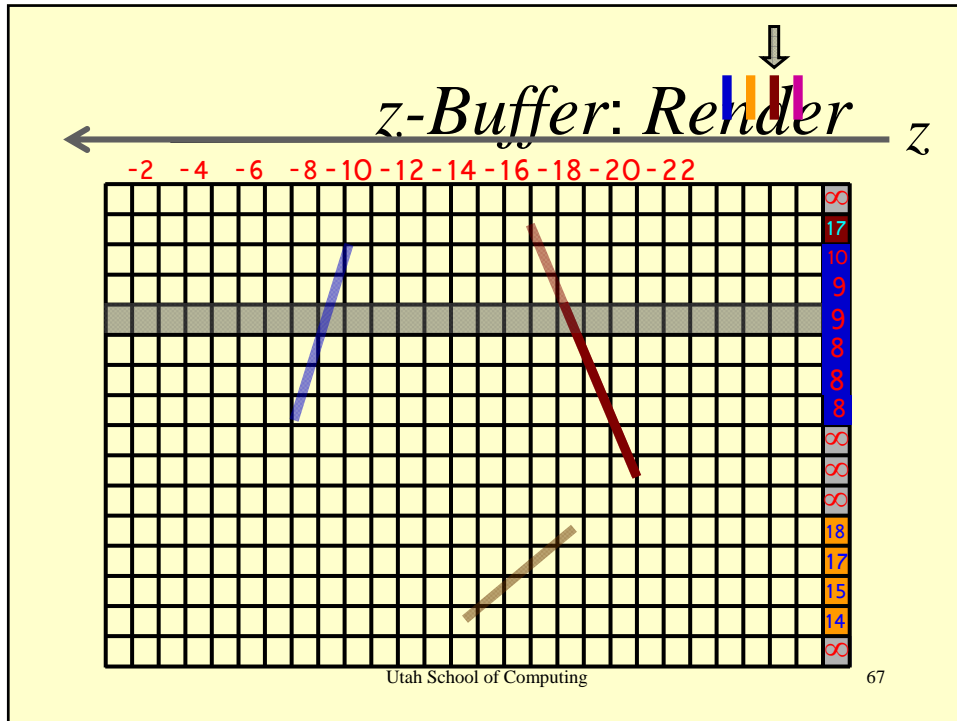


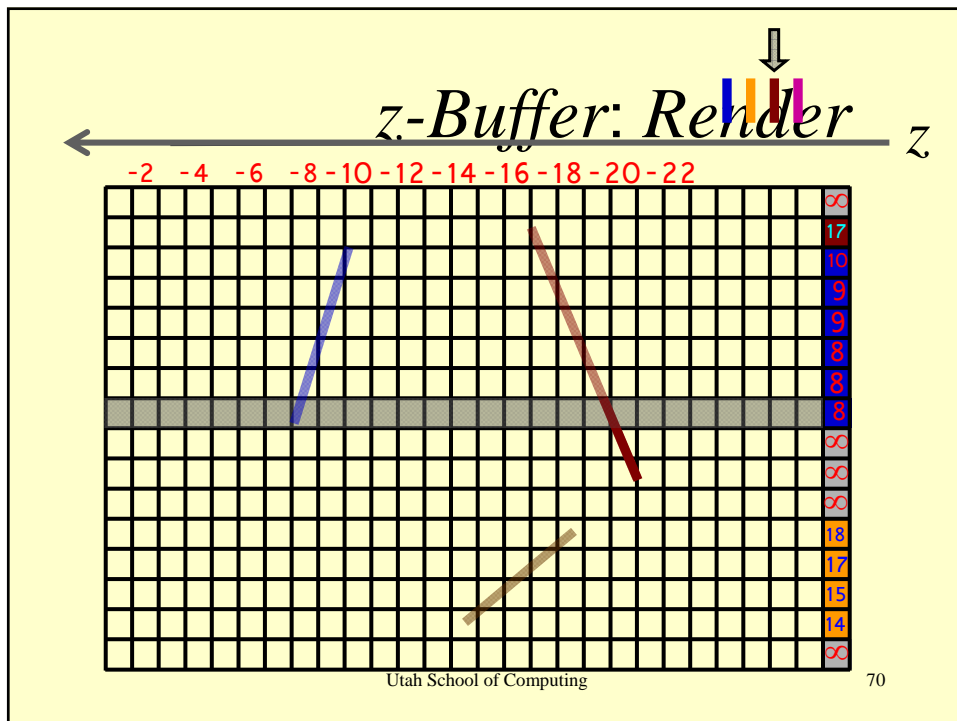
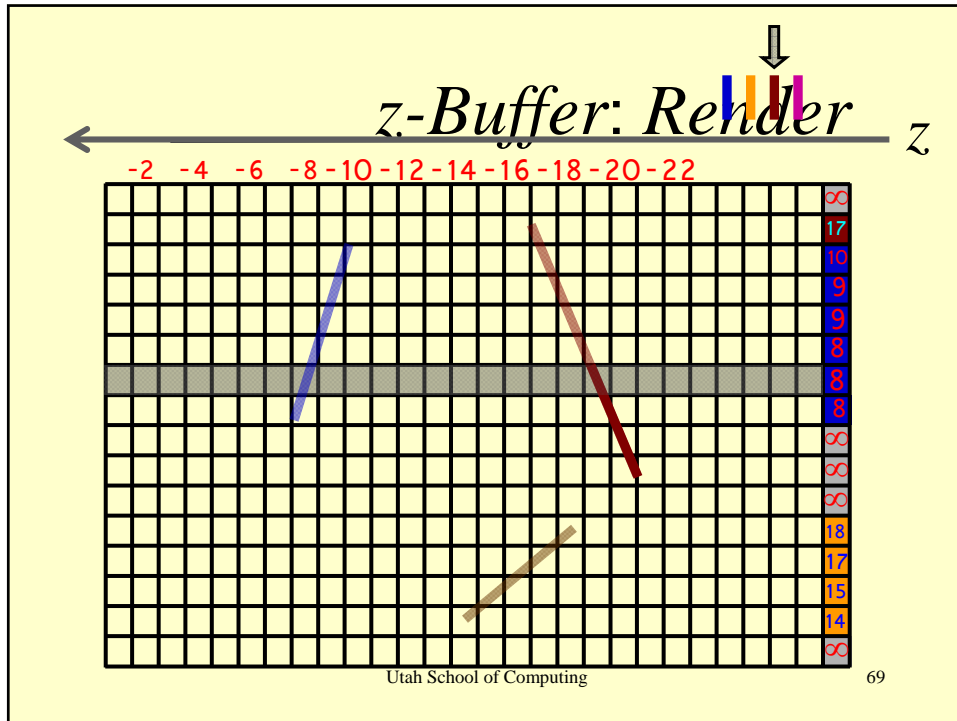


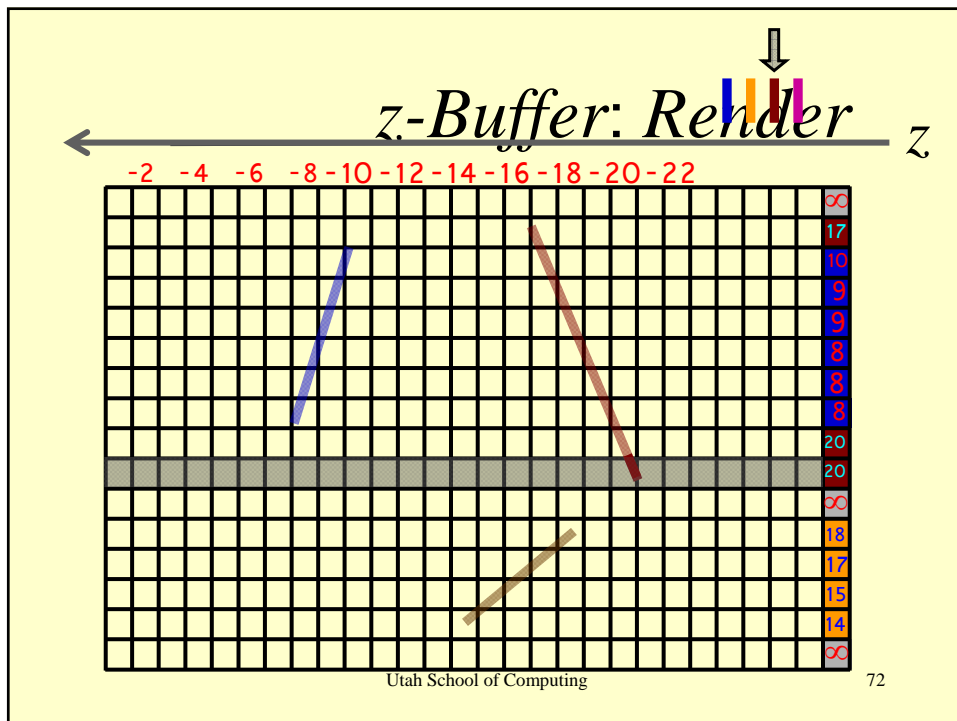
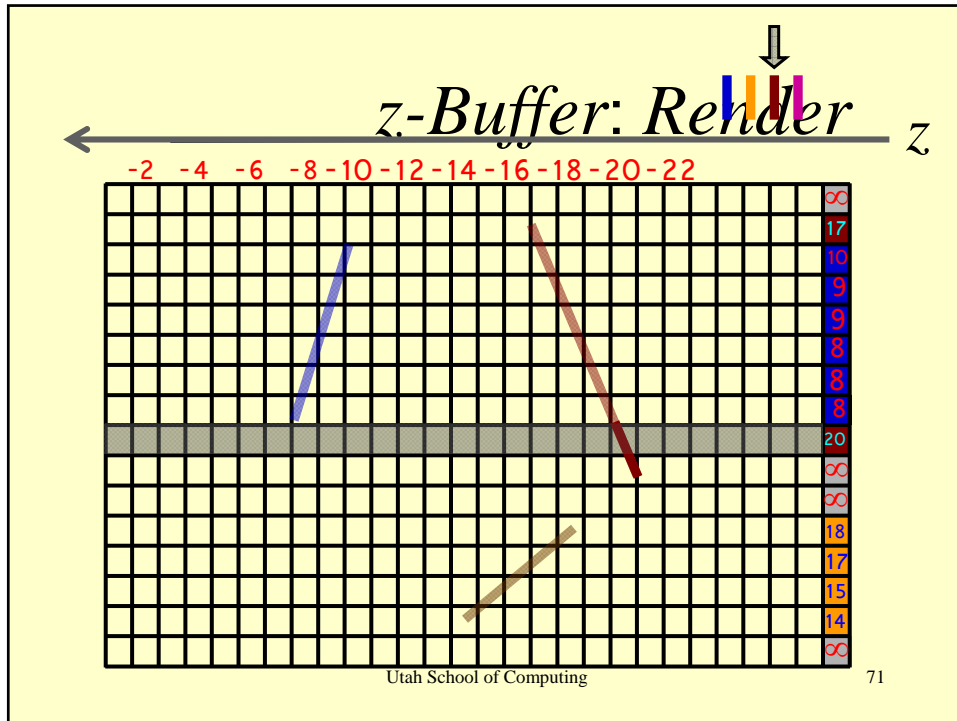


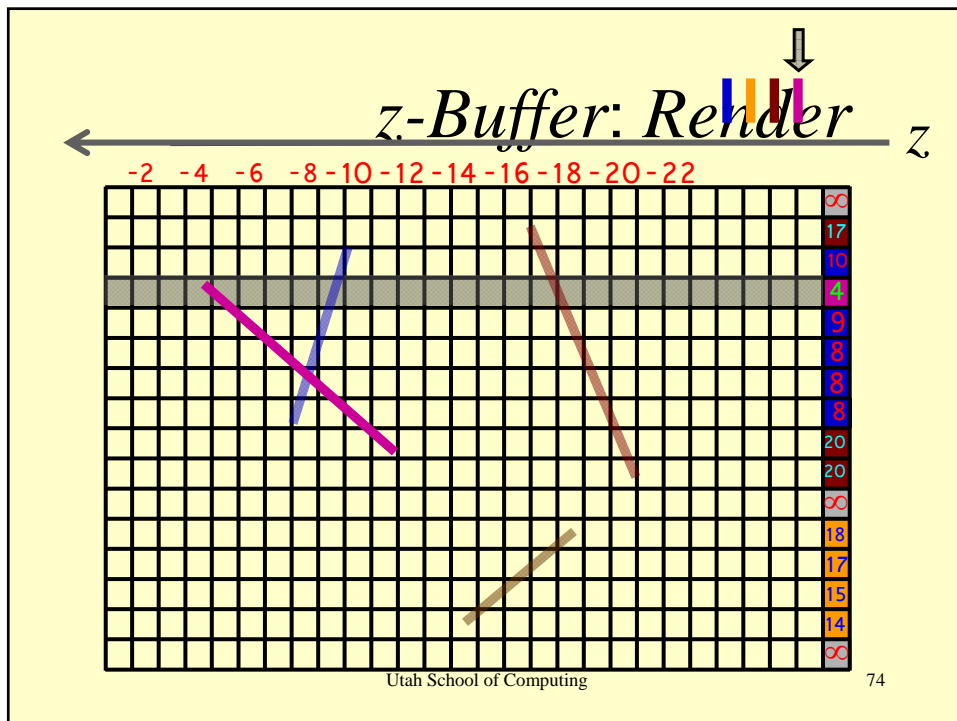
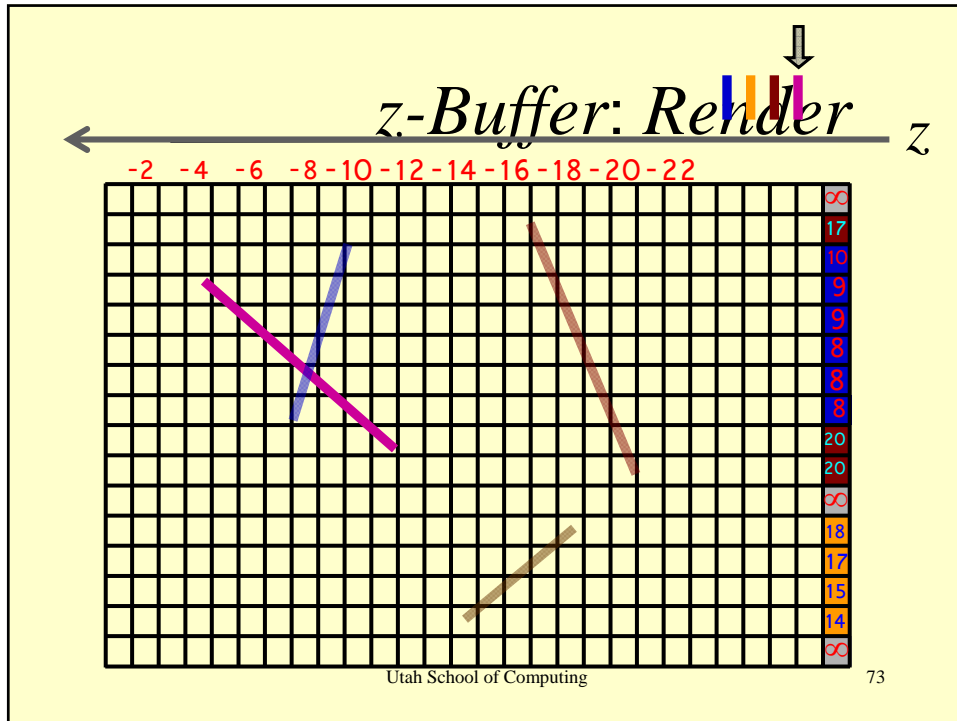


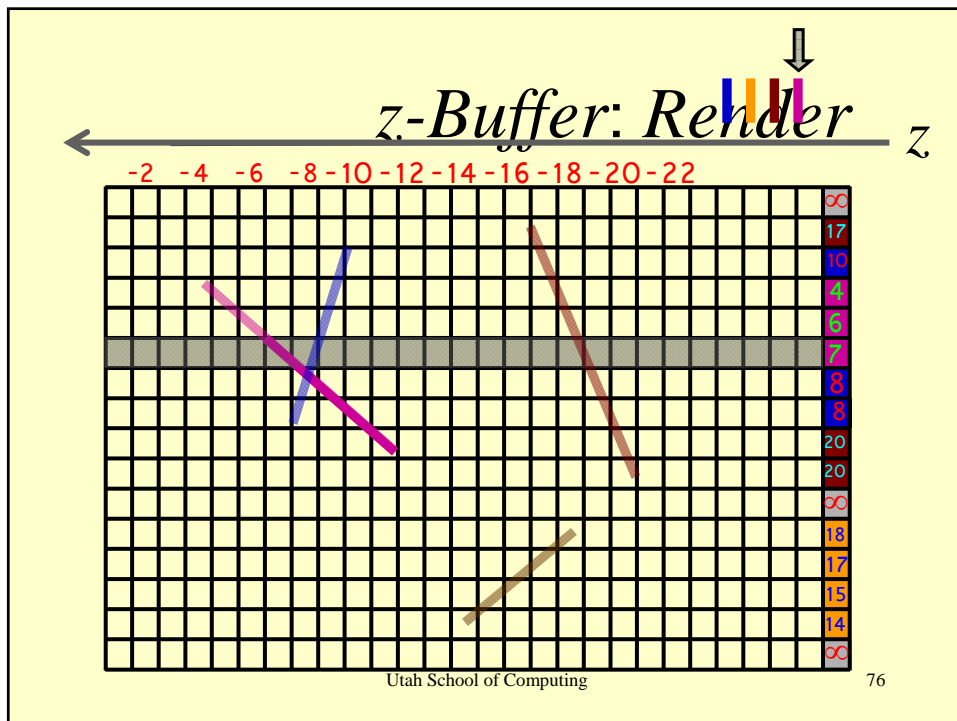
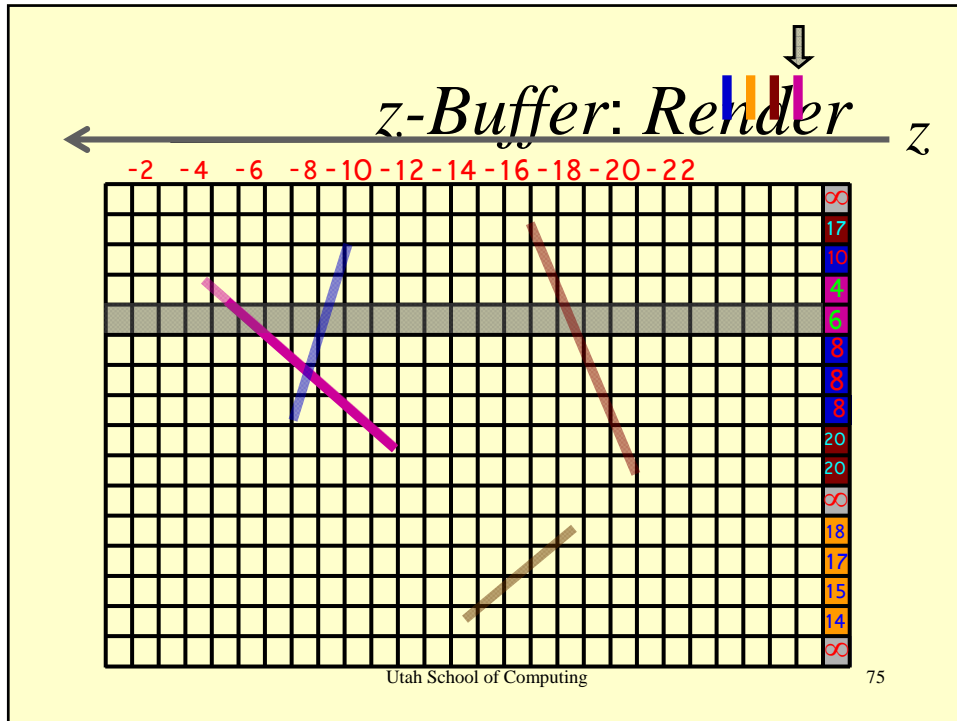


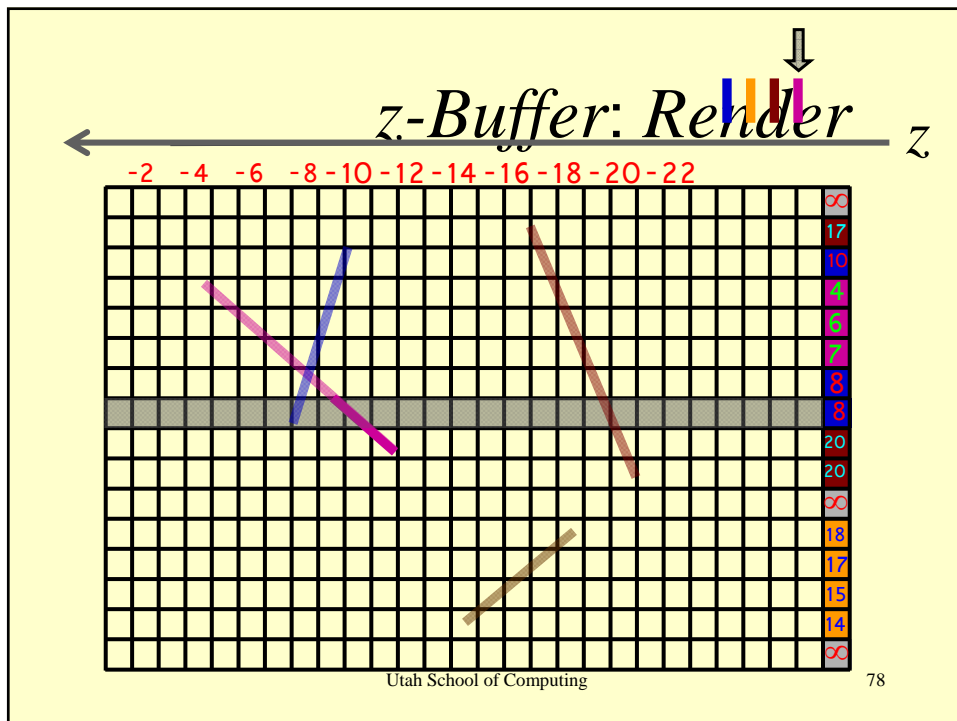
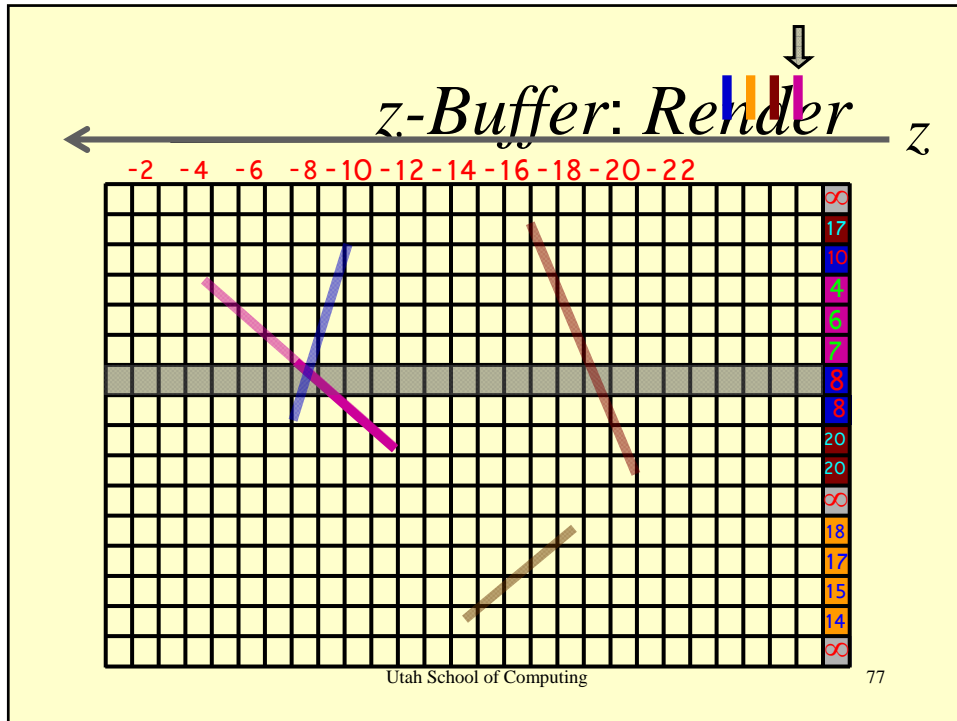


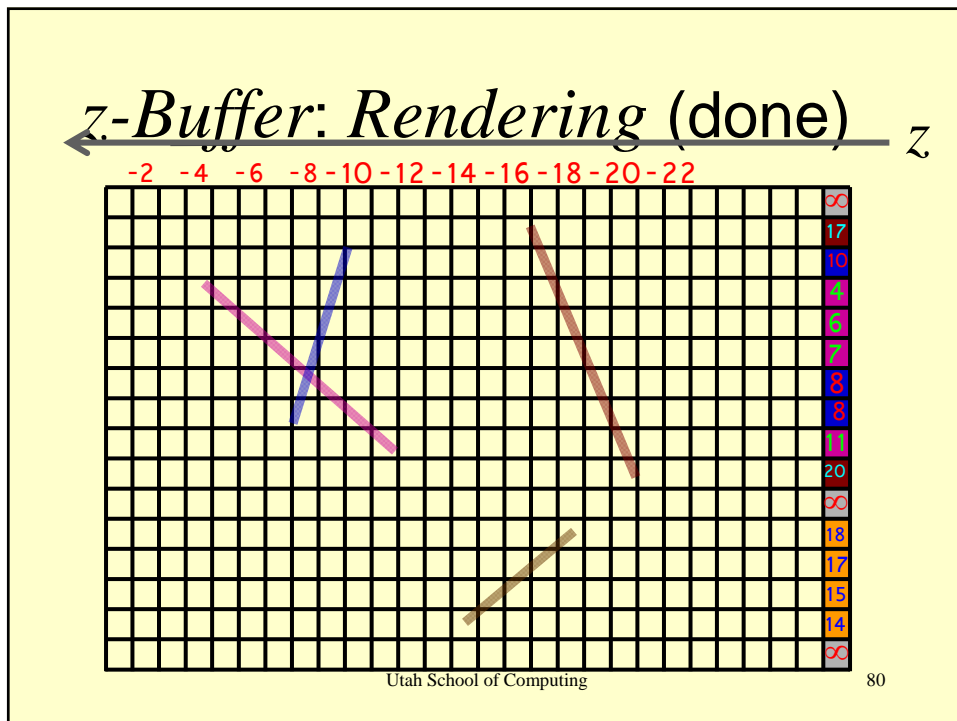
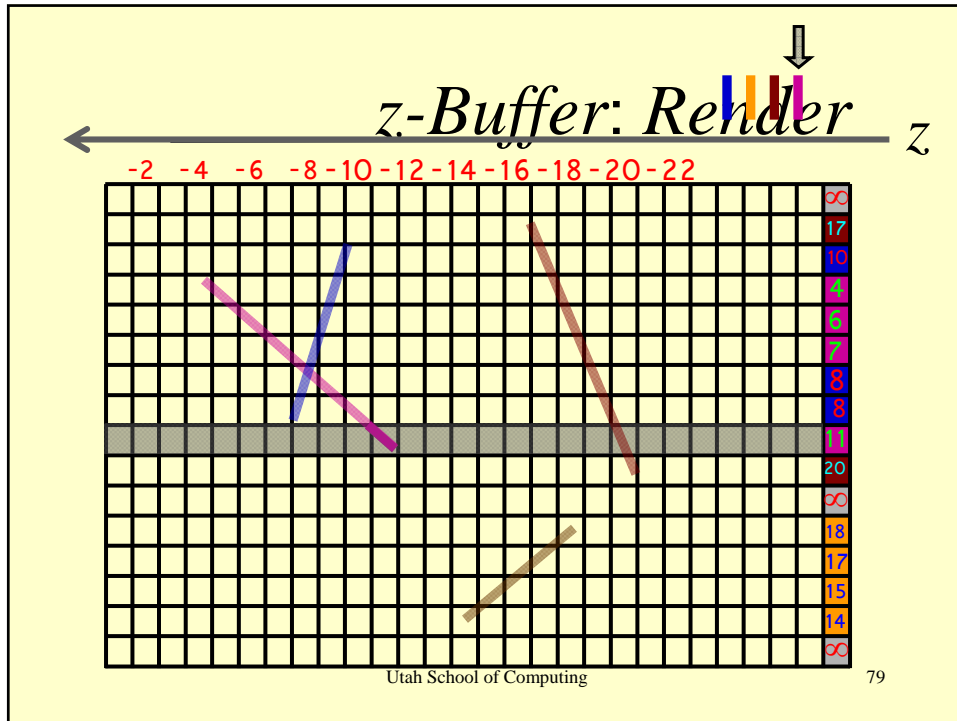


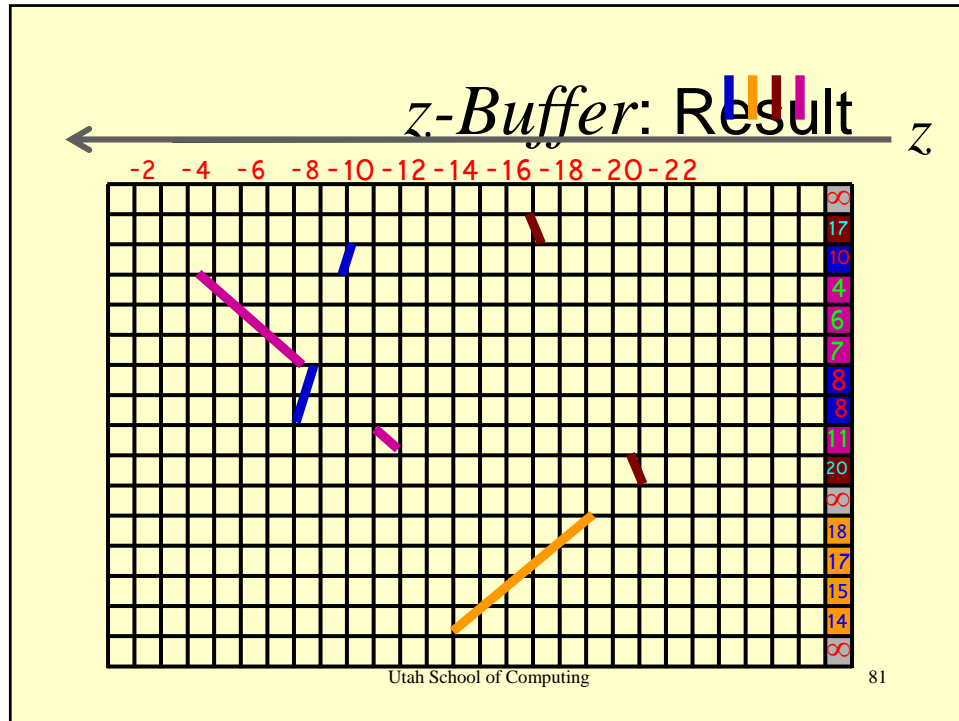












- ### *z-Buffer: Pros*
- Simple algorithm
 - Easy to implement in hardware
 - Complexity is order N , for polygons
 - No polygon processing order required
 - Easily handles polygon interpenetration
- Utah School of Computing
- 82

← *z-Buffer: Cons* z

- Memory intensive
- Hard to do antialiasing
- Hard to simulate translucent polygons
- Precision issues (scintillating, worse with perspective projection)

← *z-Buffer Algorithm* z

- Initialize buffer
 - Set background *intensity, color* $\langle r, g, b \rangle$
 - Set *depth* to *max (min)* values

← *z-Buffer* Algorithm → z

- 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

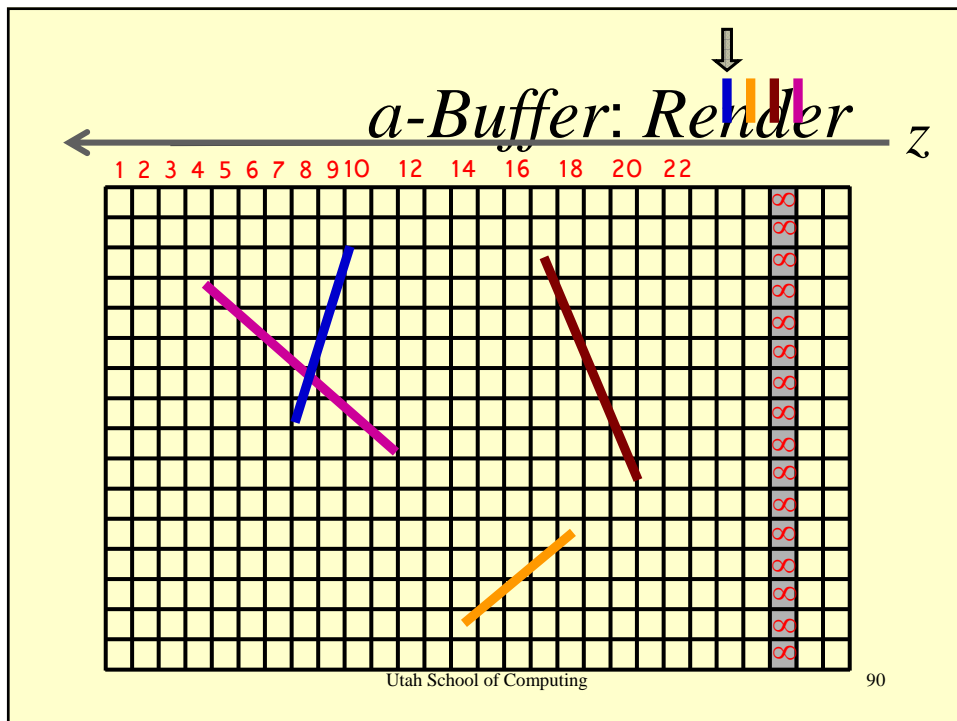
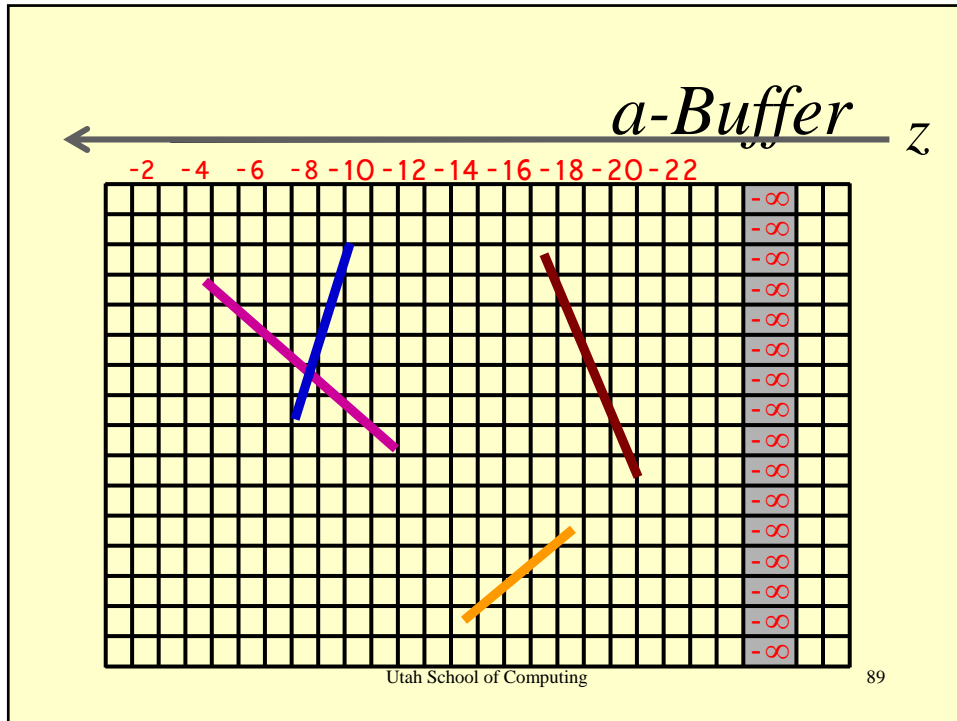
← *z-Buffer* Algorithm → z

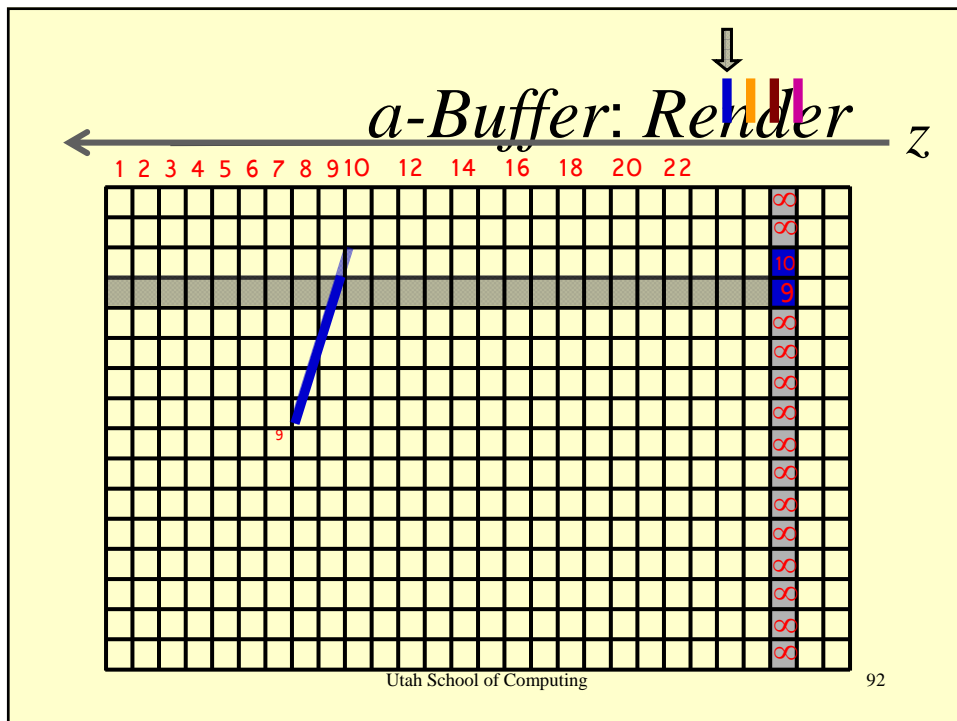
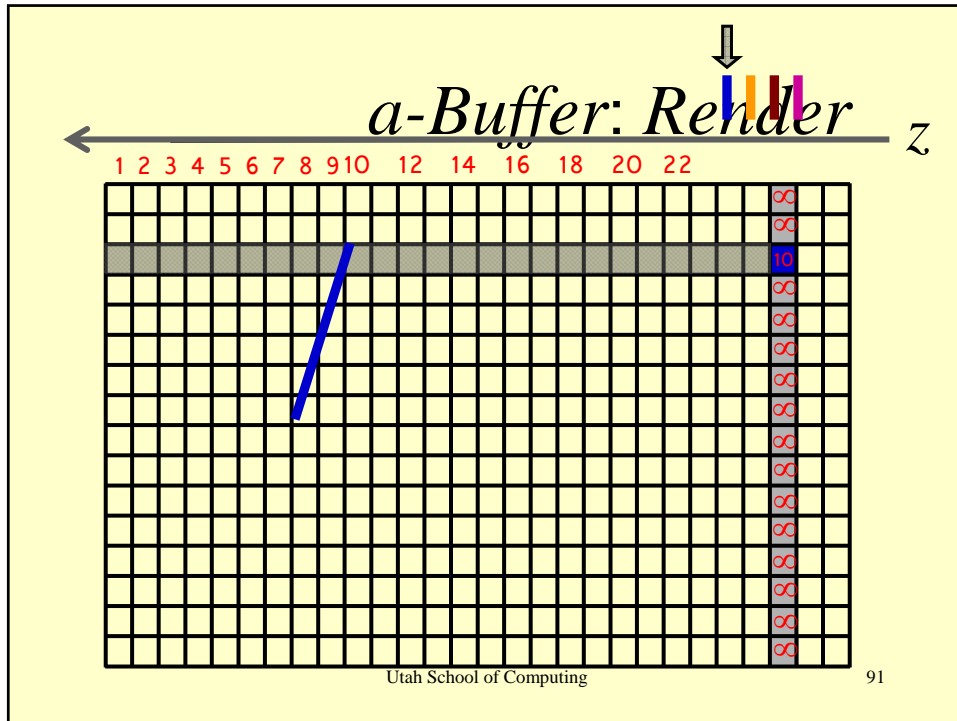
- *Convert all polygons*
- Correct image gets generated when *done*
- OpenGL: depth-buffer = z-Buffer

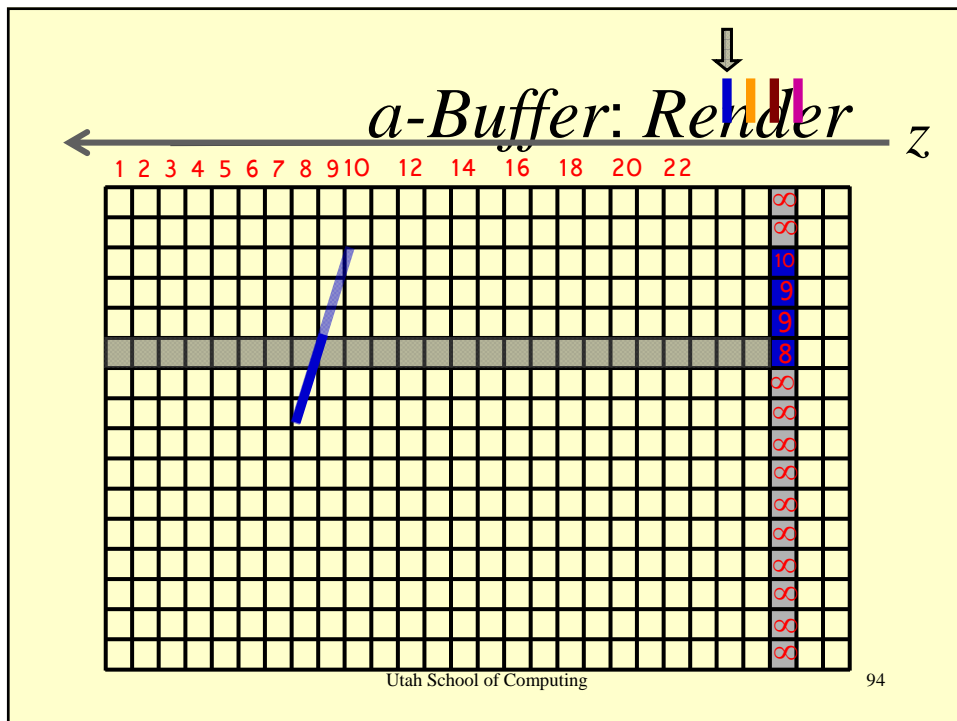
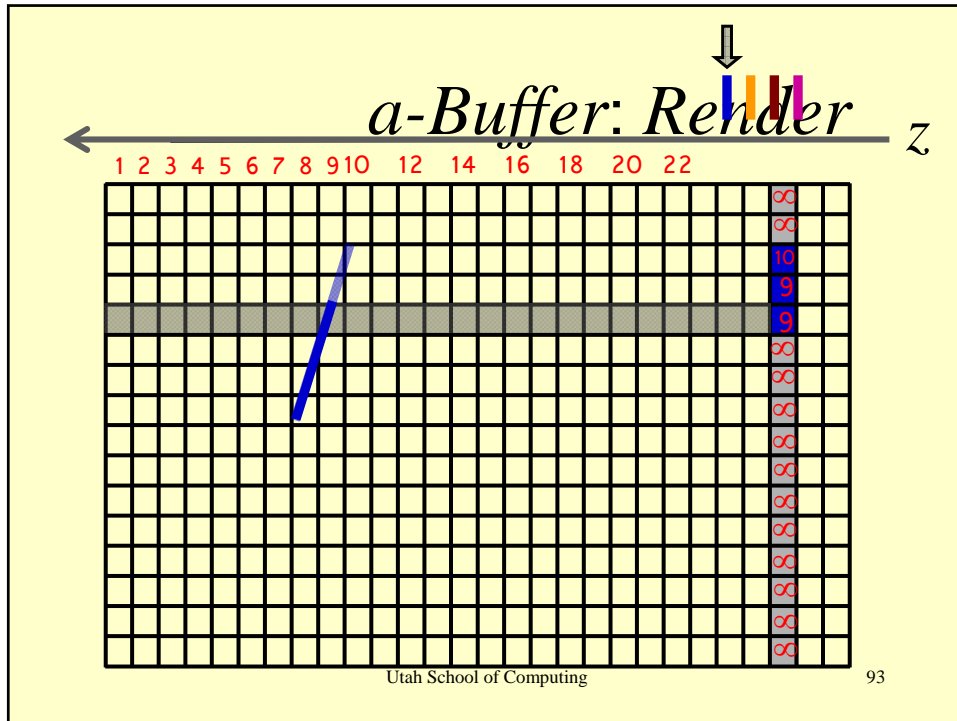
\leftarrow *a-Buffer* Algorithm \rightarrow z

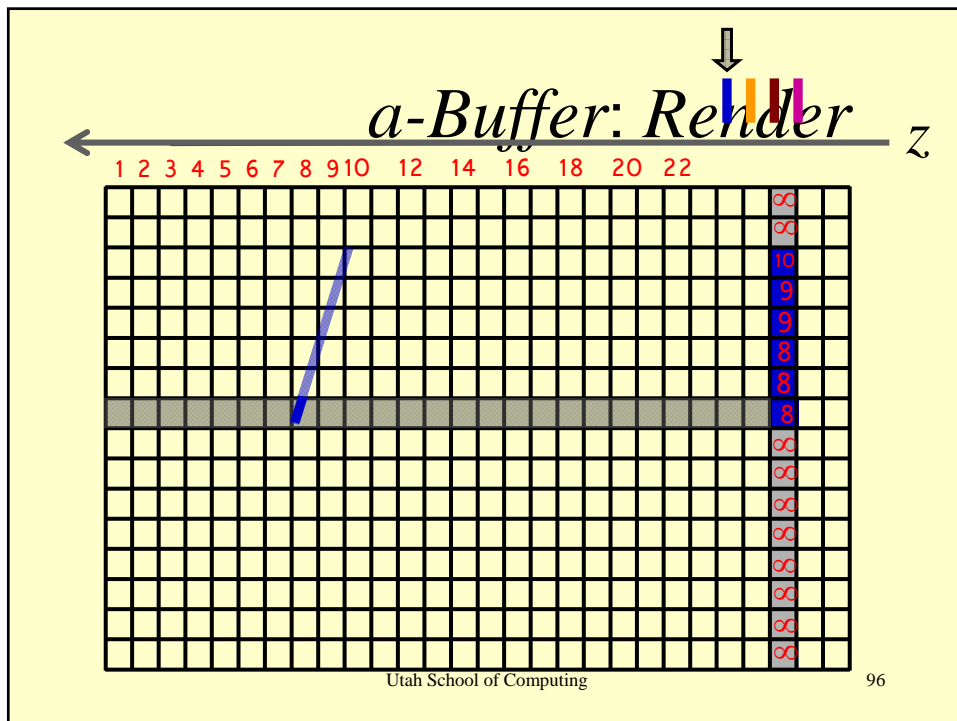
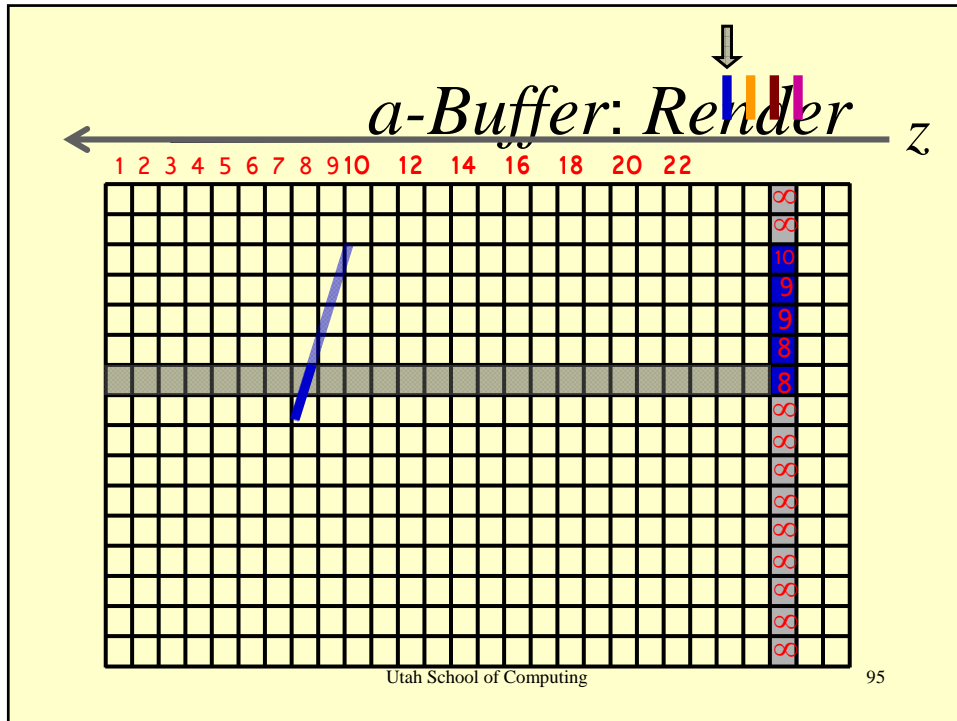
- Generates linked list for each pixel
- Memory of all contributions allows for proper handling of many advanced techniques
- Even *more* memory intensive
- Widely used for high quality rendering

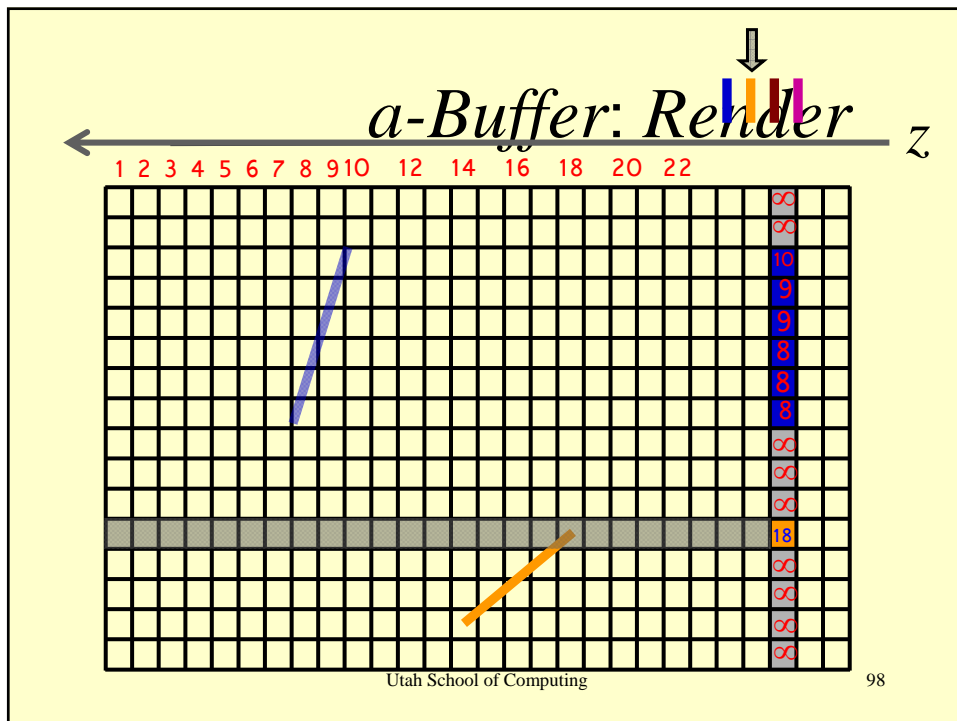
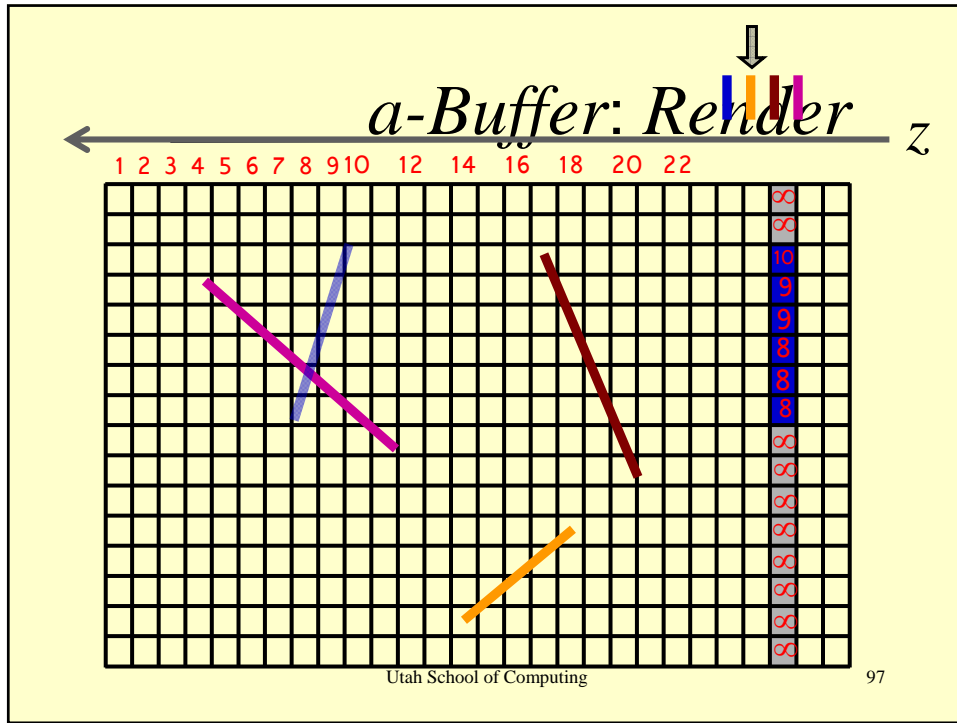
\leftarrow *a-Buffer* Algorithm: Example \rightarrow z

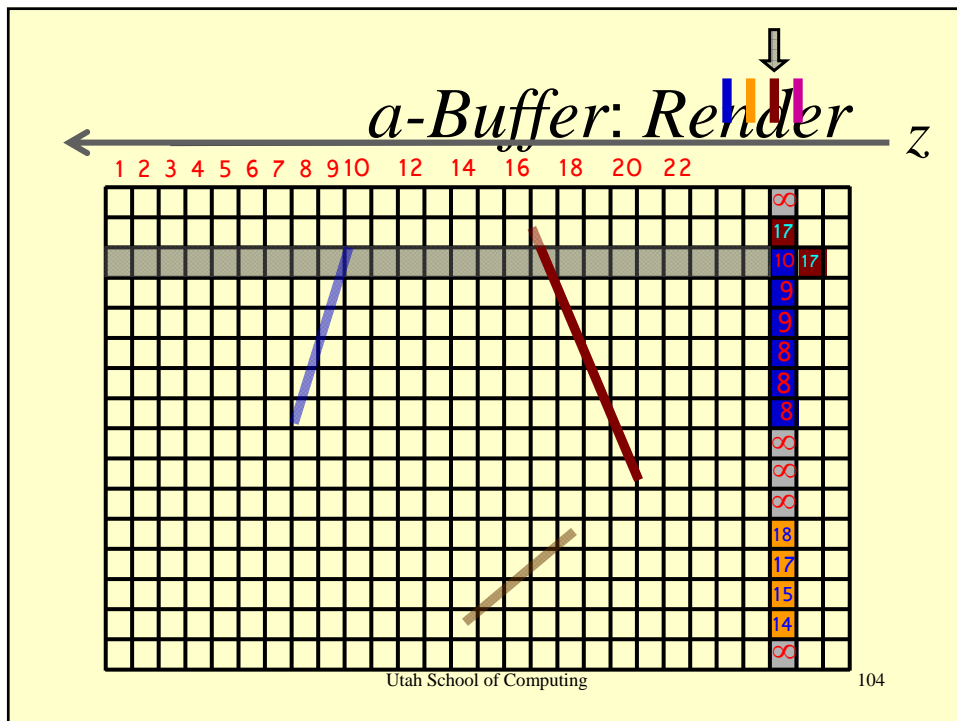
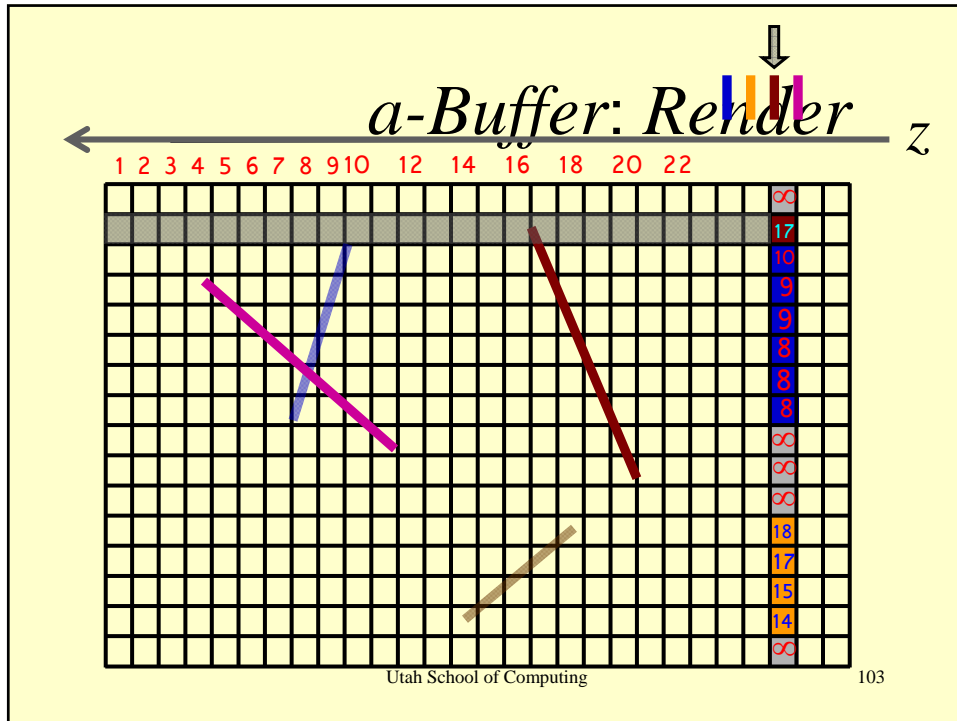


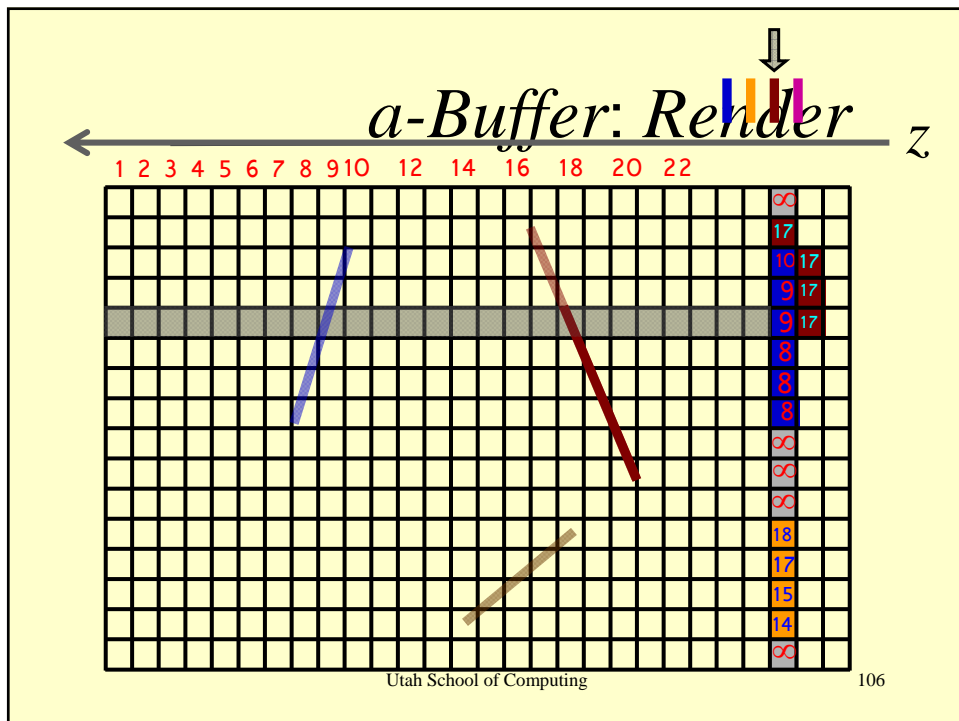
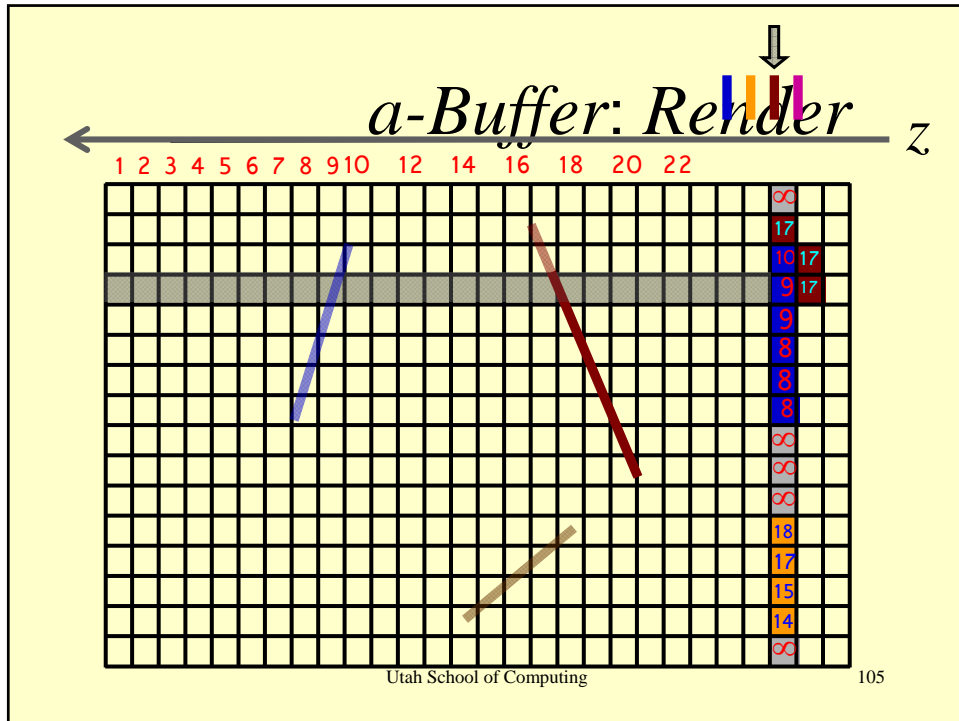


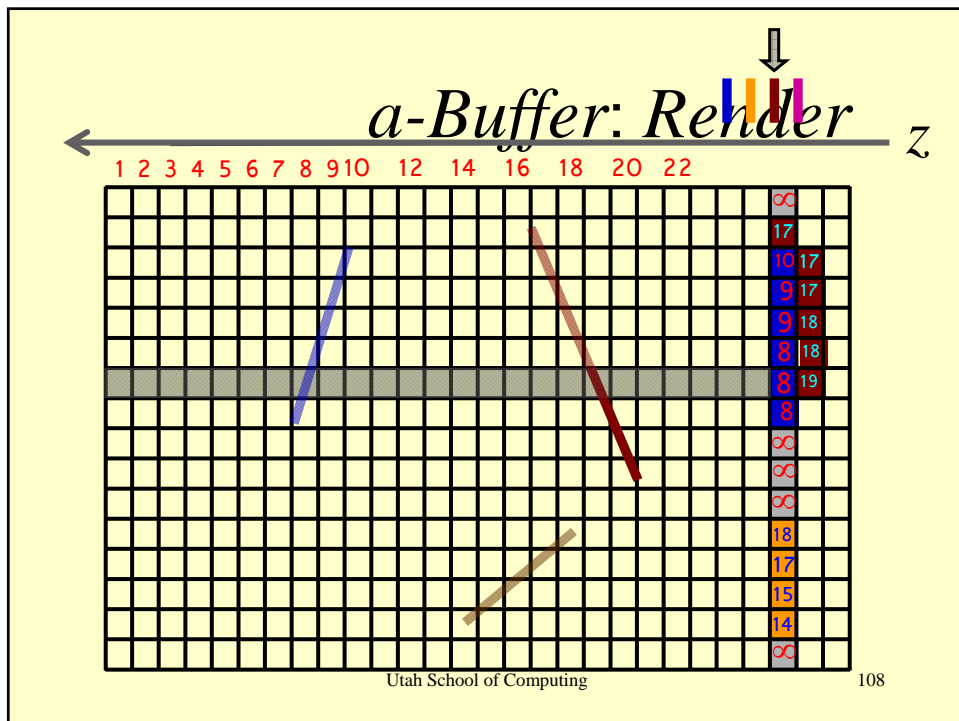
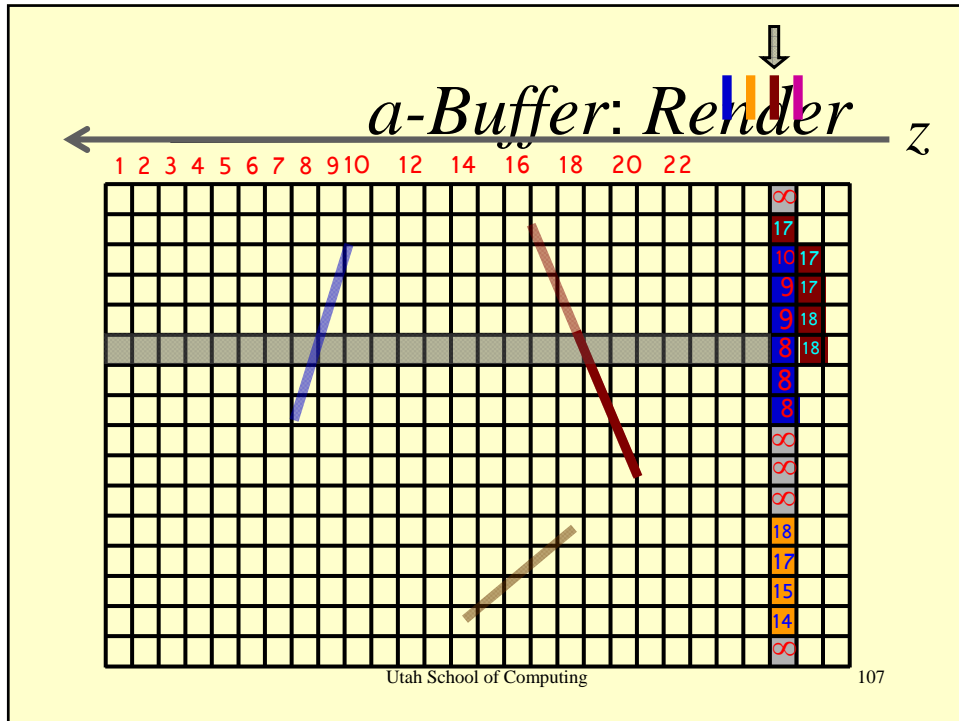


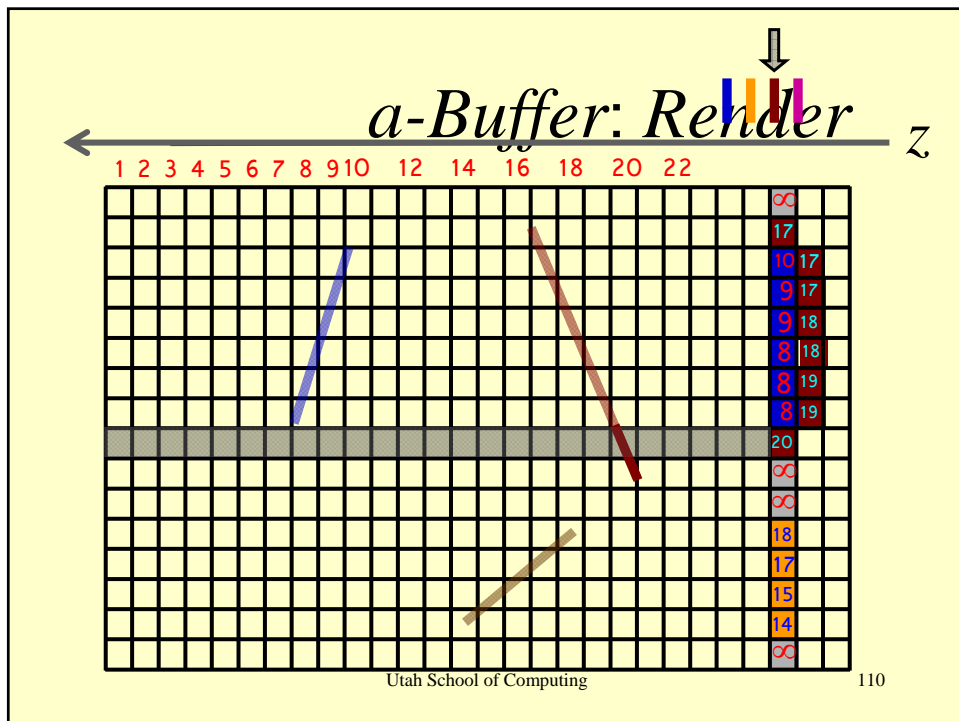
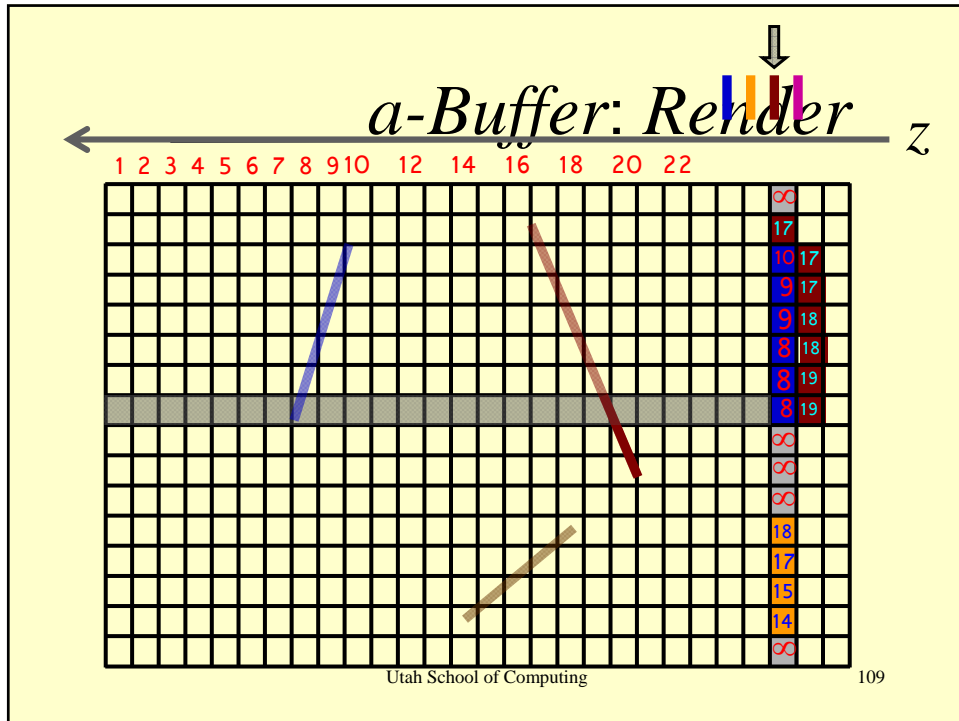


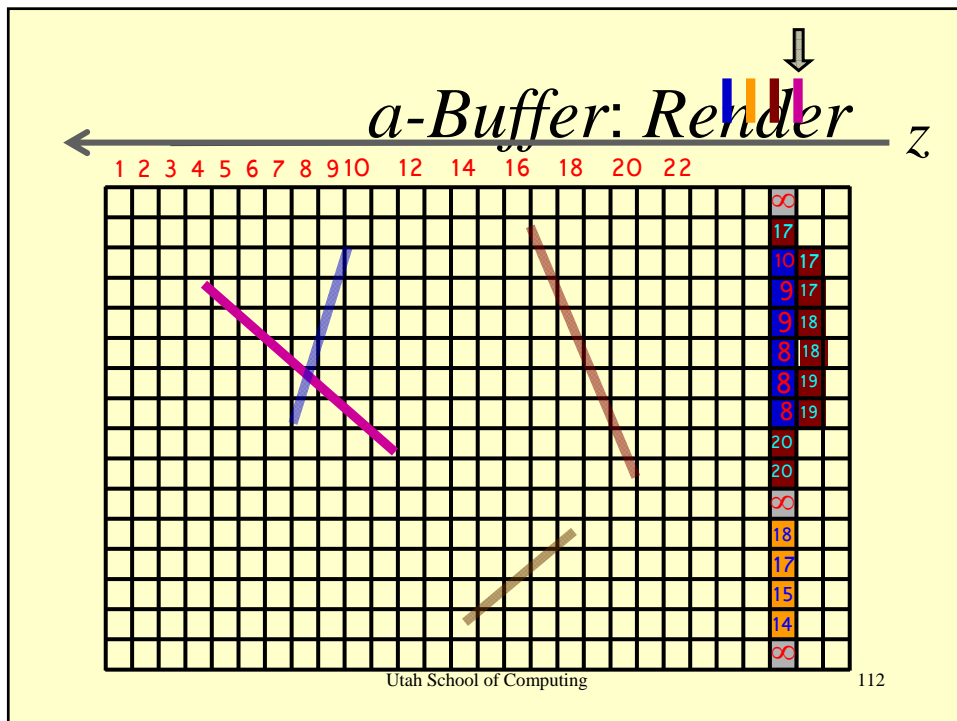
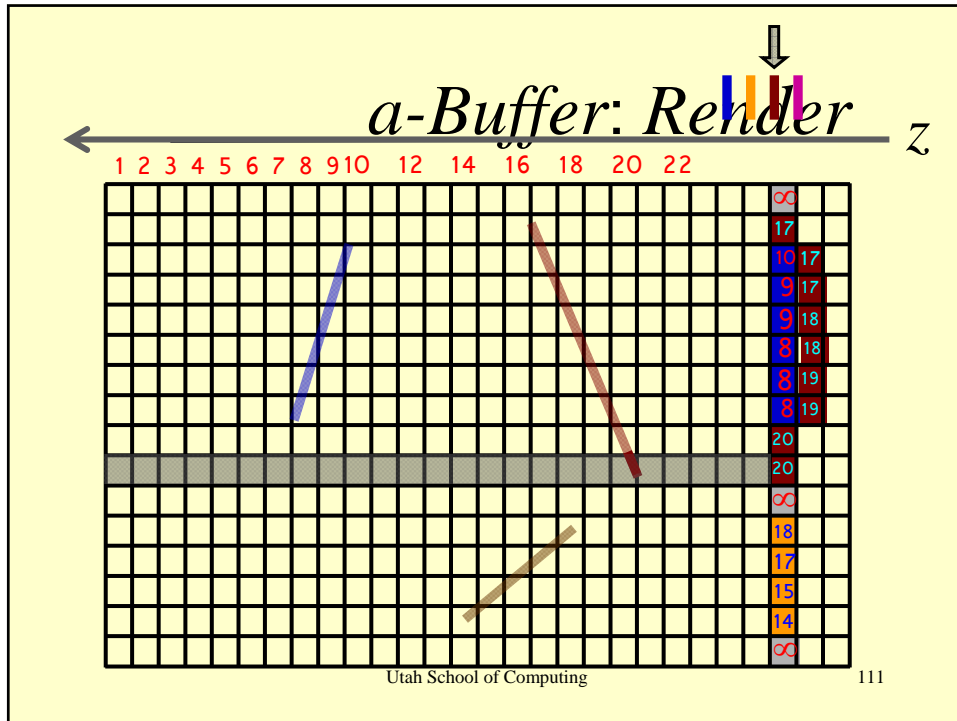


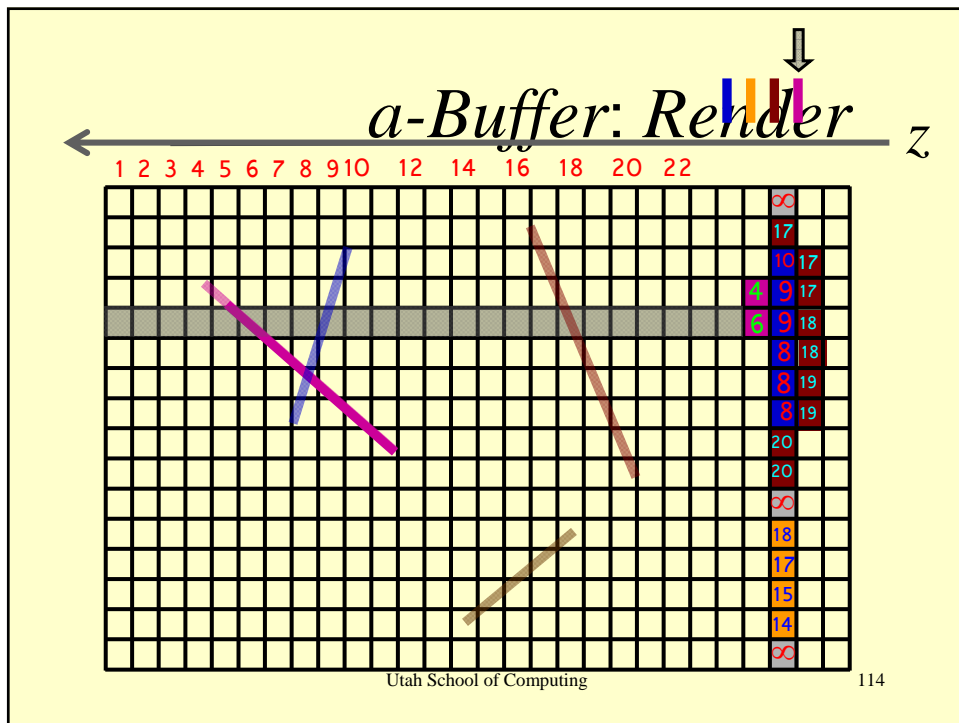
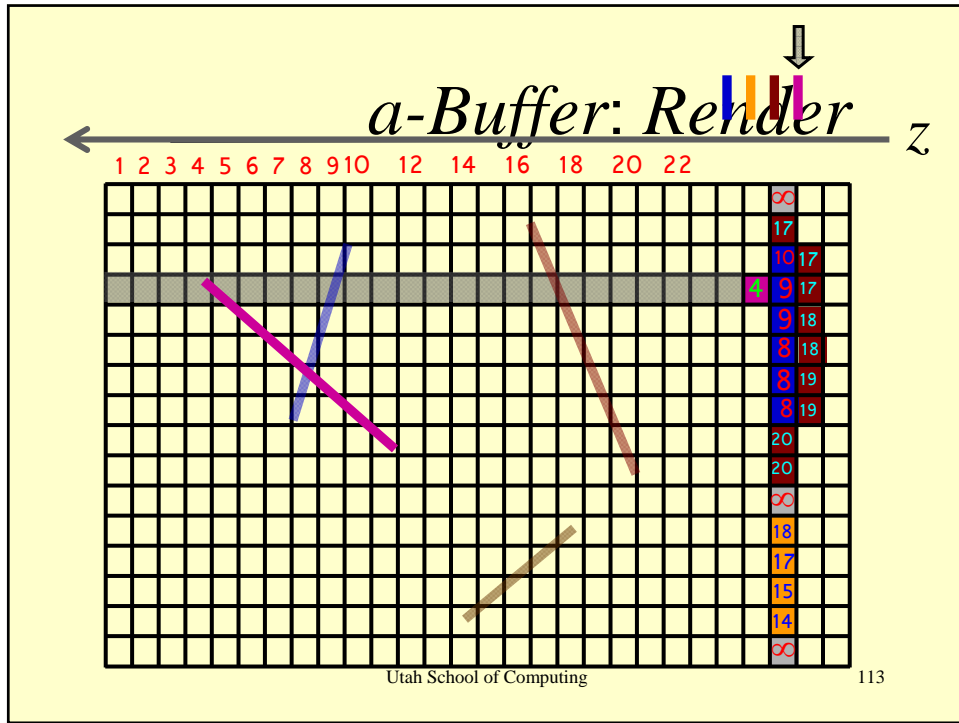


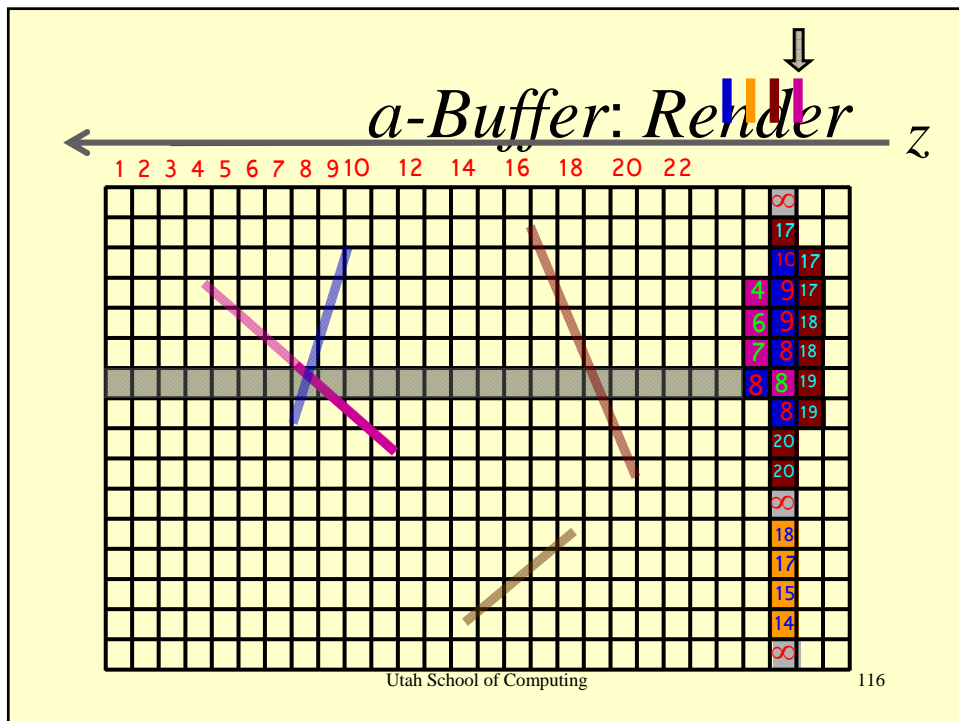
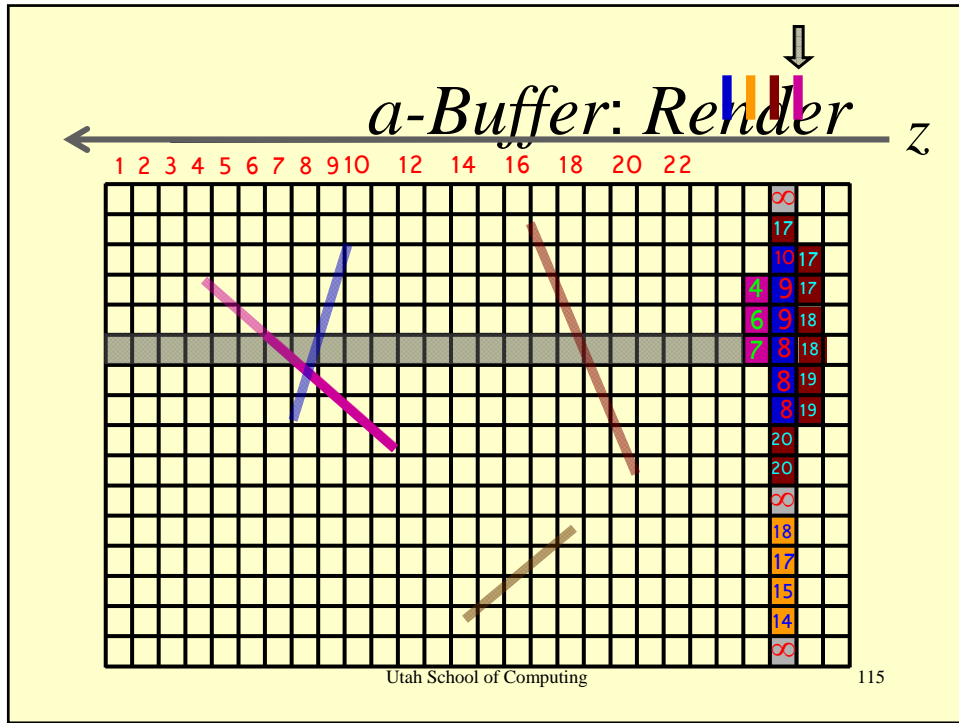


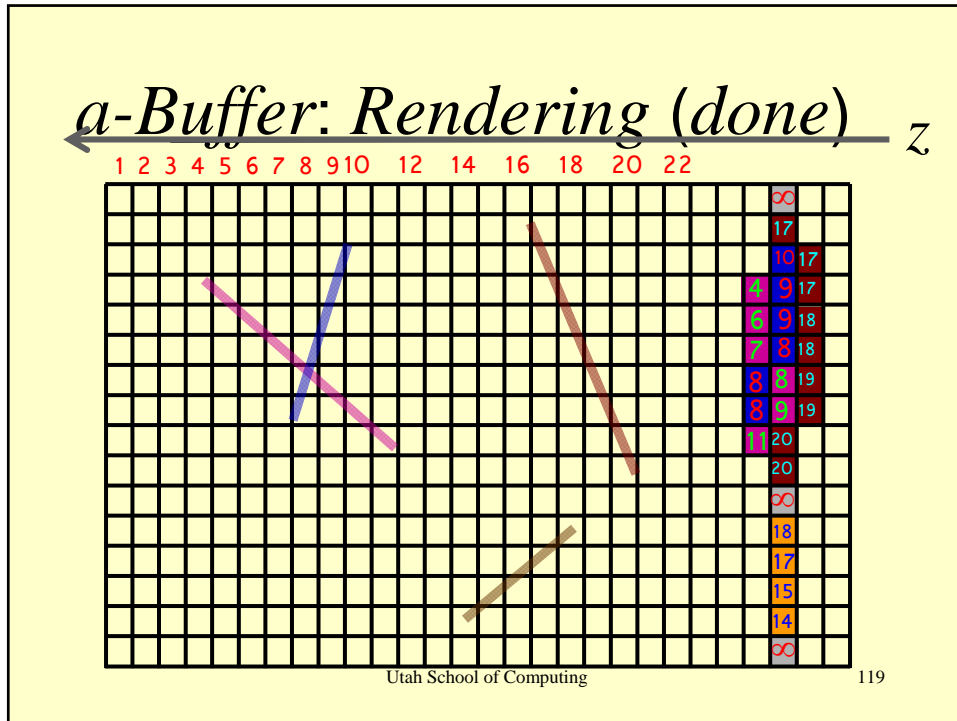




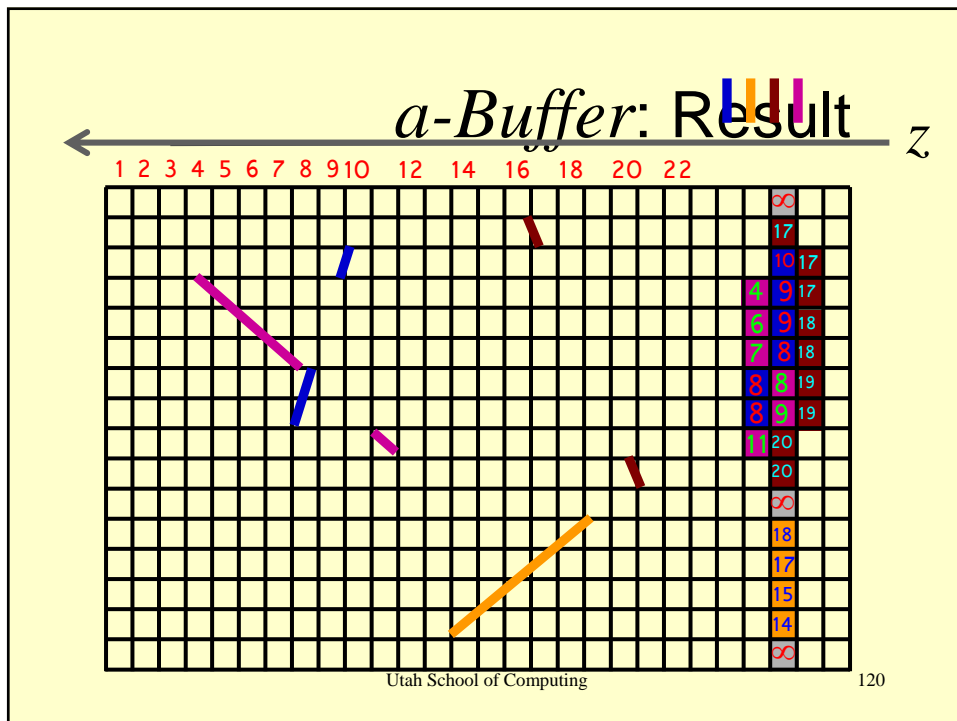








119



120

Lecture Set 11

The End
*Visible Surface
Determination*

121