CS380: Introduction to Computer Graphics

Hello World 3D

Chapter 6

Min H. Kim

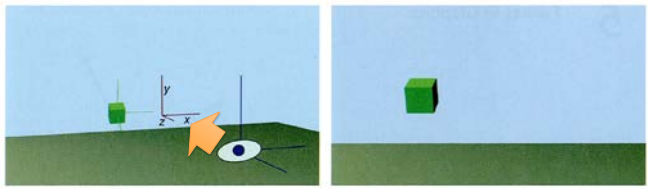
KAIST School of Computing

Respect

SUMMARY

The eye's view

- The world frame is in red
- The object frame is in green
- The eye frame is in blue
  - The eye is looking down its negative z toward the object.



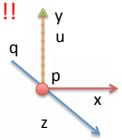
(a) The frames (b) The eye's view

The eye matrix (camera transform)

- Specifying the eye matrix $\vec{\mathbf{e}}^t = \vec{\mathbf{w}}^t E$ by:
  - the eye point $\tilde{p}$
  - the view point (where the eye looks at) $\tilde{q}$
  - the up vector $\vec{u}$

NB P. 35 in our textbook contains errors (see errata)!!!

$$\mathbf{z} = normalize(p - q)$$
$$\mathbf{x} = normalize(\mathbf{u} \times \mathbf{z})$$
$$\mathbf{y} = \mathbf{z} \times \mathbf{x}$$
$$normalize(\mathbf{c}) = \mathbf{c} / \sqrt{c_1^2 + c_2^2 + c_3^2}$$



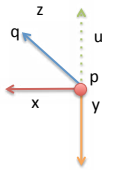
NB matrix sent to the vertex shader is *E^{-1}*O*

$$E = \begin{bmatrix} x_1 & y_1 & z_1 & p_1 \\ x_2 & y_2 & z_2 & p_2 \\ x_3 & y_3 & z_3 & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The view matrix (gluLookAt)

- Specifying the view matrix $V = E^{-1}$
  - the eye point $\tilde{p}$
  - the view point (where the eye looks at) $\tilde{q}$
  - the up vector $\vec{u}$

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = E^{-1} O \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

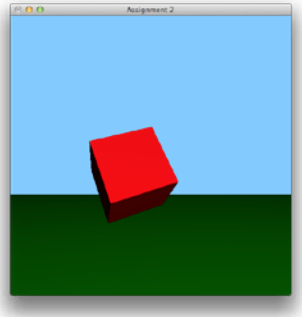


$$\mathbf{z} = normalize(q - p)$$
$$\mathbf{x} = normalize(\mathbf{u} \times \mathbf{z})$$
$$\mathbf{y} = \mathbf{x} \times \mathbf{z}$$
$$normalize(\mathbf{c}) = \mathbf{c} / \sqrt{c_1^2 + c_2^2 + c_3^2}$$

NB gluLookAt is not the part of modern OpenGL!

Chapter 6

HELLO WORLD 3D



Coordinates: cvec.h

- In our homework 2
- Coordinate vectors are defined as a C++ template class, Cvec, in the 'cvec.h' file.
- For 2D vectors (Cvec2), we have implementations, additions, multiplications, etc.
  - u + v, r * v, …
- For 4D vectors (Cvec4), we call the entries: x, y, z, w
  - A point: w = 1
  - A vector: w = 0

Matrices: matrix4.h

- Matrix4: affine matrix
  - Right-multiplication with a Cvec4: M * v
  - Multiplication of two Matrix4s: M * N
  - Inverse operation: inv(M)
  - Transpose operation: transpose(M)
- [Homework 2-2]
  - Download: hw3d.zip
  - Complete $A = TL$ M = transFact(M) * linFact(M)
  - Write transFact(M): a translational factor of M
  - Write linFact(M): a linear factor of M

Matrices: matrix4.h

- The inverse transpose of the linear factor of M: normalMatrix()

$$\begin{bmatrix} nx' \\ ny' \\ nz' \end{bmatrix} = l^{-t} \begin{bmatrix} nx \\ ny \\ nz \end{bmatrix}.$$

```
inline Matrix4 normalMatrix(const Matrix4& m) {
  Matrix4 invm = inv(m);
  invm(0, 3) = invm(1, 3) = invm(2, 3) = 0;
  return transpose(invm);
}
```

- Do *M* to *O* with respect to *A*: doMtoOwrtA()

$$AMA^{-1}O$$

```
static RigTForm doMtoOwrtA(const RigTForm& M, ... A) {
  return A * M * inv(A) * O;}
```

- Make a mixed frame of O and E: makeMixedFrame(O,E)

$$(O)_T(E)_R$$

```
static Matrix4 makeMixedFrame(...) {
  return transFact(objRbt) * linFact(eyeRbt);}
```

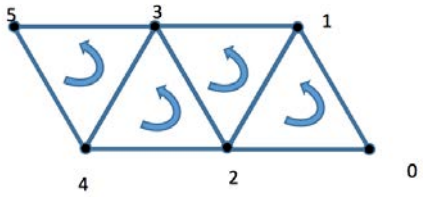

Drawing a shape

- Clear image colors
- Clear z-buffer
- Enable back-face culling
- Set the greater z value to mean closer

```
static void initGLState() {
  glClearColor(128./255., 200./255., 255./255., 0.); //  RGBA
  glClearDepth(0.0);
  glCullFace(GL_BACK); // back-face culling
  glEnable(GL_CULL_FACE); // back-face culling
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_GREATER); // closer
}
```


Drawing a shape

- OpenGL Order:
  0, 1, 2,
  2, 1, 3,
  2, 3, 4, ...
- Consist orientation of the triangles

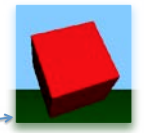


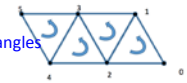
Drawing a shape

- The cube is made up of six squares (12 triangles)

```
GlfloatfloorVerts[18] =
{
 -floor_size, floor_y, -floor_size,
  floor_size, floor_y,  floor_size,
  floor_size, floor_y, -floor_size,
 -floor_size, floor_y, -floor_size,
 -floor_size, floor_y,  floor_size,
  floor_size, floor_y,  floor_size
}; // 18 = 2*3*3 // two triangles
GlfloatcubeVerts[108] =
{
 -0.5, -0.5, -0.5,
 -0.5, -0.5, +0.5,
 +0.5, -0.5, +0.5,
 ...
// 99 more vertices not shown
}; // 108=2*3*3*6*3
```

```
GlfloatfloorNorms[18] =
{
 0, 1, 0,
 0, 1, 0,
 0, 1, 0,
 0, 1, 0,
 0, 1, 0,
 0, 1, 0
}; // 18 = 2*3*3 // two triangles
GlfloatcubeNorms[108] =
{
 +0.0, -1.0, +0.0,
 +0.0, -1.0, +0.0,
 +0.0, -1.0, +0.0,
 ...
// 99 more vertices not shown
}; // 108=2*3*3*6*3
```





Initialize VBOs

- Initialize the VBOs of the floor:

```
static GLuint floorVertBO, floorNormBO, cubeNormBO, cubeNormBO;

static void initVBOs(void) {
  // floor vertices
  glGenBuffers(1, &floorVertBO);
  glBindBuffer(GL_ARRAY_BUFFER, floorVertBO);
  glBufferData(GL_ARRAY_BUFFER, 18*sizeof(GLfloat), floorVerts, GL_STATIC_DRAW);

 // floor normals
  glGenBuffers(1, &floorNormBO);
  glBindBuffer(GL_ARRAY_BUFFER, floorNormBO);
  glBufferData(GL_ARRAY_BUFFER, 18*sizeof(GLfloat), floorNorms, GL_STATIC_DRAW);

...
```


Draw VBOs

- Initialize the VBOs of the cube:

```
 // cube vertices
  glGenBuffers(1, &cubeVertBO);
  glBindBuffer(GL_ARRAY_BUFFER, cubeVertBO);
  glBufferData(GL_ARRAY_BUFFER, 108*sizeof(GLfloat), cubeVerts, GL_STATIC_DRAW);

 // cube normals
  glGenBuffers(1, &cubeNormBO);
  glBindBuffer(GL_ARRAY_BUFFER, cubeNormBO);
  glBufferData(GL_ARRAY_BUFFER, 108*sizeof(GLfloat), cubeNorms, GL_STATIC_DRAW);
}
```


Draw VBOs

- Draw an object using the VBOs:

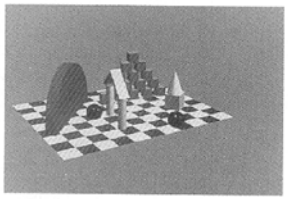
```
void drawObj(GLuint vertbo, GLuint normbo, int numverts){
  glBindBuffer(GL_ARRAY_BUFFERS, vertbo); // vertex buffer object
  safe_glVertexAttribPointer(h_aVertex);
  safe_glEnableVertexAttribArray(h_aVertex);

  glBindBuffer(GL_ARRAY_BUFFERS, normbo); // normal buffer object
  safe_glVertexAttribPointer(h_aNormal);
  safe_glEnableVertexAttribArray(h_aNormal);

  glDrawArrays(GL_TRIANGLES, 0, numverts);

  safe_glDisableVertexAttribArray(h_aVertex);
  safe_glDisableVertexAttribArray(h_aNormal);
}
```

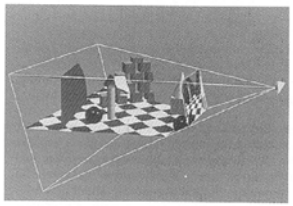

Index Buffer Object (IBO)

- IBO points to the vertex data
- to make up the triangles.
- IBOs allow for vertex sharing.
- General 3D model includes: (1) vertices, (2) index of vertices, (3) normals, (4) texture coordinates.
- So 4 verts can be stored for a quad instead of 6.
- The vertex would not be duplicated in the VBO
- At corners, the position of the vertices in the 3 faces are identical, but they have different normals, so we will not share.

Camera projection (overview)

- Intrinsic properties of the camera (the eye)
  - A field of view
  - Window aspect ratio
  - Near/far field in Z (clipping)
  - Represented as camera frustum
- sendProjectionMatrix: send the camera matrix to the vertex shader
- uniform mat4 uProjMatrix;





Modelview matrix

- Modelview matrix (*MVM*) $E^{-1}O$
  - Describes the orientation and position of the view E^{-1} and the orientation and position of the object O with respect to the eye frame $\vec{\mathbf{e}}^t$

$$\tilde{p} = \vec{\mathbf{o}}^t \mathbf{c} = \vec{\mathbf{w}}^t O \mathbf{c} = \vec{\mathbf{e}}^t E^{-1} O \mathbf{c}$$

  - The vertex shader will take these vertex data and perform the multiplication $E^{-1}O\mathbf{c}$, producing the eye coordinates used in rendering
  - uniform mat4 uModelViewMatrix;
- normalMatrix() produces the inverse transpose of the linear factor to get uniform normal *NMVM*
  - uniform mat4 uNormalMatrix;

Display VBOs

- Display the objects

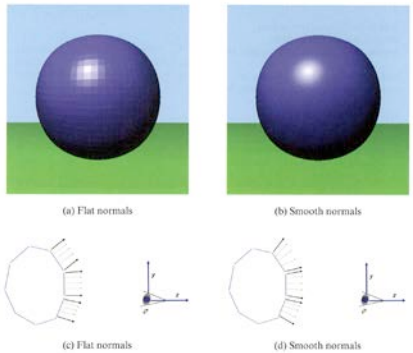
```
Static void display(){
  safe_glUseProgram(h_program_);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  Matrix4 projmat = makeProjection(frust_fovy, frust_ar, frust_near, frust_far);
// arguments: field-of-view in the y axis, aspect ratio, zNear, zFar
  sendProjectionMatrix(projmat);
// send this matrix to the vertex shader

  Matrix4 MVM = inv(eyeRbt); // the view matrix (omit the object mat for floor)   E^-1 O
  Matrix4 NMVM = normalMatrix(MVM); // inverse transpose matrix of the linear factor
  sendModelViewNormalMatrix(MVM, NMVM); // to the vertex shader

  safe_glVertexAttrib3f(h_aColor, 0.6, 0.8, 0.6); // set the color of the floor object
  drawObj(floorVertBO, floorNormBO, 6);

  MVM = inv(eyeRbt) * objRbt; // model view matrix
  NMVM = normalMatrix(MVM);
  sendModelViewNormalMatrix(MVM, NMVM);
```


Display VBOs

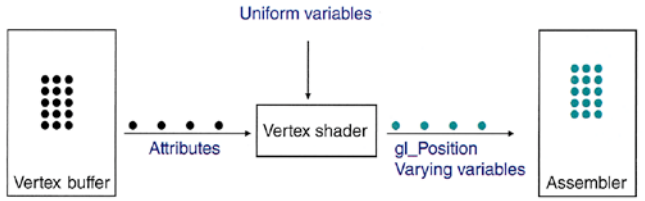
```
  safe_glVertexAttrib3f(h_aColor, 0.0, 0.0, 1.0);
  drawObj(cubeVertBO, cubeNormBO, 36);

  glutSwapBuffers();

  if (glGetError() != GL_NO_ERROR){
    const Glubyte *errString;
    errString=gluErrorString(errCode);
    printf("error: %s\n", errString);
  }
}
```

Interpolating normals in FS

- Use interpolated normals in fragment shader



Vertex shader variables



- Three types of variables:
  - uniform: does not change per primitives; read-only in shaders
  - in (vertex sh.): input changes per vertex, read-only;
  - in (frag. sh.): interpolated input; read-only
  - out: shader-output; VS to FS; FS output.

Vertex shader (3D → 2D)

- Takes the object coordinates of every vertex position and turns them into eye coordinates, as well as the vertex's normal coordinates

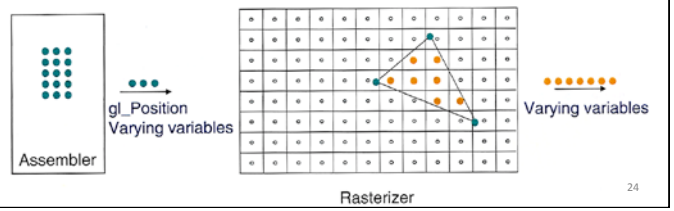
```
#version 130

uniform Matrix4 uModelViewMatrix;
uniform Matrix4 uNormalMatrix;
uniform Matrix4 uProjMatrix;

in vec3 aColor;
in vec4 aNormal;
in vec4 aVertex;

out vec3 vColor;
out vec3 vNormal;
out vec3 vPosition;

void main()
{
  vColor = aColor;
  vPosition = uModelViewMatrix * aVertex;
  vec4 normal = vec4(aNormal.x, aNormal.y,
aNormal.z, 0.0);
  vNormal = vec3(uNormalMatrix * normal);
  gl_Position = uProjMatrix * vPosition;
}
```

$E^{-1}O\mathbf{c}$

$PE^{-1}O\mathbf{c}$

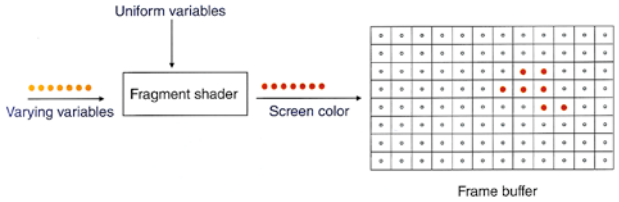
Clip coordinates

- 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

- Compute material appearance
- By using uniform variables (light and color)
- NB We use the eye coordinates of variables (light, normal, position) in FS $E^{-1}Oc$ since rasterization was already done.



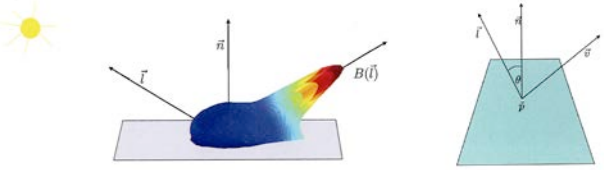
Fragment shader (pixel color)

```
#version 130

uniform vec3 uLight1, uLight2; // the eye coordinates of lights
uniform vec3 uColor; // use glVertexUniform3f call to pass them to the shaders

in vec3 vNormal; // eye coordinates
in vec3 vPosition; // eye coordinates

out fragColor; // output variable of the interpolated color data per pixel

void main()
{
  vec3 toLight = normalize(uLight – vec3(vPosition)); // direction to the point light
  vec3 normal = normalize(vNormal);
  float diffuse = max(0.0, dot(normal, toLight1)); // negative clamping of the cosine law
  diffuse += max(0.0, dot(normal, toLight2)); // add light2 energy
  vec3 intensity = uColor * diffuse; // only apply diffuse reflectance function of BRDF
  fragColor = vec4(intensity.x, intensity.y, intensity.z, 1.0); // 4th element is opacity.
}
```

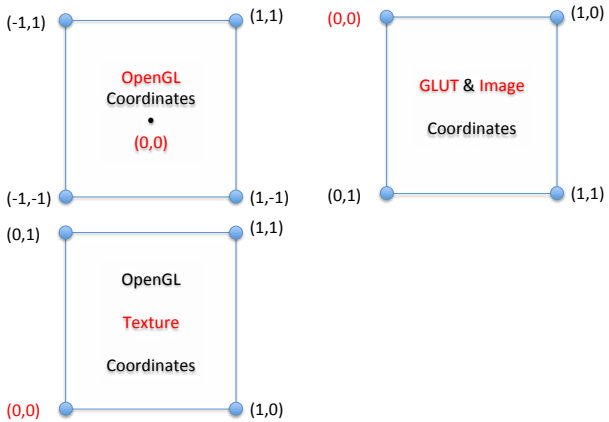

Material appearance (overview)

- Bidirectional reflectance distribution function (BRDF)
- Simple reflectance model = diffuse + specular
- The appearance change by the view vector $\vec{v}$ is managed by OpenGL

$$L_{diffuse} = R_{diffuse}\frac{\cos(\theta)}{d^2} \qquad \cos(\theta) = \vec{n}\cdot\vec{l}$$



The origin in 2D coordinate systems



Motion callback

- We need to register two callback functions:
  - glutMouseFunc(mouse);
  - glutMotionFunc(motion);

```
void mouse(int button, int state, int x, int y){
 g_mouseClickX = x;
 g_mouseClickY = g_windowHeight - y - 1;
// conversion from GLUT window-coordinate-system to OpenGL window-coordinate-system

 g_mouseLClickButton |= (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN);
 g_mouseRClickButton |= (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN);
...
}
```

```
void motion(int x, int y){
 const double dx = x - g_mouseClickX;
  const double dy = g_windowHeight - y - 1 - g_mouseClickY;
...
(do something)
...
 g_mouseClickX = x;
 g_mouseClickY = g_windowHeight - y - 1;
}
```


Motion callback

- Move the object

```
Matrix4 M = makeXRotation(deltay) * makeYRotation(deltax);
Matrix4 A = makeMixedFrame(objRbt, eyeRbt);
objRbt = doMtoOwrtA(M, objRbt, A);
```

- Move the eye

```
objRbt = doMtoOwrtA(inv(M), eyeRbt, A);
```

- Perform ego motion

```
objRbt = doMtoOwrtA(inv(M), eyeRbt, eyeRbt);
```

- NB For the eye motions, we invert the M so that the mouse movements produce the image movements in more desired directions.

Assignment2 video

Assignments2 hints

- you will draw 2 objects,
- you will be able to use the sky-cam or either object 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, 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