Lighting: Spot Light, Point Light, Infinite Light (Directional Lighting) Fragment vs Vertex Shader, Direction to the Light vs Location of the Light. Importance of Normalizing our Vectors.

- **Spot Light:**
  - a) Strong beam of light that illuminates a well-defined area. OpenGL models spotlights as restricted to producing a cone of light in a particular direction
  - b) Direction to the spotlight is not the same as the focus direction of the cone from the spotlight unless you are looking from the middle of the “spot”
  - c) Using cosine computed as a dot product will tell us to what extent these two directions are in alignment. This is what we need to know to deduce if we are inside or outside the cone of illumination
  - d) We can sharpen the light falling within the cone by raising the cosine of the angle to higher powers. This allows control over how much the light fades as it gets near the edge of the cutoff
  - e) The beginning and end of the shader look the same as point-light. The differences are in the middle of the shader.
    - i) We take the dot product of the spotlight’s focus direction with the light direction and compare it to a precomputed cosine cutoff value “SpotCosCutOff” to determine whether the position on the surface is inside or outside the spotlight.
      1) If it is outside the spotlight attenuation is set to 0
      2) Otherwise this value is raised to a power specified by “SpotExponent”.
      3) The resulting spotlight attenuation factor is multiplied by the previously computed attenuation factor to give the overall attenuation factor.
- **Point Light:**
  - a) Mimics lights that are near the scene or within the scene like lamps, street lights or ceiling lights.
  - b) Direction of the light is different for each point on the surface, cannot be represented by a uniform direction.
  - c) The light is expected to decrease on the surface the farther away the object is from the light
  - d) Steps to compute the Point light from directional light
    - Compute the light-direction vector from the surface to the light position
    - Compute the light distance by using length() function.
    - Normalize the light-direction vector so we can use it in a dot product to compute a proper cosine
    - Compute the attenuation factor and the direction of maximum highlights
    - Then multiply diffuse and specular terms by the attenuation factor
- **Infinite Light (Directional Light)**
  - a) FragShader:
    - i) Uniform Variables
      1) Vec3 Ambient, vec3 LightColor, vec4 LightDirection, vec3 HalfVector, float shininess, float strength
      2) Compute diffuse = max(0.0, dot(Normal, LightDirection)), speculat = max(0.0, dot(Normal, HalfVector))
         - (a) If diffuse is equal 0.0, specular = 0.0 (because its facing away from light)
         - (b) Else, specular = pow(specular, shininess);
      3) Computer scatteredLight = LightColor * specular * strength; reflectedLight = LightColor * specular * Strength
      4) Then fragColor = min(vColor.rgb * scatteredLight + reflectedLight, vec3(1.0));
  - b) Vertex Shader:
    - i) We get a vertexPos, vertexNormal, vertexColor
    - ii) Our out color is the vertex color
    - iii) Our out vec3 normal = normalize(normalMatrix * VertexNormal)
      1) NormalMatrix = mat3
      2) vertexNormal = vec3
- **Direction to the Light**
  - a) The direction is important for certain lights because the direction will determine what sides get light and how much of it. Sides might get some light but not as much as a side directly facing the light
- **Location of the light**
  - a) Location of the light differs from direction depending on how close or far the light is
    - i) If the light is close, the sides will attract and reflect more light
    - ii) If a light is far, the sides getting the light will decrease in the amount of light received
- **Importance of Normalizing our Vectors(No clue if this is correct)**

  - To make sure the vectors unit length is 1, since for lighting it needs this to compute the correct result.
  - Also every vector pointing in the same direction gets normalized to the same vector
  - Its important to normalize vectors to represent a direction, as for infinite light or directional light, we couldn't give it precise coordinates, but we can verify where to find it from a particular point by using the normal vector

Instancing: Buffer Data vs. BufferSubData vs Instanced Drawing

- **Buffer Data:** Allocate and load data (if non-NULL pointer provided)
  - You need at least one glBufferData to allocate the buffer space
- **BufferSubData:** Load data only
  - After using bufferData to allocate buffer space, its more efficient to use glBufferSubData to change the contents. Mostly because glBufferData is often provided a NULL pointer - so all loading is then deferred to another command such as glBufferSubData
  - When replacing the entire data store, use BufferSubData instead of recreating with glBufferData to avoid the cost of reallocating the data store
- **Instanced Drawing:** similar to BufferSubData in the idea of updating the data to be redrawn
  - Used to render a large amount of geometry with very few calls
  - Using glDrawArraysInstanced is similar to glDrawArrays, but with the addition of primCount that specifies the count of the number of instances that are to be rendered. It will draw that object the many number of times
  - So instead of continually updating the buffers for a new object, you can just use this to draw multiple of that same object. This method has many functions that fundamentally all do the same thing such as:
    - glDrawElementsInstancedBaseVertex(GLenum mode, count, type, indices*, instanceCount, baseVertex)

Texture Mapping: Connecting Texture to the Image, Loading Texture into GPU Memory, Activating the Texture, How the Fragment Shader Accesses the Texture.

- **Connecting Texture to the image**
  a) Int width, int height, int numColorChannels
  b) Unsigned char* bytes = stbi_load(filePath, &width, &height, &numColorChannels, 0);
     i) Stbi is a 3rd party library that will pull out the texture we have to be added to the object
  c) GLuint texture; then glGenTextures(1, &texture);
  d) glActivateTexture(GL_TEXTURE0);
  e) glBindTexture(GL_Texture_2D, texture);
  f) glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);, glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);, glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);, glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  g) glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, bytes);
  h) Then in the vertices array for the object we want the texture to be on, add the coords for the texture to the array, cords higher than 1 will make the texture repeat itself
- **Loading Texture into GPU Memory**
  a) Add a new uniform variable to the vertex shader and modify the vertex buffer object to account for the vertices the texture needs and send them to vertex shader
  b) Output the texture cords to the frag shader from the vertex shader
  c) Create an in variable vec2 for the text coords in the frag shader and add a uniform variable with a type sampler2D named tex0
  d) Then change the FragColor out variable to be = texture(tex0, texCoords);
- **Activating the Texture**
  - Make sure to activate the texture before setting uniforms
  a) In the main file, connect the uniform variables from the shaders into the main file
  b) Then bind the texture in the main function by doing glBindTexture(GL_TEXTURE_2D, texture);
- **How the frag shader accesses the texture**
  a) We pass the texture to the frag shader as a sampler2D which applies the texture to the object based on the texture coords we passed into the frag shader from the vertex shader

Animation: Concept of changing transform in the draw loop

- Have a function that will either rotate, translate, or scale the object that is being animated like an arm or leg
- Depending on the state the object is in (like where it is currently) do one of the following transformations to its next state. So if an arm is facing upwards, transform it to be facing behind the person or so its facing straight down from where it started when calling the update object
- Once the object has been translated, call the function that will redraw the object to its specified location after its been updated.
- Then just keep doing that over and over depending on what type of transformation you use for that part of the picture.

Hierarchical Modeling: How the transformations are connected and how it is modeled.

- **How transformations are connected**
    a) Each level of the tree is a different transformation, each branch is a different object.
    b) So it might start at the top which would be the translation level, once it moves to the next object and level in that branch it would rotate it, then the next level would do the same but scale it. Once it traverses back up the branch, it would move to the next object on a different branch and do the same thing.
        i) It should visit all “kids” on its branch before going back up and moving to next object
- **How it is modeled**
    a) It is modeled as a tree, for example it could be modeled as a struct looking like:
        i) Containing the variables as floats or something similar for the 3 transformations
        ii) Maybe other characteristics such as color
        iii) Should contain the object type
        iv) Then 2 pointers, one pointing to the next object(same level), and one pointing to the next level(object at next level)
    b) Then to actually work with it you would have a function that would obtain the object it is currently supposed to draw,
        i) If the object isnt empty, we would get the transformations for the object, draw the object, restore the original transformation, then recall the function to render the next object (all this is one big recursion loop).
            1) **This will end once it reaches an object that is empty**
        ii) Once it reaches an empty object in that branch, go back up the branch to the top of the tree, move to next branch and repeat
        iii) If no branches remain, drawing is complete

Objects: Creation of Sphere and Cylinder

- **For Sphere**
    a) Start with a octahedron, for each face compute midpoints and subdivide into 16 triangles
    b) Once you have each face subdivided, normalize the vector based on center point so they are all the same distance
    c) Then expand the points by scaling the vectors so they are all the same length
- **For Cylinder**
    a) Start with a diamond, Subdivide the line segments either 2 or 3 times
    b) Normalize and Scale the vectors to all be the same distance
    c) Do it again so you end up with two circles of the same size so the circles are on top one another separated by a certain distance
    d) Then join the two circles with rectangles to finish the cylinder

Normals: Computing using the Cross-Product.
- Set 2 vectors, U and V
- U = (vector 2.xyz - vector 1.xyz), V = (vector 3.xyz - vector 1.xyz)
- Normal.x = (U.y * V.z) - (U.z * V.y)
- Normal.y = (U.z * V.x) - (U.x * V.z)
- Normal.z = (U.x * V.y) - (U.y * V.x)
- Return the Normal.xyz