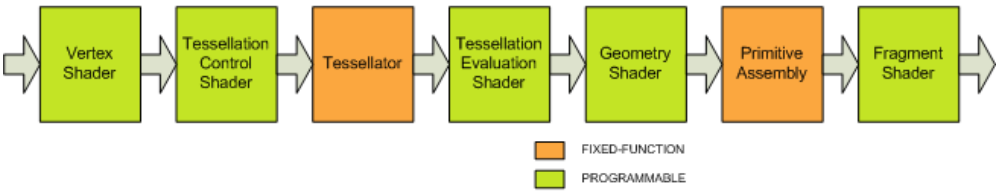
CS174A Midterm Study Guide – Nathan Tung

Lecture 1:

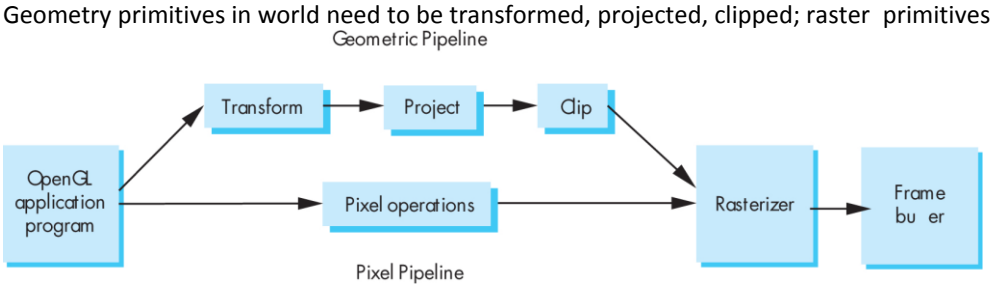
- Interaction/event loop
  - Collecting user input from external hardware (e.g. mouse, joystick, etc.)
  - Sends data down imagine/rendering pipeline to display and repeats
- Imaging pipeline (shaders)
  - Geometry data passed in is projected, rasterized, colored (visibility), and displayed
  - Vertex, geometry, and fragment shaders are programmable



- Camera model
  - No light rays in computer, so use math (geometry, linear algebra) to approximate
  - In a camera, moving image plane closer reduces size of image
  - Clipping/Culling (2D/3D): eliminating parts of the image/view that is out of bounds
  - We can move our camera position in space or parameters of our camera system, like size or position of window within image plane
  - Our "camera" is made up of the display "window" and the "eye-point"

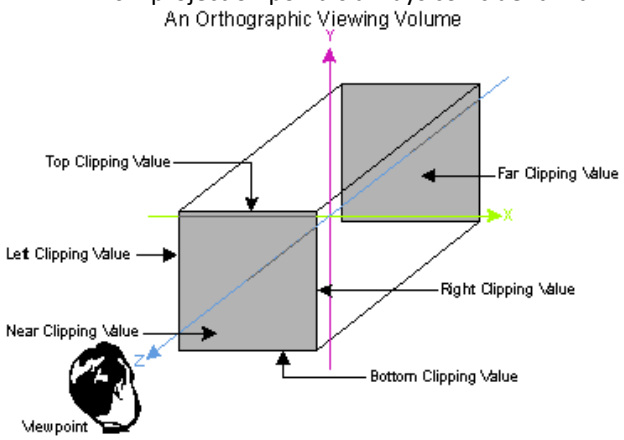
Lecture 2:

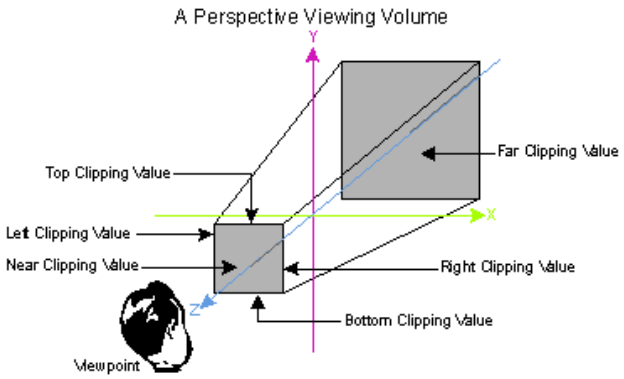
- Different Modes
  - Immediate Mode: points generated and sent to GPU each time; no storage needed
    - Lots of work
  - Retained Mode #1: points generated once, transferred to GPU each time; storage for all points needed
    - Less work, but needs storage and transfer to GPU each time
  - Retained Mode #2: points generated once, transferred to GPU once; storage for all points once
    - Lots of work, temporary storage on host and GPU, single transfer time
- Transformation Functions: translate, scale, rotate (these are used heavily by viewing)
  - Occur in application itself or in shader
- Input for interaction (i.e. keyboard, mouse)
- OpenGL is a state machine, a black box, that accepts two inputs
  - Something that alters internal state, like changing or returning state (parameter setting, enable/disable features)
    - State is persistent: stays set until you change it
    - State is not bound to primitives ("cube is yellow, grass is green, etc." - we have to manage this ourselves)
  - Something that causes output, like specifying primitives going through pipeline
- GLUT (simple system independent windowing and input processing), GLEW (removes OS specified dependencies)
- Coordinate Systems (device independent graphics: we don't specify pixels on display, and no units in OpenGL)
  - World: geometry is defined here, represented with floats
  - Window: physical display, width and height in pixels
- Use transformations to do world → screen coordinates
  - Geometry primitives in world need to be transformed, projected, clipped; raster primitives already in screen space



- Primitives include points, lines, triangles, triangle strips (these render very fast and can approximate complex ones)
  - Primitives made up of vertices (attributes define how GPU processes these primitives)
- CG is additive color (nothing=black, add color), as opposed to subtractive color (nothing=white, subtract from white)
  - Black is (0, 0, 0, 1) and white is (1, 1, 1, 1)
  - Red, green, blue (RGB), 8-bits per color plus another for alpha in RGBA

- Perspective vs. Orthographic Projection (for viewing)
    - Perspective: single projection point, appearance of depth like realistic vision
    - Orthographic: projection point is infinitely far away, so projection lines are parallel to image plane
        - As if projection point is always coincident with image plane



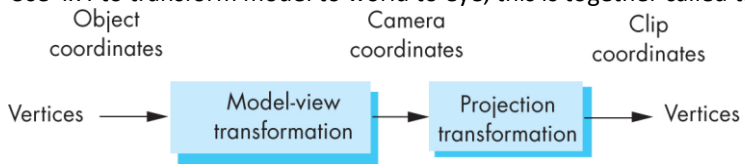


    - Orthographic creates 2D data with z=0, like text on glass (score) or HUD (head-up display)
        - Near clip, far clip, and sides make up the viewport
- Shaders (set of shaders exist within a program that executes on GPU)
    - Vertex Shader
        - On glDrawArrays(), each vertex triggers, execution of vertex shader
        - gl_position is an OpenGL state variable to pass final transformed vertex down pipeline to rasterizer
    - Fragment shader
        - Executes on output of the rasterizer; rasterizer generates fragments for primitives in pipeline
        - gl_frag_color must be output for fragment at minimum per execution

<u>Lecture 3:</u>

- Spaces
    - Vector, Affine, Euclidian Space
    - Each builds on the last; geometric objects exist within these spaces
- Vector Space
    - Vectors only have direction and magnitude; addition/multiple allowed
    - Addition (head-to-tail axiom), multiplication (scalar, changes magnitude)
    - Two ways to understand
        - Directed line segments (helpful to understand, not practical)
        - N-tuples of real numbers like v=(v1, v2, …., vn) (OpenGL representation)
    - This space is called “R^n”, where matrix algebra can manipulate vectors
    - Linear independence defines dimension
        - Case where only set of scalars a1u1 + a2u2 + … + anun =0 is a1=a2=an=0
        - In n=3 (or 3D), the only set of u is (1, 0, 1), (1, 1, 0), (0, 1, 1)
        - Largest number of linearly independent vectors in a space gives the dimension
    - A representation of a vector v is given by the Bi scalars with respect to v1, v2, … vn basis (head-to-tail)
    - Given a vector space of dimension *n*, then *any* set of *n linearly independent vectors* forms a *basis*
    - Basis gives us a representation (and dimension); matrix multiplication changes/transforms representations

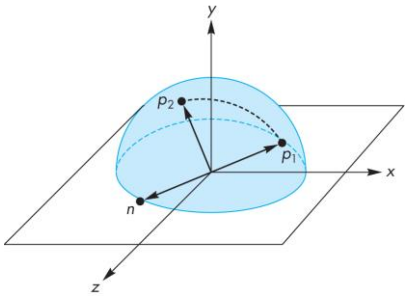
        - Can convert a frame to another
    - Vector space doesn't have location, vectors float with direction/magnitude (can have dimension though)
- Affine Space
    - Introduces concept of point to vector space; the point gives us a location
        - Operating on the point (like point subtraction – not addition – between two points) gives a vector
    - Affine space defined in terms of frames (frames: all coordinate systems existing in a basis R^n)
        - Frame consist of a point P0 (origin) and basis; other points are defined as P = P0 + B1v1 + … + Bnvn
    - Affine space doesn't have distance or length
- Euclidian Space
    - Consists of (real) vectors and scalars only; defines dot product to combine two vectors to form a real
    - Having $u \cdot v = 0$ means u, v are orthogonal; $u \cdot u = |u|^2$ tells us the square of the length!
    - And $u \cdot v = |u||v|cos\theta$ to give us the angle between two vectors
    - Adding in affine space's key concept of points, we can get distance between points
        - $|P - Q| = \sqrt{(P - Q) \cdot (P - Q)}$, the distance between P and Q
- Planes (an addition to vectors, points, lines)
    - Exists in affine space and defined by three points P, Q, and R
        - Those three points essentially form a triangle; notice that (Q-P) and (R-P) are arbitrary vectors
    - Plane is defined by two non-parallel vectors and a point
        - "Barycentric coordinate" representation is a simplified equation for planes
    - A vector n, or "normal," exists that's orthogonal to our plane using cross product: $n = u \times v$
- Homogeneous Coordinates
    - Frame in affine space: for R^3, vector v=[a1, a2, a3, 0] and point p=[b1, b2, b3, 1]; 0 is nowhere, 1 plants it in space
    - This is a 4D vector! Now we can do transformations
    - Scale: S(a, b) = [a, 0; 0, b] such that [a 0; 0 b][x; y] = [ax; by]
    - Rotation: R(theta) = [cos theta, -sin theta; sin theta, cos theta]
    - Translation can't be done with matrix-vector multiplication, but with homogeneous coordinates and [x; y; 1]
        - S(a, b) = [a, 0, 0; 0, b, 0; 0, 0, 1] multiplied gives us [ax; by; 1]
        - Translation uses [1, 0, j; 0, 1, k; 0, 0, 1], multiplied gives us [x+j, y+k, 1]
- Six Coordinate Frames used in OpenGL pipeline
    - Object (model), World, Eye (camera), Clip, Normalized device, Window (screen)
    - Object is represented in object/model coordinates, a local frame of the object to model geometry
    - Objects placed in world coordinates, a global frame to position relative objects via scaling, translating, rotating
    - Eye coordinates need to be placed to see object/world (whose coordinates are ideal for modeling, not viewing)
        - Use 4x4 to transform model to world to eye; this is together called the model-view transformation



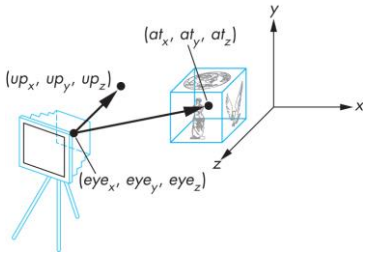
    - Clip coordinates reject primitives outside of view volume after projection transformation
        - Easiest when done after transforming view volume into origin-centered cube; clipping is done quickly here
    - After projection transformation, clip coordinates are still homogenous coordinates, so we remove w component to obtain a 3D point within normalized device coordinates
    - This is finally transformed into 3D window coordinates
        - Window (screen) coordinates are 2D, so pipeline drops the z (depth) value; window defined by viewport
    - No actual camera; everything is seen by manipulating model-view transformation
        - Same thing: moving world to eye, or eye to world coordinates frames
        - Transform world frame by -2 when camera fixed at origin; Transform eye frame by 2 if world is fixed
- Transformations
    - All transformation matrices are 4x4 (OpenGL uses column vectors for matrices like T[3][1])
        - Homogeneous translation matrix: [1 0 0 x'; 0 1 0 y'; 0 0 1 z'; 0 0 0 1]
        - Scale: [Bx 0 0 0; 0 By 0 0; 0 0 Bz 0; 0 0 0 1], Rotation [cosx –sinx 0 0; sinx cosx 0 0; 0 0 1 0; 0 0 0 1]
    - Matrix multiplication is associative (concatenate matrices for better performance in transformations)
        - q = CBAp is the same as q=C(B(Ap))
    - Not commutative; order of different transformation type matters
        - q=TRSTp is NOT the same as q=TSRTp
    - Applied last to first, since post-multiplied with current transformation matrix
    - To transform in geometry, send data once to update transformation via GPU uniform variable

Lecture 4:

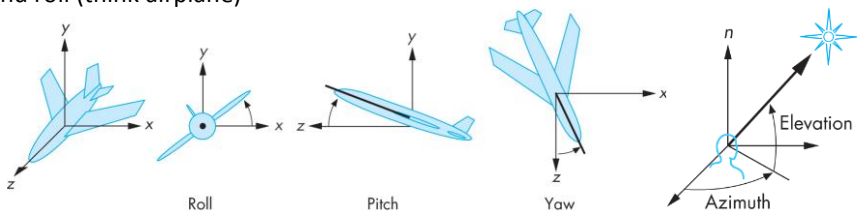
- Trackball
  - Place unit hemisphere in xz plane, reconstructing y using mouse positions (x2+y2+z2=1; $y = \sqrt{1 - x^2 - z^2}$)
  - Given two positions on hemisphere p1 and p2, we can find axis of rotation by the cross product of p1 to p2
    - That's the normal, or $n = p1 \times p2$, and for small rotations, θ and $\sin\theta$ are about equivalent



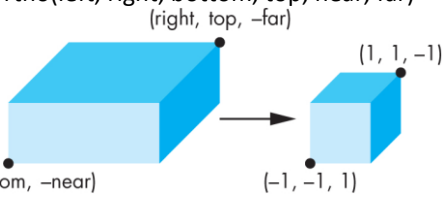
- Quaternions do the same thing with less work; q=w+xi+yj+zk (imaginary terms), where w is the rotation
  - But q must be normalized to q=|q^2|
  - Quaternions allow smooth interpolation with rotation R and (small) increment R1; renormalize R occasionally
- Viewing: perspective and parallel both have objects, projection lines and planes, and eye (center of projection – COP)
  - Model-view transformation doesn't go to clip coordinates; we need projection transformation
    - The former gets objects in front of camera; the latter defines how they appear
  - Again, transformations are specified in reverse (so specify camera position first)
  - Look-At: location of EYE, point the eye is looking AT, and the UP direction of camera (0, 1,0) or (0, 1, 0, 0)
    - AT and EYE points form the view-plane-normal, or vpn
    - Look-At is not smooth for moving camera



  - Yaw, pitch, and roll (think airplane)

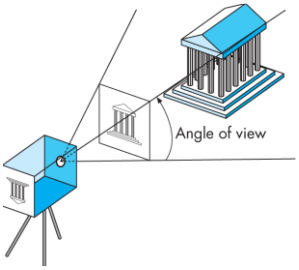


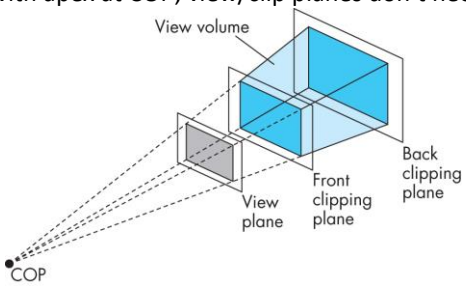
    - Use polar coordinates; a simplified version uses azimuth (side to side rotation) and elevation
    - Translate camera to eye point and rotate to direction we want (reality: perform the inverse)
      - We're really moving the world to the camera
  - Now we have our camera coordinates and we need a projection transformation to get clip coordinates
- Parallel (orthographic) projection/viewing
  - M = [1 0 0 0; 0 1 0 0; 0 0 0 0; 0 0 0 1]; this transformation is applied in hardware after vertex shader
  - Then we scale in order to fit/include within the canonical view volume (-1, 1), (-1, 1), (-1, 1)
  - Use glOrtho(left, right, bottom, top, near, far)

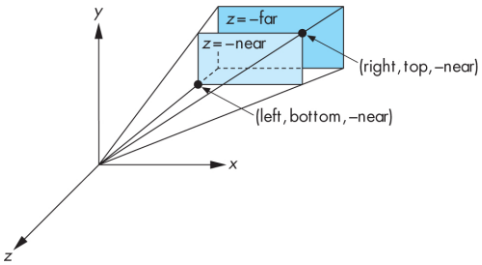


- Perspective projection/viewing
  - Basic symmetrical perspective projection; project (x, y, z) through projection plane to eye point (or COP)
  - M = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 1/d 0]
    - The 1/d term means our homogeneous coordinate will be modified
  - So far we only have points on projection plane, not clip coordinates; we also need a view volume

- View volume is a pyramid with apex at COP; view/clip planes don't need to be the same or coincident



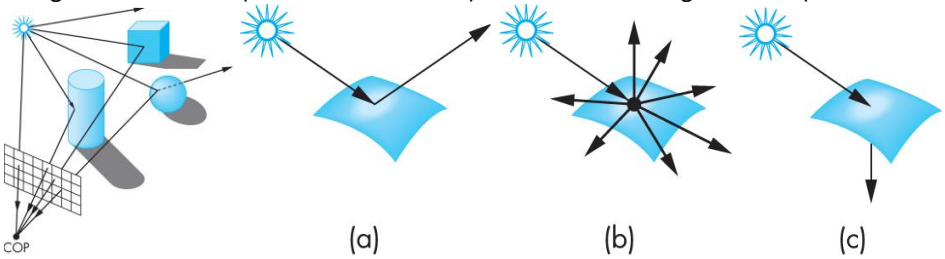




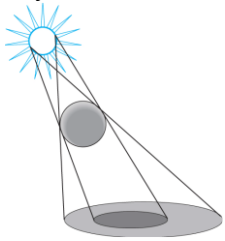
  - So we use perspective(fovy, aspect, near, far), where aspect is the aspect ratio of view volume (width/height)
  - Matrices are passed to vertex shader as uniform variables, as before

Lecture 5:

- Lighting and Shading
  - Shading needed in 3D to enhance effect of dimensionality; real-world color is rarely flat and uniform
  - Approximate how light interacts…
    - Different types of light sources
    - Properties of a surface that affects how light reflects off of it
  - Use local interaction, since global light models are too computationally intensive to be interactive
    - Global might be physics of light (surfaces emit, absorb, reflect; light reflects, scatters, bleeds, etc.)
    - Not fast enough (not even offline rendering); GPUs can speed this up for only simple/moderate things
    - Even approximations are too slow (radiosity – thermodynamics, ray tracing – light rays)
  - Simplify to single interactions between light source and surface
    - Need two models: light source, and how a surface reflects that light source
  - We can see light of the source (color of source itself) and reflection of light source (some absorbed/bounced)

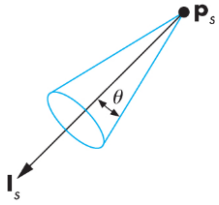


- Surface Types
  - Specular (a): most light reflected in narrow range, like a mirror
  - Diffuse (b): light is reflected everywhere, with a flat appearance
  - Translucent (c): light is refracted, some may be reflected
- Light Source Types (each have color and luminance, or brightness, each represented as RGB)
  - Ambient: uniform illumination (cloudy day or indirect lightning in house)
    - Light is scattered in all directions, modeled with luminance Ia
    - Every point receives same illumination; surfaces may reflect ambient light differently
  - Point
    - Light emitted equally in all directions, with a similar ambient luminance with location I(p0)
    - Amount of light received by a point is proportional to distance from point light source
    - $d = |p - p0|^2$, $i(p, p0) = \frac{1}{a+bd+cd^2} I(p0)$, where a, b, and c soften the light
    - Most real point sources are finite in size and not an actual point
      - Object shadow casts an (inside) umbra and (outside) penumbra

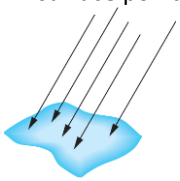


  - Spotlight
    - Have a position like point lights, but these actually have a direction defined by an angle
    - Intensity of light attenuated by magnitude of angle between direction vector and point of surface

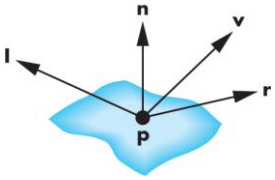
- This is the “falloff”



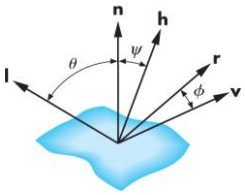
  - The angle is defined as $cos\theta = u \cdot v$
- Distant/Infinite
  - When light source is very far away (like the sun); light rays are parallel to each other in direction
  - All surface points use that direction vector instead of p0



- Shading – Phong Reflection Model
  - Rather than exact physical model, use efficient approximation
  - Use four vectors to enable calculation of a color value for any point on a surface
    - Vector n, normal at point P on surface
    - Vector v, from point P to eye
    - Vector I, from point P to the light source (direction)
    - Vector r, the perfectly reflected ray from I (computed from n and I)

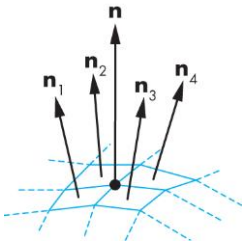


  - Supports ambient, diffuse, and specular light types
    - These 3 “color” values are stored for a surface
    - Generally, define a material and associate with surface so surface can have multiple colors
  - To compute light (color) at some surface point…
    - Sum contributions from each source interacting with surface (R) and global ambient term
    - $I = \Sigma(I_{a_i} + I_{d_i} + I_{s_i}) + I_a$, where $(I_{a_i} + I_{d_i} + I_{s_i}) = (L_{a_i}R_a + L_{d_i}R_d + L_{s_i}R_s)$
  - Ambient Reflection
    - Intensity of ambient light at each point on surface is equal
    - Amount that’s reflected is given by coefficient ka
    - For any light source, resulting reflected light intensity is $I_a = k_a L_a$ where $0 \leq k_a \leq 1$
  - Diffuse Reflection
    - No preferred direction of reflection; behavior modeled by Lambert’s law
    - Only see vertical component of reflected light, but light gets dimmer as angle becomes larger
      - Reason: light is spread out over larger area
    - Direction of light source l and surface normal n determine amount of reflected light
      - Take ito account reflection coefficient kd
    - Also include quadratic attenuation term to account for surface’s distance from light
      - $I_d = \frac{k_d}{a+bd+cd^2}\max((l \cdot n)L_d, 0)$, where max accounts for light when it’s below the horizon
  - Specular Reflection
    - Adding shininess to a smooth surface; reflection of part of light source itself
      - Smoother surface means more concentrated
      - Mirror is a perfect specular surface
    - Phong approximates this (can’t mirror)
    - Depends on angle between perfect reflection vector r, viewer direction v, ks (reflection coefficient of specular reflection), and α (the shininess coefficient which controls how narrow/broad the reflection is)
    - $I_s = k_s L_s \max((r \cdot v)^{\alpha}, 0)$
  - Full Phong Model
    - Combines diffuse, specular, ambient terms and computes it for each light source in scene
  - Blinn-Phong Model
    - Usually we compute the reflection vector r for each point on surface
    - When l, n, r, and v are in same plane, halfway vector h can be used instead of computing r



- $h = \frac{l+v}{|l+v|}$ and $\phi = 2\psi$
- The angle between n and h is smaller than angle between r and v, however
- So shininess coefficient α has to be raised to get approximately the same result as pure Phong

- Normal
  - For flat surfaces we mean normal to plane
  - For three non-collinear points, normal n = (p2-p0)x(p1-p0), where order matters; determines direction of n
  - For curved surfaces, depends on how surface represented; normal at point on a unit sphere is n = p
    - Approximations are needed (often using points/planes) since we represent objects using primitives
- Flat
  - Vectors l, n, and v vary as we move across surface
    - But flat surface with far light source and viewer means vectors can be held constant
  - Then every point on surface primitive receives the same shading, or "flat shading"
  - Flat shading is called per primitive in OpenGL, since a single color (light) is applied to entire primitive
- Smooth
  - Provide color and normal information per vertex to achieve a smoother shading
  - Let rasterizer interpolate color values across surface of primitive
    - How do we compute normal for a surface?
  - Gouraud shading tells us to use the average of surface normal of the primitive they are a part of
    - For a vertex shared by four surfaces, we compute the vertex normal as
      - $n = \frac{n_1+n_2+n_3+n_4}{|n_1+n_2+n_3+n_4|}$



  - Phong shading, a variation of Gouraud, interpolates the normals themselves (one step further)
    - Requires us to compute lighting at vertex or fragment level

Lecture 6:

- Frame (Color) Buffer
  - Manages RGB or RGBA color values on screen
- Depth Buffer
  - Stores normalized z values from clip volume
  - Used for hidden surface elimination; allows drawing objects without respect to space position
    - Order independent
  - When enabled, z value written in buffer for every pixel
    - If incoming pixel with z value less than value already in buffer, pixel written into frame (color) buffer, and z value is updated in depth buffer
    - Otherwise, pixel is rejected (not written to frame buffer) and depth buffer remains unchanged
  - Without depth buffer, last write to frame buffer wins; drawing becomes order dependent (render back-to-front)
  - Problem cases: inter-penetrating objects, moving objects
  - Entails requesting depth buffer (initialization time) and enabling it; also clear buffer (bit) before new image
- Back to Frame Buffer
  - Uses double buffering: front and back color buffers
    - Rendering happens on back buffer, front/back can be swapped, and front gets displayed
  - Transparent object rendering
    - Enable blending to consider alpha values
    - Blending involves RGBA values of pixels in pipeline ("source") and current frame buffer ("destination")
    - Blending adds these two pixels together after multiplying each by its blending factor
      - Prevents saturating (going above alpha of 1.0)

        - Don't want to update depth buffer; set gl.depthMask to false to make it read-only
        - Example: to render a frosted window, see what's behind the window and not see the window if it's behind other opaque objects
            - This requires depth test, but not writing depth value of window
        - Therefore, we render transparent objects AFTER opaque objects
        - Also sort and render transparent objects back-to-front (remember, no depth buffer here!)
        - Once transparent objects rendered, renable depth buffer writing (mask) and disable blending
            - Leaving blending on just makes it unnecessarily slow
- Mapping Methods
    - Texture, Environment, and Bump Mapping
        - All three can be combined
        - All are performed at fragment stage
        - All can be stored in 1, 2, or 3 dimensional buffers (maps)
    - Three variations of same mechanism to add detail and complexity
        - More complex geometry, details assigned to vertex colors, etc.
    - Imagery adds detail and appearance of complexity to geometry without actually modeling it
    - Maps can modify shading directly by altering color values, or alter surface material or normal properties
- Texture Mapping: use image to alter surface color values
    - Consider 2D form; individual pixels of the image are called texels
    - Texture map described by T(s,t) - texture coordinates s and t defined over [0,1] - and mapped onto point p(u,v)
    - Assign texture coordinates to vertices (like color or normal), which are interpolated over rendered surface and made available in fragment shader
    - Define/download image to GPU → assign texture coordinates to geometry → apply texture to fragments
        - First, create texture object (container objet describing texture)
        - After it's created and bound(active), define image data itself (as raw RGB) and copy into GPU
        - Texture is defined and coordinates are assigned to vertices statically; they can be used like colors/normal
        - Vertex shader: pass texture coordinates through to fragment shader, allowing s and t to be interpolated
        - Fragment shader: set up a way to control texture object to use; then set it
            - Define texture reference as a type sampler2D
            - Built in function texture 2D samples referenced texture at coordinate specified
            - Color and texture can be combined with each other and with the normal
    - For (s,t) values outside [0,1], use texture wrapping by either…
        - Clamp (reuse edge texel, smears)
        - Repeat
        - These are set separately for s and t, performed when texture is defined
    - Mapping between [0,1] and discrete texels
        - Sampling
            - Point, nearest neighbor
            - Linear, bi-linear interpolation
        - Minification and magnification
- Texture Mapping Overview
    - What kind of sampling to use? Depends on quality and type of image used in texture map.
    - Avoidance of sampling artifacts (Moire patterns)
    - Use experimentation
    - Consider resource usage (mipmaps use 1/3 extra storage)
    - Multi-texturing GPUs today can process several textures at once
        - Activate hardware texture unit, then bind texture to it (may need multiple sets of texture coordinates)
        - Up to fragment shader to determine how to process/combine result
    - Usually texture mapping is used as a tool to make fine adjustments
    - Load image from disk (no OpenGL mechanism for this – use a library; but WebGL is pretty easy)
- Environment Mapping: give objects the appearance of reflection
    - This reflection of fixed environment is sometimes called "reflection mapping"
    - Can be computed via ray tracing, but usually too slow for real-time; texture map can approximate the effect
    - Reflection vector is calculated by eye→surface→reflect→world
        - Intersect that ray with scene and compute the color (shading) value; but this is essentially ray tracing
        -
- Bump Mapping: distort normal vectors during shading process