Raster image – made of pixels (picture elements)
Scan line – row of pixels
Pixel – (r, g, b) 0-255
(255, 150, 150) = pink
(0, 0, 0) = black

Coordinates in Processing

size(400, 400); creates window → origin in top left, pixels go 0-399 in each axis
rect(100, 200, 10, 10); = x, y of top left, width, height
ellipse(200, 200, 20, 20); = x, y of center, horiz diameter, vert diameter
background(255, 255, 255); = color of background
- This **resets the window** (removes previous)

setup() {} called once at start of program
draw() {} called repeatedly

noFill(); - removes shape fill

Transpose [-2, 2] range to [0, 400] → add 2, mult by 100 (or div by 100.0)

Position on circle: (x, y) = (cos, sin) – **radians**

Vectors
v1 dot v2 = sum of products of entries
dot to 0 → perpendicular

Transformations
Translate, rotate, scale

Translate: move x to x' = x + dx, y to y' = y + dy
P' = P + T (with vectors)

Scale:
$S = \begin{matrix} sx & 0 \\ 0 & sy \end{matrix}$ → P' = SP

Rotate (counterclockwise around origin): P' = RP
x' = xcosA – ysinA → cos -sin
y' = xsinA + ycosA sin cos

Homogenous Coordinates
Point (x, y) → vector [x, y, 1]
Scale matrix → [sx 0 0; 0 sy 0; 0 0 1]
Rotation matrix → [cos -sin 0; sin cos 0; 0 0 1]
Translation matrix → [1 0 dx; 0 1 dy; 0 0 1] (this works better with homog. coords)

Shear: Farther from origin → shifts more
x' = x + ay
y' = y

Reflection: x' = -x, y' = y (y-axis reflection)
x' = x, y' = -y (x-axis reflection)

Transformations: Translate to origin, then rotate, then translate back
Multiply by appropriate matrices in order

P''' = T2 *(R * (T1 * P)) = (T2 * R * T1) * P
T1 is translation to origin, R is rotation, T2 is translation back to start pt. (reverse order)

OpenGL: glScale(2.0, 2.0), glTranslate(-2.0, 0.0), draw_triangle();
*** **for translate to happen first, write it second** (draw triangle last)

Commutative

	Translation	Rotation	Uniform scale	Non-unif. scale
Translation	Yes	No	No	No
Rotation		Yes (No in 3D)	Yes	No
Uniform scale			Yes	Yes
Non-unif. scale				Yes

Draw a line:
gtBeginShape(GT_LINES ← placeholder);
gtVertex(100.0, 400.0); ← gtVertex: connects pairs of points together
gtVertex(100.0, 100.0);
gtEnd();

Draw a circle (with a bunch of lines);

```
void circle() {
        gtBeginShape(GT_LINES);
        xold = 1
        yold = 0
        for (i = 1; i <= 20; i++) {
                theta = 2 * pi * i / 20;
                gtVertex(xold, yold);
                gtVertex(cos theta, sin theta);
                xold, yold = cos theta, sin theta;
        }
        gtEndShape();
}
```

Matrix Stack

CTM = current transformation matrix (product of matrices)
gtPushMatrix() – replicate CTM and pushes onto top of stack
gtPopMatrix() – pop off top of stack (and throw it away)
gtTranslate(x, y, z) – translate by x, y, z units
gtScale(x, y, z) – scale by x, y, z in each dimension
gtRotate(theta) – rotate theta around origin
gtVertex(x, y) – position is multiplied by CTM before drawing point

translate, scale, rotate create transformation matrix and multiply <u>on right</u> of CTM
****<u>First thing to affect square in hand() function is T3 (rightmost matrix on stack)</u>****

<u>Note: translate moves origin!!</u>

Drawing circle with two half-size circles stacked inside:

```
void circle_image() {
        gtPushMatrix()    ← Stack is just the identity, add identity on top of that
        gtTranslate(0.5, 0.5)    ← top of stack becomes I * T
        gtScale(0.5, 0.5)    ←  top of stack becomes I * T * S
        circle() ← draw the circle centered at 0.5, 0.5 with radius 0.5
        gtPopMatrix() ← take off top element (I * T *S), stack becomes just identity

        push
        translate(0.5, 0.25)
        scale(0.25, 0.25)
        circle()
        pop
        push
        translate(0.5, 0.75)
        scale(0.25, 0.25)
        circle()
        pop
}
```

Matrix Stack Uses:
-change coordinate system
-instantiation (object reuse)
-hierarchy (objects made of sub-objects)
Ex: Person composed of torso + 2 arms (extent + hand (fingers + thumb))

Drawing a body (<u>see handout</u>):

```
void square() {
        gtBeginShape(GT_LINES);
        Draw square from (-1, -1) to (1, 1) with gtVertex commands
        gtEnd();
void extent() {
        push()
```

```
        scale(3, 0.5)
        translate(1, 0)
        square()
        pop()
}
```

3D Coordinates
Right-handed - Z is left (middle finger), X is right (thumb), Y is up (index finger)

P = [x; y; z; 1]
T = [1 0 0 dx; 0 1 0 dy; 0 0 1 dz; 0 0 0 1]
S = [Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1]
P' = TP or SP

Rotation is 3 diff. matrices: Rx, Ry, Rz
Counterclockwise – start on positive axis (x, y or z) and rotate towards negative axis
Rx : [1 0 0 0; 0 cos -sin 0; 0 sin cos 0; 0 0 0 1]
Ry: [cos 0 sin 0; 0 1 0 0; -sin 0 cos 0; 0 0 0 1]
Rz: [cos -sin 0 0; sin cos 0 0; 0 0 1 0; 0 0 0 1]

Parallel Projection
Map pts. on object to pts. on view plane using projector lines from object through plane

Perspective projection – project objects onto view plane based on center of projection (all projectors go through center of projection and hit pts. on object)
- -Farther away objects look smaller on view plane
- -Simulates vision or camera
- -Use similar triangles to reduce measurements appropriately to view plane

Orthographic Projection (-Z to the right, +Z to the left, Y up and down)
view plane – projects points onto xy plane
orthographic view volume – region of space we can see
- -bottom left is (l, b, n), top right is (r, t, f)
- -x in [left, right], y in [bottom, top], z in [near, far]
- -view plane top/bottom, left/right go to screen 0/height, 0/width and left/right, bottom/top
- -Must translate/scale window to match screen
- -anything farther than far or closer than near is clipped away in z

Field of view – angle for width of visibility
- -height of view plane = 2Rtan(theta/2) if view plane is R away from center of projection
- -mapping y to screen: [-Rtan(theta/2), Rtan(theta/2)] → [0, h]

y' = y / |z|, x' = x / |z|
y'' = h(y' + k) / (2k), x'' = w(x' + k) / (2k)
k = tan(theta / 2)

v = (x, y, z)
v-hat = V / |V| = (x, y, z)/(x^2 + y^2 + z^2)
v-hat dot v-hat = 1
orthogonal – A dot B = 0

cross product – V1 x V2 = another vector orthogonal to V1, V2
V1 x V2 = (y1z2 – y2z1, x1z2 – x2z1, x1y2 – x2y1)
Perpendicular vectors in 2D **dot to 0**

Rotation matrix: [ax ay 0; bx by 0; 0 0 1]
Or [ax ay 0; -ay ax 0; 0 0 1]
These rotate A to x-axis (kill y term)
Create 2 unit vectors perpendicular to each other, put in 1st 2 rows

Rotate theta around given axis vector A
Rotate to x-axis, Rotate around x-axis, rotate back

Choose N not parallel to A, let B = (A x N)/|A x N|
Let C = A x B
A, B, C are unit length and mutually orthogonal

R1 = [ax ay az 0; bx by bz 0; cx cy cz 0; 0 0 0 1]
R1A = [A dot A A dot B A dot C] = [1 0 0]
R2 is rotation matrix around x-axis
$R3 = R1^{-1} = R1^{T}$
Overall matrix = $R_3R_2R_1$
RX = rotate X around this axis

glRotate(theta, ax, ay, az) to rotate around that axis

Making N: N = (1, 0, 0) if ax is 0, N = (0, 1, 0) otherwise

View Transformation
Camera(ex, ey, ez, cx, cy, cz, ux, uy, uz) – coords are eye position, center, and up
Usually assume y-axis is up

gluLookUp(from, at, up)
Assume perspective projection

camera(0, 0, 8, 0, 0, 0, 0, 1, 0) is same as translate (0, 0, -8)

camera(0, 0, 0, 1, 0, 0, 0, 1, 0) is same as rotate(90, 0, 1, 0) – around y-axis

View matrix: $M_{view} = R * T$
T = translate(-ex, -ey, -ez)

W = -g / |g| (unit vector for gaze direction)
U = t x w / |t x w|
V = w x u

R = [ux uy uz 0; vx vy vz 0; wx wy wz 0; 0 0 0 1]
View matrix puts gaze point at origin, facing -z axis (R dot g = [0 0 -1 1])

A x B = [**i j k**; a1 a2 a3; b1 b2 b3] (**i** * det bottom right + **k** * det bottom left + **j** * det cols 1, 3)

Output Devices
LCD, E-Ink

CPU connects to Bus, which connects to Frame Buffer (stored image, memory)

Double buffering - Memory 1 used for display, Memory 2 for creating next image
→ directed to video controller, then monitor
If you don't double buffer: partially blank screen, image tearing

LCD – liquid crystal display
Uses molecules that can change configuration from pressure, temp., electric fields

Light source → polarizer → hits liquid crystal material (no voltage) → polarizer → hits eye

Orientation of polarizer causes each lens to block light in different direction
- One is horizontal, one vertical → orient glasses vertically one lens blocks light, horizontally the other one does

Pixel intensity – controlled voltages to partially un-twist crystals
Full LCD screen uses thin film transistors
- Switch on one row, send charge down column → affect one pixel
Color LCDs have RGB filters (3 sub-pixels in each pixel)

E-Ink (electronic paper)
-Invented Xerox
-Just reflects light, doesn't create it
-Based on pigment particles (some bright, some dark)
-Black particles are positively charged, white negative
-Reads in bright sunlight, low energy consumption
-Cons: slow image switching

Lines
Parametric: $(x(t), y(t)) = (x_1, y_1) + t(x_2 - x_1, y_2 - y_1)$
Implicit: On line when $f(x, y) = 0$ (otherwise not)
$f(x, y) = ax + by + c$

```
void line(x0, y0, x1, y1) {
        dx = x1 – x0, same for y
        length = max(fabs(dx), fabs(dy))       fabs – float abs. value
        xinc = dx / length, same for y
        x, y = x0, y0
        for (I = 0; i <= length; i++) {
                gtWritePixel(round(x), round(y), 3 color values);
                x += xinc;
                y += yinc;
        }
}
```

aliasing – jaggedness in lines due to pixel values being integers (can't do fractional points)
rasterization – turning line/polygon into pixels

```
for (y = ymin; y < ymax; y++)
        for (x = xmin; x < xmax; x++)
                gtWritePixel(x, y, RGB color)
```

** This doesn't draw upper/right edges of rectangle! **

Only 1 rect. should touch a pixel (want transparency) → don't do upper/right edges

Polygons

Convex vs. concave
- Can add features (holes)
- non-exterior, even-odd filling rules

Rasterizing polygons: fill one scanline at a time, from bottom to top, fill between intersections

Whenever scanline intersects with an edge, fill in pixel nearest to intersection & <u>fill all pixels between them</u>

```
for (y = ymin; y <= ymax; y++) {
        -find x intersections with edges, sort by x-value
        -fill all pixels between pairs of intersections
```

Can go from one intersection point to the next along a certain edge:

xdiff / 1 = total dx / total dy → increment y by 1 and x by xdiff to get next intersection

-Graphics cards use <u>rasterization by z-buffer</u> algorithm to draw

Start at bottommost scanline → label left x intersection xleft and right x intersection xright

If each intersection is on its own line, (like a V), then need two dxleft and dxright (xdiff)

<u>Rasterize polygon</u>
find ymin, ymax
find xleft, x right, dx left, dx right

```
for (int y = ceiling(ymin); y < ymax, y++) {
        for (int x = ceiling(xleft); x < xright; x++) {
                writePixel(x, y, color stuff)
        // maybe switch edges
        xleft += dxleft
        xright += dxright
```

Hidden vs. Visible Surfaces

Note: eye is +z direction (out of frame)

Painter's Algorithm

Sort polygons comprising object by increasing avg. z of their pixels → draw them in this order

-Draws polygon from back to front, so front ones cover up back ones

Flaws with Painter's: Using centroid → objects partially behind others can be drawn over them and more complex overlapping fails

- Binary space partition can fix this issue

z-buffer (graphics cards)

```
for each pixel (x, y)
        Write Pixel(x, y, color)
        WriteZ(x, y, far)
for each polygon
        for each pixel (x, y) in polygon → polygon rasterization
                pz = polygon z-value at x, y
                if (pz >= ReadZ(x, y)) {
                        WriteZ(x, y, pz)
                        WritePixel(x, y, polygon color)
                }
```

Replace each pixel's z-value/color with a new one if there is a polygon further forward

Unit vectors A, B → A dot B = cos theta (angle between them)

Surface normal for triangle with points A, B, C:

$E_1 = B - A$ (vector is diff. between coordinates)

$E_2 = C - A$

$N = E_1 \times E_2 / |E_1 \times E_2|$ (unit length perpendicular to triangle ABC)

Surface Shading: color of object, how bright

- Care about light sources and property of surface
- Surfaces can reflect or absorb light
  - diffuse – reflects light equally in all directions (chalk, matte paint, paper)
  - shiny – reflects at same angle as light hit

light source hits vertical surface A → hits surface tilted by theta → A / cos (theta) is bigger than A so larger surface

- Results in fewer photons per unit area

L points towards light source, N is normal to surface, N dot L = cos theta between those vectors

Diffuse surface: C (final color) proportional to C_r (diffuse color) * (N dot L)
For light of color C_L: $C = C_r C_L$(N dot L)
$C = C^R, C^G, C^B$
$C^R = C_r^R C_L^R$(N dot L), same for G and B
C = max(0, N dot L) to avoid negative light

Indirect Illumination Cheat:
$C = C_r(C_a + C_L * \max(0, N \text{ dot } L))$ where C_a is ambient light and C_L is light source
-Makes shadows not completely black
-Should use photon mapping, radiosity, metropolis light transport instead of ambient

Shiny Surfaces (Phong + Blinn Illumination Models)
- metals, plastics, mirrors
- angle of impact matches angle of reflection
- viewer sees the most when light comes in perpendicular to normal, less as it moves away from that

Possible function: $(\cos(\pi/2 - \text{theta}))^P = (E \text{ dot } R)^P$
P is specular exponent
This gives $C = C_L * \max(0, E \text{ dot } R)^P$ Phong illumination

$R = 2N(N \text{ dot } L) - L$

Shiny Surfaces (Blinn)
Create half-way vector between L (light source) and surface normal vector

$H = (L + E)/\|L + E\|$ (unit vector)
$(H \text{ dot } N)^P = (\cos B)^P$
$C = C_p(H \text{ dot } N)^P$

Combining: $C = C_r (C_L + C_P * \max(0, N \text{ dot } L)) + C_P C_H (H \text{ dot } N)^P$
C_P is color of light, C_H is color of highlight
highlight color – (1, 1, 1) if plastic (white highlights), (1, 0, 0) if metal (reddish)

$(\cos \text{theta})^P = (R \text{ dot } E)^P$ for shiny surfaces

specular exponent p – medium → rough surface, high → smooth surface (between 0 and 1)

Shading Equation: where to apply it?
- Per polygon (flat shading)
- Per vertex (Gouraud interpolation)
- Per pixel (Phong interpolation)

Per Polygon Shading (flat)
-one normal for whole polygon
-viewer + light source are far from object
-shading changes abruptly at polygon boundaries

Per Vertex Shading (Gouraud)
-polygon edges smoothed off
Calculate N at each vertex
Apply shading equation for each vertex
- Color per vertex (interpolate color across polygon)
- $C_a = C + (c2 – c1) * (y – y1) / (y2 – y1)$ interpolation between c1 and c2
- C_b is similar to C_a on different edge of polygon
- C_p is between C_a and C_b : $C_p = C_a + (C_b – C_a) * (x – x_a) / (x_b – x_a)$

Per Pixel Shading
Phong interpolation – interp. surface normal of each vertex across polygon, then shade each pixel
-different surface normal at each point along an edge based on endpoints
-points inside polygon: interpolate surface normals of points on edges along scanlines
-use interpolated surface normal to shade pixel
-captures highlights better than Gouraud (glint/glare of objects)
-costs more to compute (more calculations per pixel for interpolation)

photoreceptors – light sensitive cells in retina
- rods – dark vision (mostly away from fovea)
- cones – light vision, 3 types: short, medium, long wavelength (mostly in fovea)
-give 3D color vision
-short wavelength are mostly blue, medium mostly green, long mostly red
-all cones trigger slightly for all colors, but short trigger more for blue, etc.

Spectral response: medium wavelengths receive green light, do so most strongly → green backgrounds give best visibility

CIE Chromaticity Diagram
-x, y coordinates correspond to visible light spectrum
-White in center of oval-ish shape, other colors go around perimeter
-**complementary colors** – mix to form white (across each other on oval-ish shape)
green-magenta, blue-yellow, cyan-red
Have RGB distributions that sum to 255, 255, 255 (approximately)

gamut– range of colors a device can display
subsection of chromaticity diagram (triangle where vertices are max. blue, red, green)

Additive display of colors - devices that produce photons of given wavelength (LCD, projector)

-red + blue = magenta, etc.

RGB color cube

-Green (0, 1, 0), Red (1, 0, 0), Blue (0, 0, 1), Black (0, 0, 0), other colors are along edges/inside cube

-Cyan (0, 1, 1), Magenta (1, 0, 1), White (1, 1, 1), Yellow (1, 1, 0)

-Along 0, 0, 0 to 1, 1, 1 is **grayscale**

Subtractive colors – materials that absorb photons (paint, ink)

-Primary colors become **cyan**, **magenta**, **yellow → CMYK** (black) for printers

-RGB are results of combining these primary colors, black produced by all 3

(R, G, B) = (1, 1, 1) – (C, M, Y)
red paint absorbs green and blue, cyan absorbs red

metamers –spectral distributions that look same to humans
Intensity ~ Brightness ~ Lightness ~ Value

HSV – hue, saturation, value

-value is vertical axis (0 is black, 1 is white)

-saturation goes from 0 to 1 (gray/washed out to vibrant): distance from center of cone

-hue is angle from 0 (counterclockwise)

red is 0, yellow is pi/3, green is 2pi/3, cyan is pi, blue is 4pi/3, magenta is 5pi/3

grayscale is vertical line at saturation 0 from value 0 → 1

-helps people select colors!

Note: HSV transforms into RGB color cube

HLS – hue, lightness, saturation

-double cone (one pointed down, another pointed up sitting on top)

-saturation is dist from center of cone (0 to 1)

-lightness goes from black (0) at bottom to white (1) at top of second cone

Rasterization + z-buffer vs. **Ray tracing**

fast	slow
GPU	not on GPU (uses CPU)
good image quality	great image quality
used in video games (faster)	used in film (have plenty of time)

Ray Tracing

Draw rays from eye through view plane to objects

for each pixel (x_s, y_s)

create ray R from eye thru (x_s, y_s)

for each object O in scene

if R intersects O_i and is closest to eye so far, record intersection (color, location)
// (only want closest b/c that's what's visible to eye)
shade pixel based on nearest intersection

z-buffer:
for object
for pixel <u>inside</u> objects boundary

<u>Ray Description</u> (parametric):
$x(t) = x_0 + t(x_1 - x_0) = x_0 + tdx$, same with y(t), z(t)
$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ (vectors)

-Objects are usually described implicitly (e.g. x^2 + y^2 + z^2 = 1)

(x0 + t * dx)^2 + (y0 + t * dy)^2 + (z0 + t * dz)^2 – 1 = 0 → solve for t to get when sphere intersects with ray
One intersection → tangent, none → no intersection

(x – a)^2 + (y – b)^2 + (z – c)^2 = r^2 → radius r, center at (a, b, c)

<u>Intersect Ray with Polygon</u>
1. Intersect ray with plane of polygon
2. Project triangle and point of intersection to 2D
3. Is point inside triangle? (point in polygon test)
Plane equation: ax + by + cz + d = 0 → replace x with x0 + t * dx, etc. → solve for t

If N = [a b c] dot d = [dx dy dz] = 0, then ray is parallel to plane and no intersection

Projecting triangle in 3D onto each plane: which projection has highest area?
-Examine surface normal N, drop coordinate with largest absolute value
-Other two planes are where the projection is largest

<u>Point-in-Polygon</u> (2D)
Crossing test: If point crosses polygon even number of times, it's outside (odd # → inside)
Messes up at corners
Winding number: connect pt to all vertices of polygon → acute angles formed by lines sum to nonzero is inside, zero is outside
<u>Half-plane test</u>: Use all edges of polygon as lines → get eqs ax + by = c
Check if point is on positive or negative side of each line
All positive → inside, otherwise outside
Do by = -ax + c, determine if right is greater/less than left
Less than → under, greater than → over

<u>Line Equation</u>
f(x) = ax + by + c = 0 with points (x0, y0) and (x1, y1)

V = (P1 – P0) / |P1 – P0| (unit length vector) → (vx, vy)

W orthogonal to V → W = (-vy, vx) so rotation from W = RV is R = [0, -1, 0; 1, 0, 0; 0, 0, 1]
W = (a, b) gives a and b in equation, c is distance to origin (plug in x0, y0)

Plane Equation
ax + by + cz + d = 0
-E1 = (P1 – P0) / |P1 – P0|, E2 = (P2 – P0) / |P2 – P0|
-Normal N = (E1 x E2)/|E1 x E2| → N = (a, b, c) for function
-Plug in one point to find d after finding a, b, c

Eye Rays
focal length – distance from eye to view plane

Screen pixel coordinates: (i, j) from 0 to w and 0 to h
View plane: (u, v) from -1 to 1 and -1 to 1
view plane coords = (u = 2i/w – 1, v = 2j/h – 1)

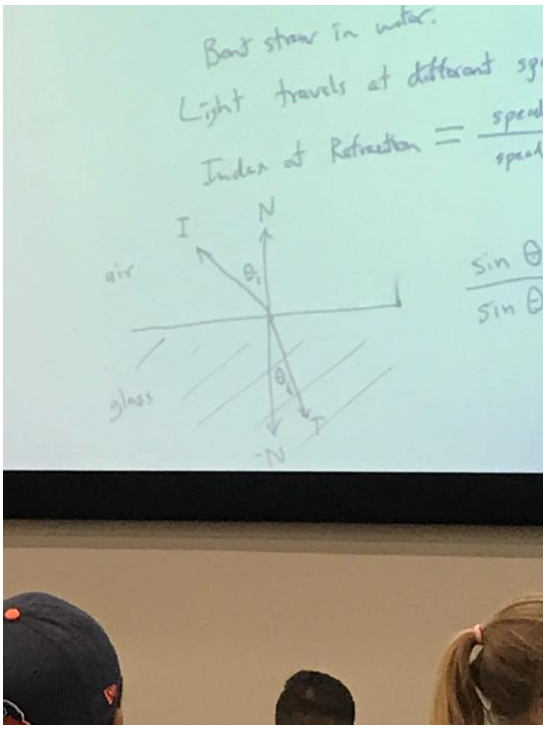
Ray origins are e (eye location), direction is -d**w** + u**u** + v**v** (w, u, v are **vectors**)
d = focal length
u, **v**, **w** are orthonormal coordinate frame
Field of view angle theta is angle made by eye rays to top and bottom of view plane
→ tan (theta / 2) = 1 / d

Recursive Ray Tracing
Reflection (mirror): draw ray to mirror, then draw ray from mirror to object at reflection angle
R = 2(N dot E)N – E, assuming N and E are unit length

Color of eye ray = ambient + diffuse + specular + $K_{reflection}$ * $C_{reflection}$ (perfectness, color of mirror) + $K_{transmitted}$ * $C_{transmitted}$ (colors from other rays)

When to stop recursion with multiple reflective surfaces?
- At certain depth # of recursions
- When contribution to 1st ray is small (product of $K_{reflection}$ is small)

Transparent Surfaces
Light travels at different speed through different media
- Index of Refraction = speed of light in vacuum / speed of light thru material
air: 1.0003, water: 1.33, glass: 1.5, diamond: 2.4, ice: 1.3



Shiny surface → reflected ray
Transparent surface → transmitted ray (bent)

$\sin \text{theta}_1 / \sin \text{theta}_2 = \text{index}_2 / \text{index}_1$ for angles to surface normal with light angle

Surface between air and glass: <u>critical angle</u> of light impact so that no light escapes (total internal reflection)

<u>Hard Shadows</u>
-Point light source casting shadow of object onto a surface
-When is ray to surface blocked by object?

Eye ray <u>from eye</u> intersects with shadow ray <u>from light source</u> (determines color of eye ray)
Object completely blocks out light

$C = C_L C_R$ (N dot L) * Visible(P, L) Visible is 0 if blocked, 1 if unblocked

<u>Soft Shadows</u>
Area light source shining through an object onto <u>diffuse</u> surface

Umbra – total shadow
Penumbra – partial shadow (cast by one side of light source hitting opposite side of object, then surface)

Shoot n rays from light source through object

→ Color = 1/n * sum from i = 1 to n of C_{Li} * visible(P, L_i) (if point is visible to that ray)
-Average together contributions of each ray → **Distribution Ray Tracing**

Glossy Reflection
Instead of one eye ray reflecting off of perfectly smooth surface, one eye ray turns into several reflecting off of a glossy surface

Traditional: C = amb + diff + spec + KreflC_R
Distribution: C = amb + diff + spec + 1/n * sum from i = 1 to n of Krefl * C_{Ri}

Motion Blur
Distribute rays in time, as object moves through a space (sometimes ray hits it, sometimes not)

Traditional: C = $C_l C_R$(N dot L)
Distribution: C = 1/n * sum from i = 1 to n of $C_L C_{Rt}$ (N_t, L_t)

Add a primitive:
-Intersects ray with object → return intersection point, surface normal
-Return bounding box

Fast to ray-trace: sphere, triangles/polygons, cylinder, box, ellipsoid
Slow to ray-trace: torus, fractals, blobby spheres, cubic patch (curved sheets)

```
for each pixel
    create ray thru pixel
    for each object in scene
        intersect ray with object
```

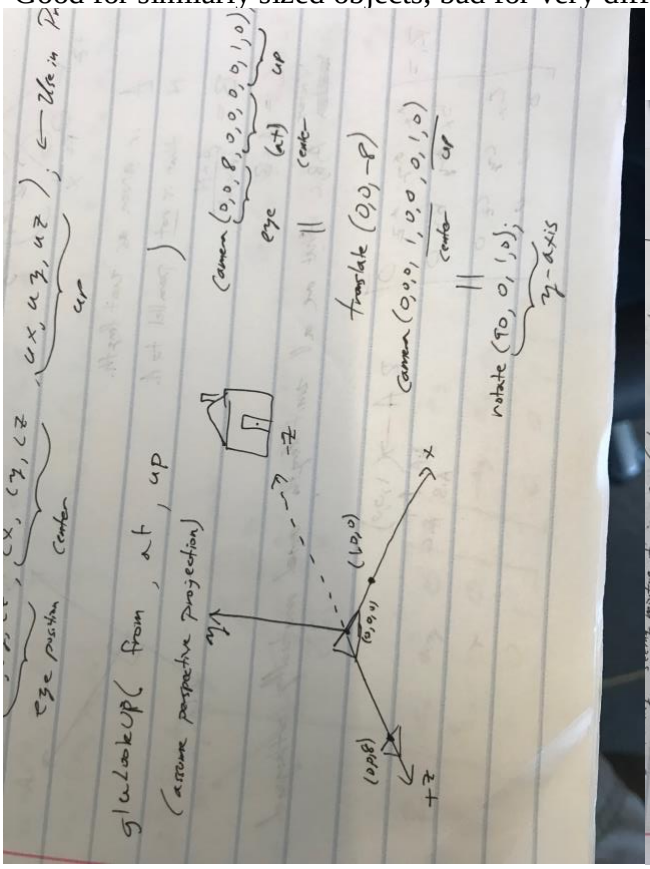
Bounding Volumes → fewer expensive tests
-Put box around object, see if ray intersects it (if not, don't worry about the object)
-Want tight bounds to minimize false positives
-Speeds up ray tracing of one object

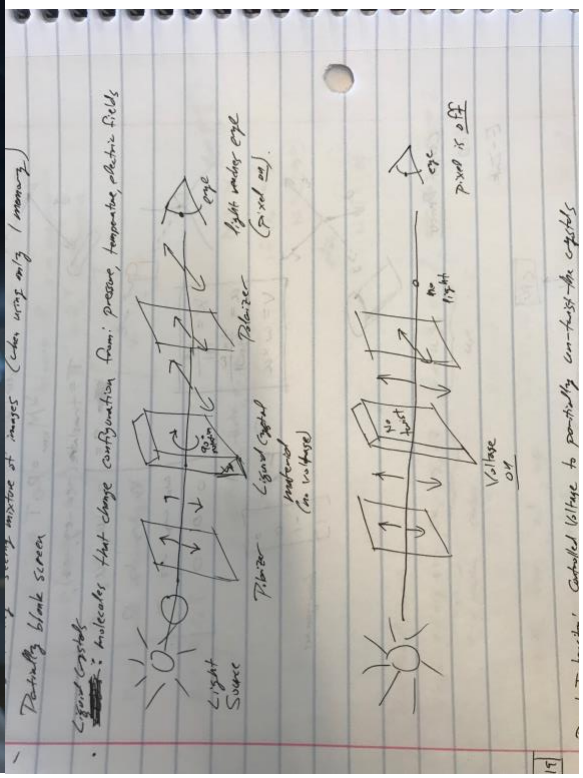
Bounding hierarchy: One box contains two subboxes, each containing their own subboxes, leaves of the tree are objects

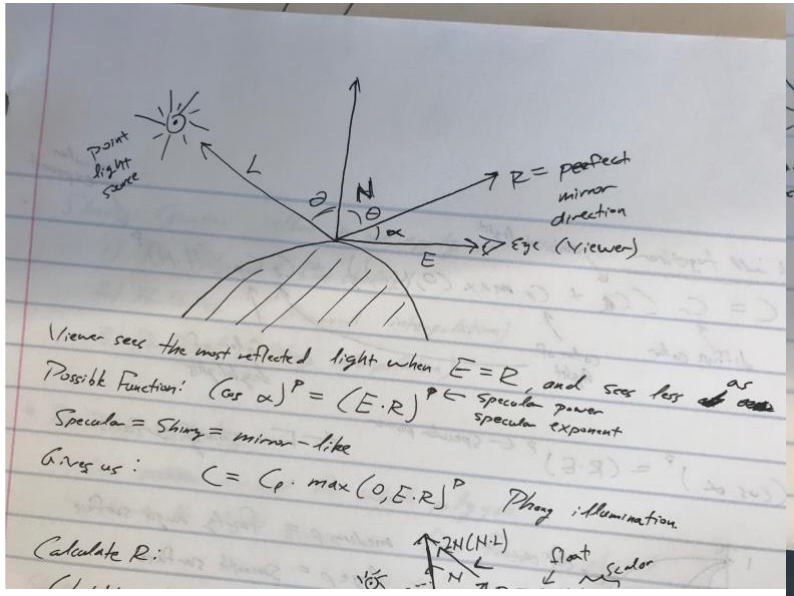
```
hierarchy_traverse(ray r, node n) {
    if r intersects n's bounding volume
        if n is leaf, intersect r with ns object
        else for each child c of n, hierarchy_traverse(r, c)
```

Building Boundary Hierarchy
1. bottom up: pair 2 nearest objects until you're done with that, then move onto boxes containing smaller boxes, etc.
2. top down: box all objects, then divide box into 8 parts → boxes around objects in each part

Grids (Spatial Partition)
-Divide space into grid of cells
-Each cell has list of objects partially/completely in it
-Move from cell to cell similar to rasterization
-Good for similarly sized objects, bad for very different sizes







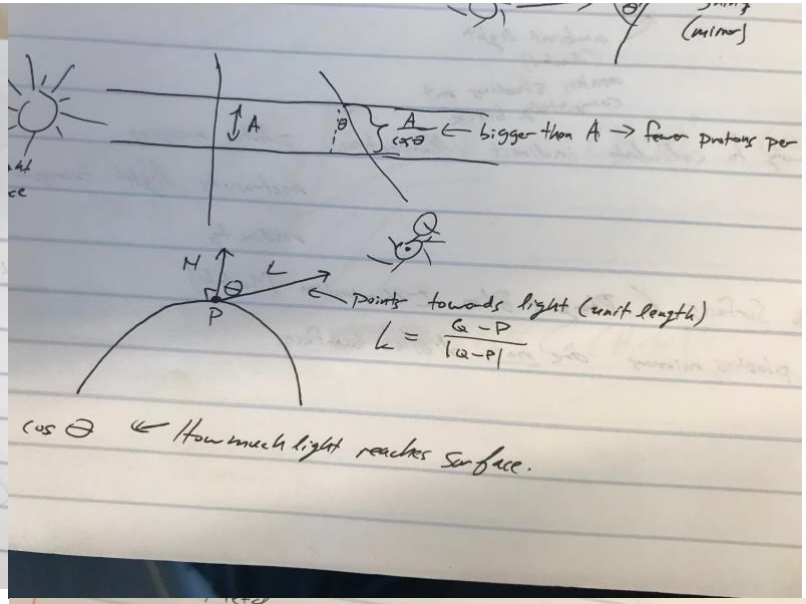
Viewer sees the most reflected light when $E = R$, and sees less as

Possible Function: $(\cos\alpha)^p = (E \cdot R)^p$ ← specular power / specular exponent

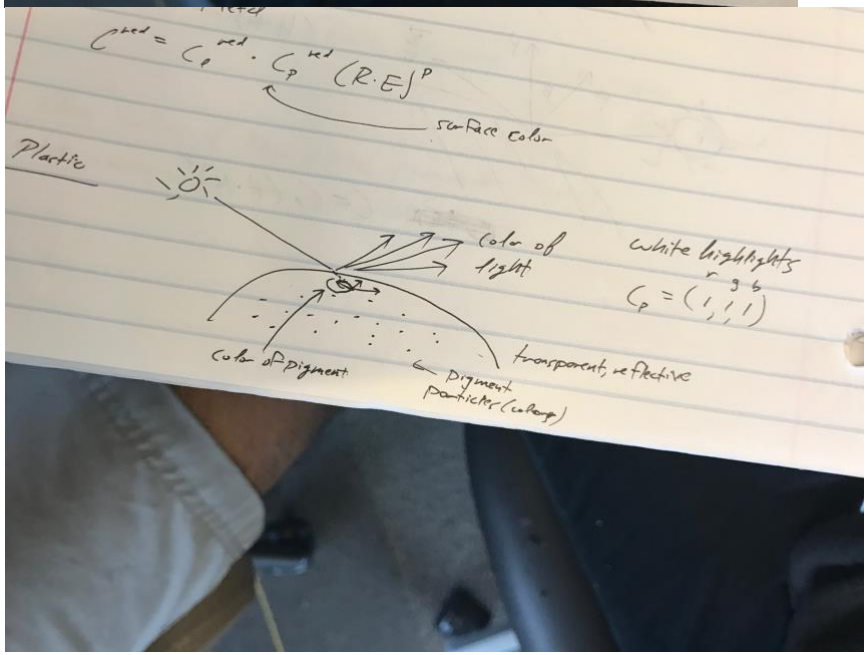
Specular = Shiny = mirror-like

Gives us: $C = C_p \cdot \max(0, E \cdot R)^p$ Phong illumination

Calculate R:



cos θ ← How much light reaches surface.



Shadows using Rasterization
1. Shadow volumes
2. Shadow mapping

visible(P, Q) = visible(Q, P) shadows → visibility
N dot L < 0 → faces away from light
Attached shadow – easier, not hidden by anything just not visible (backside of objects)
vs. cast shadow – blocked by an object, need to do more visibility checks to light source

Shadow mapping: **2-pass z-buffer** (Lance Williams 1978)
1. Render scene from light source
2. Render scene from eye/camera
3. Find which pixels in 2 are hidden from light using 1

World Space (3D) → eye space (2D, z-buffer & color buffer) or light space (2D, z-buffer)
Want to go from P = (x, y, z) in eye space to P' = (x', y', z') in light space

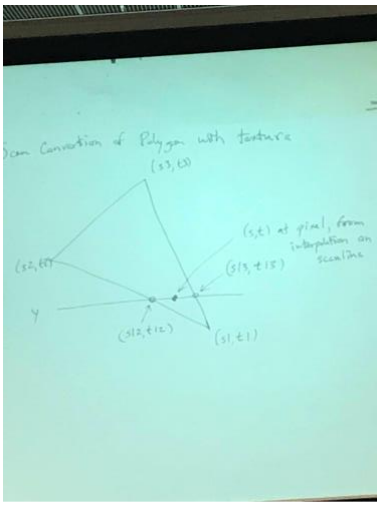
-Transform visible pixel coordinates to light space (P → P')
-Test z-value (z') against light's z-value at (x', y')
$z' < z_{light}(x', y')$ → shadow

$P' = LV^{-1}P$ (V = function to eye space, L = function to light space)

This process is expensive!

Texture Mapping
Texture space – color at each texel
Screen space – what's seen on screen
Use **texture lookup** to get correct texture for point on screen from texture space

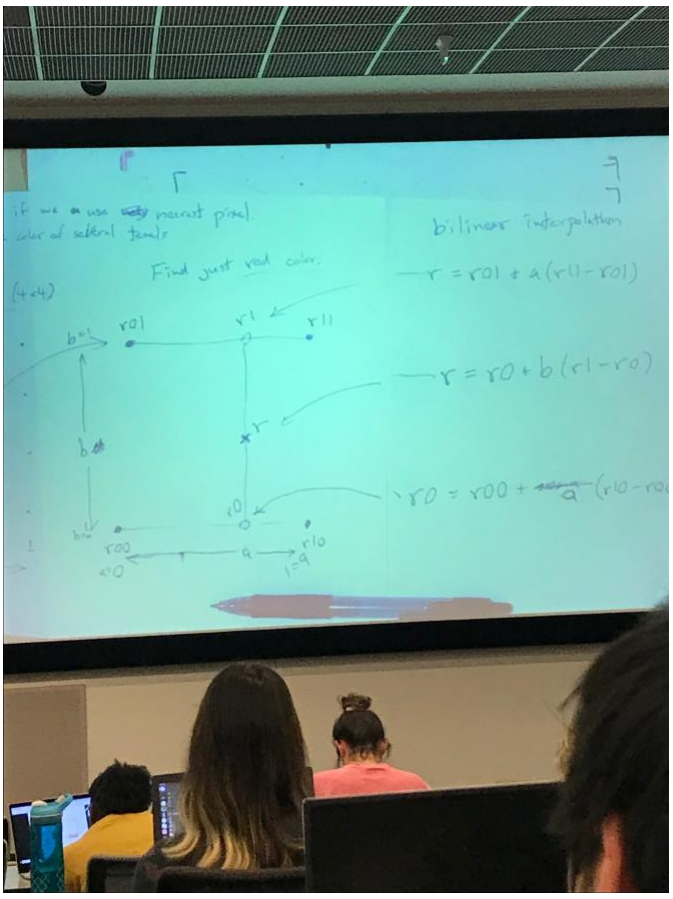
1. Interpolate correct coordinates in texture space across polygon (during rasterization)
   - Texture space goes from 0 to 1 on s and t axes, map to surface in screen space
2. Find color from texture (texture lookup)
3. Color pixel based on texture + shading model of surface (e. g. diffuse)



What s, t to use for interpolation?
x' = x/|z|, y' = y/|z|, z' = 1/|z|
1. divide s and t by |z| → s', t'
2. interpolate s' and t' linearly during rasterization
3. divide by z' per pixel (mult. by z)
4. texture lookup + shading

Using nearest pixel → texture looks blocky
Solution: Interpolate color of several texels!
Take weighted average of colors of 4 nearest texels

$r_x = r_{01} + a(r_{11} - r_{01})$
$r_y = r_{00} + b(r_{01} - r_{00})$
where a, b are in [0..1] (dist from r to bottom left of square)



This is <u>bilinear interpolation</u>: important for texture magnification (blowing up image)
vs. texture minification: creates sparkle (bad)

Solution: create multiple resolution textures (image pyramid, MIPmap)

MIPmap to prevent noise/sparkle in background of images → GPUs

128 x 128 → 64 x 64: average every square of 4 pixels to make 1 pixel in smaller image

<u>trilinear interpolation</u>: interp. between x, y coords, and between levels of minimization

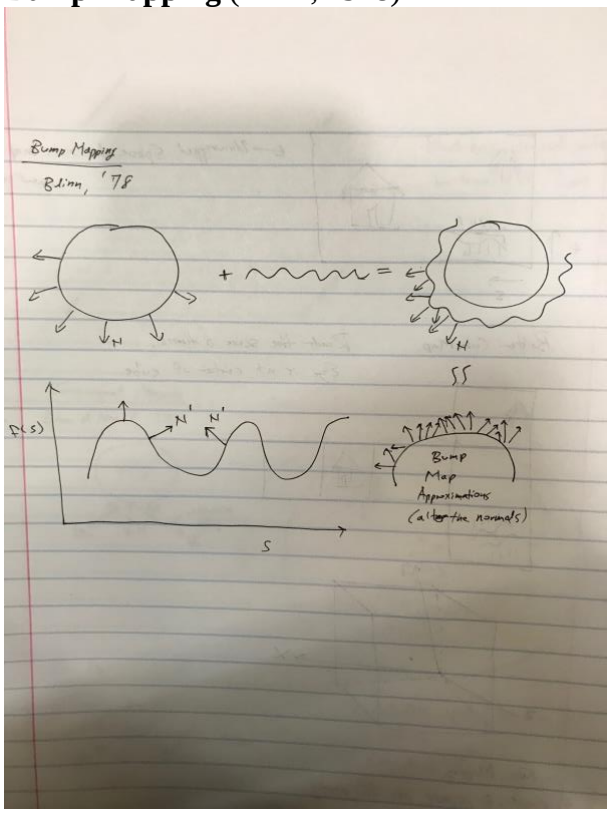
Cube Mapping – render scene 6 times, eye at center of cube

Environment Mapping (Blinn + Newell, 1976)
-good if objects are far away

Shiny surface: Use reflective ray that bounces off at same angle, and normal vector
Ray: direction (rx, ry, rz)

1. Calculate vector R from N and V (where V is ray reflected off of surface)
2. Use R to look up color from environment map

Bump Mapping (Blinn, 1978)



Add bumps to surface → normals get changed

Fake normal N' = N – f'(s) * T (derivative of bump equation to get normal at that point)
Tangent: [1, 0] (if flat, then this is tangent)

2D Bump Mapping: have map with u and v directions
*** fake bumps**: <u>no change to object silhouette</u> *
B_u is slope in u direction, B_v is slope in v direction (dB/du, dB/dv)
Bu, Bv play same role as f'
P_t is tangent direction (tilt in this direction)
$N' = N – (B_uP_s + B_vP_t)$ B's are scalars, others are vectors
Ps, Pt are tangent lines, N is regular normal

GPUs – Graphics Processing Unit
Host CPU → enter GPU at vertex processors
-Send graphics primitives (e.g. triangles)
-Vertex processors do transforming, shading, 2D projection

Vertex processors → send screen-space primitives to rasterizer (scan conversion)

Rasterizer sends fragments (almost-pixels) to fragment processors (shade, texture, transparency)
Frag. processors access texture mapping by sending texels

Fragment processors → final image (sends pixels)

GeForce 6800 (2004)
Vertex processors (6 units)
3D transforms, per-vertex shading, operate independently of each other (MIMD), 4 mults/adds vector unit, scalar sqrt, sin, cos, etc., user programmable
Fragment processors (16 units)
per-fragment shading, texture mapping, 2 vector units (4 mults/adds), operate in lockstep (SIMD), user programmable
Rasterizer
fixed-function (can't program), interpolates attributes across polygon, creates fragments (per-pixel data from primitive: color, depth, texture coords)

GeForce 8800 – unified graphics architecture (vertex + frag. processors are same)
128 streaming processor cores (SP)
add and multiply, single integer operations
16 streaming multiprocessors (SM) – independent instruction streams
8 SP's, 2 SFUs (special function units) for sqrt, cos, exponent, log

GeForce 2080
18 bil. transistors, 4608 cores (72 streaming multiprocessors w/ 64 float pt. cores each)

GLSL – OpenGL Shading Language
	C, C++, Java	"look and feel"
	float, int, bool
	vec2 (2D vector: texture coords, etc.), vec3, vec4 (rgba for color + transp. / coverage)
	mat3, mat4 for transformations
	uniform sampler2D – texture map
	operators: * is matrix multiplication, inversesqrt = 1 / sqrt, floor, ceil, pow, etc.
	length, dot, normalize, reflect
	texture2D(pass in sampler, texture coord)
DirectX – HLSL

Input (vertex info): gl Vertex (3D), gl Color (RGB), gl Normal, texture coords

→ Vertex Program → output: gl Position (2D), gl FrontColor, gl BackColor, texture coords

→ Rasterizer: interpolates values across polygon, creates input to fragment programs

→ Input to fragment program: gl Color, texture coords

→ Fragment Program → Output: gl Fragment color

Vertex Program (diffuse shader)
```
varying vec3 normal
varying vec3 vertex_to_light (points to light source)
void main() {
	gl Position = gl ModelViewProjectionMatrix * gl Vertex (transform + project into 2D)
	normal = gl NormalMatrix * gl Normal
	vec4 vertex_modelview = gl ModelViewMatrix * gl Vertex
	vertex to light = vec3(gl Light Source[0].position – vertex_modelview)
}
```

Fragment Program (diffuse shader)
```
# PROCESSING COLOR SHADER (tells processing to stick some stuff in front of code)
varying vec3 normal
varying vec3 vertex_to_light (points to light source)
void main() {                                          // in separate class
	const vec4 diffuseColor = vec4(1, 0, 0, 1) // red
	vec3 n normal = normalize(normal)
	vec3 n vertex to light = normalize(vertex to light)
	float diffuse = clamp(dot(n normalize, n vertex to light), 0, 1) // <0 → 0, >1 → 1
	gl Frag Color = diffuse * diffuseColor
}
```

C = red * (N dot L)

Vertex Program – Twisting

```
void main() {
        vec4 pos = gl.ModelViewProjectMatrix *gl_vertex
        vec4 center = vec4(512, 512, 0, 0)          ← center of 1024x1024 2D space
        vec2 diff = vec2(pos.x – center.x, pos.y – center.y)
        float angle = twist * length(diff)
        c = cos angle, s = sin angle
        gl_position.x = c * pos.x – s * pos.y
        gl_position.y = s * pos.x + c * pos.y
        gl_position.z, gl_position.w = pos.z, pos.w
        glFrontColor = gl_BackColor = glColor          // copy color to output
}
```

Moving outward from center, vertices get twisted counterclockwise more and more

Input to Fragment: gl Color, texture coords → output gl.FragColor

Vertex Program – Texture

```
#define PROCESSING_TEXTURE_SHADER
varying vec2 texture_coord
void main() {
        gl_Position = gl_as;ldkfjsadf * gl_Vertex
        textureCoord = vec2(gl_MultiTexCoord0)
}
```

Fragment Program – Texture

```
varying vec2 textureCoord
uniform sampler2D myTexture
void main() {
        gl_FragColor = texture2D(my_texture, texture_Coords)
}
```

Vertex Program – Two Textures

```
varying vec2 leftcoord, rightcoord
void main() {
        gl Position = gl asd;lfkjasdf * gl_vertex
        vec2 textureCoord = vec2(gl_multiTexCoord0)
        leftcoord = textureCoord + vec2(-0.2, 0)          // on [0, 1] scale
        rightCoord = textureCoord + vec2(0.2, 0)
}
```

Fragment Program – 2 Textures

```
#define PROCESSING_TEXTURE_SHADER
varying vec2 leftCoord, rightCoord
uniform sampler2D myTexture
void main() {
        vec4 leftColor, rightColor = texture2D(myTexture, leftCoord/rightCoord)
        gl FragColor = 0.5 * (leftColor + rightColor)
}
```

Splits one circle with 2 textures into Venn diagram with each texture

Coloring a Ray – input ray to color routine, output color
- -color routine sends ray to ray/scene intersection routine, which returns Hit object with t, point of intersection, normal, surface info
- -send hit object to shader, which returns color to color routine to return to user
- -shader can send reflected ray back to color routine as well

B = (xb, yb, zb) is vertex of cone
Cones: $(x-xb)^2 + (z-zb)^2 = (k(y-yb))^2$ (circle at each height)
Get a, b, c → solve for t
y = yb + h = y0 + t * dy → solve for t → d = dist(intersection pt. P, center C of cone base)
d < radius? → Normal N = (0, 1, 0) **always**

T_2: tangent alongside surface of cone
P: point at base of cone
N: normal vector from P pointing perpendicular to cone surface
D: vector from center of base to start of P
B: coordinates of tip of cone

D = (px – bx, 0, pz – bz)
T_1 = D rotated by pi/2 = (-dz, 0, dx)
T_2 = B – P
$N = T_1 \times T_2 / ||T_1 \times T_2||$

---

polyhedron – surface composed of polygons (made of vertices + edges)
Used to approximate smooth surfaces
manifold – looks like (possibly bent) plane everywhere on surface
can walk around a vertex and come back to starting point
manifold edge – edge where all points satisfy manifold property (ex shared base of cones)
boundary edge – like edge of a half-plane (on outside of manifold)

manifold surface (ex cube) – all edges are manifold, all vertices are manifold
manifold with boundary (ex remove two faces from cube) – all outer edges are boundary edge

Laplacian smoothing – make sizes of polygons similar by moving one vertex closer to others
- improves triangle shape, but shrinks surface
- v = 1/5 * (v1 + v2 + v3 + v4 + v5)

Face subdivision – subdivide edges into segments, connect them to form inner polygons
- create geodesic sphere from octahedron

Triangulation – on each face, divide it into triangles by drawing s – 3 edges from one vertex

Polygon Soup – no connectivity between faces

```
class Face {
    float x1, y1, z1, x2, y2, z2, x3, y3, z3
}
```

polyhedron = list of faces (common file format to feed graphics hardware)

shared vertex/indexed face set – common as file format (.obj)
- -list of vertices, then list of faces that reference 3 vertices at a time
- -more complex & compact than soup, but has face connectivity
- -vertices usually used by 6 diff. triangles

corners (Rossignac) – triangles only
- -corner: association between triangle and vertex (one for each vertex of each triangle)
- -Geometry Table (G): list of 3-coordinate vertices
- -Vertex Table (V): list of integers, indices 0-2 make up triangle 1, 3-5 are triangle 2, etc.
    - -values are vertex IDs for each corner, build triangle with refs. to corners in G
    - -order matters for corner values in each triangle (clockwise/counterclockwise)
- -corner.next = 3 * (c / 3) + (c + 1) mod 3
- -corner.previous = corner.next.next

adjacency with O table
- -for each corner, store corner.opposite (integer ID of opposite corner in an adj. triangle)
- -Table: at each index, have vertex associated with and opposite corner of that index

Computing O table from V table

```
for each corner a
    for each corner b
        if (a.next.vert == b.prev.vert and a.prev.vert == b.next.vert) O(a) = b, O(b) = a
```

Neighbors

right neighbor c.right = c.next.opposite
left neighbor c.left = c.prev.opposite
swing c.swing = c.next.opposite.next ← allows moving between corners surrounding a vertex

Read: Curves in textbook

Convex Hull – smallest shape containing all pts. in a set

convex combination – given two points p1, p2, pt. a can be written in the form $a = w_1p_1 + w_2p_2$
- If $w_1 + w_2 = 1$ and they are nonnegative, a is on p_1p_2 line segment
- 3 points p1, p2, p3 → b = xp1 + yp2 + zp3 is inside triangle if x + y + z = 1, x, y, z >= 0
- **In general**: p1 … pn, then q = lin. comb. of p_i points is inside convex hull if sum of weights = 1 and all weights >= 0

For a triangle, weights w1, w2, w3 for point q are barycentric coordinates
-Draw lines from each vertex of triangle to q → get 3 smaller triangles
-w1 = a1 / A, w2 = a2 / A, w3 = a3 / A, where ai = areas of triangles opposite p1, p2, p3

Bézier Curves
Degree 1 Bézier are line segments, we care about degree 3

p1, p2, p3, p4 → Bézier curve hits p1, p4 and curves towards p2, p3

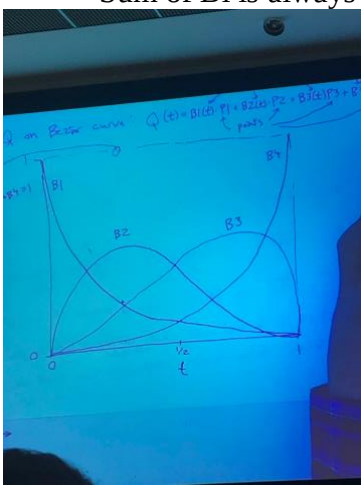
1. Bezier curve interpolates (passes thru) p1, p4
2. Tangent at endpts. to segments P1-P2 and P3-P4
3. Pts. on curve are inside convex hull of P1, P2, P3, P4
4. Cubic Bezier curves give indep. endpoint + tangent controls (lowest degree that this works)

Can chain together cubic Bezier curves by using endpoint as start pt for next curve

Parametric Line Segment
Q(t) = [x(t), y(t)] for x, y coords of two points
x(t) = x1 * (1 – t) + x2 * t
y(t) = y1 * (1 – t) + y2 * t
Q(t) = P1 * L1(t) + P2 * L2(t), where L1 = 1 – t, L2 = t (moving along P1-P2 segment)

Cubic Bezier: 4 control points, 4 basis functions B1, B2, B3, B4

Q on Bezier curve: Q(t) = B1(t) * P1 + B2(t) * P2 … where B1, Bi(t) are scalars, Pi are points
Sum of Bi is always 1 → convex combination → stays in convex hull of 4 points

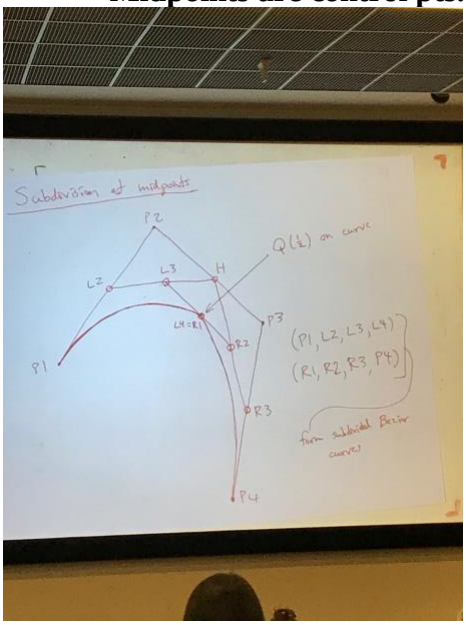


$B1(t) = (1 - t)^3$
$B2(t) = 3t(1 - t)^2$
$B3(t) = 3t^2(1 - t)$
$B4(t) = t^3$
B1, B4 and B2, B3 symmetric around t = ½

Subdivision at midpoints – draw lines between midpoints of each of 3 lines, then create midpoints for those lines and connect them, then midpoint of that line is Q(½)

-**Bezier curve goes through that pt.**

-**Midpoints are control pts. for two subdivided (recursive) Bezier curves**



Uses for curves: font, architecture, drawing diagrams/art

General subdivision: do ¾ point instead of midpoint → Q(3/4), etc. for anything between 0 and 1 → construct entire Bezier curve!

<u>Quadratic Bezier</u> – only 3 points → 2 lines instead of 4 points

Midpoint subdivision creates one line whose midpoint the curve goes through

<u>Polynomial Evaluation</u>

f(t) = at^3 + bt^2 + ct + d t^3 = t^2 * t, t^2 = t * t

5 multiplications, 3 adds

Horner's Rule: ((at + b)t + c)t + d → 3 mults, 3 adds

Linear: f(t) = at + b

f(h) = a * h + b, f(2h) = f(h) + ah, each time add ah → 1 add, 0 mult

Quadratic: f(t) = at^2 + bt + c

f(t + h) = f(t) + 2ah + ah^2 + bh, since the latter part is linear it can be done with one add → quadratic takes 2 adds, 0 mult

finite difference method

Draw Bezier Curves
1. Direct evaluation of cubic polynomials (Horner's rule)
2. Finite difference (fast)
3. curve subdivision

Complex Numbers
$|z| < 1$ → z^2, z^3, z^4, etc. spiral inward toward origin (angle changes same amount each time)

$f(z) = z^2 + c$ → Do iterations of f stay near origin or diverge?
Based on c → **Julia set** for that value of c

Box from (-1.5, -1.5) to (1.5, 1.5) just chillin in the notes

Julia Set
Different values of c → different shapes
z_0 – start value of z comes from screen position (texture coordinates)
Start with one value of c for whole picture

If z0 = (0, 0) iterations stay near origin, Julia set is connected (otherwise, fractal dust)

Parametric Surfaces
$Q(t) = p1 * (1 - t) + p2 * t$

Ex: parallelogram (1 x 1 square → transform to bottom left vertex with vectors to top left and bottom right)
Take 0 → 1 in t, 0 → 1 in s and transform it to (x1, y1, z1)

$x(s, t) = x1 + s * diff(x2, x1) + t * diff(x3, x1)$
y(s, t) is similar but with y values, z is similar but with z values

Ex: cylinder (1x2pi t x s square → cylinder coordinates)
t → height, s → rotation angle
$x(s, t) = cos(s), y(s, t) = sin(s), z(s, t) = t$

Ex: sphere (t from -pi/2 to pi/2, s from 0 to 2pi)
t = 0 corresponds to equator, pi/2 is north pole
s value is which longitude point is at
$x(s, t) = cos(t) * cos(s), y(s, t) = cos(t) * sin(s), z(s, t) = sin(t)$

Cubic Bezier Patches - Bezier inside Bezier
-4 Bezier curves → Bezier patch (16 control pts. in 3D) by "sweeping" through the curves
-2 parameters: s, t in [0, 1] with t controlling point along all curves, s controlling which curve
s = 0 → P0, s = 1 → P3, s = 0.8 = partway between P2 and P3, etc.

Calculate 3D Q(s, t):
1. Create 4 Bezier curves G1, G2, G3, G4
2. calculate 4 points on curves P1(t), P2(t), P3(t), P4(t)
3. create new Bezier curve for 4 pts.
4. Calculate Q(s, t) on that curve from s → Q on Bezier patch

Problems with Bezier Patches:
- Must chop up surface into quad patches
- Making continuous surface is messy (many control point constraints)

Subdivision surfaces – first appeared 1978 (Catmull-Clark, Dao-Sabin) → advances in 90's (Tony de Rose, Jos Stam)
- Can turn any polygon into smooth surface
- Triangles, grids, polyogons, etc. okay
- Continuity (smoothness) is free
- Easy to program

Loop Scheme: (triangles only) C^2 continuous almost everywhere (second deriv. is continuous)
- Subdivide adjacent triangles into smaller ones at midpoints of each edge

1. Compute positions of new vertices on edges
2. Move positions of old vertices
3. Make smaller triangles

Place new vertex on edge between two triangles: $v = 3/8 * (v1 + v2) + 1/8 * (v3 + v4)$
- v1, v2 are endpts of the shared edge

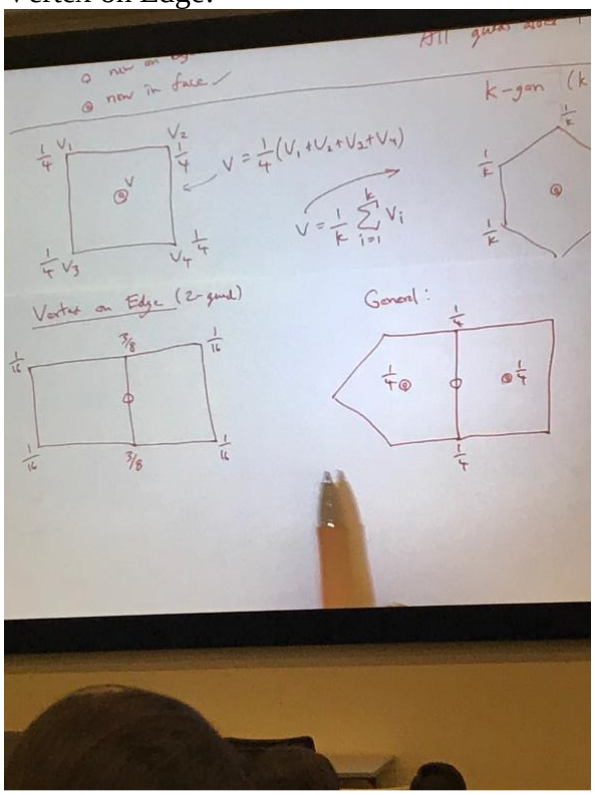
Move old vertex (**valence** k - # triangles around it): $v' = (1 – kB)v + B(v1 + v2 + … vk)$
- $B = 3/(8k)$ for $k > 3$, $3/16$ for $k = 3$

Note: Most vertices will have valence 6, all others called extraordinary pts. (C' continuous here)

Catmull-Clark (mostly quadrilaterals) C^2 continuous nearly everywhere
-Create new vertex at center of mass of each face, connect to midpts. of each edge → all quads

k-gon: center $v = (v1 + v2 + … + vk)/k$

Vertex on Edge:



Move Old Vertices:
Let E = (e1 + e2 + … + ek)/k, where ei are midpts. of edges to center from each vertex
Let F = (f1 + f2 + … + fk)/k, where fi are centroids of new polygons created by edges
v' = (E + F + v(k-2))/k
Note: Most Vs will have valence 4, others are extraordinary, only C' continuous at these places

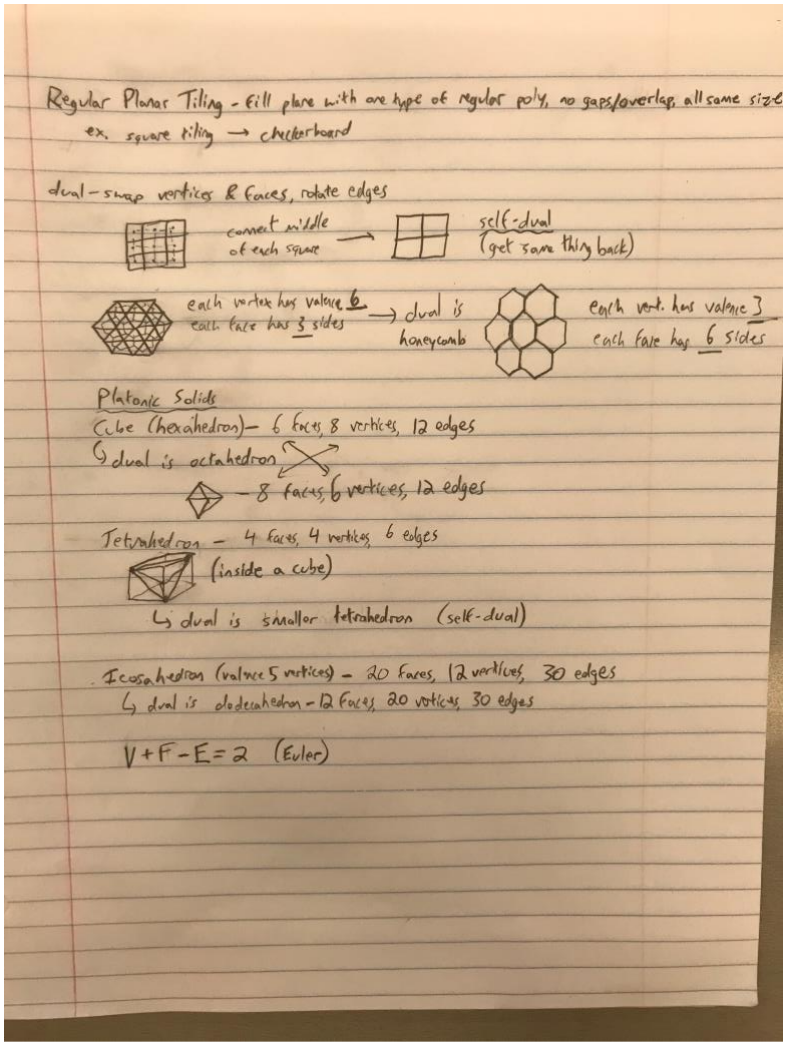
Sharp Crease
"Tag" edges to be sharp (ex. edge of table)

Vertex with 2 sharp edges: only controlled by the vertices on other end of sharp edges
v' = (6v + v1 + v2)/8 where v1, v2 are the sharp edge vertices

Semi-Sharp Edges
s = 0 → not that sharp, s = infinity → infinitely sharp (corner)

Use sharp edge rule s times, then smooth rule after

Regular Polygons = convex, equal sides & angles



Planar tiling platonic solids

Fractal – infinitely detailed geometric object, approximate natural phenomena, math. curiosities
use recursion, subdivision, iteration

Georg Cantor (1883) – line segment subdivision
Start with line segment, remove middle third, remove middle thirds of subsegments, etc.
Result is Cantor Dust (infinite # points, fractal "dust") with **measure 0** (sum to 0)

Sierpinski carpet (1916) – remove middle ninth of square, middle ninth of other eight, etc.
Still one unbroken object with holes everywhere

Peano/space-filling curves (1890) – draw snakelike curve through a unit square → subdivide into 9 squares and draw curves through those + connect them → continue down levels (9 → 81, etc.)
infinitely wiggly, passes through all points in square

von Koch snowflake (1904) – replace middle third of segment with triangular part, then repeat for all segments

infinite length, tangent is never defined

Fractal Dimension

-Square linearly scaled 2x → 4 copies

-Haussdorf-Besicovich (HB) Dimension D = log(replication factor)/log(linear scale factor)

von Koch snowflake - replication factor is 4 (4 copies when done once), linear scale factor is 3 (each replica is 1/3 size of original)

Mandelbrot – IBM research fellow, coined term fractal (70's), connected fractals and nature

coastlines, veins and arteries, streams

Carpenter: fractal mountains

- 1D: divide line segment, move midpt to random height & repeat for subsegments
range for random heights is ½ for first iteration, then ¼, 1/8, etc.
- 2D: triangle → draw middle triangle, non-corner vertices (midpts.) up/down like in 1D
Repeat within each triangle for next iterations

Mandelbrot Set – fix z0 = (0, 0), test different c values for convergence

Values of c for which Julia set is connected

Volume Rendering

volume data – values given at collection of grid points (3D)

-usually 3D rectilinear grid, 128^3 or 512*512*1024 or something

-each grid element is **voxel** (volume element), holding one float value

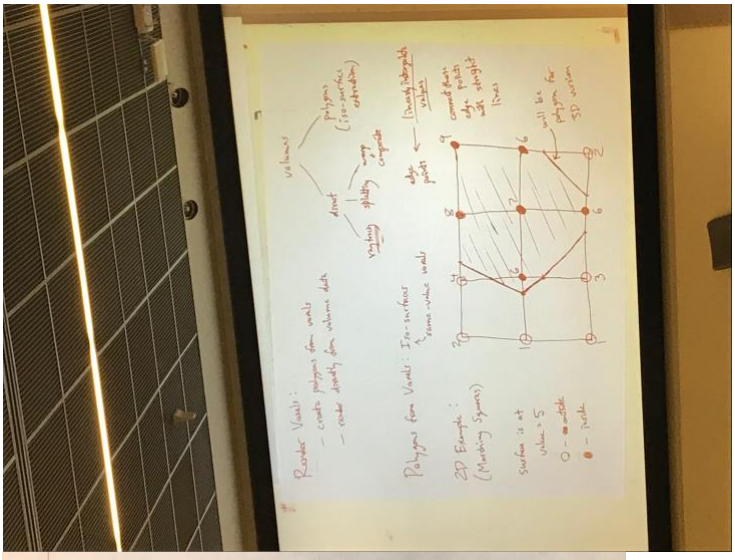
Data from: CAT scan data – permeability to x-rays, MRI –hydrogen in parts of body, Fluid sims

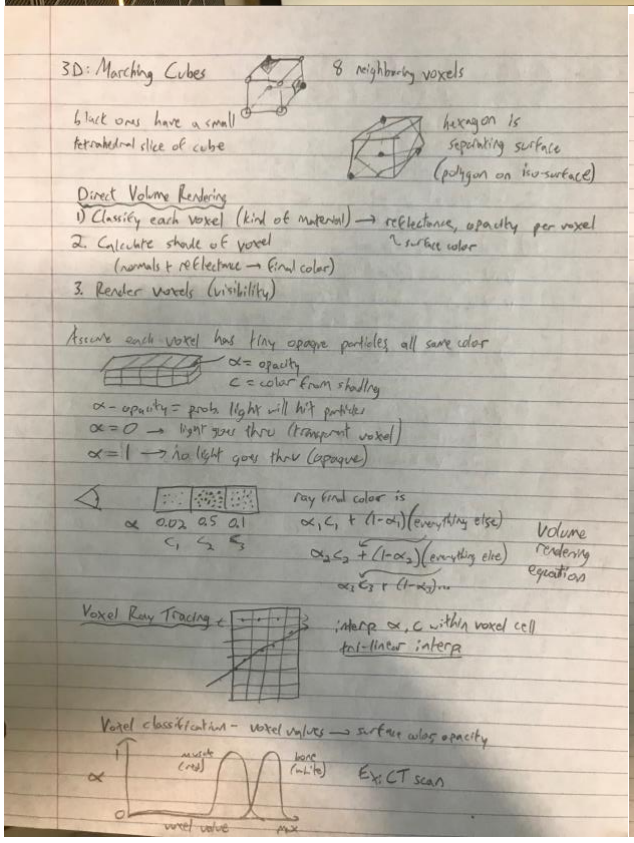
Rendering voxels: create polygons from voxels, render directly from volume data

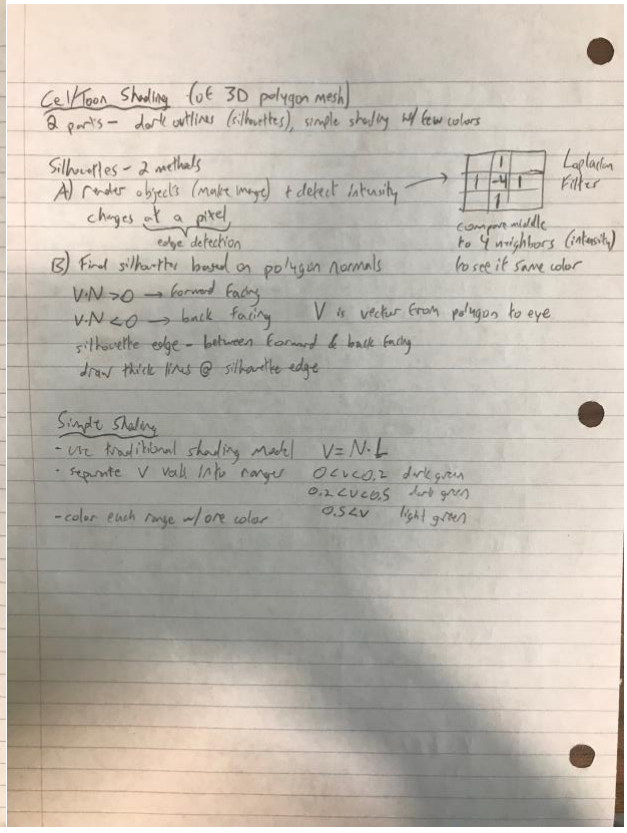
volumes → direct (ray tracing, splatting, warp & ??) or polygons (iso-surface extraction)

polygons from voxels – iso-surfaces (same-value voxels)

2D Example (marching squares): each voxel has value, want surface at value >= 5







Marching Cubes

The algorithm proceeds through the scalar field, taking eight neighbor locations at a time (thus forming an imaginary cube), then determining the polygon(s) needed to represent the part of the isosurface that passes through this cube. The individual polygons are then fused into the desired surface.

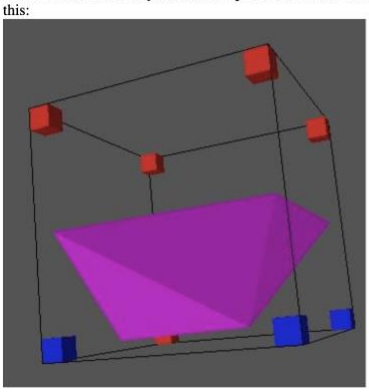
This is done by creating an index to a precalculated array of 256 possible polygon configurations (2^8=256) within the cube, by treating each of the 8 scalar values as a bit in an 8-bit integer. If the

scalar's value is higher than the iso-value (i.e., it is inside the surface) then the appropriate bit is set to one, while if it is lower (outside), it is set to zero. The final value, after all eight scalars are checked, is the actual index to the polygon indices array.

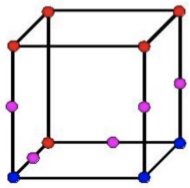
Finally each vertex of the generated polygons is placed on the appropriate position along the cube's edge by linearly interpolating the two scalar values that are connected by that edge.

The gradient of the scalar field at each grid point is also the normal vector of a hypothetical isosurface passing from that point. Therefore, these normals may be interpolated along the edges of each cube to find the normals of the generated vertices which are essential for shading the resulting mesh with some illumination model.

Which doesn't look very much like a piece of a circle. Instead, what we want to do is triangulate the cube where filled triangles will represent the surface passing through the cube. Something like this:



Implementing the algorithm in 3D works much the same as it did in 2D. For slice data like the Visible Human Male dataset, you stack the slices in 3D, knowing each slice is 1mm or 3 pixels appart. In order to be able to test the vertices of each cube, you must choose your cube size to align with the slices, either using 1mm cubes (or rectangles 1mm high and 1/3rd of a mm thick since pixels are only 1/3rd of a mm wide), or some multiple of 1mm cubes (ie 2mm or 3 mm) so that each horizontal side of your cubes falls on the plane of a slice. You then can test each vertex by going to the masked slices corresponding to each cube's top and bottom z values. You now have a bunch of cubes with labeled corners. For each cube, you know the surface intersects the cube along the edges in between corners of opposing classifications. Each cube should look something like this:



In 2D to approximate the surface we simply had to draw lines between each purple dot. In 3D this would give us a strange line drawing like this: