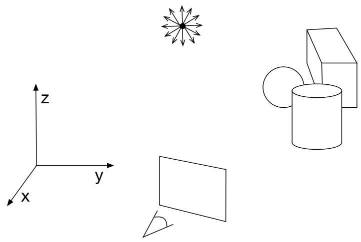
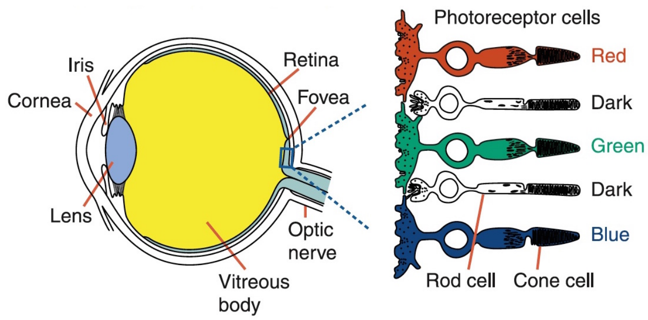
The Fundamental Problem



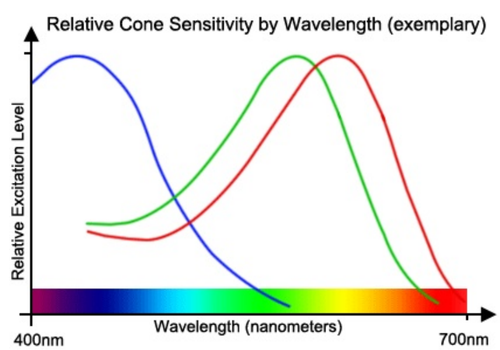
Given: model, material properties, eye/camera, lights

Generate 2-D image

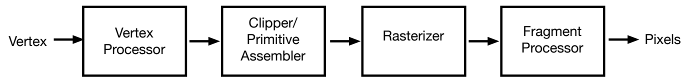
3 Color Sensors (Cones)



How Cones See Spectrum



Graphics Pipeline



- transformation, rasterization
- originally fixed-function VLSI
- pipeline, parallelism
- GPUs evolve -> more powerful, programmable
- vertex shaders, fragment shaders

Shaders

- **Vertex Shaders:** programs that describe the traits (position, colors, and so on) of a vertex. The vertex is a point in 2D/3D space, such as the corner or intersection of a 2D/3D shape.
- **Fragment Shaders:** programs that deal with the per-fragment processing such as lighting. The fragment is a WebGL term that you can think of as a kind of pixel and contains color, depth value, texture coordinates, and more.

Coordinate Systems

- Model: where you define object
- World: place objects, define eye/camera, define light positions, perform lighting operations
- Eye: define view volume, perform lighting operations
- Canonical View Volume: clip
- Screen: device specific coordinates

Want Matrix Representation

why?

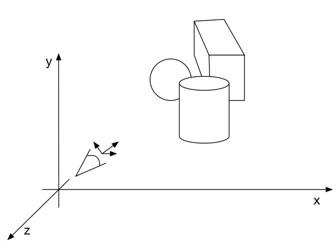
$$\left(TM_3\left(TM_2\left(TM_1\right)\right)\right)\begin{bmatrix} x \\ y \end{bmatrix} = (TM_3 TM_2 TM_1)\begin{bmatrix} x \\ y \end{bmatrix} = TM_{combined}\begin{bmatrix} x \\ y \end{bmatrix}$$

Homogeneous Coordinates

- want to represent all transformations with a matrix
- P(x, y) ⇔ P(w·x, w·y, w), w ≠ 0
- i.e. go up 1 more dimension
- we can always go back by dividing by w
- let's use w = 1
- eg. P(3, 4) ⇔ P(3, 4, 1)

Eye Coordinate System

- eye is at origin
- eye is looking along z axis (l.h.s.?)
- x-axis is horizontal
- y-axis is vertical



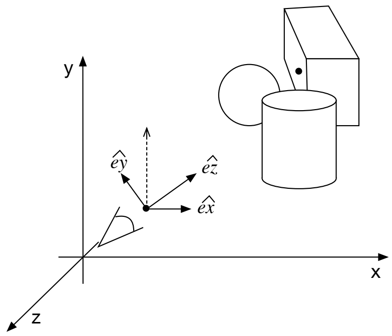
Viewing Transformation

given: $lookFrom, lookAt, \overrightarrow{lookUp}$

$$\widehat{ez} = \frac{lookAt - lookFrom}{\|lookAt - lookFrom\|}$$

$$\widehat{ex} = \frac{\widehat{ez} \times \overrightarrow{lookUp}}{\|\widehat{ez} \times \overrightarrow{lookUp}\|}$$

$$\widehat{ey} = \widehat{ex} \times \widehat{ez}$$

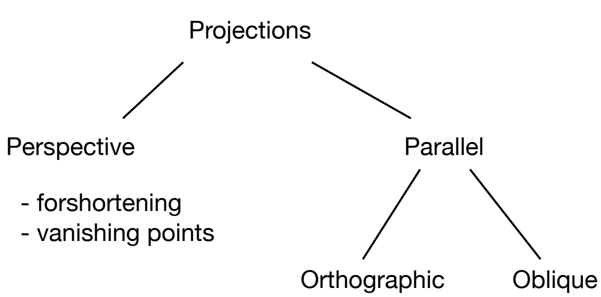


Direct Matrix Creation

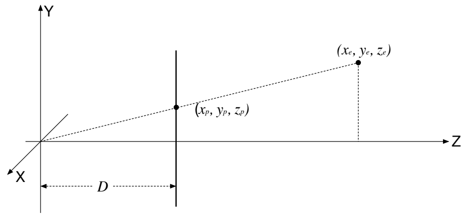
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} ex_x & ex_y & ex_z & d_x \\ ey_x & ey_y & ey_z & d_y \\ ez_x & ez_y & ez_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$d_x = -\widehat{ex} \bullet lookFrom$$
$$d_y = -\widehat{ey} \bullet lookFrom$$
$$d_z = -\widehat{ez} \bullet lookFrom$$

Projections



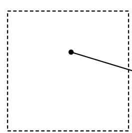
Perspective Projection



$$\frac{y_p}{D} = \frac{y_e}{z_e} \longrightarrow \quad x_p = \frac{D \cdot x_e}{z_e} \qquad y_p = \frac{D \cdot y_e}{z_e}$$

What if z_e is < 0 or = 0?

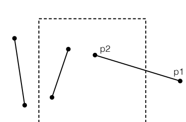
Clipping



- removing parts of object that are outside field of view
- related terms:
  - culling: quickly determining in/out
  - bounding box: axis-aligned box containing object
  - scissoring: combing clipping with scan conversion

Clipping Lines: Cohen-Sutherland

1. compute labels for p1 & p2
2. determine if total visible or trivial reject
3. if p1 not outside, swap p1 & p2
4. find edge p1 is out
5. replace p1 with intersection of p1-p2 and edge
6. compute new label for p1



1001	1000	1100
0001	0000	0100
0011	0010	0110

if both labels 0
→trivial accept

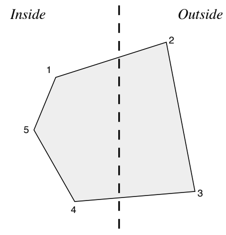
if label(p1) ∩ label(p2) ≠ 0
→trivial reject

Hardware acceleration

Polygon Clipping: Sutherland-Hodgman

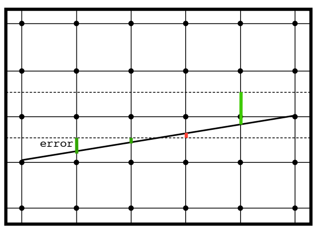
- convex clipping region
- clip against one edge at a time

P	C	Output
inside	inside	c
inside	outside	intersection point
outside	outside	-
outside	inside	intersection point; c



Bresenham's Alg

```
DrawLine(int x1,int y1,int x2,int y2){
 int x, y, dx, dy, error
 dx = x2-x1
 dy = y2-y1
 error = 2*dy-dx
 y = y1;
 for (x = x1; i <= x2; x++) {
  SetPixel(x, y)
  if (error > 0) {
     y++
     error = error - 2*dx
  } else
     error = error + 2*dy
 }
}
```

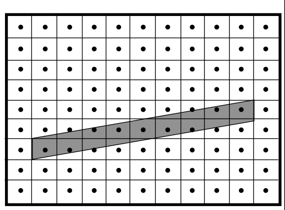


- *multiply error by 2*dx (only care about sign)*
- *developed in 1960s*
- *pen plotters*

Area Averaging

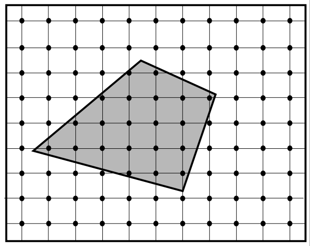
- contribution proportional to area within pixel square

$$I'(x_0,y_0)=\int_{x_0-0.5}^{x_0+0.5}\int_{y_0-0.5}^{y_0+0.5} I(x,y)dxdy$$



Convex Polygons

- find top vertex
- go down left and right sides
- compute intersections with scanline
- draw horizontal runs on each scan line



- *incremental*
- *amortize edge computations*

Newell & Sequin

- VLSI
- compute “winding number” as you go along scanline
- edge going up: +1
- edge going down: -1
- draw non-zero spans

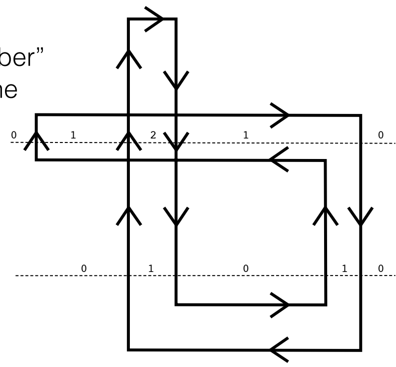
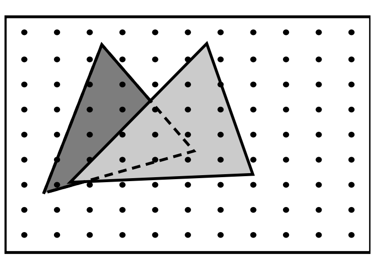


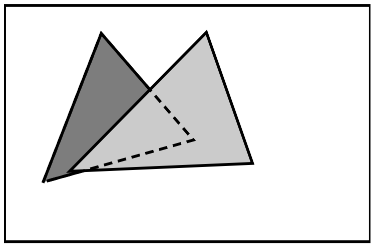
Image Space

- for each pixel in image:
  - determine object closest to viewer
  - draw pixel appropriate color



Object Space

- for each object in scene:
  - determine parts of object that are unobstructed
  - draw those parts appropriate color

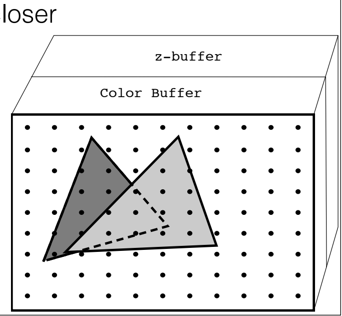


Techniques for Efficient Visible Surface Algorithms

- **coherence**: degree to which parts of environment or its projection exhibit local similarities
- examples of types of coherence: object, face, edge, scan-line, area, depth, frame

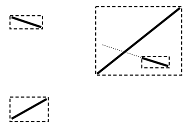
VSA: Z-Buffer (depth buffer)

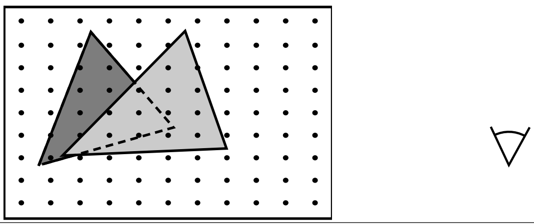
- 2 buffers: color buffer, z-buffer
- compare during scan conversion:
  - if depth of new fragment is closer
    - update color buffer
    - update z-buffer



VSA: Painters Alg

- sort polygons back to front
- resolve any ambiguities
  (use extents, clip if necessary)
- scan-convert polygons back to front

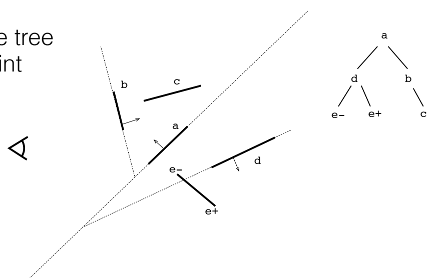




BSP-Trees

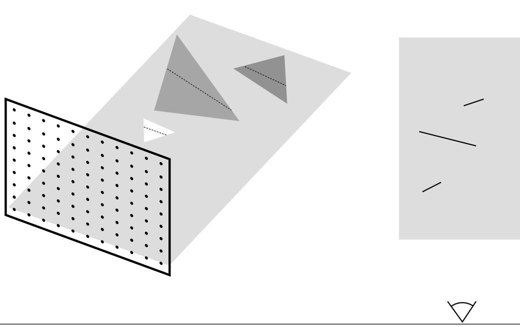
Draw in priority order (back to front)

- create tree of subspaces
  (nodes store polygon,
  separating plane)
- recursively traverse tree
  using lookFrom point
- visit order:
  - far
  - plane
  - near



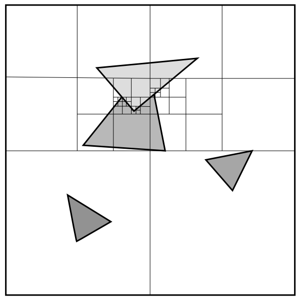
VSA: Scanline Alg

- scanline at a time
- reduce dimension: 3D->2D



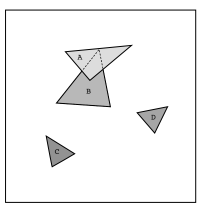
VSA: Warnock's Alg

- recursively subdivide screen
- stop when "simple" or at pixel
- at pixel draw closest object



VSA: Weiler-Atherton

- fast sort of polygons by z
- select "closest" polygon
- use it to clip the rest
- if any poly inside clipping poly closer -> initial sort wrong
  - use it as clipping poly first
- otherwise discard those inside
- draw clipping poly

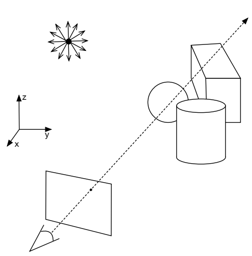


VSA: Ray Casting

(Ray Tracing)

- shoot ray ray from eye through screen into world
- intersect objects with ray
- find closest intersection
- do shading/lighting calculation

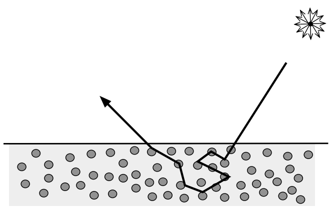
- very floating-point intensive

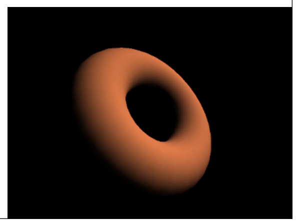


Diffuse Reflection

(Lambertian Reflection)

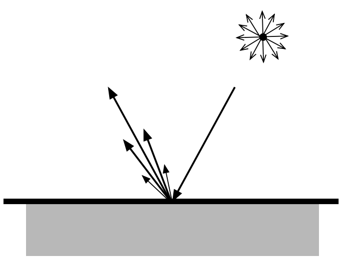
- dull, matte surfaces
- reflect light equally in all directions
- light enters object, scatters internally
- eg: plastic, paint, paper, vegetation, snow, etc





Specular Reflection

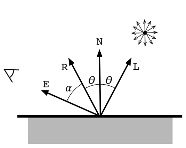
- shiny surfaces, highlights



Diffuse + Specular

$$I = I_{light} \cdot (K_d \cdot \cos(\theta) + K_s \cdot \cos^n(\alpha))$$

[0…1]

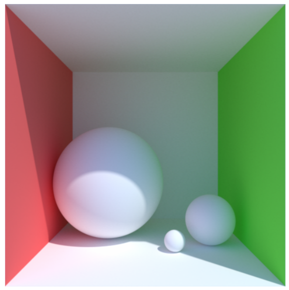


$\cos(\alpha) = \text{dot}(\texttt{E},\texttt{R})$

Ambient Reflection

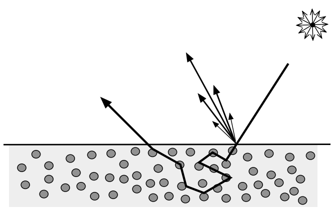
- modeling inter-reflection is hard
- parts in shadow are black (looks bad)
- approximate indirect lighting
- simplification:
  use constant K_a

[0…1]



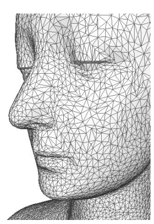
Color

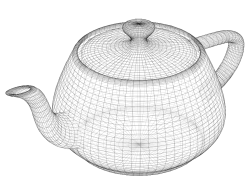
- K_a, K_d, K_s depend on λ, I_{light} depends on λ
- highlights: white for many objects (actually, color of light)



Gouraud Shading

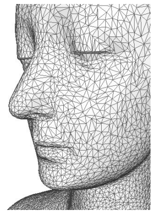
- compute vertex normals
  - average of polygons around vertex
  - directly from model during tessellation
- perform lighting operation at vertex
- linearly interpolate resulting vertex color
  (linear interpolation not correct)

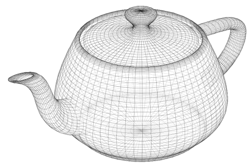




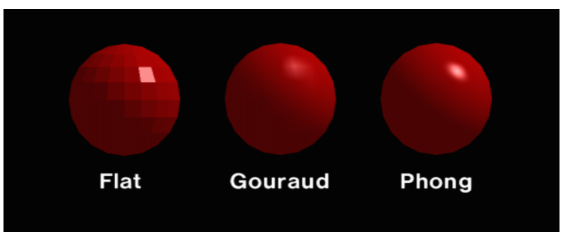
Phong Shading

- compute vertex normals
- linearly interpolate vertex normals
  (linear interpolation not correct)
- perform lighting operation per pixel





Shading Models for Polygons



Cook-Torrance

reflectance (Fresnel) is also a function of wavelength λ

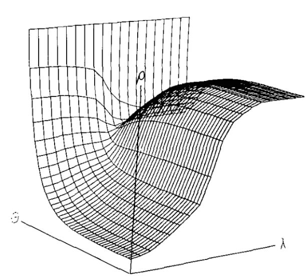


Figure 5a. Reflectance (ρ) of a copper mirror as a function of wavelength (λ) and incidence angle (θ).