Prof. Reuven Aviv
Department of Computer Science
Tel Hai Academic College

Computer Graphics

Visual Realism Techniques

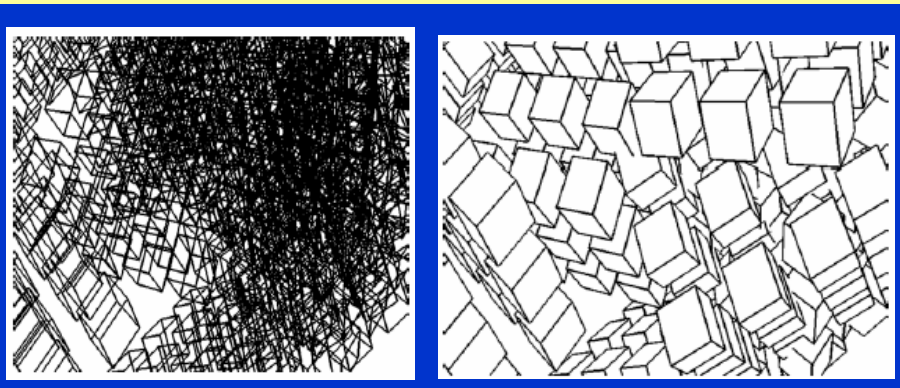
Slides adapted from F. Hill, S. Kelley Computer Graphics

Visual Realism Requirements

- Light Sources
- Materials (e.g., plastic, metal)
- Shading Models
- Depth Buffer Hidden Surface Removal
- Textures
- The Graphics pipeline revisited

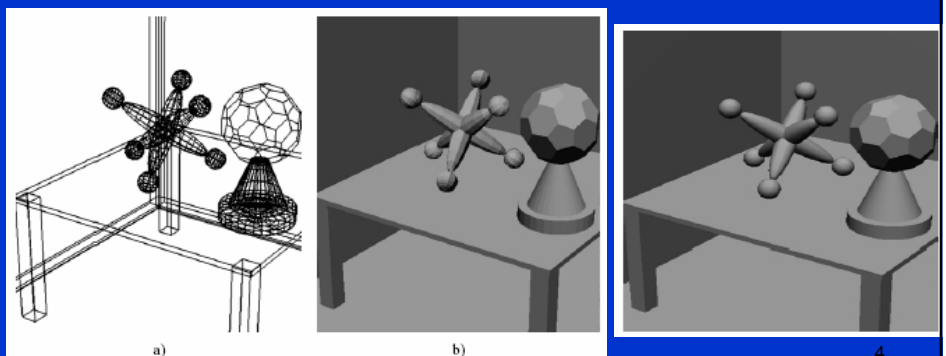
Rendering

- Rendering: deciding how a pixel should look
- Example: compare wireframe (left) to wire-frame with hidden surface removal (right)



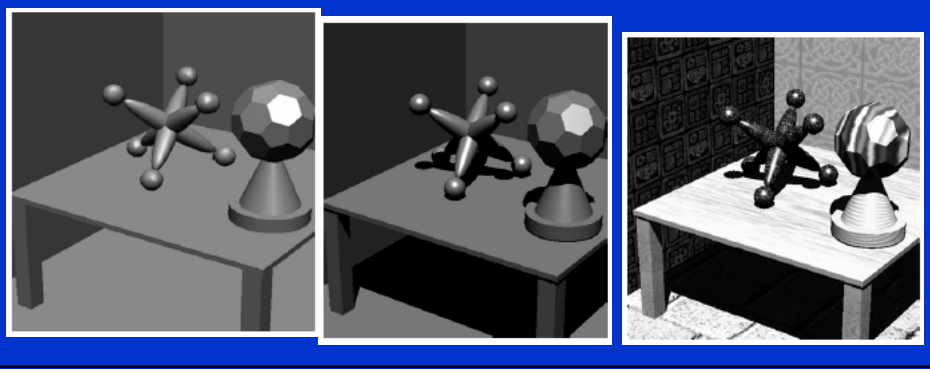
Rendering (2)

- wire-frame (left)
- flat shading (middle)
- smooth (Gouraud) shading (right)



Rendering (3)

- specular highlights
- Shadows
- textures



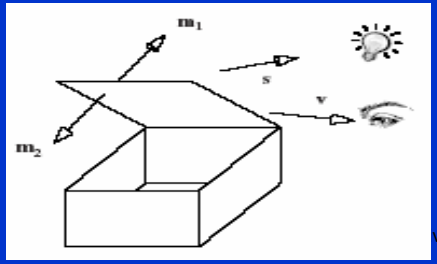
Shading: Determination of the color of every point in the model

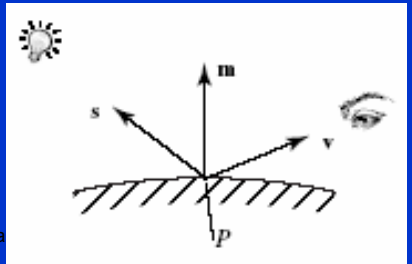
Shading Models: Introduction

- Basic Modeling Assumptions (for now):
- 1. light has no color, R = G = B; it has intensity (brightness)
- 2. point source of light (we'll add another source later)
- *what happens when light hits an object?*
- absorbed, reflected, refracted
- *what's the color of an object absorbs all the light?*
- reflected light that reaches the eye is the sum of
  - **Diffuse** reflection: reflected from inside, uniformly in all directions. *in what color?*
  - **Specular** (miror like) reflection: reflected from surface, approximately the same color as source. Shiny surface

Reflected Light

- Light from Source, reflected at Point **P**, reaches **eye**
- Depends on 2 directions from **P**: **s** to **Source**, **v** to **Eye**
  - Intensity depends on (**v**·**m**), (**s**·**m**) **m** normal at **P**
  - (**v**·**m**), (**s**·**m**) must be positive, else reflected is 0
- Normally eye sees outside surface, not always

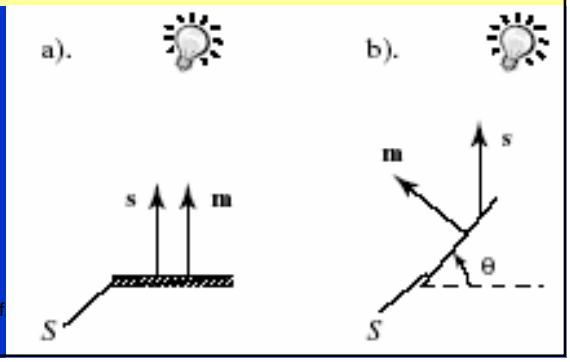




Diffuse Light (reflected in all directions)

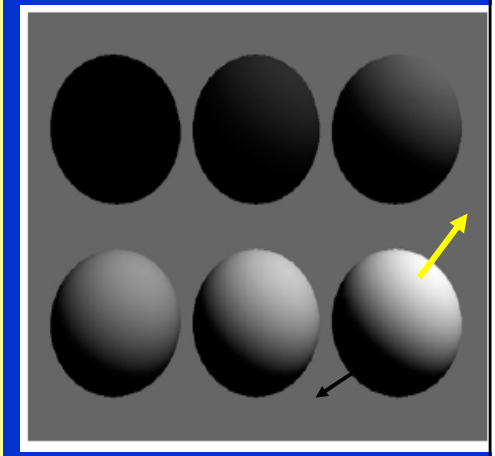
- I_d assumed to be independent of eye direction **v**
- I_d proportional to cos of the angle between **s** and **m**
  - Lambert Law: $I_d = I_s \rho_d (\mathbf{s} \cdot \mathbf{m})/(|\mathbf{s}||\mathbf{m}|)$ *why?*
- ρ_d diffuse reflection coefficient of material

- $\mathbf{s} \cdot \mathbf{m} < 0 \rightarrow I_d = 0$.
- $I_d = I_s \rho_d$ max $[(\mathbf{s} \cdot \mathbf{m})/(|\mathbf{s}||\mathbf{m}|), 0]$.
- Note: distance from source ignored!



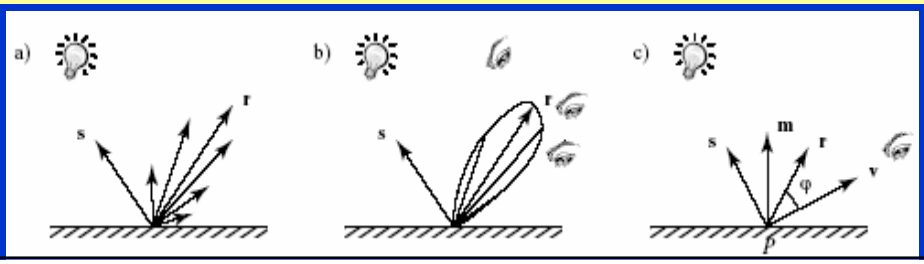
Example: Spheres modeled with Diffuse Light.

- $\rho_d = 0$ to 1 by 0.2 from top left to bottom right
- Source intensity $I_s = 1.0$
- Background ρ_d 0.4.
- Top left sphere $\rho_d = 0.0$
  - $I_d = 0$ (Black)
- bottom half of all spheres:
  - $(\mathbf{s} \cdot \mathbf{m}) < 0$)
  - Also $I_d = 0$ (black)



Specular Light

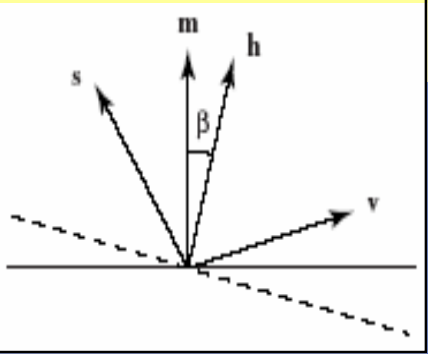
- light reflects at all angles, highest intensity at direction **r**
  - max at **(r·m) = (s·m)** equal incident & reflection angles
- I_{sp} decreases as φ between **r** and **v** increases.
- Phong Model: intensity decreases as $\cos^f \varphi$ f = 1…200
- $\cos \varphi = \mathbf{r}\cdot\mathbf{v}/(|\mathbf{r}||\mathbf{v}|)$ require **(r·v) > 0**
  - $I_{sp} = I_s \rho_s (\mathbf{r}\cdot\mathbf{v}/(|\mathbf{r}||\mathbf{v}|))^f$ *what's the effect of large f*



Speeding up Calculations for Specular Light

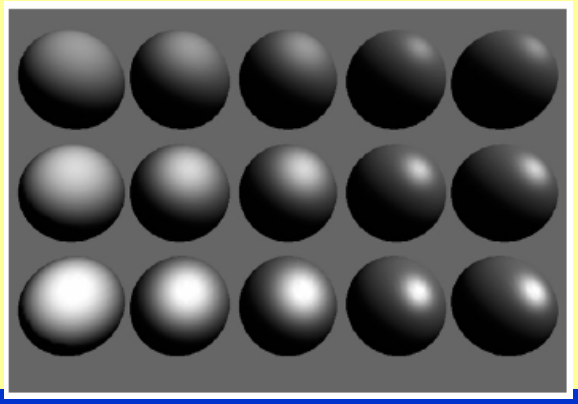
- Need to calculate $\mathbf{r} = -\mathbf{s} + 2\,\mathbf{m}\,(\mathbf{s}\cdot\mathbf{m})/(|\mathbf{m}|^2)$ (lecture 2)
  - For each point on each surface. Lengthy calculation
- Alternative: use the halfway vector **h** = **s** + **v**.
- Replace φ by β (they are not the same)

$I_{sp} = I_s \rho_s \max[(\mathbf{h}\cdot\mathbf{m}/(|\mathbf{h}||\mathbf{m}|))^f, 0]$



Example Reflected light

- From top to bottom, $\rho_s = 0.25, 0.5, 0.75$
- From left to right, f = 3, 6, 9, 25, 200.
- $\rho_a = 0.1$
- $\rho_d = 0.4$

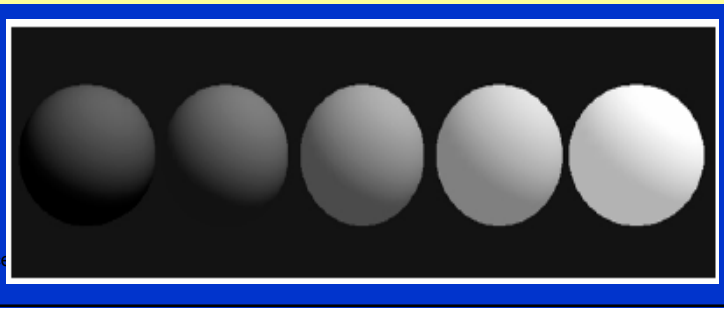


Ambient Light

- Point source light is not sufficient
  - E.g., generated shadows unrealistically sharp
- Scenes always bathed in some non-directional light.
  - Generated by multiple reflections & many light source
- We model this by **ambient light,** not situated at any particular place, spreads in all directions uniformly
- The source is assigned an intensity, I_a.
- Each point **P** at face reflects ambient light, $I_a \rho_a$
  - ρ_a ambient reflection coefficient of material at **P**
- The total reflected light reaching the eye is the sum of diffuse, specular and ambient light

Example

- Diffuse and ambient sources have intensities 1.0
- $\rho_d = 0.4$.
- $\rho_a = 0, 0.1, 0.3, 0.5, 0.7$ (left to right).
- Modest ambient light softens shadows; too much ambient light washes out shadows.



Combining Light Contributions; Color

- $I = I_a \rho_a + I_d \rho_d$ x *lambert* + $I_{sp} \rho_s$ x $phong^f$
- *lambert* ≡ max[($\underline{\mathbf{s}}\cdot\underline{\mathbf{m}}$)/(|$\underline{\mathbf{s}}$||$\underline{\mathbf{m}}$|), 0]
- *phong* ≡ max[($\underline{\mathbf{h}}\cdot\underline{\mathbf{m}}$/(|$\underline{\mathbf{h}}$||$\underline{\mathbf{m}}$|), 0]
- Color: combine 3 separate intensities like above
  - one each for Red, Green, and Blue
- light sources have three color intensities:
- ambient = (I_{ar}, I_{ag}, I_{ab}),
- diffuse = (I_{dr}, I_{dg}, I_{db}), specular = (I_{spr}, I_{spg}, I_{spb})
  - For each source; usually $I_{di} = I_{spi}$ for i = r, g, b

The 9 Reflection coefficients

- Ambient: ρ_{ar}, ρ_{ag}, and $\rho_{ab;}$
- Diffuse: ρ_{dr}, ρ_{dg}, and ρ_{db}
- Specular: ρ_{sr}, ρ_{sg}, and ρ_{sb}
  - Ambient & diffuse coefficients usually the same
  - They determine the color of the surface in white light
- Example: modeling sphere 30% red, 45% green, 25% blue
  - diffuse reflection: $(\rho_{dr}, \rho_{dg}, \rho_{db}) = (0.3K, 0.45K, 0.25K)$
- model white light: $I_d = I_s = (1, 1, 1)$
  - → diffused light: (0.3 K, 0.45 K, 0.25 K)* *lambert*
  - →Surface color is 30% red, 45% green, and 25% blue
  - (ignoring specular reflection: *phong* = 0)

Modeling red object in green light

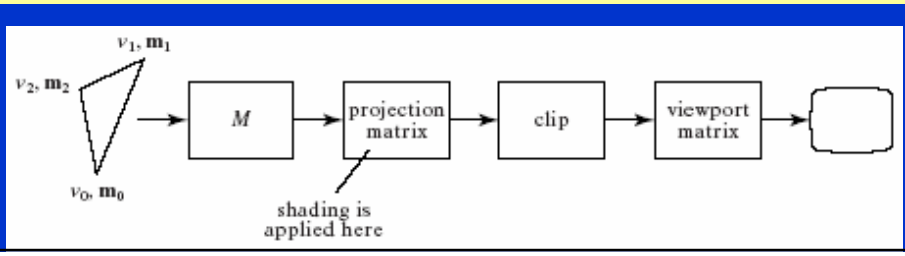
- sphere ambient/diffuse reflection coefficients (0.8, 0.2, 0.1),
  - mostly red when bathed in white light.
- Assume greenish light souce $I_s = I_d = (0.15, 0.7, 0.15)$
- 0.15 x 0.8 = 0.12
- 0.7 x 0.2 = 0.14
- 0.15 x 0.1 = 0.015
- Reflected diffuse light: (0.12, 0.14, 0.015)* *lambert*
- a fairly even mix of red and green
- and would appear yellowish (from color theory)

Specular reflection of colored light

- specular reflection is mirror-like
  - Its color is often the same as that of the light source.
  - **the specular highlight seen on a glossy red apple when illuminated by a yellow light is yellow, not red.**
- Same phenomena on most surfaces
- set the *specular reflection coefficients* $\rho_{sr} = \rho_{sg} = \rho_{sb} = \rho_s$
  - specular reflection coefficients are "neutral"
  - do not alter the color of the incident light.
- choose $\rho_s = 0.5$ for a slightly shiny plastic surface,
- $\rho_s = 0.9$ for a highly shiny surface.

Shading and the Graphics Pipeline (1)

- after modeling each vertex has its position and its normal, **n**
- The modelview matrix M now transforms vertices, light sources and normals (**n**)
  - n → M^{-T}**n** M^{-T} = transpose of the inverse matrix M^{-1}
- All coordinates are "eye coordinates"
- **Color (shading) calculated at this point, for each vertex**
- Results (color values) are kept



Shading and the Graphics Pipeline (2)

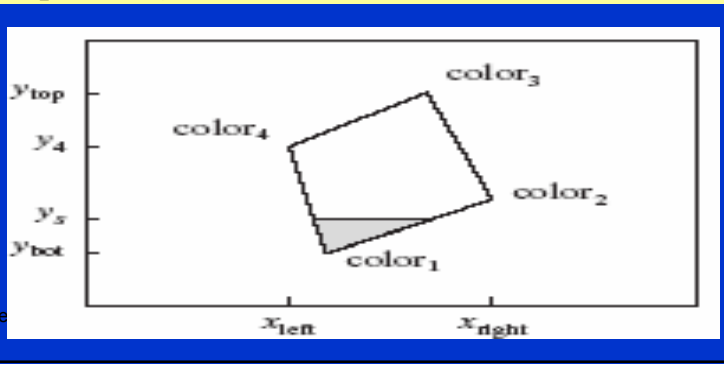
- Next perspective transformation, then clipping
  - Clipping may create vertices which need to have colors, calculated by weighted average of initial vertex colors.
  - E.g. if the new vertex **A** is 40% of the way from $\mathbf{V}_0$ to $\mathbf{V}_1$, the color **(A)** is a weighted average:
  - 60% of (r_0, g_0, b_0) and 40% of (r_1, g_1, b_1)
- Next: viewport transformation → vertices map to screen coordinates
  - Each having pseudodepth, between 0 and 1
- Next hidden surface are removed (later)
- Next: 2-D objects scanlined to a buffer (next topic)

Normal vectors of vertices

- Flat faces in a model are realy flat (e.g. house) or a mesh approximation for a surface
- For a flat face we attach same normal to all its vertices.
- If a face approximates an underlying surface (e.g sphere or a torus), each vertex gets the real normal to the underlying surface
- in later lecture we talk about calculating normals to surfaces

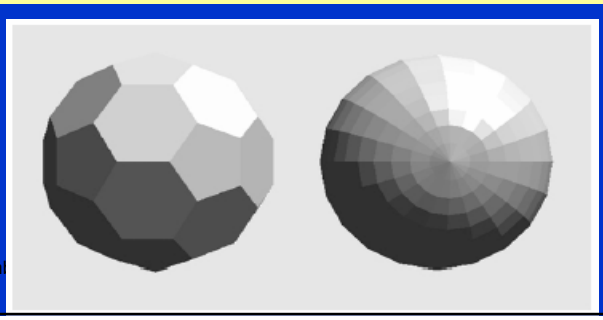
coloring a convex face: scanline

- faces are *scanlined*, putting colors into *frame buffer*
- What are the colors of intermediate pixels?
  - Flat shading: all have same color (of first vertex)
  - Smooth shading: colors are weighted averages (interpolation) of vertex colors



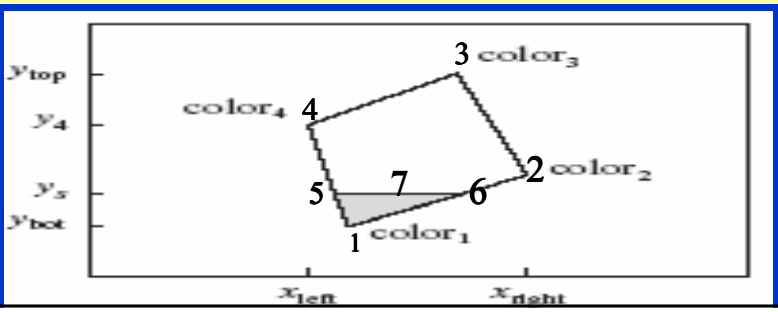
Shading Models: Flat Shading

- Edges appear pronounced (sharpened)
- Specular highlights
- It appears in all points of the face or none at all depending if first vertex has specular component
- Flat shading usually do not include specular light



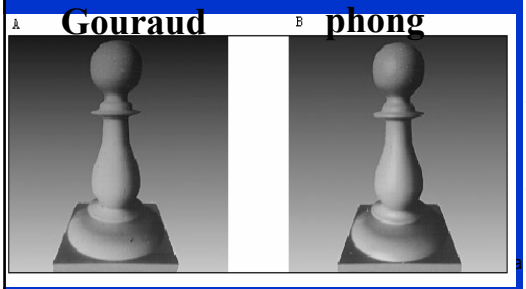
Smooth Shading: Gouraud

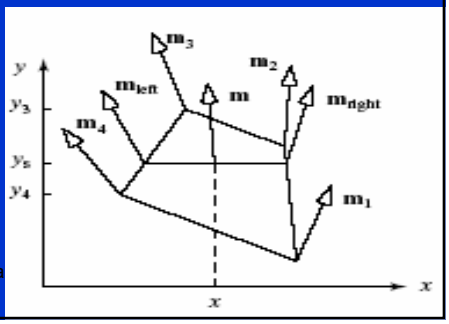
- colors of left edge point C5 interpolate between colors of edge vertices C1 C4 (linear interpolation)
  - C5 = c1 + (c4 - c1)*f*, where $f(y_5) = (y_5 - y_1)/(y_4-y_1)$
- Similarly c6 = c1 + (c2-c1)*g* where $g(y_6) = (y_6-y_1)/(y_2-y_1)$
- then: c7 = c5 + (c6-c5)*h* $h(x_7) = (x_7-x_5)/(x_6 - x_5)$
- Color transition across edges is smooth



Phong Shading: Interpolating normals

- normal vectors on face are interpolation between normal vectors of vertices. Then calculate colors
- Accurate, but very time-consuming! Takes 6 to 8 times longer than Gouraud shading.
- Good with specular light

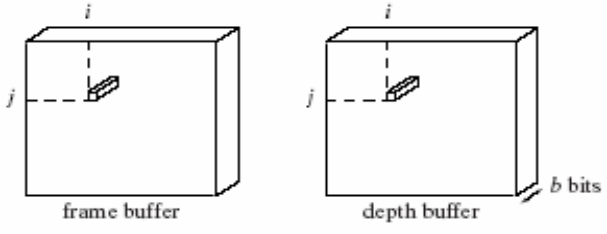




Hidden Surface Removal

Hidden Face removals

- *Frame buffer*: for each pixel [i,j] it has its color c[i, j]
- *Depth buffer*: for each pixel it has its pseudodepth, d[i, j]
- When scanning a face, at point [i,j]:
  - Checks if pseudodepth of [i,j] is smaller than $d[i][j]$
  - If so, the color of [i,j] replaces $c[i][j]$ and the psudodepth of [i,j] replaces the old value in $d[i][j]$. Else nothing



What are the Initial values Of the buffers?

Hidden Face removals (2)

- Recall: Pseudodepth is the 3rd component of the [i, j] point:
  - $d(P_z) = (aP_z + b)/(-P_z)$; a, b were chosen so that $0 \leq d \leq 1$.
  - pseudodepth of vertices are known
  - pseudodepths of internal points are calculated by interpolation – like colors. (not precise; d non-linear)
- Faces can be scanned in any order
  - If remote face is scanned first, color of some of its pixels might later be replaced by colors of a nearer face pixels
- Algorithm works for objects of any shape including curved surfaces, because it test for closenes by point-by-point test
- The content of the frame buffer is copied into the screen
  - Either synchronously in real time or asynchronously

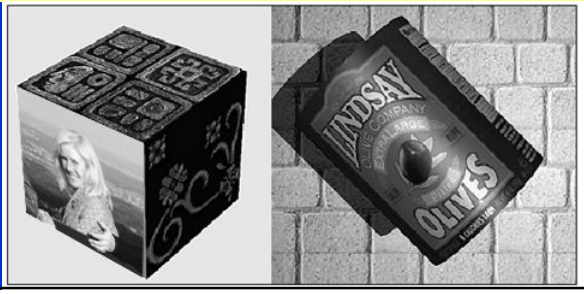
Painter algorithm: summary

- uses depth buffer to remove hidden surfaces
- Also called z-buffer algorithm
- Principal limitations:
  - It requires a large amount of memory.
  - It often renders an object that is later obscured by a nearer object (so time spent rendering the first object is wasted)
  - If interpolation is used, it is not precise

Adding Texture

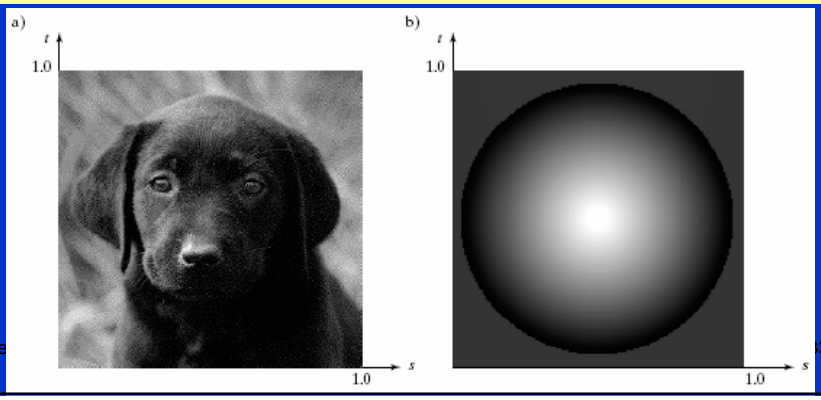
Adding Texture to Faces

- Makes surfaces look more realistic: e.g., brick, wood, or simply pictures are painted on or wrapped around surfaces.
- Texture uses *tex*(s, t) which sets a color or intensity value between 0 and 1, for each value of s and t between 0 and 1
- The (s,t) points are attached to surfaces of the model



How to generate texture

- *tex*(s,t) is a function on *points in texture space (s,t)*
  - Each point (s,t) is called texel
- digitizing image → bitmap tex[i, j] → *tex*(s,t)
- Mathematical procedure → *tex(s,t)*



Procedural and image textures

- **mathematical procedural texture**
- *tex* (s, t) {

  r = sqrt((s - 0.5)*(s - 0.5) + (t - 0.5)*(t - 0.5));

  if (r < 0.3) return 1 - r/0.3; else return 0.2; }
  - *tex* varies: white at center to black at edges of sphere.
- **Image texture**
- Digitizing image → Bitmap of color values
  - *col*[x][y] x, y integers
- Define $s = x/x_{max}$ and $t = y/y_{max}$.
- Divide values of *col* by its max values to get *tex*

Usages of Textures

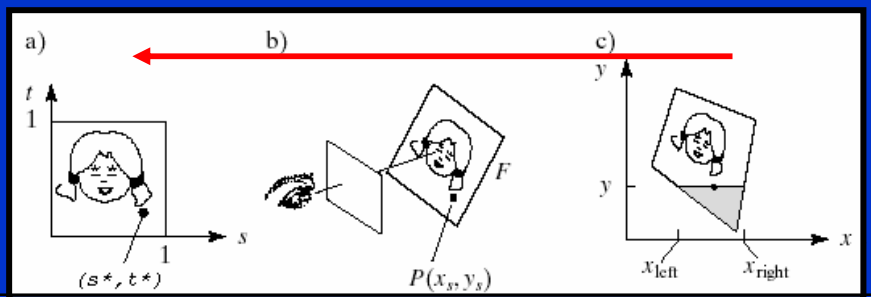
- *tex(s,t)* can be used as the color of the face itself
  - e.g. the face will be glowing at certain points
- *tex(s,t)* can be the ambient, diffuse, or specular reflection *coefficients*
  - to modulate the amount of light reflected from the face
- *tex(s,t)* can be used to alter the normal vector to the surface to give the object a bumpy appearance.

Using Texture with Mesh Objects

- Mesh objects are list of faces (later lecture)
- Data structure of a mesh: 3 lists:
  - Vertex list, face list, normal list
  - Face points to vertices, vertex points to a normal
- Now add a list of texture coordinates, (s_i, t_i)
  - Each vertex points to its (s_i, t_i) - during modeling phase
- Usage 1: flat faces
  - each face points to 1 normal, and to a texture
- Usage 2: flat faces that approximate underlying surface.
  - Each vertex points to 1 normal (of the underlying surface)
  - Each vertex points to its (s_i, t_i)

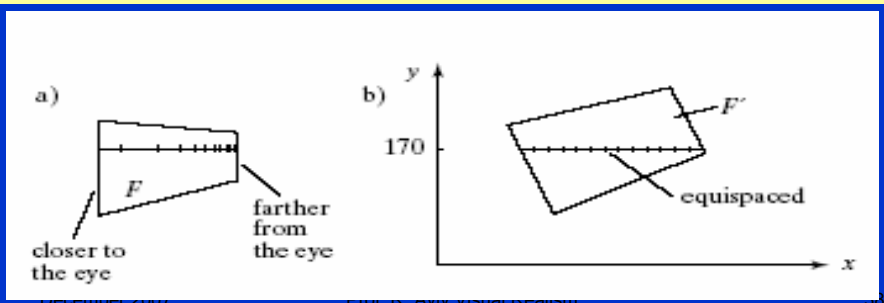
Rendering texture (flat faces)

- Goal: calculate (s*,t*) for each screen point (x_s,y_s) of face
- Basic idea: scan lines of the face
  - (s*,t*) of edge points interpolate (s*,t*) of vertices
  - (s*,t*) of internal points interpolate (s*,t*) of edge points
- Same as calculating colors and pseudodepths of points



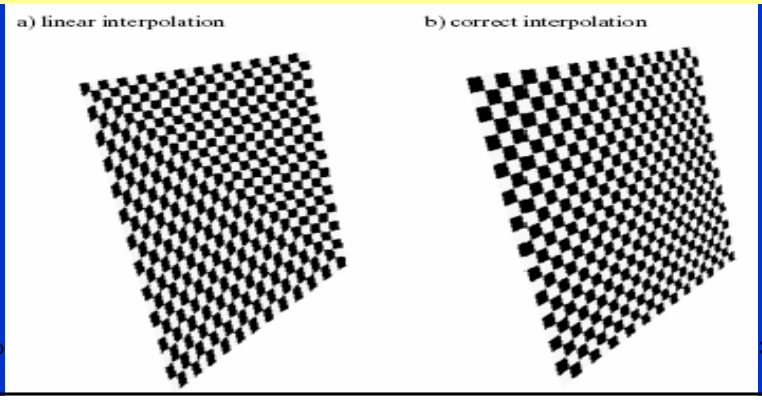
Rendering texture: a problem

- The face on the screen is a *projection* of the real face
- Consider equispaced points on the screen
- are corresponding points on real face equispaced?
- in general, NO
- Linear interpolation assumes that they are



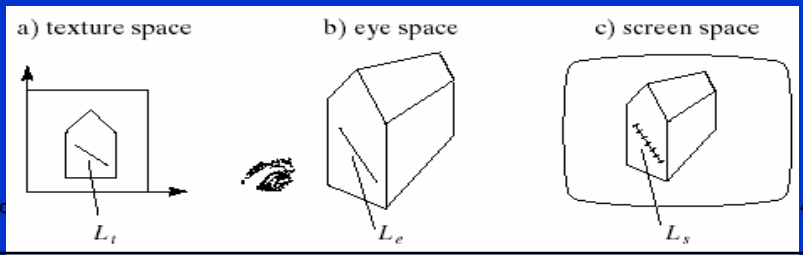
Rendering texture: a problem (2)

- Example: object is rotated checkerboard, projected to screen
- Linear interpolation (left):
  - equispaced, same size squares on screen
  - Farther away squares are not smaller
  - annoying, wrong



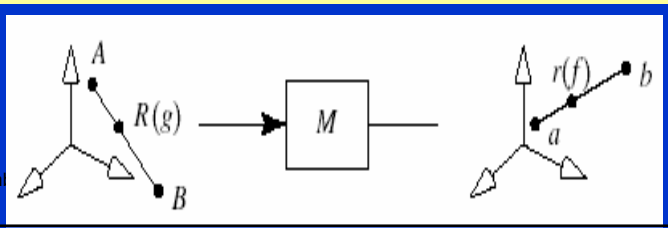
Analysis of the problem (1)

- We attach texel points to points in eye coordinates
  - Line segment of texels, L_t attached to line segment L_e
- But objects are transformed to screen coordinates
  - Affine & projective transformations: Lines → lines
  - L_e in eye space projects to line segment L_s in screen space
- Consider equispaced pixels on line segment L_s (on screen)
- Where are the corresponding points on Le (or Lt)?



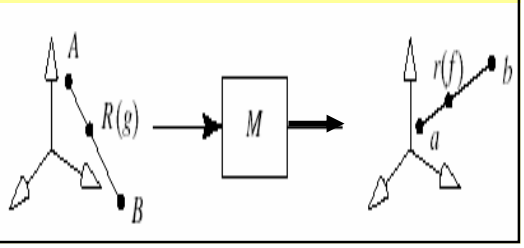
Solving the problem

- 3D segment **AB** transformed into segment **ab,** by matrix M
- **A** maps to **a**, **B** maps to **b**, **R**(g) maps to **r**(f)
- **R**(g) = **A** + (**B** – **A**)g ≡ lerp(**A**, **B**, g) 0<= g <= 1
- **r**(f) = **a** + (**b** – **a**)f ≡ lerp(**a**, **b**, f) 0 <= f <= 1
- fractions f and g are ***not*** the same. g nonlinear funcion of f
- The problem: Find the function $g = g(f)$
- At edge points: $f = 0 \rightarrow g = 0$; $f = 1 \rightarrow g = 1$



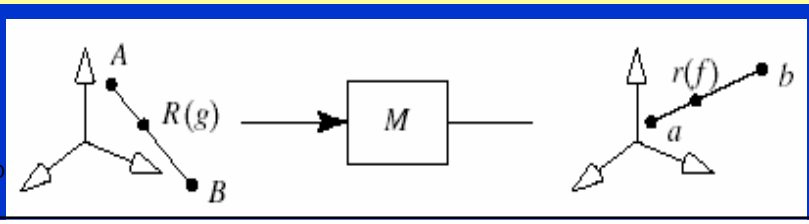
Solving the problem (1)

- Homogeneous coordinates of **A, R** are the quartets
  - **A** = (A_1, A_2, A_3,1) **R** = (R_1, R_2, R_3,1)
- homogeneous coordinates of **a** are the quartet
  - α = (α_1, α_2, α_3, α_4) note that α_4 not necessarily 1
  - **a** = (α_1/α_4, α_2/α_4, α_3/α_4)
- Similarly
- **b** = (β_1/β_4, β_2/β_4, β_3/β_4)
- **r** = (ρ_1/ρ_4, ρ_2/ρ_4, ρ_3/ρ_4)



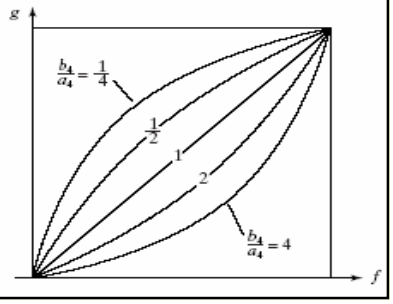
Solving the problem (2)

- The transformation maps **A** to α, **B** to β, **R** to ρ
  - $\alpha = M\mathbf{A}^T \quad \beta = M\mathbf{B}^T \quad \rho(f) = M\mathbf{R}(g)^T$
- But: $\mathbf{R}(g) = \mathbf{A} + (\mathbf{B}-\mathbf{A})g \equiv \text{lerp}(\mathbf{A}, \mathbf{B}, g)$ *check this*
  - (note: $\rho(f) \neq \alpha + (\beta - \alpha)f$ *check this)*
  - $\rho(f) = M^*\text{lerp}(\mathbf{A}, \mathbf{B}, g)^T \quad \text{lerp}(M\mathbf{A}^T, M\mathbf{B}^T, g)$
    - $\rho(f) = \text{lerp}(\alpha, \beta, g)$ (four components equation)



Solving the problem (3)

- The four components of $\rho(f) = \text{lerp}(\alpha, \beta, g)$ are:
- $\text{lerp}(\alpha_1, \beta_1, g)$, $\text{lerp}(\alpha_2, \beta_2, g)$, $\text{lerp}(\alpha_3, \beta_3, g)$, $\text{lerp}(\alpha_4, \beta_4, g)$
- Do perspective division to get actual coordinates of **r**
  - $r_1(f) = \text{lerp}(\alpha_1, \beta_1, g)/\ \text{lerp}(\alpha_4, \beta_4, g)$
- But $\mathbf{r}(f) = \mathbf{a} + (\mathbf{b} - \mathbf{a})f = \text{lerp}(\mathbf{a}, \mathbf{b}, f)$
  - $r_1(f) = \text{lerp}(\alpha_1/\alpha_4, \beta_1/\beta_4, f)$
- Compare two expressions to get
- $g = f/[\beta_4/\alpha_4 + (1 - \beta_4/\alpha_4)f]$
- $g = f/\text{lerp}(\beta_4/\alpha_4, 1, f)$



What determines β_4/α_4 ?

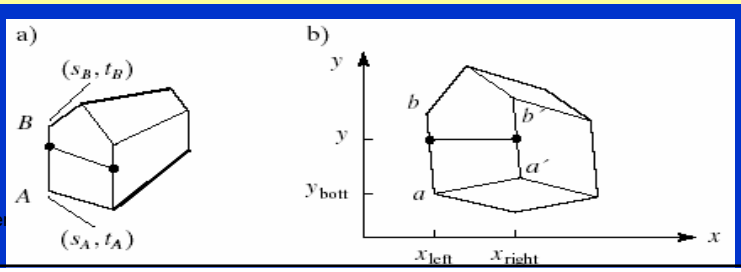
- 1) If M is affine (not a perspective projection)
  - $\alpha_4 = \beta_4 = 1$ → $g = f$ check this
- 2) M is a perspective transformation
  - $\alpha_4 = -A_3$ check this
  - $\beta_4 = -B_3$
  - These are the Z coordinates of points **A**, **B**
- The relative depth of **A**, **B** determine the relation between f and g

The hyperbolic interpolation

- Given a point **r**(*f*) in the screen coordinates, what is the corresponding point that was transformed to it, **R**
  - in real 3D coordinates **R** = ($R_1(f)$, $R_2(f)$, $R_3(f)$)
- Starting point: $R_1(f) = A_1 + (B_1 - A_1)g(f)$
- Plug in the formula for *g* to get
  - $R_1(f) = \text{lerp}(A_1/\alpha_4, B_1/\beta_4, f)/\text{lerp}(1/\alpha_4, 1/\beta_4, f)$ check this
  - Similar expressions for $R_2(f)$, $R_3(f)$
- This is the hyperbolic interpolation.

Example: attaching texture to a barn

- we want to attach a line segment [(s_{left}, t_{left}) , (s_{right}, t_{right})] in texture plane to line segment [(x_{left} , y), (x_{right}, y)] in the screen (frame) buffer
- left edge of the face in the screen has endpoints **a** and **b**.
- First find the texture coordinates (s_{left}, t_{left}) (s_{right}, t_{right})
- then interpolate across the scan-line.



Example: attaching texture to a barn (2)

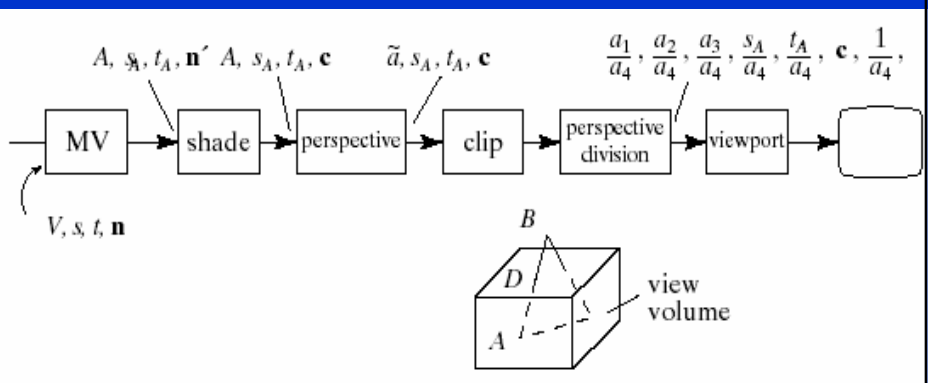
- *Assume* (S_A , T_A) (S_B , T_B) were attached to vertices **A**, **B**
  - during modeling phase
  - *they were passed down the pipeline along with **A** and **B***
  - they are now attached to projected vertices **a** and **b**
  - no transformation were done on them
- Now scanline is done
- scan line at y is a fraction *f* of the way between y_{bott} and y_{top}
  - $f(y) = (y - y_{bott})/(y_{top} - y_{bott})$
  - $s_{left}(y) = lerp(S_A/\alpha_4, S_B/\beta_4, f(y))/lerp(1/\alpha_4, 1/\beta_4, f(y))$
- similar expression for s_{right} dependence on $S_{A'}$, $S_{B'}$, y

Example: attaching texture to a barn (3)

- similar expression for t_{left} and t_{right} dependence on T_A, T_B, y
- then s(x) are calculated for the fixed y
  - by hyperbolic interpolations between s_{left}, s_{right}
- similarly t(x) is calculated by hyperbolic interpolation between t_{left}, t_{right}
- Repeat for next y

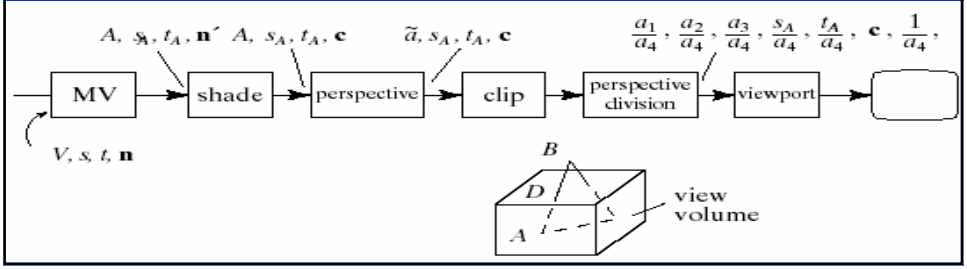
Graphics Pipeline: Sart

- The pipeline: Various points in the pipeline are labeled with the information available at that point.
- Start: Each vertex V is associated with a texture pair (s, t) and a vertex normal, **n**



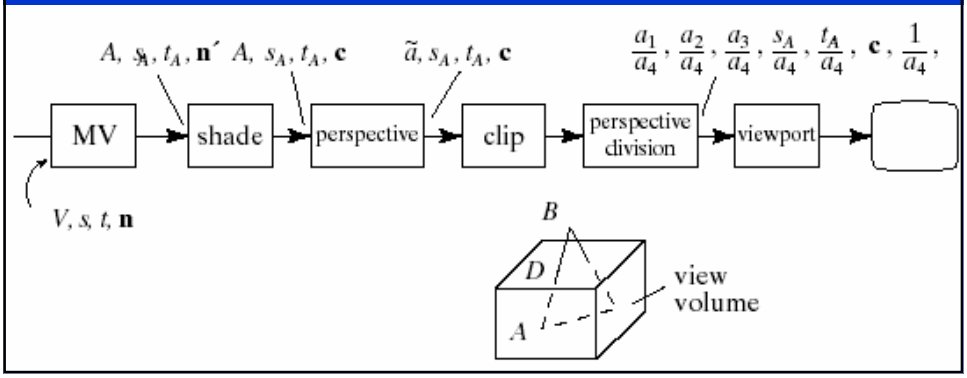
Graphics Pipeline: Modeling, shading

- vertices **V** transformed by the modelview matrix M
- normals multiplied by the inverse transpose of M
  - giving vertex **A** = (A_1, A_2, A_3) and a normal **n**' in eye coordinates.
- Shading calculations are done using this normal, producing the color **c** = (c_r, c_g, c_b).
- The texture coordinates (s_A, t_A) (same as (s, t)) are still attached to A.



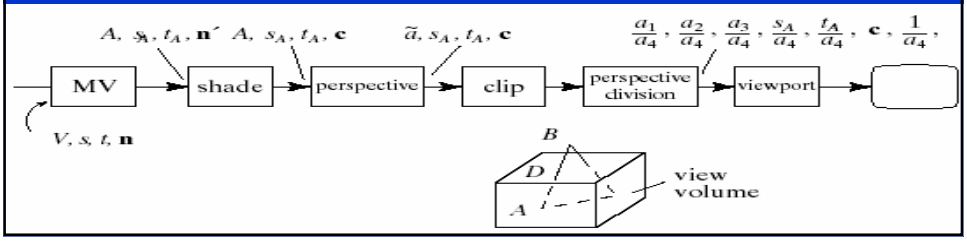
Graphics Pipeline: Perspective Transformation

- Perspective Transformation:
  - Vertex A transformed into $\alpha = (\alpha_1, \alpha_2, \alpha_3, \alpha_4)$
  - colors and texture points unchanged
- information is now α, S_A, T_A, **c**



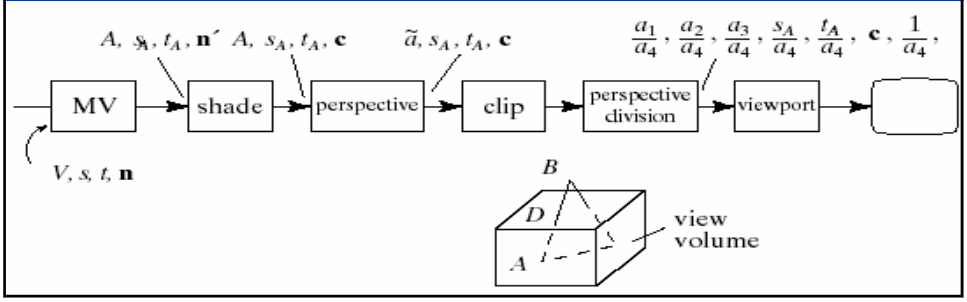
Graphics Pipeline: Clipping

- Clipping:
  - some vertices disappear others created
- position of new vertex D (d_1, d_2, d_3, d_4) by linear interpolation $d_i = lerp(a_i, b_i, t)$, for $i = 1,.., 4$
  - color and texture points are also calculated by linear interpolation, and attached
- information is in the array $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, s_A, t_A, \mathbf{c}, 1)$.



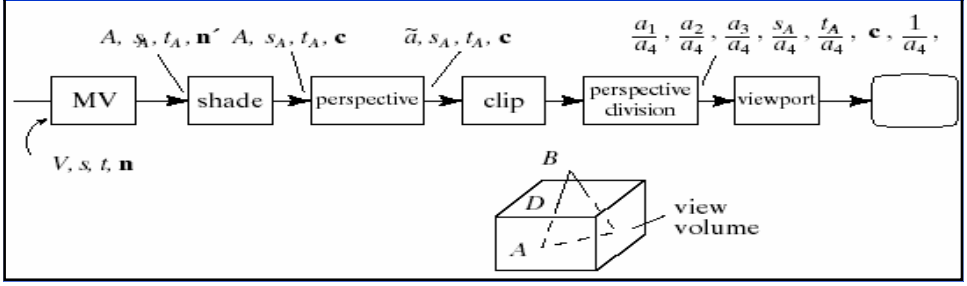
The Graphics Pipeline: perspective division

- hyperbolic interpolation need terms such as s_A/α_4, $1/\alpha_4$, ...
- we divide *every* item in the array that we wish to interpolate hyperbolically by α_4
- result $(\alpha_1/\alpha_4, \alpha_2/\alpha_4, a_3/a_4, 1, s_A/\alpha_4, t_A/\alpha_4, \mathbf{c}, 1/\alpha_4)$
- first three are position in Normalized Device Coordinates
  - third is pseudodepth



The Graphics Pipeline: Viewport transformation

- first two components scaled and shifted by the viewport transformation
- we call the position now x, y, z
- information per vertex: $(x, y, z, 1, s_A/a_4, t_A/a_4, \mathbf{c}, 1/a_4)$
- It is simple to render texture using hyperbolic interpolation, since the required values s_A/a_4 and $1/a_4$ are available for each vertex.



The Graphics Pipeline: Scanline

- information per vertex: $(x, y, z, 1, s_A/a_4, t_A/a_4, \mathbf{c}, 1/a_4)$
- Scanline: determine properties of internal points
  - hide back surfaces: use z (depth) buffer
  - shading (calculate colors): can use color buffer)
  - add texture: use the texture function
- Push color buffer to the screen