hello world 3d: basic approach

- object (say a cube) will be made up of triangles, each with three vertices, each with known object coordinates.
- object coordinates of vertices will be put in an OpenGL buffer object.
- C++ code will maintain object matrices `g_objectRbt[i]` for each object i and sky matrix `g_skyRbt` for a sky camera. (show).
- typically we will use the sky camera's matrix as the eye's matrix E, `eyeRbt=g_skyRbt` but we can also use any other object as the eye.

interaction

- in the beginning we initialize the object and sky matrices.
- we will use mouse motion to update them
  - here very simply, but in your assignment with more sophistication.

drawing

- to draw an object, we will pass $E^{-1}O$ as a uniform variable to the vertex shader.
- called a modelview matrix `MVM`
- vertex shader will transform the object coordinates into eye coordinates and pass these out as varying variables.
- to get "perspective effect" we will also create a special projection matrix P, `projmat` and pass it to the vertex shader. (much more later)
- vertex shader will multiply eye coordinates by P to get data to set gl_Position.

for normals

- explicit normal data will also be placed by us in the openGL buffers.
- we will compute and pass the inv tpos MVM as a uniform variable `NMVM`.
- vertex shader will transform the object coordinates of the normal into eye coordinates of the normal and pass these out as varying variables.

vertex shader code

- does the matrix multiplies
- sets gl_Position.
- also sets the eye coordinates of the vertex and the normal.

fixed function

- finds screen pixels inside of triangle
- interpolates values for the varying variables.
- vPosition at each pixel corresponds to geometric position of the point in the triangle observed at the pixel (more later)

fragment shader code

- takes the position and normal information, as well the eye coordinates of the position of 2 light sources, as well as the underlying surface color
- does some math to simulate the observed color value (more later)
- output goes to screen and is properly z-buffered (more later)

cvec

- we give you a `Cvec2, Cvec3, and Cvec4` data type.
- entries can be accessed with `v[i]` or `v(i)`
- cvecs can be added, and scalar multiplied
- we also give you dot and cross and normalize

Matrix4

- we will give you a Matrix4 data type
- default constructor gives identity
- give creators such as `makeXRotation`..., `makeTranslation`
- entries can be accessed as `M(i,j)`
- matrices can be multiplied together
- matrix can be multiplied by a cvec4
- we give code for `inv(M)`
  - only works on affine matrices.
- we give code for `transpose(M)`
- we give code for `normalMatrix(M)`
- we also give special code for `makeProjection`
- you will code `transFact(A)` and `linFact(A)` to implement $A = TL$

geometry data types

- in our code we use a `VertexPN` type to store the position and normal. (show)
- we will pass this data to OpenGL buffers
- a few differences from asst1.
- instead of using multiple VBOs for each attribute (position, normal), we pack them in a single VBO.
- we will use an indexed buffer object, IBO, to point to the vertex data making up the triangles.

digression

- IBOs allow for vertex sharing
- so 4 verts can be stored for a quad instead of 6
- at corners, the position of the vertices in the 3 faces are identical, but they have different normals, so we will not share
- show figure
- for a smooth object, we could store a single normal which represents the "true" normal of an underling smooth surface
  - often just an average of the surrounding faces' flat normals
- in this case, the normals do not agree at the vertices of a triangle.
- the normal data is interpolated and we get a smooth appearance
- so the vertex need not be duplicated in the VBO

geometry object

- a `Geometry` object (show)
- created by passed an array of VertexPNs and an array of unsigned shorts,

- during construction, these are placed in one VBO and IBO
- a `Geometry` object will be drawn by wiring the VBO to the appropriate attribute variables
  - this requires stride information due to the interleaving
- and wiring the IBO to the appropriate slot
- then we call `glDrawElements` (instead of glDrawArrays).
- we also give you functions that fills in cube and sphere geometry into an array. (show initCubes)

code specifics

- `initGLState`, now sets up some special stuff for z-buffer and "back face culling"
- our ShaderState struct now has a constructor which reads and loads the shaders, and grabs the handles.
- `initGeometry` initializes a ground geometry and a cube geometry. lets look.
- `drawStuff` sets up matrices and then draws geometry
  - note that we pass eye Coordinates of the light position, for use in the fragment shader.
- motion: in the starter code, we just we simply post multiply an M action to O.
  - not desired.

your code

- you will draw 2 cubes
- you will be able to use the sky-cam or either of the cubes as the eye
- you can move either object or the skycam.
- assuming that we are looking at the scene from the skycam:...
- when moving an object, you will do this using wrt the cube-sky frame we discussed
- when moving the sky-cam, you will do this wrt world-sky, as well as sky.
  - this will require the factoring routines
  - you will need to code `doMtoOwrtA`.
- for more details see spec