Modelling, Viewing, and Projection

8 March 2007
CMPT370
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Review last time

- Rotations in 3D
  - Euler angles, gimbal lock problem
  - Virtual trackball
    - 2D mouse coords -> pt on hemisphere
  - Axis-angle representation
- Quaternions
  - Converting from axis-angle to quaternion
  - Using quaternions for rotations
  - Constructing a 4x4 rotation matrix
    - GameDev article on quaternions

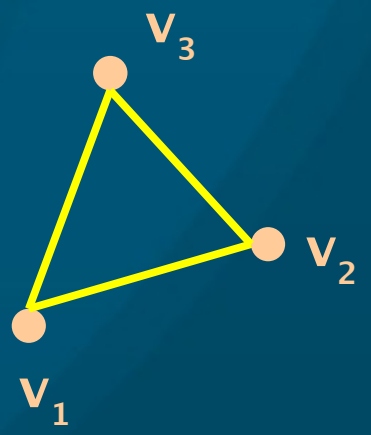
Outline for today

- Modelling: vertex lists, face lists, edge lists
  - OpenGL vertex arrays
  - OpenGL display lists (see RedBook ch7)
- Viewing: (see RedBook ch3)
  - Positioning the camera: model-view matrix
  - Selecting a lens: projection matrix
  - Clipping: setting the view volume
    - See UC-Davis ECS175 graphics course

Modelling polygons

- Simple representation (see CubeView):

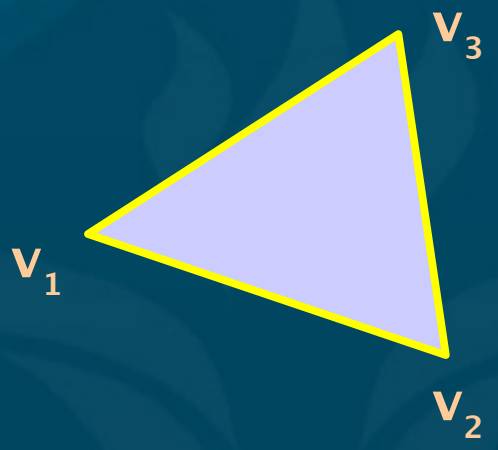
```
glBegin( GL_POLYGON );
    glVertex3f( 0.0, 0.0, 0.0 );
    glVertex3f( 1.0, 1.5, 2.2 );
    glVertex3f( -2.3, 1.5, 0.0 );
glEnd();
```



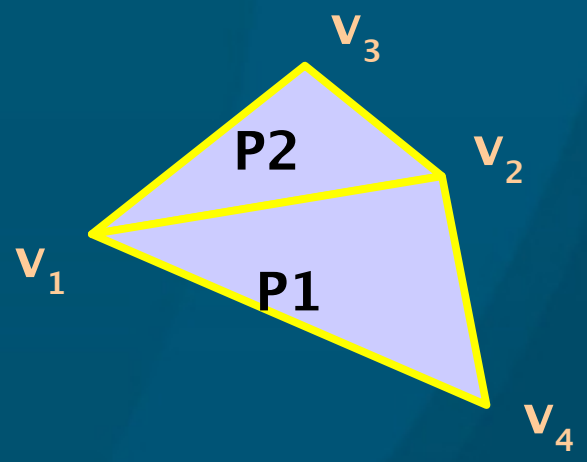
- Problems: inefficient, unstructured
  - What if we want to move a vertex to a new location?

Inward/outward facing polygons

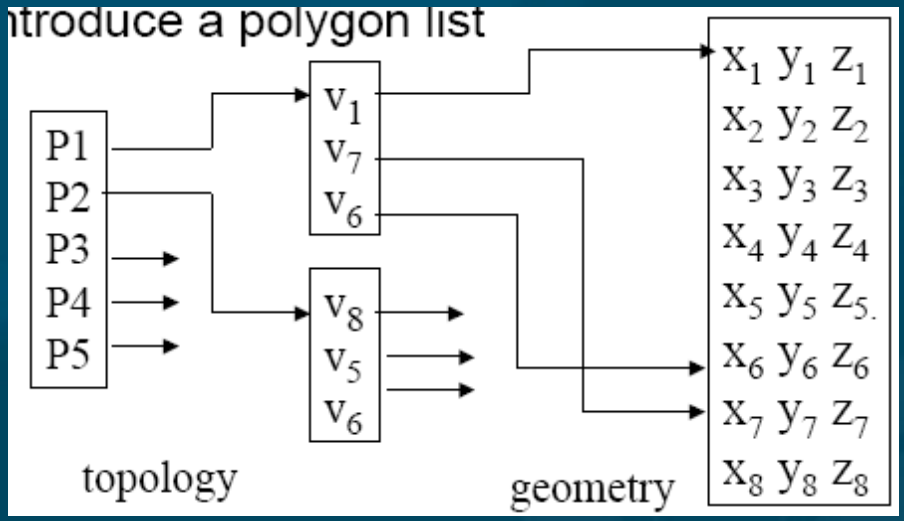
- The normal vector for a polygon follows the right-hand rule
- Specifying vertices in order (v_1, v_2, v_3) is same as (v_2, v_3, v_1) but different from (v_1, v_3, v_2)
- When constructing a closed surface, make sure all your polygons face outward
- Backface culling may mean inward-facing polygons don't get rendered



Vertex lists and face lists

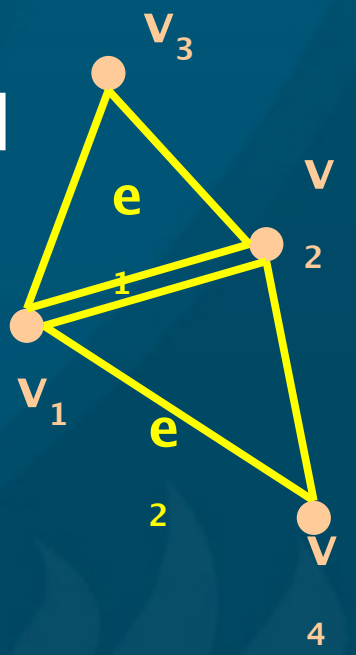


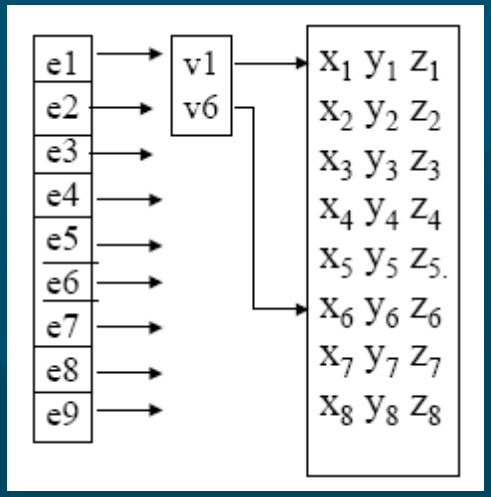
- Separate geometry from topology
  - Vertex coords are geometry
  - Connections between vertices (edges, polygons) are topology
- Vertex list:
  - $v_1 = \{x_1, y_1, z_1\}$
  - $v_2 = \{x_2, y_2, z_2\}$
- Polygon/face list:
  - $P_1 = \{v_1, v_2, v_3\}$
  - $P_2 = \{v_1, v_4, v_2\}$



Edge lists

- If only drawing edges (wireframe):
  - Many shared edges may be duplicated
  - Similar to face list but for edges:
    - Does not represent the polygons!
- Vertex list:
  - $v_1 = \{x_1, y_1, z_1\}$
  - $v_2 = \{x_2, y_2, z_2\}$
- Edge list:
  - $e_1 = \{v_1, v_2\}$
  - $e_2 = \{v_1, v_4\}$





OpenGL vertex arrays

- Stores a vertex list in the graphics hardware
  - Six types of arrays: vertices, colours, colour indices, normals, texture coords, edge flags
- Our vertex list in C:
  - GLfloat verts[][3] = {{0.0, 0.0, 0.0}, {0.1, 0.0, 0.0}, ...}
- Load into hardware:
  - glEnableClientState(GL_VERTEX_ARRAY);
  - glVertexPointer(3, GL_FLOAT, 0, verts);
    - **3: 3D vertices**
    - **GL_FLOAT: array is of GLfloat-s**
    - **0: contiguous data**
    - **verts: pointer to data**

Using OpenGL vertex arrays

- Use glDrawElements instead of glVertex
- Polygon list references indices in the stored vertex array
  - GLubyte cubeIndices[24] = {0,3,2,1, 2,3,7,6, 0,4,7,3, 1,2,6,5, 4,5,6,7, 0,1,5,4};
  - Each group of four indices is one quad
- Draw a whole object in one function call:
  - glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);

OpenGL display lists

- Take a group of OpenGL commands (e.g., defining an object) and store in hardware
- Can change OpenGL state, camera view, other objects, etc. without redefining this stored object
- Creating a display list:
  - GLuint cubeDL = glGenLists(1);
  - glNewList(cubeDL, GL_COMPILE);
    - glBegin(); ....; glEnd();
  - glEndList();
- Using a stored display list:
  - glCallList(cubeDL);

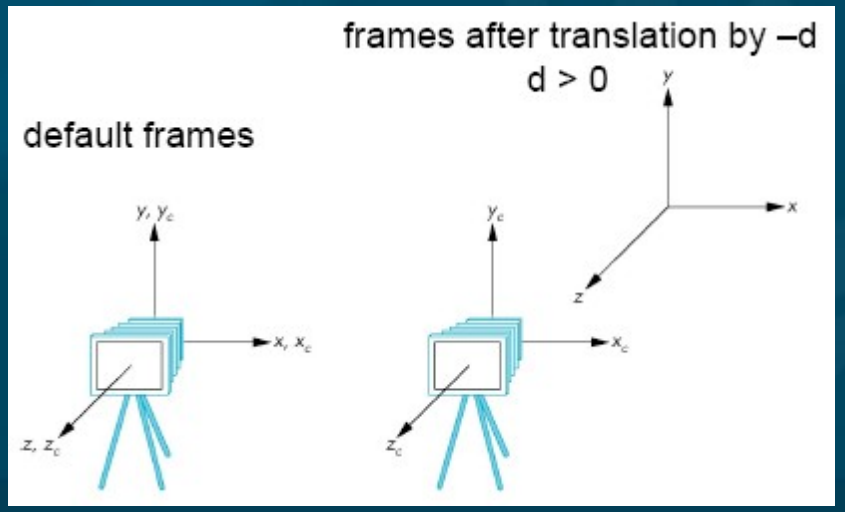
See RedBook ch7

Outline for today

- Modelling: vertex lists, face lists, edge lists
  - OpenGL vertex arrays
  - OpenGL display lists
- Viewing:
  - Positioning the camera: model-view matrix
  - Selecting a lens: projection matrix
  - Clipping: setting the view volume

Positioning the camera: model-view

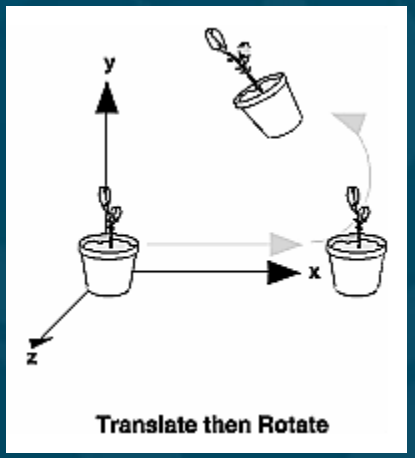
- The model-view matrix describes where the world is relative to the camera
  - Initially identity matrix: camera is at center of world, facing in negative z direction
- Say we want to see an object placed at the origin:
  - Move the camera in the +z direction, or
  - Move the world frame in the -z direction
  - Both are equivalent: glTranslatef(0., 0., -d);



Order of transformations

- C: current model-view matrix
- M: new additional transformation,
  via glMultMatrix, glTranslate, glRotate, etc.
- v: vertex to be transformed

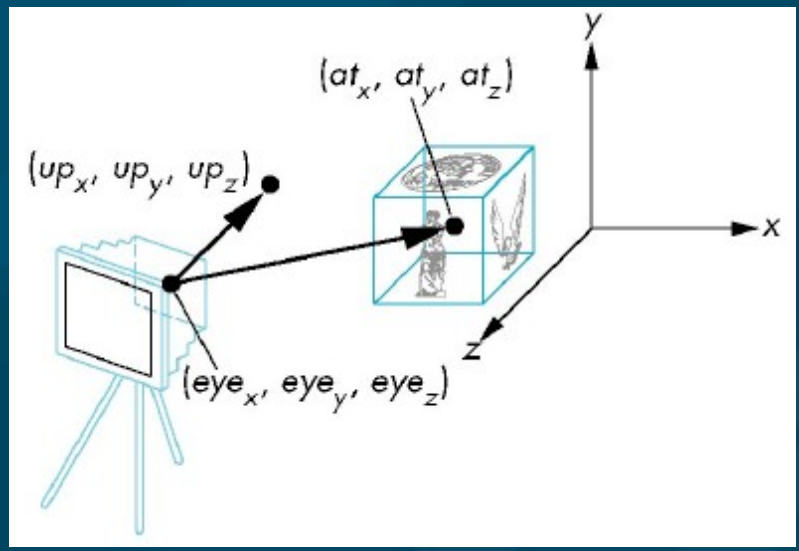
- OpenGL applies transforms in the order: CMv
- So the last transform is applied first!
  - glMatrixMode(GL_MODELVIEW);
  - glLoadIdentity();
  - glRotatef(60., 0., 0., 1.);
  - glTranslatef(10., 0., 0.);
  - glBegin(GL_POINTS);
    - glVertex3fv(vert);



Translate then Rotate

gluLookAt

- Handy helper function for setting up model-view
  - #include <GLU.h>
- Specify eye coords, where you want to look at, and direction of "up" vector:
  - glMatrixMode(GL_MODELVIEW);
  - glLoadIdentity();
  - gluLookAt(eye_x, eye_y, eye_z, at_x, at_y, at_z, up_x, up_y, up_z);

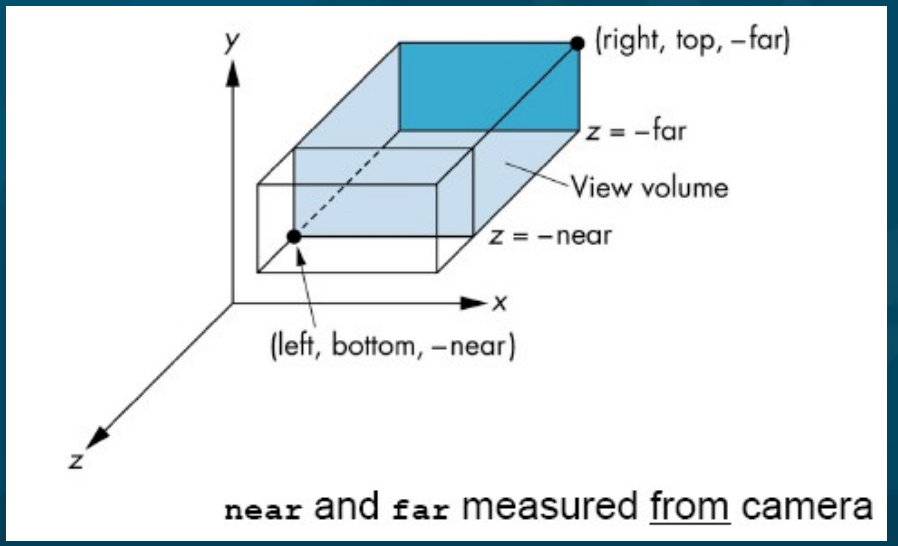


Selecting a lens: Projection

- The projection matrix maps 3D points in the camera's frame to 2D points on the image plane
  - Input to projection matrix is homogeneous coords after model-view matrix is applied
  - After multiplying by projection matrix,
    - Divide to ensure homogeneous coords: [x y z 1]
    - Take just the (x, y) coords as coords on image plane
  - Default projection matrix is the identity
    - Orthographic projection onto the xy plane

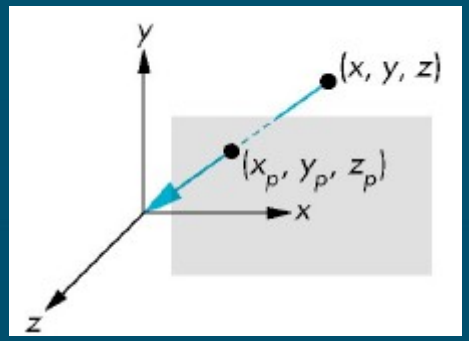
Orthographic projection

- The manual way:
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity();
  - glMultMatrix(...);
- The easier way with glOrtho():
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity();
  - glOrtho(left, right, bottom, top, near, far);

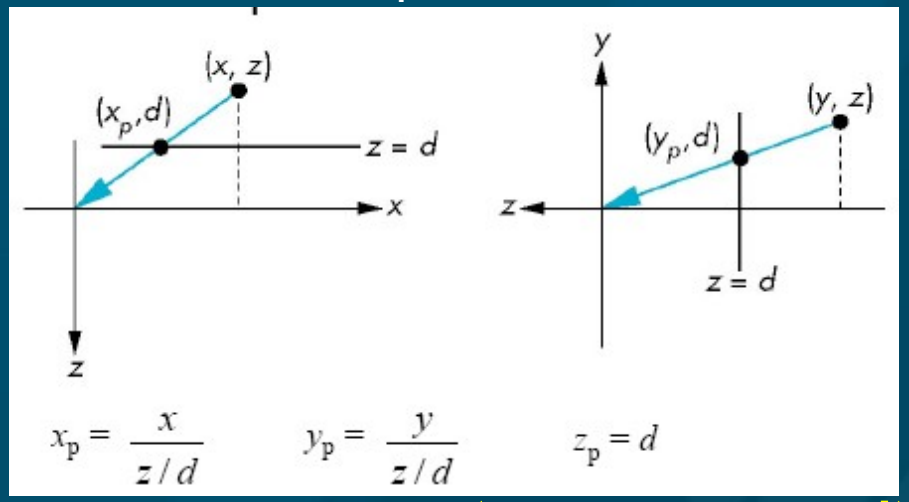


Perspective projection

- Consider a perspective projection with center of projection (CoP) at origin, and image plane at z=d:



View from top: View from side:

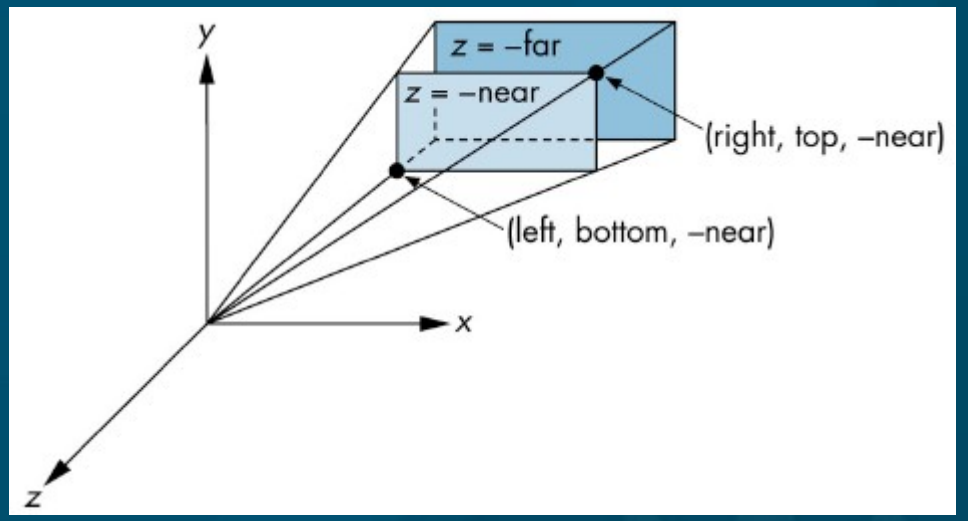


- A point p = (x,y,z) projects to q = (x_p, y_p, z_p=d) via

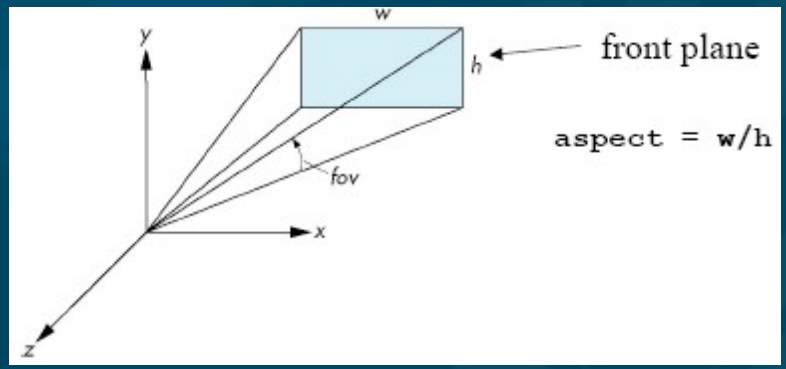
$$q = Mp, \text{ where } M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix}$$

Specifying perspective projection

- Can also do this manually with glMultMatrix()
- Or use glFrustum():
  - glFrustum(left, right, bottom, top, near, far)



- Or use gluPerspective():
  - gluPerspective(fov, aspect, near, far);



- Easier to use than glFrustum()

Outline for today

- Modelling: vertex lists, face lists, edge lists
  - OpenGL vertex arrays
  - OpenGL display lists
- Viewing:
  - Positioning the camera: model-view matrix
  - Selecting a lens: projection matrix
  - Clipping: setting the view volume

TODO

- Lab4: due next week Thu 15Mar
  - Add a virtual trackball using quaternions