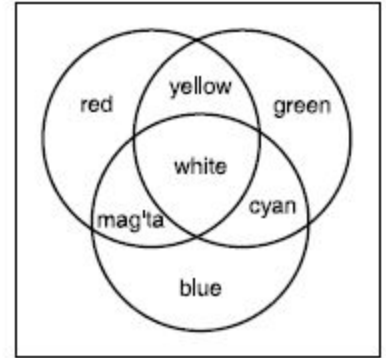


Feel Free to join chat, Please Contribute

<https://quizlet.com/96709210/midterm-flash-cards/>

- Pixels

- Used in Raster Displays
 - Raster displays show images as a rectangular array of pixels
- Screen made of large amount of smaller, single color, squares, called pixels
- 1Color of each pixel set with RGB (Red, Green, Blue) Value
- Pixels normally indexed with rows and columns, Processing used Top-Left as (0,0)
- Scan line = one line, or row, of pixels
 - When you write to a display it is written one line at a time, not one pixel at a time (usually).



- Transformations

- Homogeneous Coordinates
 - Goes from [-1, 1] on X, Y, Z axes
 - in computer graphic we use homogeneous coordinate to capture the concept of infinity.
 - makes calculation of geometry possible in projective space
 - point (x,y,z) ==> $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$
- Translate
 - Moves image.
- Scale
 - Changes the size of the image.
 - Can be used to invert an image by scaling by -1 on desired axis.
- Rotate
 - Rotates Image
- Shear (we haven't used this)
 - distorts image along an axis
 - x-shear
 - $x^1 = x + ay$
 - $y^1 = y$

- Composition of Transformations

Translate

1	0	0	dX
0	1	0	dY
0	0	1	dZ
0	0	0	1

Scale

sX	0	0	0
0	sY	0	0
0	0	sZ	0
0	0	0	1

Rotate(Z)

cos	-sin	0	0
sin	cos	0	0
0	0	1	0
0	0	0	1

Rotate(Y)

cos	0	sin	0
0	1	0	0
-sin	0	cos	0
0	0	0	1

Rotate(X)

1	0	0	0
0	cos	-sin	0
0	sin	cos	0
0	0	0	1

When translating a vector (point), always use [x,y,z,1], translation doesn't work if using 0

- Matrix Stack and Transformation Hierarchy
 - <https://processing.org/tutorials/transform2d/> → good tutorial on matrix stack
 - Matrices are used to modify shapes using matrix arithmetic
 - Transformations go in reverse order
 - Example: Transformation Matrix = (Rotate)(Scale). The scale would be applied before the rotation.
 - Transforms are applied from the right side first. So the matrix M= RS first applies S and then R.

- Projections

- Orthographic

- Useful in construction/modeling where there shouldn't be any size distortion due to distance from camera

$$P = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Perspective

- Mimics real world view where objects appear to shrink in size at a greater distance from the camera

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- Axis Review:

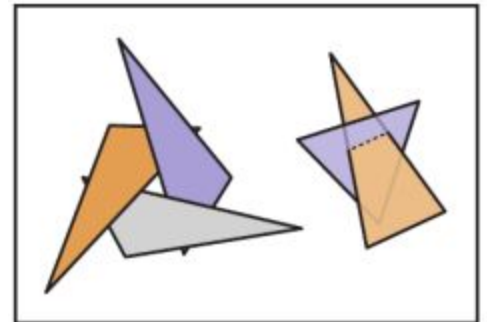
- x <- [left, right] = [l, r]
 - y <- [bottom, top] = [b, t]
 - z <- [near, far] = [n, f]
 - Right-handed coordinate system - similar to the right hand rule used in physics, but z is the direction to and from the "camera."

- x' = x / |z|
 - y' = y / |z|
 - z' = -1
 - k = tan(FoV / 2)

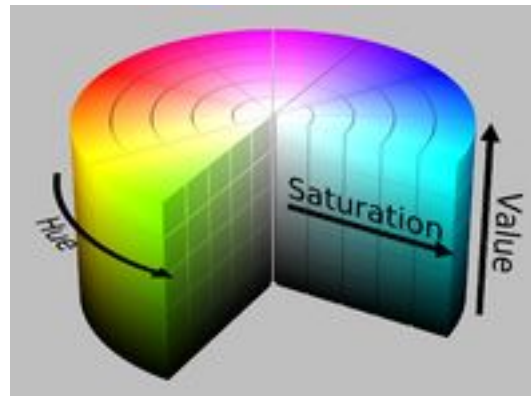
- Projection and Mapping to Screen

- Viewing Transformations

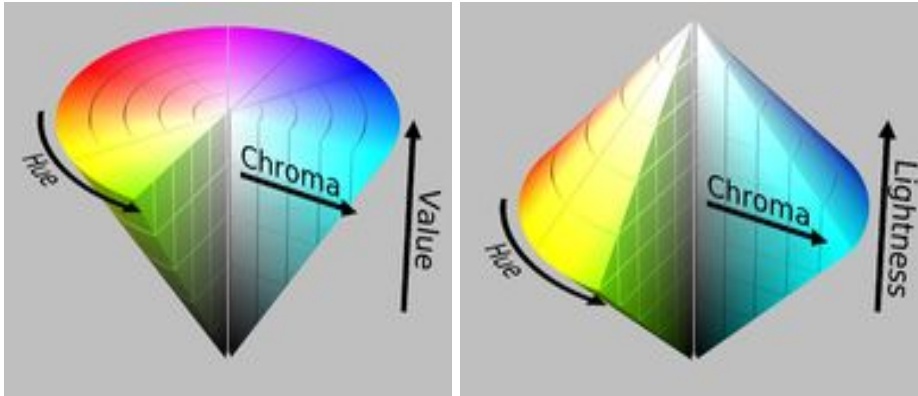
- LCD's
 - Liquid Crystal Displays
 - Molecules move like a liquid, maintain aviation like a solid
 - Affected by electric and magnetic fields, temperature, & pressure
 - Alternative to phosphorus based CRT's
 - E-Ink
 - Either reflects or absorbs light depending on pigment particles
- Lines
 - Uses coordinate System with 2 (x,y) values to determine how to draw the line using pixels.
 - Common algorithm for line drawing is midpoint algorithm (implicit equation)
 - two types of line equations: implicit and parametric
 - Parametric: $y = mx + b \leftarrow$ (Note: this may not be true. $Y = mx + b$ is formed by the implicit equation).
 - Implicit = 2 (x,y) values, line drawn through points
- Line and Polygon Drawing
- Polygon Scan Conversion
 - Fill one scanline at a time
 - Go bottom to top/sort intersections on x-coords
 - Fill between intersections
- Hidden Surfaces
 - Hidden Surfaces are surfaces and parts of surfaces that are not visible from a certain viewpoint. We need to identify these surfaces and determine which of these need to be visible to users and which shouldn't.
 - Painter's Algorithm (rarely used): This algorithm makes it so that object primitives must be drawn in back to front order, kind of like a painter who paints the background of a painting first and then the foreground over it. It has many drawbacks: Cannot draw triangles that intersect with each other and triangles can't be arranged in an occlusion cycle (AKA overlapping objects are a no no)
 - BSP Trees for Visibility
 - Ray Tracing is also another Hidden Surface Algorithm
 - Z-Buffer Algorithm is more commonly used for hidden surface removal.
 - At each pixel we keep track of the distance to the closest surface that has been drawn so far, and we throw away fragments that are farther away than that distance. The closest distance is stored by allocating an extra value for each pixel, in addition to rgb values. The z buffer is initialized to the maximum depth.
 - Graphics Card use the Z buffer algorithm, not Raytracing.



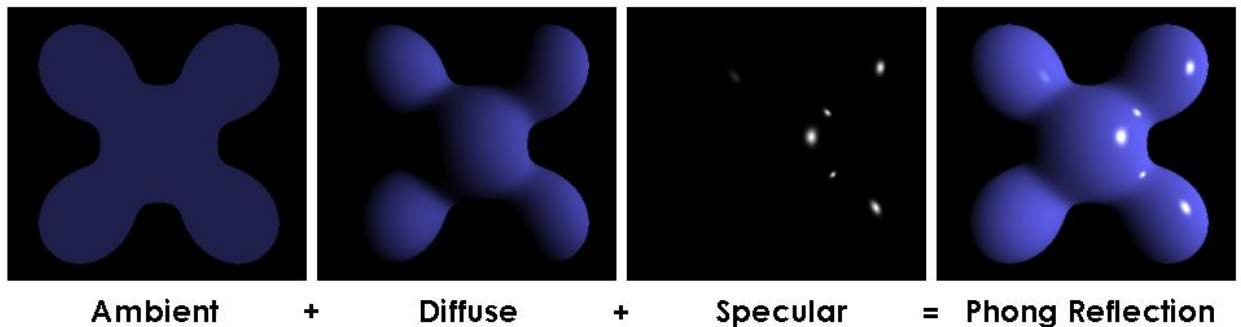
- Radiometry
 - This refers to measuring light (counting photons)
 - Radiance - fundamental quantity of light
 - it's a function of position and direction
 - Properties of Radiance
 - Radiance along unblocked ray is constant
 - Response of sensor is proportional to radiance of visible surface
- BRDF's - Bidirectional Reflectance Distribution Function
 - Reflectance Equation (Past Midterm Question)
 - $$L(x, k_o) = \int_{\Omega} L(x, k_i) p(k_i, k_o) \cos\theta_i dk_i$$
 - BRDF Laws
 - Helmholtz reciprocity - photons travel same path, forwards & back
 - Conservation of Energy
 - BRDF is the constant of proportionality for outgoing & incoming radiance
- Color
 - RGB
 - The simplest color space, triplet of three values (R, G, B)
 - All zero is black, all maximum is white
 - Additive colors, contrasted with subtractive colors (CMYK)
 - HSV
 - Hue, saturation, value
 - Hue is the actual color, the theta around a cylinder
 - Red at 0, Green at 120, Blue at 240, back to red at 360
 - Value is bottom to top base of cylinder, gives black from bottom to full color at the top
 - Saturation is along the radius, gives white at center to full color at outside of cylinder
 - HSL
 - Similar to HSV
 - HSL: hue, saturation, lightness



- Primary difference is that full color saturation is at an L value of 0.5, L of 1 is white, L of 0 is black
- Both HSL and HSV can be represented by cones and cylinders
- HSV is a cone with its point down, while HSL is a bicone
- HSL and HSV can both also be represented by hexagonal pyramids, not just cones/cylinders



- Surface Shading
 - Shading models
 - Shading Components: diffuse, ambient, specular
 - Diffuse Color: How an object would look under completely white light.
 - Is the color that an object reflects when illuminated
 - Specular: How much light an object reflects, basically how “mirrored” an object is.
 - Ambient Light: The light that is all around a scene. **Ambient color is basically a darker shade of whatever the color of an object is.**



- Where to Shade: per-polygon (fastest), per-vertex, per-pixel(looks best)
- Interpolation for Shading
 - Linear Interpolation is perfect for triangles
 - flat shading
 - one normal/light vector/color -> per polygon
 - problem: shady discontinuity “jumps”
 - Gouraud Interpolation

- **per vertex** shading
 - estimate N at each vertex
 - with N, calculate shading “C” for each vertex
 - Interpolate the colors across the shape.
- Phong Interpolation (normal interp)
 - estimate normal **per pixel**
 - then shade **per pixel**
 - Interpolate the normals across the shape. Find colors from those normals.
- **Ray Tracing**
 - Pseudocode for basic raytracer

```

for each pixel of the screen {
  Final color = 0;
  Ray = { starting point, direction };
  Repeat {
    5
    for each object in the scene {
      determine closest ray object/intersection;
    }
    if intersection exists {
      for each light in the scene {
        if the light is not in shadow of another object {
          add this light contribution to computed color;
        }
      }
    }
  }
  Final color = Final color + computed color * previous reflection factor;
  reflection factor = reflection factor * surface reflection property;
  increment depth;
} until reflection factor is 0 or maximum depth is reached;
}

```

- Recursive Ray Tracing
 - angle of incidence = angle of reflection
- Fast Ray Tracing
 - For each pixel (x, y) create ray
 - For each object in scene (may be thousands!)
 - test ray/object intersection
 - Solutions: divide up 3D space

Three common data structures that can be used to accelerate:

 1. Uniform grid
 2. Bounding Volume Hierarchy

3. KD tree - similar to BST

- Cost of intersection
 - Expensive: torus (donut)
 - Cheap: sphere
- Ray Sphere Intersection
 - Sphere formula: $(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0$
 - Sphere center at: (x_c, y_c, z_c)
 - Assuming ray with origin: (x_0, y_0, z_0) and direction: (dx, dy, dz)
 - Plug in $x_0 + t \cdot dx$ for x in sphere formula
 - Solve for t
 - Intersection is at: $(x_0, y_0, z_0) + t \cdot (dx, dy, dz)$

Bounding Volumes

- Advantages:
 - nice reflection, refraction, shadow
 - easy to code
 - many different primitives
 - object oriented
 - excellent image quality
- Disadvantages
 - Computationally expensive
 - If using too many consecutive bounding volumes, you can end up doing more work than if you just tested against the object in the first place
 - Hard to implement in hardware (Graphics hardware uses rasterization)
- GPU Programming
 - Vertex program: converts 3d points to 2d points, gives output to rasterizer
 - Frag program: takes rasterizer output and calculates fragment color
 - The frag program and vert program do not run on the entire mesh, or entire screen, it is called per-vertex and per-fragment

Corners Representation

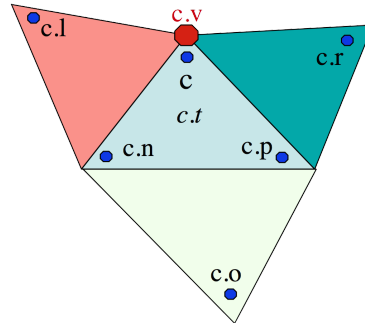
- c.v = vertex associated with corner c
- c.t = triangle associated with corner c
 - $c.t = c \text{ INTDIV } 3$
- c.n = next corner (moving counterclockwise around a tri)
 - $c.n = (3 * c.t) + ((c+1) \text{ MOD } 3)$
- c.p = previous corner
 - $c.p = c.n.n$
- c.v.g = coordinate of vertex associated with corner c (c.v.g.x, c.v.g.y, c.v.g.z)
- c.o = corner opposite corner c
 - foreach corner a
 - foreach corner b
 - if (a.n.v == b.p.v && a.p.v == b.n.v)

$$a.o = b$$

$$b.o = a$$

- c.l = corner left to corner c
 - c.l = c.p.o
- c.r = corner right to corner c
 - c.r = c.n.o

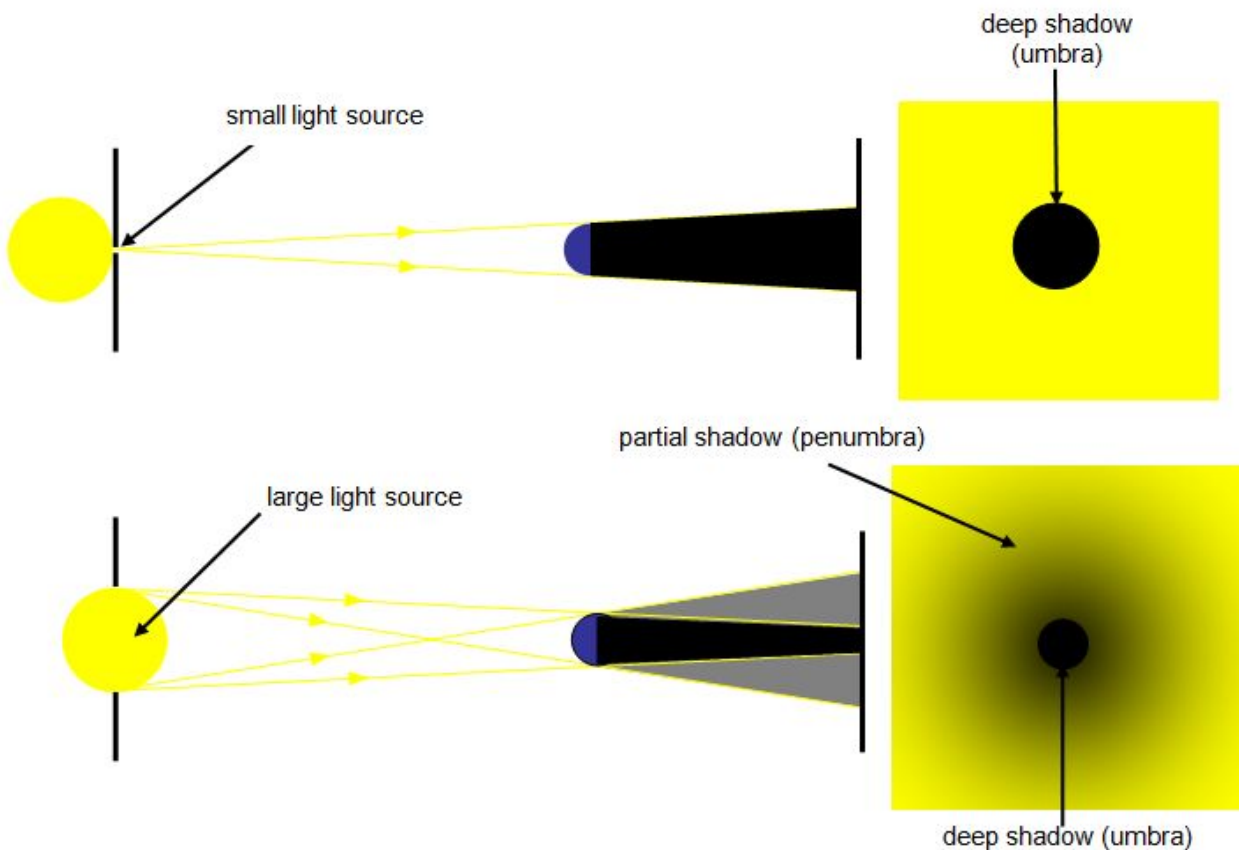
Corner Notation Summary:



Additional Topics:

Shadows

- Creating shadows is about visibility checking
- Shadow Mapping steps:
 1. Render scene from light (this allows us to test visibility: what can the light see and what can it not see?)
 2. Render scene from camera (eye)
 3. Find which pixels are hidden from the light (use 1 z buffer)
 4. Transform visible pixel coordinates to light space: $p \rightarrow p'$



Texture Mapping

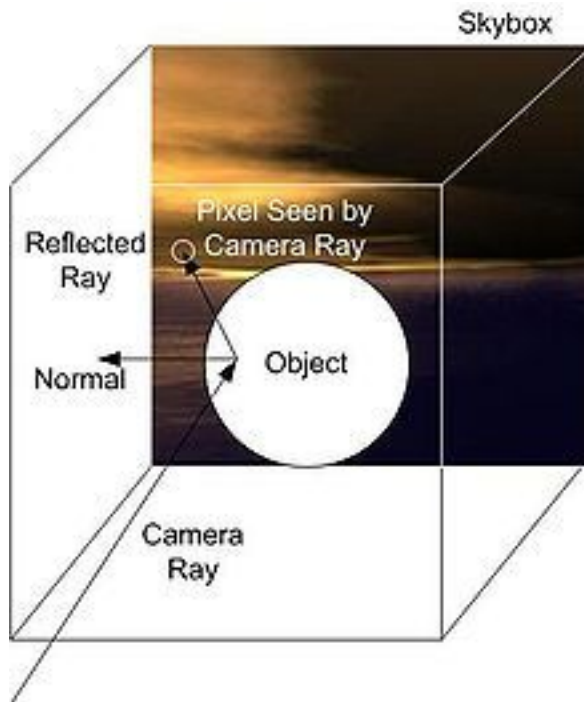
Bump Mapping

- achieved by perturbing the surface normals of the object and using the perturbed **normal** during lighting calculations.
- Because the actual shape of the object isn't changed (underlying surface isn't changed), Silhouettes and Shadows will remain the same. The left ball below uses bump mapping, but the shadow is that of a perfect sphere. The second object actually has a bumpy surface, so the shadow is more accurate.
- Bump Mapping method discussed in class (J.F. Blinn, 1978)
 1. Look up the height in the [heightmap](#) that corresponds to the position on the surface.
 2. Calculate the surface normal of the heightmap, typically using the [finite difference](#) method.
 3. Combine the surface normal from step two with the true ("geometric") surface normal so that the combined normal points in a new direction.
 4. Calculate the interaction of the new "bumpy" surface with lights in the scene using, for example, the Phong reflection model.

Notes: bump mapping doesn't actually change the shape of the object

Environment Mapping

- Used to have texture-mapped backgrounds and for objects to have specular reflections of that background.
- In order to store an environment map, we can use a **cube map**. This has 6 square texture maps (one per side in the cube).



Graphics Hardware

- Hardware components necessary to quickly render 3D objects as pixels on your computer's screen using specialized rasterization based hardware architectures. (chipsets, transistors, buses, and processors on video cards)
- There exists a graphics pipeline that ensures basic coloring, lighting, and texturing can occur very quickly.
- Important things about the graphics pipeline:
 - User programs provide information to the hardware as primitives (lines, points, polygons). Images or bitmaps are also supplied for use in texturing surfaces.
 - Geometric primitives are processed on a per-vertex basis and are not transformed from 3D coordinates to 2D screen triangles
 - Screen objects are passed to the pixel processors, rasterized and then colored on a per-pixel basis before being output to the frame buffer, and eventually to the monitor

GPU Programming

Resource : <https://www.processing.org/tutorials/pshader/>

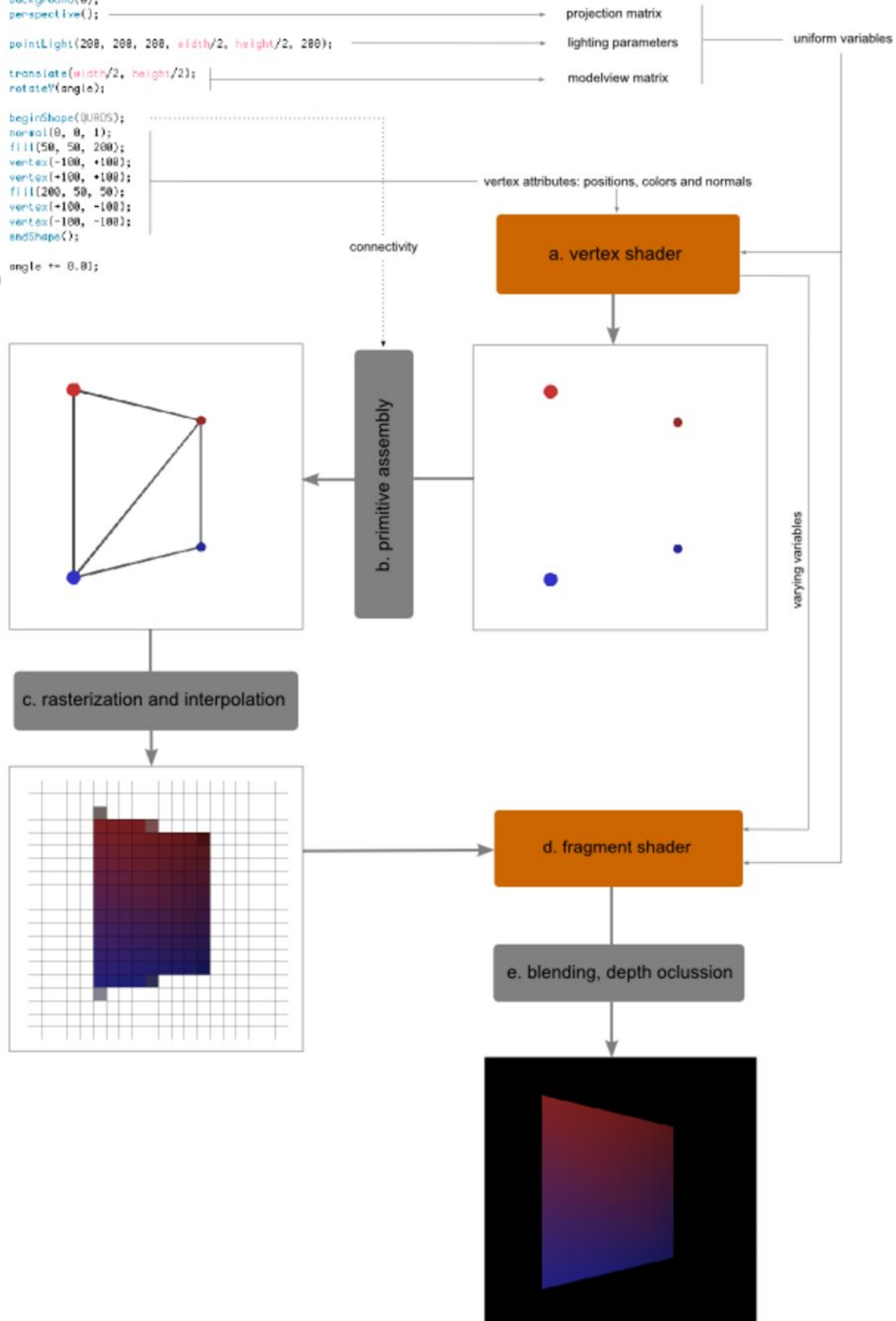
```
float angle;

void setup() {
  size(400, 400, P3D);
  noStroke();
}
```

```
void draw() {
  background(0);
  perspective();
  pointLight(200, 200, 200, width/2, height/2, 200);
  translate(width/2, height/2);
  rotateY(angle);
```

```
  beginShape(QUADS);
  normal(0, 0, 1);
  fill(50, 50, 200);
  vertex(-100, +100);
  vertex(+100, +100);
  fill(200, 50, 50);
  vertex(+100, -100);
  vertex(-100, -100);
  endShape();
```

```
  angle += 0.01;
}
```



Representing Polyhedra

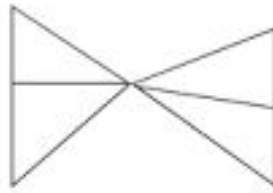
The topological class of a polyhedron is defined by its Euler characteristic and orientability. From this perspective, any polyhedral surface may be classed as certain kind of topological [manifold](#).

Manifold Meshes:

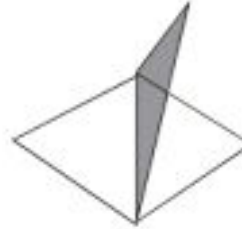
- Each edge is incident to only one or two faces
- the faces incident to a vertex form a closed or an open fan



Manifold



Non-Manifold



Non-Manifold

Operations on Polyhedra:

- Laplacian Smooth
- Face subdivision
- Triangulation

Geometry storage format in a 3D modeling:

- Polygon Soup : a group of unorganized triangles, with no connectivity info between faces

File format : stl

- Indexed face sets: have adjacency info, more compact than Polygon Soup, can create non-manifold objects.






File format: obj, ply (the one used in Project 5)

Platonic Solids

- Euler formula: $V - E + F = 2$ (V = Vertices, E = Edges, F = Faces)

- $$V = \frac{4p}{4 - (p - 2)(q - 2)}, \quad E = \frac{2pq}{4 - (p - 2)(q - 2)}, \quad F = \frac{4q}{4 - (p - 2)(q - 2)}.$$

- p = the number of edges of each face (or the number of vertices of each face) and
- q = the number of faces meeting at each vertex (or the number of edges meeting at each vertex).
- Euler's Characteristic Formula states that for any connected planar graph, the number of vertices (V) minus the number of edges (E) plus the number of faces (F) equals 2.
- Tetrahedron, Cube, Octahedron, Dodecahedron, Icosahedron (20 faces)
 - https://en.wikipedia.org/wiki/Regular_icosahedron
- Dual: interchange of faces and vertices

Polyhedron		Verti ces	Edge s	Faces	Schläfli symbol	Vertex config.
tetrahedron		4	6	4	{3, 3}	3.3.3
hexahedron (cube)		8	12	6	{4, 3}	4.4.4
octahedron		6	12	8	{3, 4}	3.3.3.3
dodecahedron		20	30	12	{5, 3}	5.5.5
icosahedron		12	30	20	{3, 5}	3.3.3.3.3

	vertices	faces	edges	dual
tetrahedron	4	4	6	tetrahedron
octahedron	6	8	12	cube
cube	8	6	12	octahedron
icosahedron	12	20	30	dodecahedron
dodecahedron	20	12	30	icosahedron

Bezier Curves (vector graphics)

- From textbook - 3 main ways to specify curves mathematically page 340 3rd edition 15.1
 $Q(t) = T * M * G$ (G contains 4 control points)
- Common representation for free-form curves (the ones mainly utilized are cubic bezier curves, where $d = 3$)
- A polynomial curve that approximates its control points (degree d has $d + 1$ control points)
- All Bezier Curves have these traits:
 - Curve interpolates first and last control points, with $u = 0$ and 1 respectively.
 (tangent at points 1 to 4 are independent)
 - First derivative of the curve at its beginning (end
- Disadvantage:
 - Bezier Curve cannot represent some curves that are too wavy since the tangent must align up for two curves

Bezier Surfaces

- $Q(s,t) = S * M * G * M^T * T^T$ (G contains 16 control points)
- Properties:
 - Interpolate corner control points
 - Surface inside convex hull of control points
 - Continuity across patches if control points “line up” across shared edges

Subdivision Surfaces

method of representing a **smooth surface** via the specification of a coarser **piecewise linear polygon mesh**

- Advantages over Bezier Patch:
 - BP must use 4-sided patches, but SS can turn any polygon mesh into smooth surface
 - BP: continuity between patches is messy, but SS: continuity comes “for free”
- Continuity

$$\frac{d^n s}{dt^n}$$

- A curve can be said to have C^n continuity if $\frac{d^n s}{dt^n}$ is continuous of value throughout the curve.
- Loop (triangles) subdivision
 - Divide all triangles into 4 new ones by creation of new vertices in the midpoints
 - C^2 continuous nearly everywhere, joined and deriv. Most everywhere, true at valence 6 vertices, C^1 continuous elsewhere
- Catmull-Clark
 - Quadrilateral based
 - for each face, a *face point* is created which is the average of all the points of the face.
 - for each edge, an *edge point* is created which is the average between the center of the edge and the center of the segment made with the face points of the two adjacent faces.
 - for each vertex point, its coordinates are updated from (new_coords):
 - the old coordinates (old_coords),
 - the average of the face points of the faces the point belongs to (avg_face_points),
 - the average of the centers of edges the point belongs to (avg_mid_edges),
 - how many faces a point belongs to (n), then use this formula:
 - $m1 = (n - 3) / n$
 - $m2 = 1 / n$
 - $m3 = 2 / n$
 - $new_coords = (m1 * old_coords)$
 - $+ (m2 * avg_face_points)$
 - $+ (m3 * avg_mid_edges)$
 - Then each face is replaced by new faces made with the new points,
 - for a triangle face (a,b,c):
 - (a, edge_pointab, face_pointabc, edge_pointca)
 - (b, edge_pointbc, face_pointabc, edge_pointab)
 - (c, edge_pointca, face_pointabc, edge_pointbc)
 - for a quad face (a,b,c,d):
 - (a, edge_pointab, face_pointabcd, edge_pointda)
 - (b, edge_pointbc, face_pointabcd, edge_pointab)


```
(c, edge_pointcd, face_pointabcd, edge_pointbc)
(d, edge_pointda, face_pointabcd, edge_pointcd)
```

Calculate Surface Normals:

- Triangles:
 - **Begin Function** CalculateSurfaceNormal (Input Triangle) Returns Vector

```
Set Vector U to (Triangle.p2 minus Triangle.p1)
```

```
Set Vector V to (Triangle.p3 minus Triangle.p1)
```

```
Set Normal.x to (multiply U.y by V.z) minus (multiply U.z by V.y)
```

```
Set Normal.y to (multiply U.z by V.x) minus (multiply U.x by V.z)
```

```
Set Normal.z to (multiply U.x by V.y) minus (multiply U.y by V.x)
```

```
Returning Normal
```

End Function

- Arbitrary:
 - **Begin Function** CalculateSurfaceNormal (Input Polygon) Returns Vector

```
Set Vertex Normal to (0, 0, 0)
```

```
Begin Cycle for Index in [0, Polygon.vertexNumber)
```

```
Set Vertex Current to Polygon.verts[Index]
```

```
Set Vertex Next to Polygon.verts[(Index plus 1) mod
Polygon.vertexNumber]
```

```
Set Normal.x to Sum of Normal.x and (multiply (Current.y minus
Next.y) by (Current.z plus Next.z))
```

```
Set Normal.y to Sum of Normal.y and (multiply (Current.z minus
Next.z) by (Current.x plus Next.x))
```

```
Set Normal.z to Sum of Normal.z and (multiply (Current.x minus
Next.x) by (Current.y plus Next.y))
```

```
End Cycle
```

Returning Normalize(Normal)

End Function

Fractals

- Mandelbrot
 - Cardioid shape
 - From HW:
 - Let $z(0) = (0,0)$. Then you look at the values $z(1)=z(0)^2+c$, $z(2)=z(1)^2+c$, etc
 - Iterate on these. If it never leave a circle radius 2, draw a dot at location
 - General definition: the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$, i.e., for which the sequence $f_c(0), f_c(f_c(0)), \text{ etc.}$, remains bounded in absolute value.
- Julia Set
 - The Julia set is now associated with those points $z = x + iy$ on the complex plane for which the series $z_{n+1} = z_n^2 + c$ does not tend to infinity. c is a complex constant, one gets a different Julia set for each c . The initial value z_0 for the series is each point in the image plane. In the broader sense the exact form of the iterated function may be anything, the general form being $z_{n+1} = f(z_n)$, interesting sets arise with nonlinear functions $f(z)$.

Volume Rendering

- Volumetric data: values of collections at given points
 - Voxel: volume element w/ scalar value
- Direct volume rendering:
 - Classify each voxel (reflectance, etc.)
 - Calculate shading->
 - Render voxels

Kinect Sensor

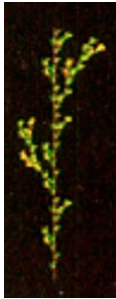
- Laser Sheet -> Range Image
- Kinect 1.0: Triangulation
 - Uses simple pythagorean formula to analyze depth
 - IR Projector projects a lot of scattered semi-random points, a camera processes point positions and calculates distance of point based on size and skew compared to neighboring dots
- Kinect 2.0: Time of Flight
 - Measures time taken for light to bounce off object

Fluid Animation

- Navier-Stokes Equations
 - $u_t = \nu \nabla^2 u - (u \cdot \nabla)u - \nabla p + f$
 - u_t -> change in velocity

- $k \nabla^2 u \rightarrow$ diffusion
- $-(u \cdot \nabla)u \rightarrow$ advection
- $-\nabla p \rightarrow$ pressure
- $f \rightarrow$ body forces

Procedural Content Generation

- Examples: Minecraft, SimCity, Spore, and No Mans Sky
- Noise pseudocode:
 - ```
float Noise1d(float x) {
 p1 = floor (x)
 p2 = floor (x) + 1
 t = x - p1
 v1 = seeded_random(p1)
 v2 = seeded_random(p2)
 return (smooth_interp (v1, v2, t))
}
```
  - Can be used to generate texture maps, bump maps, etc. to procedurally create mountains and terrain
- Simulated Erosion - simulate the slow deterioration of terrain over time to create more realistic terrain
- User guided terrain - allows an artist to create natural terrain by generally directing the generation of mountains and canyons
- Plant Modeling
  - L-System Grammar
    - w: S
    - p:  $S \rightarrow S[S]S[S]S$
  - Generates: 

### Motion Capture