3. 4.5.09
①

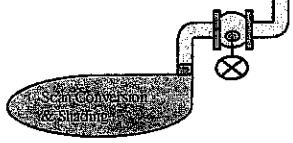
גרפיקה ממוחשבת - שיעור 7

קורס גרפיקה ממוחשבת

2009 סמסטר ב'
ליאור שפירא

3D Polygon Rendering Pipeline

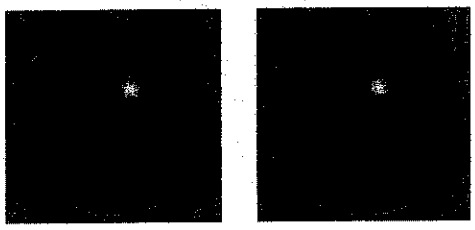
שיעור 8 חלק ראשון
השלמות...



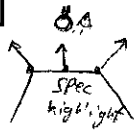
Thomas Funkhouser
Princeton University
COS 426, Fall 1999

Gouraud Shading

- Produces smoothly shaded polygonal mesh
  - Piecewise linear approximation
  - Need fine mesh to capture subtle lighting effects



Poor behavior of specular light | Same sphere – high polygon count

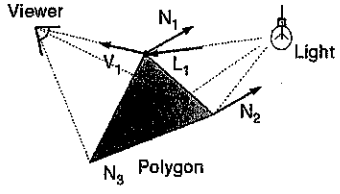


Polygon Shading Algorithms

- Flat Shading
- Gouraud Shading
- **Phong Shading**

Phong Shading

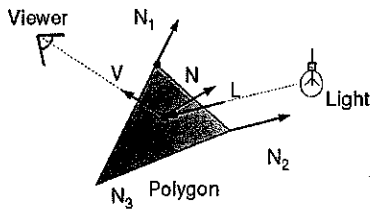
- What if polygonal mesh is too coarse to capture illumination effects in polygon interiors?



$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

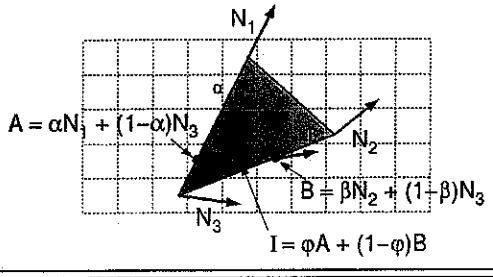
Phong Shading

- One lighting calculation per pixel
  - Approximate surface normals for points inside polygons by **bilinear interpolation of normals** from vertices

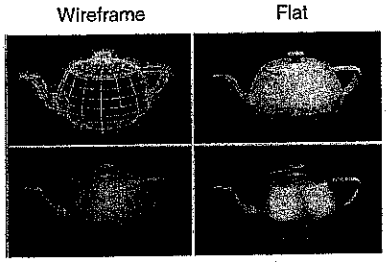


Phong Shading

- Bilinearly interpolate surface normals at vertices down and across scan lines



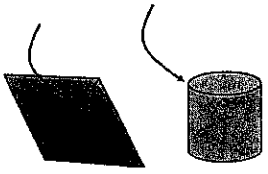
Polygon Shading Algorithms



Watt Plate 7

Shading Issues

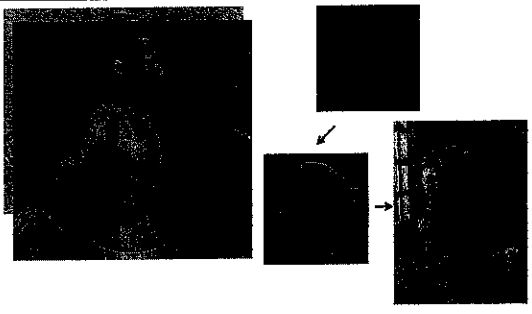
- Problems with interpolated shading:
  - Polygonal silhouettes
  - Perspective distortion
  - Orientation dependence (due to bilinear interpolation)
  - Problems at T-vertices
  - Problems computing shared vertex normals



Overview

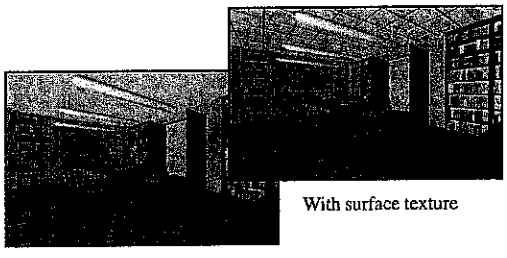
- Scan conversion
  - Figure out which pixels to fill
- Shading
  - Determine a color for each filled pixel
- Texture Mapping
  - Describe shading variation within polygon interiors
- Visible Surface Determination
  - Figure out which surface is front-most at every pixel

Surface Textures



Surface Textures

- Add visual detail to surfaces of 3D objects

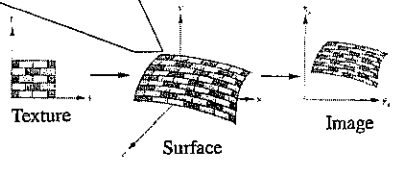


With surface texture

Polygonal model

Textures

- Describe color variation in interior of 3D polygon
  - When scan converting a polygon, vary pixel colors according to values fetched from a texture



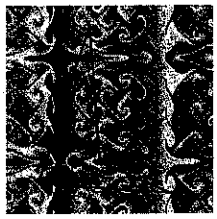
Angel Figure 9.3

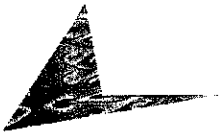
4.5.09

גרפיקה ממוחשבת - שיעור 7

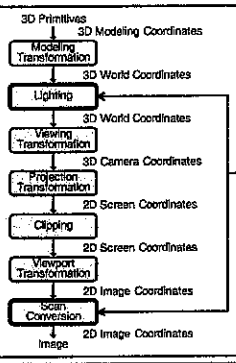
Textures

- We map coordinates in the 3D world to a **Texture**





3D Rendering Pipeline (for direct illumination)

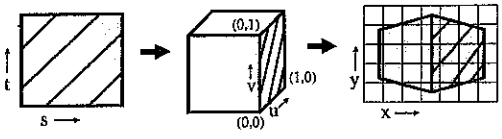


Overview

- Texture mapping methods
  - Mapping
  - Filtering
  - Parameterization
- Texture mapping applications
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Environment mapping
  - Image-based rendering
  - Non-photorealistic rendering

Texture Mapping

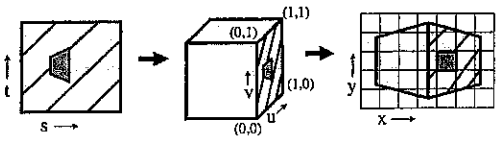
- Steps:
  - Define texture
  - Specify mapping from texture to surface
  - Lookup texture values during scan conversion



Texture Coordinate System | Modeling Coordinate System | Image Coordinate System

Texture Mapping

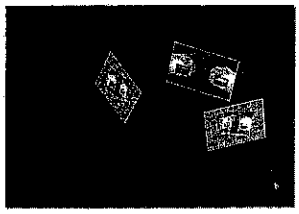
- When scan convert, map from …
  - image coordinate system (x,y) to
  - modeling coordinate system (u,v) to
  - texture image (t,s)



Texture Coordinate System | Modeling Coordinate System | Image Coordinate System

Texture Mapping

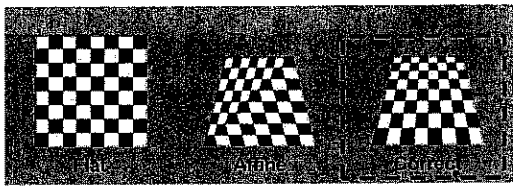
- Texture mapping is a 2D projective transformation
  - texture coordinate system: (t,s) to
  - image coordinate system (x,y)



Chris Buehler & Leonard McMillan, MIT

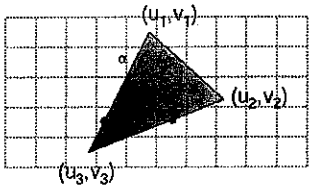
Texture Mapping

- Scan conversion
  - Interpolate texture coordinates down/across scan lines
  - Distortion due to bilinear interpolation approximation

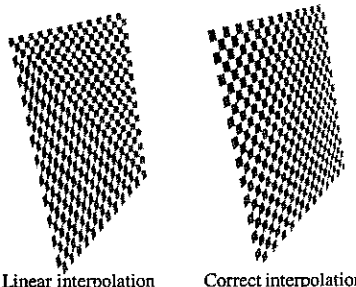


Texture Mapping

- Scan conversion
  - Interpolate texture coordinates down/across scan lines
  - Distortion due to bilinear interpolation approximation
    - Cut polygons into smaller ones, or
    - Perspective divide at each pixel



Texture Mapping



Linear interpolation of texture coordinates

Correct interpolation with perspective divide

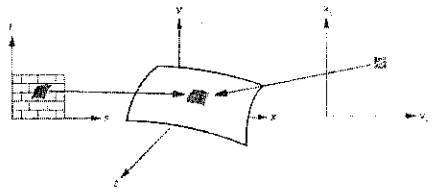
Hill Figure 8.42

Overview

- Texture mapping methods
  - Mapping
  - Filtering
  - Parameterization
- Texture mapping applications
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Environment mapping
  - Image-based rendering
  - Non-photorealistic rendering

Texture Filtering

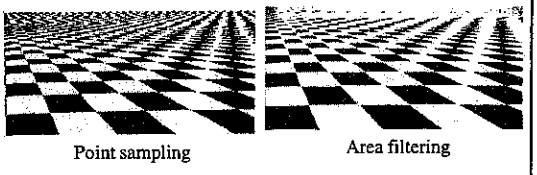
- Must sample texture to determine color at each pixel in image



Angel Figure 9.4

Texture Filtering

- Aliasing is a problem



Point sampling

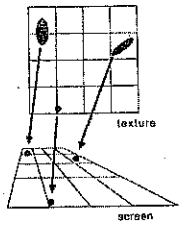
Area filtering

Angel Figure 9.5

4.5.09

Texture Filtering

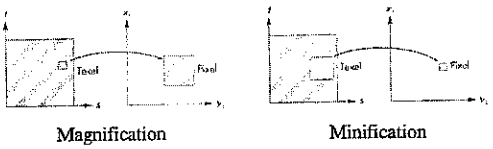
- Ideally, use elliptically shaped convolution filters



In practice, use rectangles

Texture Filtering

- Size of filter depends on projective warp
  - Can pre-filter images
    - Mipmapping
    - Summed area tables

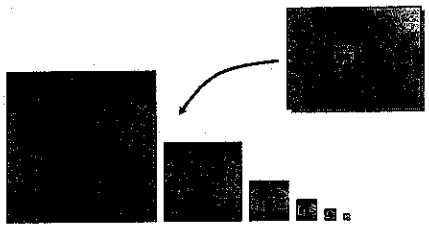


Magnification Minification

Angel Figure 9.14

Mipmapping

- Keep textures pre-filtered at multiple resolutions
  - How do we sample?
    - Nearest neighbor with Mipmapping
    - Bilinear filtering

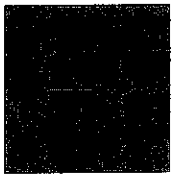


Mipmapping

- Keep textures pre-filtered at multiple resolutions
  - For each pixel, linearly interpolate between two closest levels (e.g., **tri-linear** filtering)
  - Fast, easy for hardware

Summed-area tables

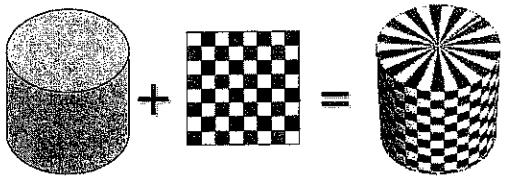
- At each Texel keep sum of all values down&right
  - We can compute sum of all values within a rectangle in constant time (four entries)
  - Better ability to capture very oblique projections
  - But, cannot store values in a single byte



Overview

- Texture mapping methods
  - Mapping
  - Filtering
  - Parameterization
- Texture mapping applications
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Environment mapping
  - Image-based rendering
  - Non-photorealistic rendering

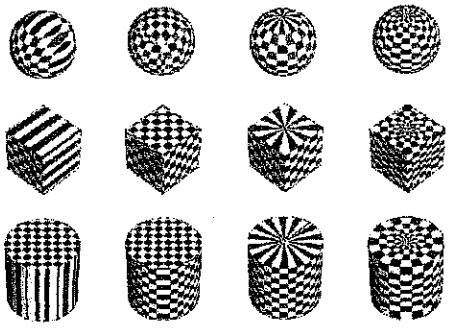
Parameterization



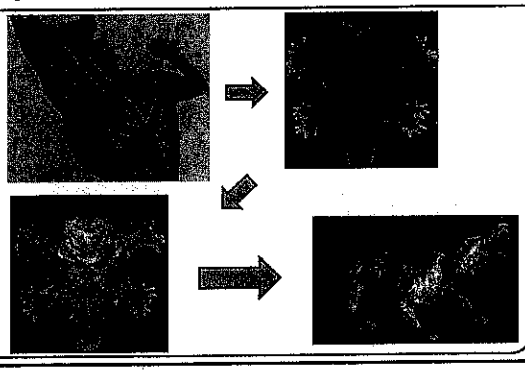
geometry image texture map

- Q: How do we decide *where* on the geometry each color from the image should go?

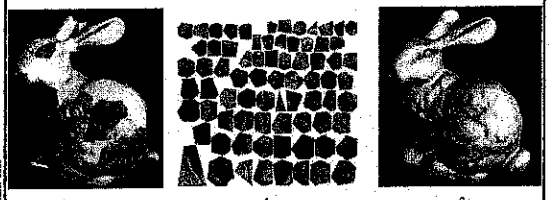
Varieties of projections



Option: unfold the surface



Option: Make an Atlas



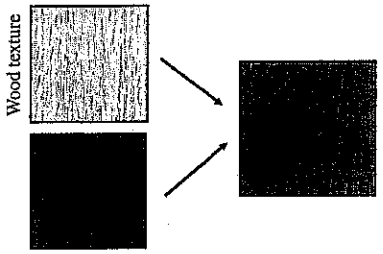
charts atlas surface

Overview

- Texture mapping methods
  - Mapping
  - Filtering
  - Parameterization
- Texture mapping applications
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Environment mapping
  - Image-based rendering
  - Non-photorealistic rendering

Modulation textures

- Map texture values to scale factor



$$I = T(s,t)(I_E + K_A I_A + \sum_L (K_D(N \bullet L) + K_S(V \bullet R)^n) S_L I_L + K_T I_T + K_S I_S)$$

4.5.09

Illumination Mapping

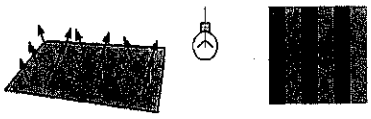
- Map texture values to surface material parameter
  - K_A
  - K_D
  - K_S
  - K_T
  - n

$K_T = T(s,t)$

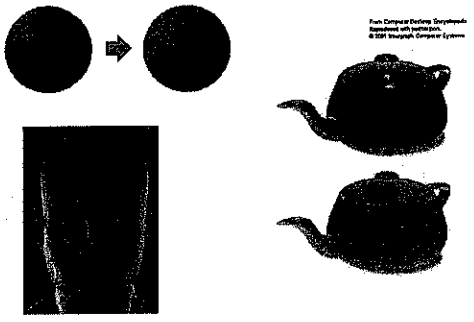
$$I = I_E + K_A I_A + \sum_L (K_D(N \bullet L) + K_S(V \bullet R)^n) S_L I_L + K_T I_T + K_S I_S$$

Bump Mapping

- Map texture values to perturbations of surface normals

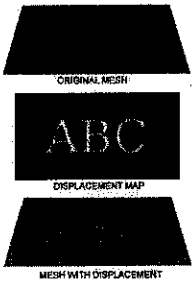


Bump Mapping



Displacement Mapping

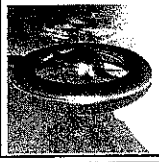
- Actually change geometric position of points over the textured surface
- Requires adaptive tessellation
- Until recently existed only in high-end rendering systems

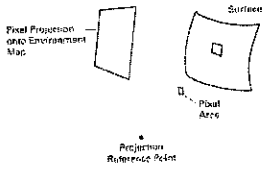


Tessellation = חלוקה

Environment/Reflection Mapping

- Map texture values to perturbations of surface normals

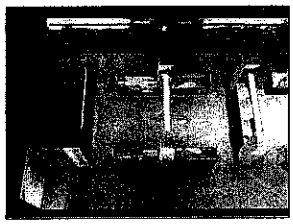




H&B Figure 14.93

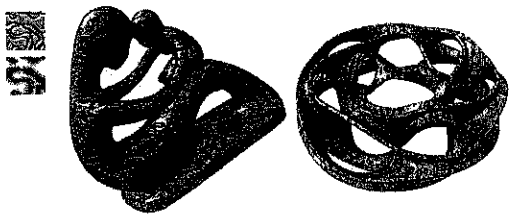
Image-Based Rendering

- Map photographic textures to provide details for coarsely detailed polygonal model



Solid Textures

- Texture values indexed by 3D location
  - Expensive storage, or
  - Compute on the fly, E.g Perlin noise

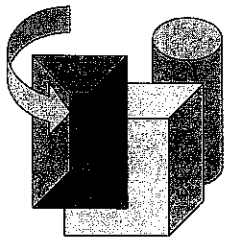


Solid Textures

- Texture values indexed by 3D location
  - Expensive storage, or
  - Compute on the fly, E.g Perlin noise

Visible Surface Detection (V.S.D)

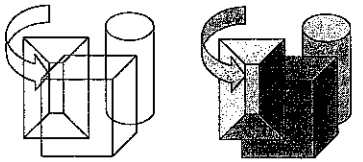
(Chapt. 15 in FVD, Chapt. 13 in Hearn & Baker)



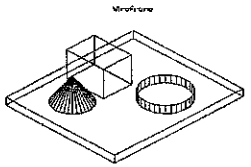
Overview

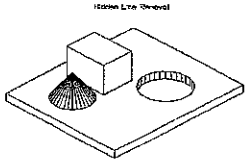
- Scan conversion
  - Figure out which pixels to fill
- Shading
  - Determine a color for each filled pixel
- Texture Mapping
  - Describe shading variation within polygon interiors
- Visible Surface Determination
  - Figure out which surface is front-most at every pixel

- Problem definition
  - Given a set of 3D objects and a viewing specifications, determine which lines or surfaces of the objects should be visible.
  - A surface might be occluded by other objects or by the same object (self occlusion)
- Two main approaches:
  - Image-precision algorithms: determine what is visible at each pixel.
  - Object-precision algorithms: determine which parts of each object are visible.

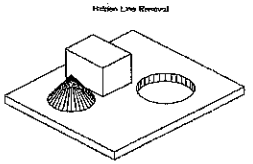


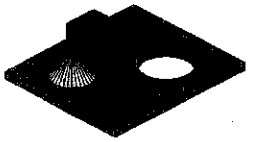
Hidden Lines Removal





Hidden Surfaces Removed



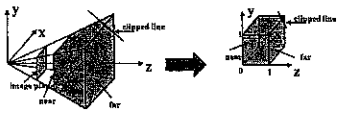


Coherence

- Most methods for V.S.D. use coherence features in the surface:
  - Object coherence.
  - Face coherence.
  - Edge coherence.
  - Scan-line coherence.
  - Depth coherence.
  - Frame coherence.

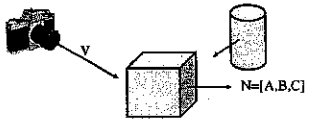
Where Are We ?

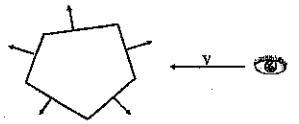
- Canonical view volume (3D image space)
- Clipping done
- division by w
- z > 0



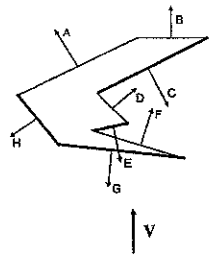
Back Face Detection

- **Observation**: In a volumetric object, you never see the "back" faces of the object (self occlusion).
- Reminder:
  - Plane equation: $Ax+By+Cz+D=0$
  - $N=[A,B,C]^T$ is the plane normal.
  - N points "outside".
- Back facing and front facing faces can be identified using the sign of $V \cdot N$



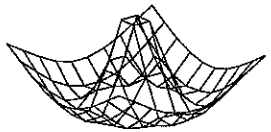


- Three possibilities:
  - $V \cdot N > 0$ back face
  - $V \cdot N < 0$ front face
  - $V \cdot N = 0$ on line of view
- Convex objects
  - For convex objects, *back face detection* actually solves the *visible surfaces problem.*
  - Back face detection is easily applied to convex polyhedral objects.
- In a general object, a front face can be visible, invisible, or partially visible.

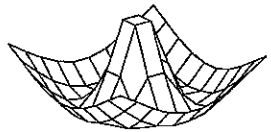


Back Face Polygons: A, B, D, F
Front Face Polygons: C, E, G, H

Single Valued Function of two variables



Without Hidden-Line Removal

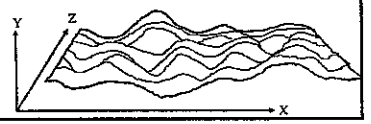


With Hidden-Line Removal

Floating Horizon Algorithm

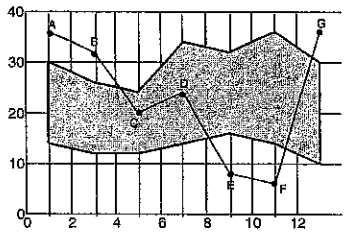
- Implicit Function: $Y=f(X,Z)$.
- Represent as 2D array of x and z values, each entry is the corresponding y-value.
- Surface = many polylines; Each polyline is constant in Z.

Algorithm:
Draw polylines of constant z from front (near z) to back (far z).
Draw only parts of polyline that are visible: ie above/below the silhouette (horizon).



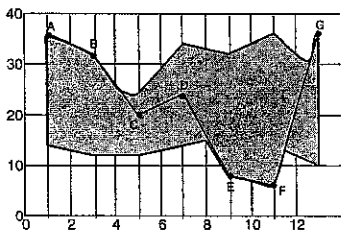
- Floating Horizon Characteristics:
  - Applied in image space (image precision).
  - Limited to explicit functions only.
  - Exploiting edge coherence.
  - Applicable for free-form surfaces.

Use 2 1D arrays YMIN and YMAX (with 1 entry for each x). When drawing a polyline of constant z, for each x-value, test if above/below YMAX/YMIN (at x location) and update arrays.

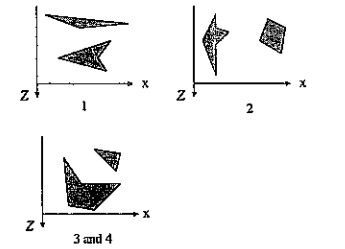


old YMAX	30	28	26	25	24	29	34	33	32	34	36	33	30
old YMIN	10	12	14	15	16	15	14	13	12	12	12	13	14
polyline	36	34	32	26	20	22	24	16	8	7	6	21	36
	A		B		C		D		E		F		G
new YMAX	36	34	32	26	24	29	34	33	32	34	36	33	36
new YMIN	10	12	14	15	16	15	14	13	8	7	6	13	14

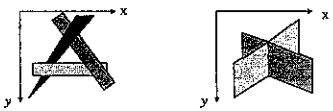
Use 2 1D arrays YMIN and YMAX (with 1 entry for each x). When drawing a polyline of constant z, for each x-value, test if above/below YMAX/YMIN (at x location) and update arrays.



old YMAX	30	28	26	25	24	29	34	33	32	34	36	33	30
old YMIN	10	12	14	15	16	15	14	13	12	12	12	13	14
polyline	36	34	32	26	20	22	24	16	8	7	6	21	36
	A		B		C		D		E		F		G
new YMAX	36	34	32	26	24	29	34	33	32	34	36	33	36
new YMIN	10	12	14	15	16	15	14	13	8	7	6	13	14

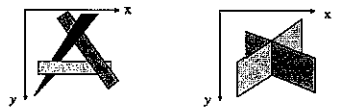


- If all the above conditions do not hold, P and Q may be split along intersection edge into two smaller polygons.



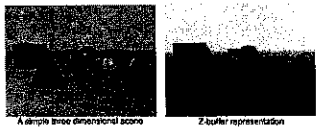
Depth Sort (Painter Algorithm)

- Sort all of the polygons in the scene by their depth.
- Draw them back to front.
- **Question**: Does a depth ordering always exist? Unfortunately, no.
  - For polygons with constant Z value, this sorting clearly works.
  - For example: window systems.



- **Question**: What if polygons are not Z constant?
- **Observation**: Given two polygons P and Q, an order may be determined between them, if at least one of the following holds:
  - 1. Z values of P and Q do not overlap.
  - 2. The bounding rectangle in the x,y plane for P and Q do not overlap.
  - 3. P is totally on one side of Q's plane.
  - 4. Q is totally on one side of P's plane.
  - 5. The bounding rectangles of Q and P do not intersect in the projection plane.

Z-buffer Method

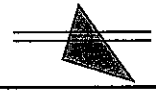


- In addition to the frame buffer (keeping the pixel values), keep a Z-buffer containing the depth value of each pixel.
- Surfaces are scan-converted in an arbitrary order. For each pixel (x,y), the Z-value is computed as well. The (x,y) pixel is overwritten only if its Z-values is closer to the viewing plane than the one already written at this location.

Algorithm:

- Initialize the z-buffer and the frame-buffer:
  depth(x,y)=MAX_Z ; I(x,y)=$I_{background}$
- Calculate the depth Z for each (x,y) position on any surface:
  - If z<*depth*(x,y), then set
    depth(x,y)=z ; I(x,y)=I_{surf}(x,y)

- For polygon surfaces, the depth-buffer method is very easy to implement using polygon scan line conversion, and exploiting face coherence and scan-line coherence :
  - Z = –(Ax+By+D)/C
  - Along scan lines
    Z'= –(A(x+1)+By+D)/C=Z–A/C
  - Between successive scan lines:
    Z'= –(Ax+B(y+1)+D)/C=Z–B/C



Z-buffer

▪Z-buffer is a 2D array that stores a depth value for each pixel.

▪***InitScreen***:

```
for i := 0 to N do
    for j := 1 to N do
  Screen[i][j] := BACKGROUND_COLOR;

  Zbuffer[i][j] := ∞;
```

▪***DrawZpixel*** (x, y, z, color)

```
if (z <= Zbuffer[x][y]) then
     Screen[x][y] := color; Zbuffer[x][y] := z;
```

Z-buffer: Scanline

I. for *each polygon* do
  for *each pixel* (x,y) *in the polygon's projection* do
    z := -(D+A*x+B*y)/C;
    DrawZpixel(x, y, z, polygon's color);

II. for *each scan-line y* do
  for *each "in range" polygon projection* do
    for each pair (x_1, x_2) of X-intersections do
      for x := x_1 to x_2 do
        z := -(D+A*x+B*y)/C;
        DrawZpixel(x, y, z, polygon's color);

If we know $z_{x,y}$ at (x,y) then: $z_{x+1,y} = z_{x,y} - A/C$

Incremental Scanline

$$Ax+By+Cz+D=0$$

$$z=\frac{-(Ax+By+D)}{C}, C\neq 0$$

On a scan line Y = *j*, a constant
Thus depth of pixel at $(x_1=x+\Delta x, j)$

$$z_1 - z = \frac{-(Ax_1+Bj+D)}{C} + \frac{(Ax+Bj+D)}{C}$$

$$z_1 - z = \frac{A(x-x_1)}{C}$$

$$z_1 = z - (\frac{A}{C})\Delta x \quad \text{, since } \Delta x = 1,$$

$$z_1 = z - \frac{A}{C}$$

Incremental Scanline (contd.)

- All that was about increment for pixels on each scanline.
- How about across scanlines for a given pixel ?
- Assumption: next scanline is within polygon

$$z_1 - z = \frac{-(Ax+By_1+D)}{C} + \frac{(Ax+By+D)}{C}$$

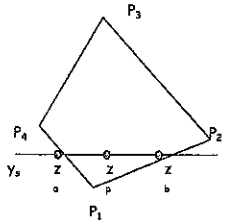
$$z_1 - z = \frac{A(y-y_1)}{C}$$

$$z_1 = z - (\frac{B}{C})\Delta y$$

, since $\Delta y = 1$,

$$z_1 = z - \frac{B}{C}$$

Non-Planar Polygons by bilinear interpolation

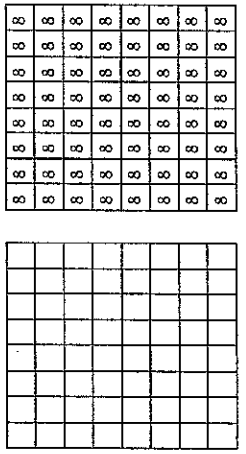


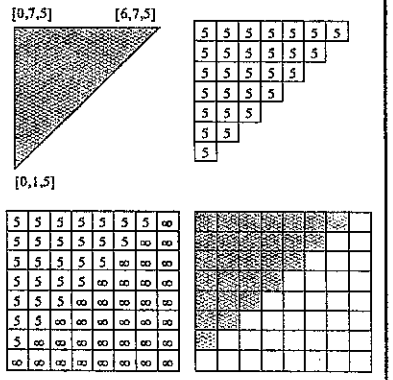
$$z_a = z_1 + (z_4 - z_1)\frac{(y_1 - y_s)}{(y_1 - y_4)}$$

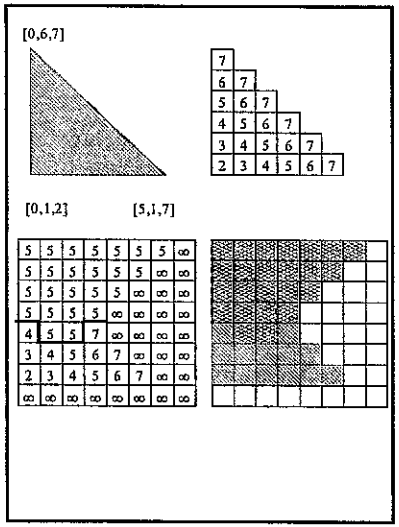
$$z_b = z_1 + (z_2 - z_1)\frac{(y_1 - y_s)}{(y_1 - y_2)}$$

$$z_p = z_a + (z_b - z_a)\frac{(x_a - x_p)}{(x_a - x_b)}$$

Z-buffer - Example







Penetrating Triangle

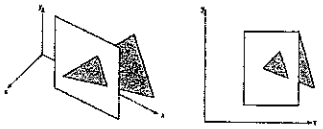
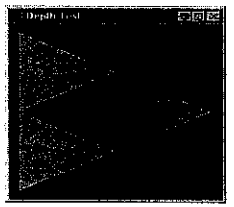


Figure 4-37 Penetrating triangle. (a) Three-dimensional view; (b) two-dimensional projection.



Z-Buffer Advantages

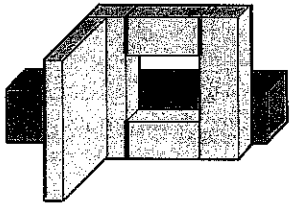
- Simple and easy to implement
- Amenable to scan-line algorithms
- Can easily resolve visibility cycles

Z-buffer Characteristics

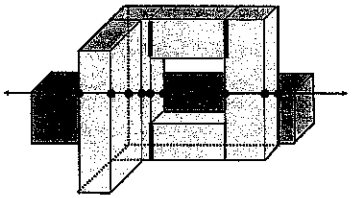
- Implemented in the image space.
- Very common in hardware due its simplicity (SGI's for example).
- 32 bits per pixel for Z is common.
- Advantages:
  - Simple and easy to implement.
- Disadvantages:
  - Requires a lot of memory.
  - Finite depth precision can cause problems.
  - Might spend a lot of time rendering polygons that are not visible.
  - Requires re-calculations when changing the objects scale.
  - Does not do transparency easily

Scan Line Algorithm

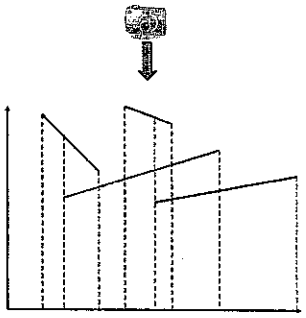
- An extension of the polygon scan conversion algorithm
- It uses the ET and AET, but for more than one polygon.
- The edge record has a link into a polygon table, which contains:
  - The plane equation (a,b,c,d)
  - The shading coefficients
  - A in/out bit

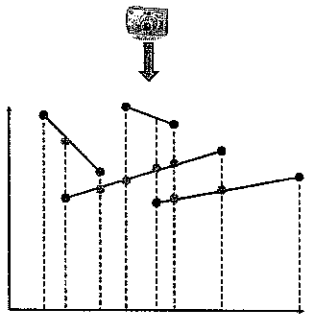


- The *active edges* are those that intersect the current horizontal slice.
- **Observations**: The visibility of an span can be changed only where it intersects an *active* edge .

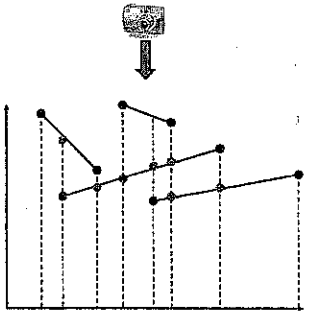


Active line segments produce span boundaries





- The span are used to subdivide the segments
- The span endpoints are *an event*



- In an event the closest segment is detected.
- **Question**: Among who?

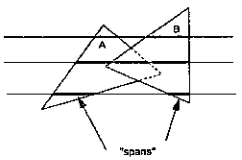
Spanning Scan-Line

Can we do better than scan-line Z-buffer ?

- Scan-line z-buffer does not exploit
  - Scan-line coherency across multiple scan-lines
  - Or span-coherence !
  - Depth coherency
- How do you deal with this – scan-conversion algorithm and a little more data structure

Scan Line Algorithm

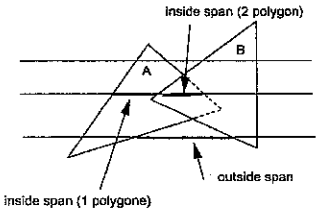
- Use no z-buffer
- Each scan line is subdivided into several "spans"
- Determine which polygon the current span belongs to
- Shade the span using the current polygon's color
- Exploit "span coherence" :
- For each span, only one visibility test needs to be done



Scan Line Algorithm

- A scan line is subdivided into a sequence of spans
- Each span can be "inside" or "outside" polygon areas
  - "outside": no pixels need to be drawn (background color)
  - "inside": can be inside one or multiple polygons
- If a span is inside one polygon, the pixels in the span will be drawn with the color of that polygon
- If a span is inside more than one polygon, then we need to compare the z values of those polygons at the scan line edge intersection point to determine the color of the pixel

Scan Line Algorithm



Determine a span is inside or outside (single polygon)

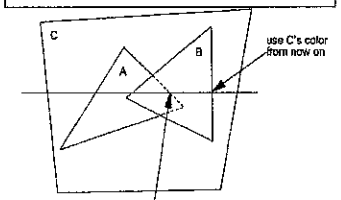
- When a scan line intersects an edge of a polygon
  - for a 1st time, the span becomes "inside" of the polygon from that intersection point on
  - for a 2nd time, the span becomes "outside" of the polygon from that point on
- Use a "in/out" flag for each polygon to keep track of the current state
- Initially, the in/out flag is set to be "outside" (value = 0 for example). Invert the tag for "inside".

When there are multiple polygon

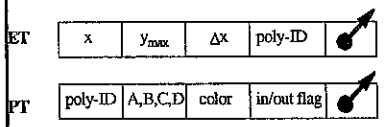
- Each polygon will have its own in/out flag
- There can be more than one polygon having the in/out flags to be "in" at a given instance
- We want to keep track of how many polygons the scan line is currently in
- If there are more than one polygon "in", we need to perform z value comparison to determine the color of the scan line span

Z value comparison

When the scan line is intersecting an edge and leaving a new polygon, we then use the color of the reminding polygon if there is now only 1 polygon "in". If there are still more than one polygon with "in" flag, we need to perform z comparison only when the scan line is leaving a non-obscured polygon

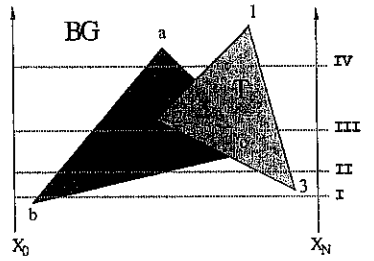


Many Polygons !



- Use a PT entry for each polygon
- When polygon is considered Flag is true !
- Multiple polygons can have flag set true !
- Use IPL as active In-Polygon List !

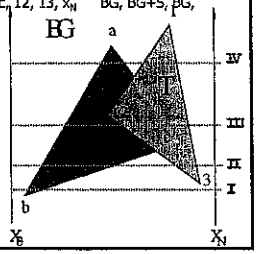
Example



Think of ScanPlanes to understand !

Spanning Scan-Line: Example

Y	AET	IPL
I	x_0, ba , bc, x_N	BG, BG+S, BG
II	x_0, ba , bc, 32, 13, x_N	BG, BG+S, BG, BG+T, BG
III	*x_0, ba , 32, ca, 13, x_N*	*BG, BG+S, BG+S+T, BG+T, BG*
IV	x_0, ba , ac, 12, 13, x_N	BG, BG+S, BG, BG+T, BG



Some Facts !

- Scan Line I: Polygon S is in and flag of S=true
- ScanLine II: Both S and T are in and flags are disjointly true
- Scan Line III: Both S and T are in simultaneously
- Scan Line IV: Same as Scan Line II

Spanning Scan-Line

```
build ET, PT          -- all polys+BG poly.
AET := IPL := Nil;
for y := y_min to y_max do
                          e1 := first_item (
AET );IPL := BG;
        while (e1.x <> MaxX) do
                e2 := next_item (AET);
                poly := closest poly in IPL at
[(e1.x+e2.x)/2, y]
                draw_line(e1.x, e2.x, poly-color);
                update IPL (flags); e1 := e2;
        end-while;
        IPL := NIL; update AET;
end-for;
```

Depth Coherence !

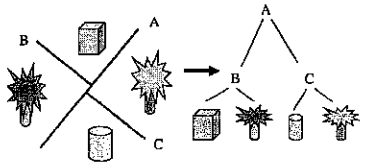
Depth relationships may not change between polygons from one scan-line to scan-line

These can be kept track using the AET and PT

How about penetrating polygons

The BSP Tree

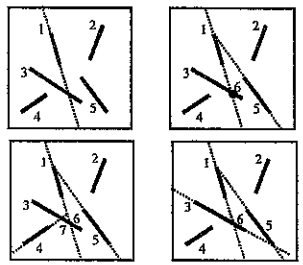
- BSP = Binary Space Partitioning.
- Interior nodes correspond to partitioning planes.
- Leaf nodes correspond to convex regions of space.



- Tests 3 and 4 in *Depth Sort* technique can be exploited efficiently:
- Let L_p be the plane P lies in: The 3D space may be divided into the following three groups:
  - Polygons in front of L_p.
  - Polygons behind L_p.
  - Polygons intersecting L_p.
- Polygons in the third class are split, and classified into the first two.
- As a result of the subdivision with respect to L_p:
  - The polygons behind L_p cannot obscure P, so we can draw them first.
  - P cannot obscure the polygons in front of L_p so we can draw P second.
  - Finally we draw the polygons in front of P.

The BSP-Tree Algorithm

- Construct a BSP tree:
  - Pick a polygon, let its supporting plane be the root of the tree.
  - Create two lists of polygons: these in front, and those behind (splitting polygons as necessary).
  - Recurse on the two lists to create the two sub-trees.
- Display:
  - Traverse the BSP tree back to front, drawing polygons in the order they are encountered in the traversal.



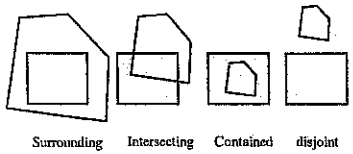
Should be prepared from the beginning !

BSP Properties:

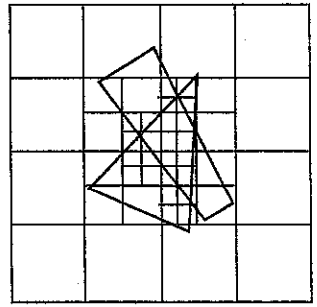
- The BSP tree is *view independent*!
- The BSP tree is constructed using the geometry of the object only.
- The tree can be used for hidden surface removal at an arbitrary direction.
- BSP = Object-precision alg.

Area Subdivision Technique (Warnock 1969)

- Subdivide screen area recursively, until visible surfaces are easy to determine.
- Each polygon has one of four relationships to the area of interest:



- If all polygons are disjoint from the area, fill area with background color.
- Only one intersecting or contained polygon: First fill with background color, then scan convert polygon.
- Only one surrounding polygon: Fill area with polygon's color.
- More than one polygon is surrounding, intersecting, or contained, but one surrounding polygon is in front of the rest: Fill area with polygon'c color.
- If none of the above cases occurs: Subdivide area into four, and recurse.
- Area subdivision = Image precision technique.



When the resolution of the image is reached, polygons are sorted by their Z-values at the center of the pixel, and the color of the closest polygon is used.