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 \( g_{\text{objectRbt}}[i] \) for each object \( i \) and sky matrix \( g_{\text{skyRbt}} \) for a sky camera. (show).
- typically we will use the sky camera’s matrix as the eye’s matrix \( E, \text{eyeRbt} = g_{\text{skyRbt}} \) 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 \( \text{MVM/umodelviewMatrix} \)
- 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 \( P, \text{projmat} \) and pass it to the vertex shader. (much more later)
- vertex shader will multiply eye coordinates by \( P \) to get data to set \( \text{gl_Position} \).

for normals

- explicit normal data will also be placed by us in the openGL buffers.
- vertex shader will transform the object coordinates of the normal into eye coordinates of the normal and pass these out as varying variables
- these will be interpolated over a triangle and used by the fragment shader for material simulation.

flat vs smooth shading

- one can calculate a triangle’s normal from the vertex position data.
  - using a cross product
- to simulate the shading of a flat facet, we pass the face’s geometric normal at all three vertices of a triangle.
  - since the data agrees at the three vertices, the interpolated value at any interior pixel will agree with this value.
- but suppose we want to give the appearance of a smooth object,
- we can calculate and pass a normal which represents the “true” normal of some underlying 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 as a varying variable and we get a smooth appearance
- see material demo

detour: transforming normals

- we use normals for shading
- how do they transform
• suppose i rotate forward
  - normal gets rotated forward
• suppose squash in the $y$ direction
  - (normalized) normal gets higher in the $y$ direction (see figure)
• what is the rule?

**computing normals**

• context: lets $\vec{f}$ be a RHON frame.
• a (non-unit) tangent is vector between two nearby points
• $\vec{t} = (\vec{p}_1 - \vec{p}_0)$
• (non-unit) normal is orthogonal to any tangent
  \[
  \vec{n} \cdot \vec{t} = 0
  \]
• in coordinates
  \[
  \begin{bmatrix}
  nx & ny & nz \\
  tx & ty & tz
  \end{bmatrix} = 0
  \]

**transform**

• let us apply a transform which acts on points as $\vec{f}c \Rightarrow \vec{f}(Ac)$
  - where
  \[
  A = \begin{bmatrix}
  l & t \\
  0 & 1
  \end{bmatrix}
  \]
• $[tx', ty', tz']^t := l[tx, ty, tz]^t$ are the coordinates of the transformed tangent.
• so lets plug in $l^{-1}l$
  \[
  (\begin{bmatrix}
  nx & ny & nz \\
  tx & ty & tz
  \end{bmatrix} l^{-1}l \begin{bmatrix}
  tx & ty & tz
  \end{bmatrix}) = 0
  \]
• define $[nx', ny', nz'] := [nx, ny, nz]l^{-1}$
  \[
  \begin{bmatrix}
  nx' & ny' & nz' \\
  tx' & ty' & tz'
  \end{bmatrix} = 0
  \]
• so $[nx', ny', nz']^t$ must be the (non unit) normal of the transformed geometry

so

\[
\begin{bmatrix}
nx' \\
ny' \\
nz'
\end{bmatrix} = l^{-t} \begin{bmatrix}
nx \\
ny \\
nz
\end{bmatrix}
\]
• as mat4, we use

\[
normalMatrix(A) = \begin{bmatrix}
l^{-t} & 0 \\
0 & 1
\end{bmatrix}
\]
inv transpose

- so inverse transpose/ transpose inverse is the rule
- for rotations, transpose = inverse.
- for scales, transpose = nothing.
- in the code we will send \( A \) and its normal matrix to the vertex shader.
- we will compute and pass the inv tpos MVM as a uniform variable \( \text{NMVM}/\text{uNormalMatrix} \)

vertex shader code

- does the matrix multiplies
- sets \( \text{gl\_Position} \).
- also outputs the eye coordinates of the vertex and the normal.
- note the glsl vec3 and vec4 casting.

fixed function

- finds screen pixels inside of triangle
- interpolates values for the varying variables.
- \( \text{vPosition} \) 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 \( \text{Cvec2, Cvec3, and Cvec4} \) data type.
- entries can be accessed with \( \text{v}[i] \) or \( \text{v}(i) \)
- cvecs can be added, and scalar multiplied (only implemented with scalar on the rhs).
- 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 \( \text{makeXRotation\ldots, makeTranslation} \)
- entries can be accessed as \( M(i,j) \)
- matrices can be multiplied together
- matrix can be multiplied by a cvec4
- we give code for \( \text{inv}(M) \)
  - only works on affine matrices.
- we give code for \( \text{transpose}(M) \)
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.

IBOs

- IBOs allow for vertex sharing
- so 4 verts can be stored for a quad instead of 6
- we want to draw the cube using flat shading
- so at cube corners, the position of the vertices in the 3 faces are identical, but they have different normals
- so we will not share these.

geometry object

- a `Geometry` object (show)
- created by passed an array of `VertexPNs` for the VBO and an array of unsigned shorts for the IBO
- 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`).
- see `initGround`
- 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
• when you are viewing from the sky-cam you can move either object or 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 use roll free orbit, and roll free egomotion.
• this will require the factoring routines
• you will need to code doMtoOwrtA.
• for more details see spec