Transform Matrices, Viewing, Modelling

19 February 2009
CMPT370
Dr. Sean Ho
Trinity Western University

Review last time

- Scalars, vectors, points
- Vector spaces, affine spaces (+point)
- Lines, rays, line segments
- Curves, surfaces
- Normal vectors
- Convex hull
- Linear independence
- Basis, frame (+point)

What's on for today

- Math for 3D graphics: homogeneous coordinates
  - 4x4 transform matrices
  - Translate, scale, rotate
- Viewing: (see RedBook ch3)
  - Positioning the camera: model-view matrix
  - Selecting a lens: projection matrix
  - Clipping: setting the view volume
- Modelling: vertex lists, face lists, edge lists
  - OpenGL vertex arrays and display lists

Homogeneous coordinates

- We use a 4-tuple as a homogeneous representation for both vectors and points
  - $[\ \alpha_1\ \alpha_2\ \alpha_3\ 0\ ]^T$ is a vector
  - $[\ \beta_1\ \beta_2\ \beta_3\ 1\ ]^T$ is a point
  - Relative to current coordinate frame
  - Any 4-tuple $[\ x\ y\ z\ w\ ]^T$ maps to a point as
    - $[\ x/w\quad y/w\quad z/w\quad 1\ ]^T$
    - If $w=0$, the 4-tuple represents a vector
  - Each point in 3D maps to a line through the origin in 4D

Changing coordinate systems

- Say we have a vector whose representation in one basis (e_1, e_2, e_3) is $v = \{\alpha_1\ \alpha_2\ \alpha_3\}$.
  - What is the representation for the same vector in a different basis, $\{d_1, d_2, d_3\}$?
- Represent each old basis vec e_i in the new basis:
  - $e_1 = \gamma_{11}d_1 + \gamma_{12}d_2 + \gamma_{13}d_3$
  - $e_2 = \gamma_{21}d_1 + \gamma_{22}d_2 + \gamma_{23}d_3$
  - $e_3 = \gamma_{31}d_1 + \gamma_{32}d_2 + \gamma_{33}d_3$

3x3 vector transform matrix

- These nine coefficients form a 3x3 vector transform matrix M:
  - $$M = \begin{pmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{pmatrix}$$
  - $w = M^T v$, where
    - **$v = \{\alpha_1\ \alpha_2\ \alpha_3\}$ is representation in old basis**
    - **$w = \{\beta_1\ \beta_2\ \beta_3\}$ is representation in new basis**

Change of frames

- Something similar happens to change frames:
  - Old frame is (P, e_1, e_2, e_3)
  - New frame is (Q, d_1, d_2, d_3)
  - Represent old frame in new basis
  - 12 degrees of freedom in affine transform

$$M = \begin{pmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{pmatrix}$$

Translation matrix

$$T = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Translate a point p by multiplying by T (=M^T):
  - p' = Tp

Scaling matrix

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Scale a point p by multiplying by T:
  - p' = Tp
- Fixed point of origin (scaling away from origin)
- Reflection is via negative scale factors

Rotation matrix

$$R=\begin{pmatrix}\cos(\theta) & -\sin(\theta) & 0 & 0\\ \sin(\theta) & \cos(\theta) & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

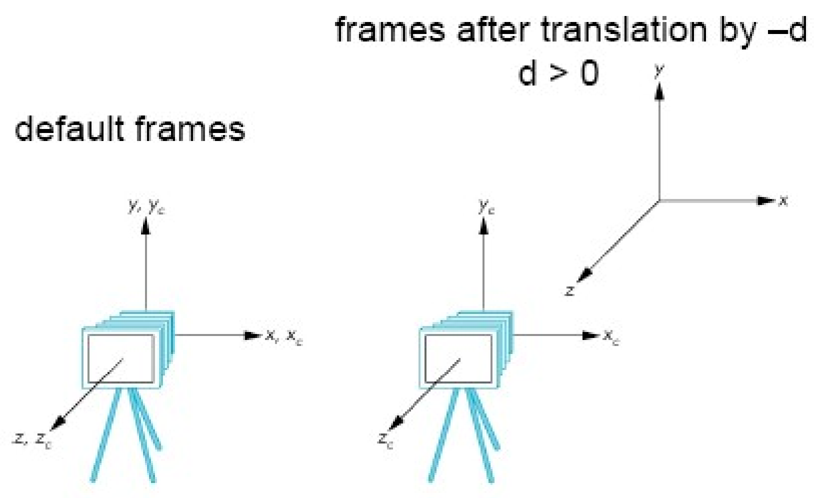
- Rotate by an angle θ about the z axis
- Similar matrices for rotating about x, y axes
- 3 Euler angles
- Order of operations is important!
  - Rotation in 3D is non-Abelian

What's on for today

- Math for 3D graphics: homogeneous coordinates
  - 4x4 transform matrices
  - Translate, scale, rotate
- Viewing: (see RedBook ch3)
  - Positioning the camera: model-view matrix
  - Selecting a lens: projection matrix
  - Clipping: setting the view volume
- Modelling: vertex lists, face lists, edge lists
  - OpenGL vertex arrays and display lists

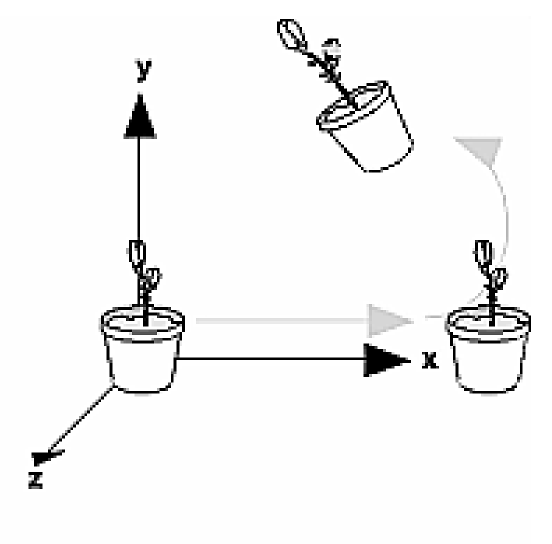
Placing the camera: model-view

- The model-view matrix describes where the world is relative to the camera
  - Initially identity matrix: camera is at origin of world, facing in negative z direction
- Say we want to see an object at the origin: either
  - 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

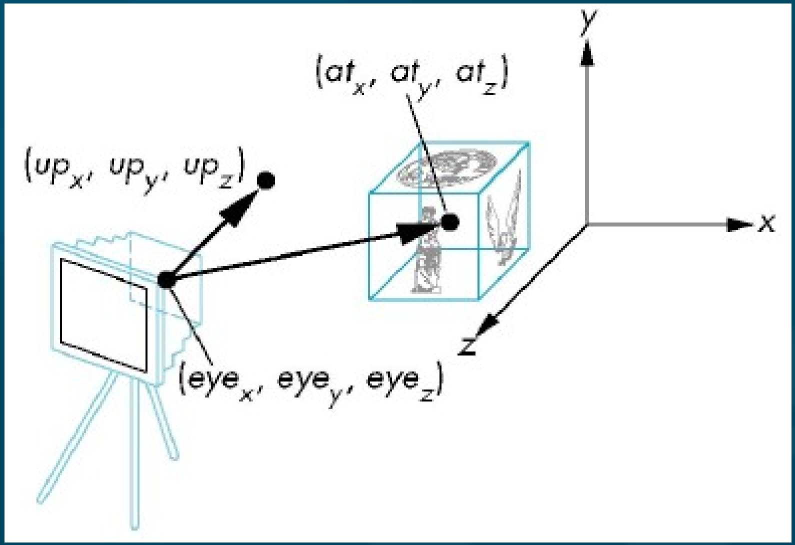
  - C: current model-view matrix
  - M: new additional transformation,
    via glMultMatrix, glTranslate, glRotate, etc.
  - v: vertex to be transformed
- OpenGL applies transforms in the order: CMv
- So the last transform is applied first!
  - glMatrixMode(GL_MODELVIEW);
  - glLoadIdentity();
  - glRotatef(60., 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);

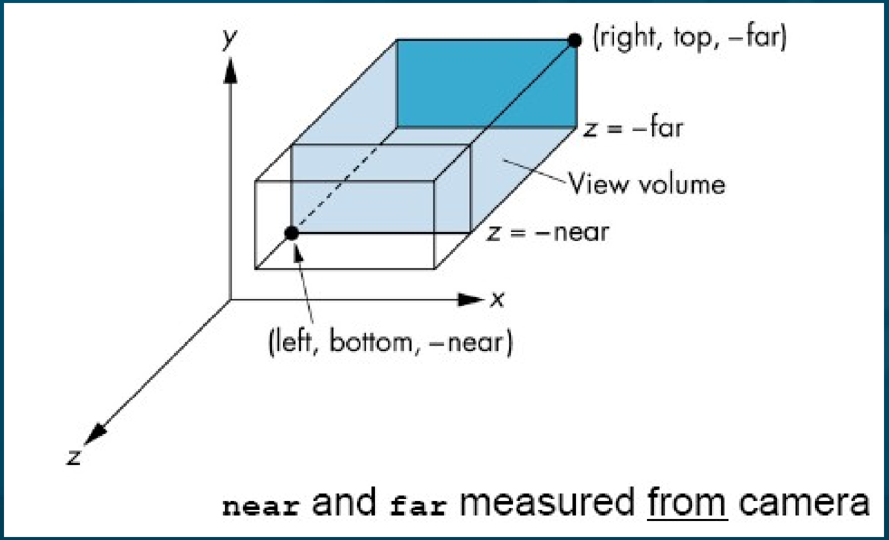


Selecting a lens: Projection

- 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

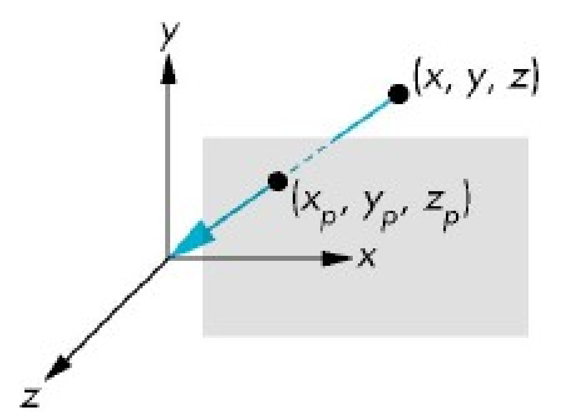
- The manual way:
  - `glMatrixMode( GL_PROJECTION );`
  - `glLoadIdentity();`
  - `glMultMatrix(...);`
- The easier way with glOrtho():
  - `glMatrixMode( GL_PROJECTION );`
  - `glLoadIdentity();`
  - `glOrtho( left, right, bottom, top, near, far );`

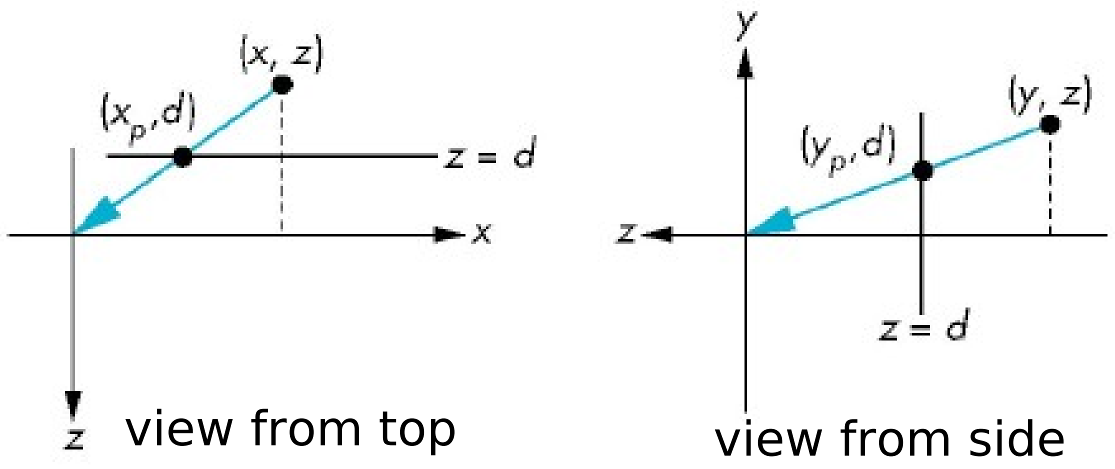


`near` and `far` measured from camera

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

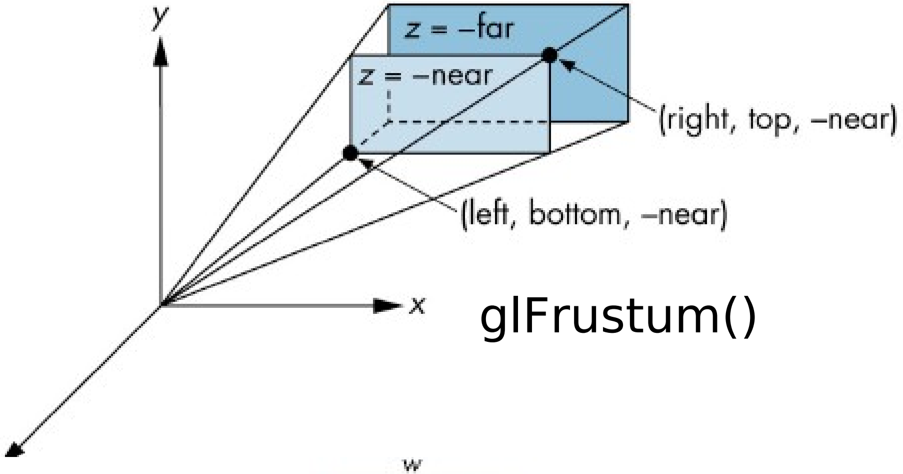
$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

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

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

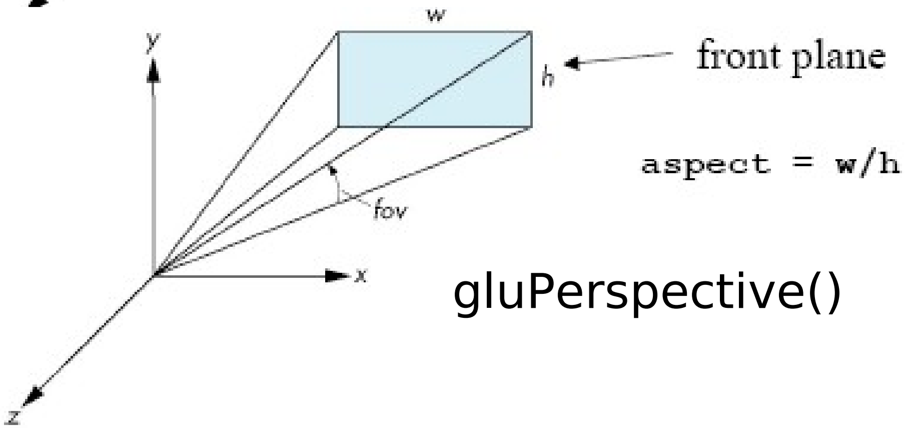
Setting perspective in OpenGL

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



glFrustum()

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



gluPerspective()

  - Easier to use than glFrustum()

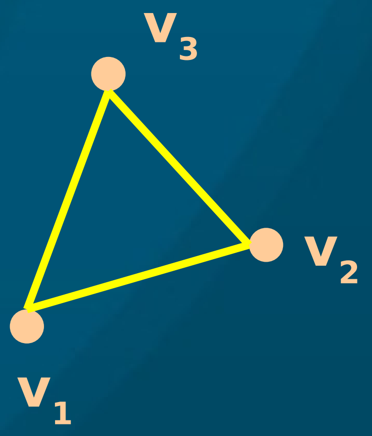
What's on for today

- Math for 3D graphics: homogeneous coordinates
  - 4x4 transform matrices
  - Translate, scale, rotate
- Viewing: (see RedBook ch3)
  - Positioning the camera: model-view matrix
  - Selecting a lens: projection matrix
  - Clipping: setting the view volume
- Modelling: vertex lists, face lists, edge lists
  - OpenGL vertex arrays and display lists

Modelling polygons

- Simple representation (see CubeView):

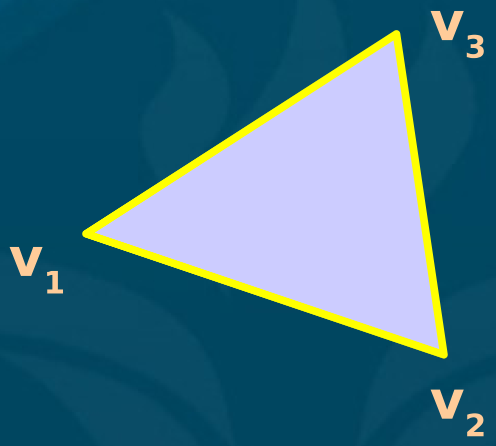
```
glBegin( GL_POLYGON );
    glVertex3f( 0.0, 0.0, 0.0 );
    glVertex3f( 1.0, 1.5, 2.2 );
    glVertex3f( -2.3, 1.5, 0.0 );
glEnd();
```



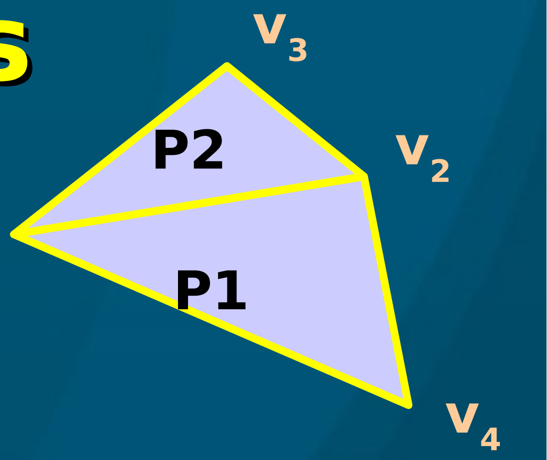
- Problems: inefficient, unstructured
  - What if we want to move a vertex to a new location?

Inward/outward facing polygons

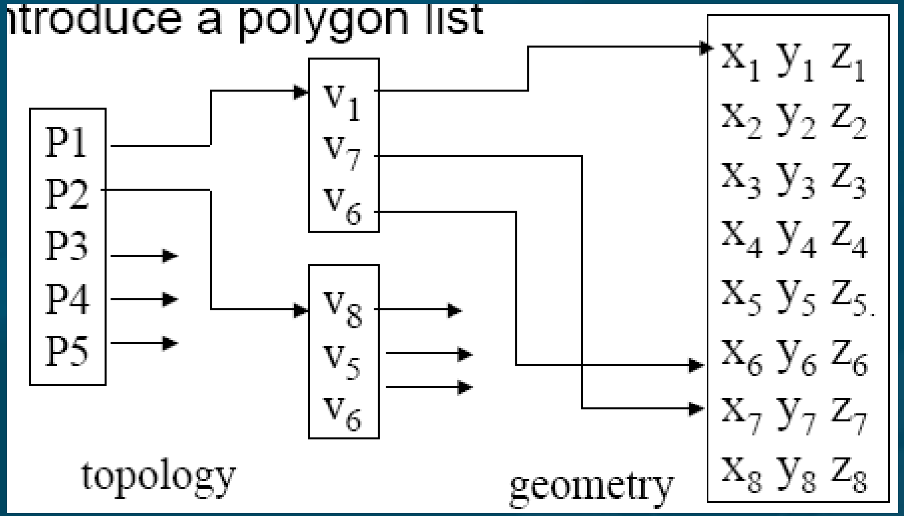
- The normal vector for a polygon follows the right-hand rule
- Specifying vertices in order (v_1, v_2, v_3) is same as (v_2, v_3, v_1) but different from (v_1, v_3, v_2)
- When constructing a closed surface, make sure all your polygons face outward
- Backface culling may mean inward-facing polygons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



Vertex lists and face lists

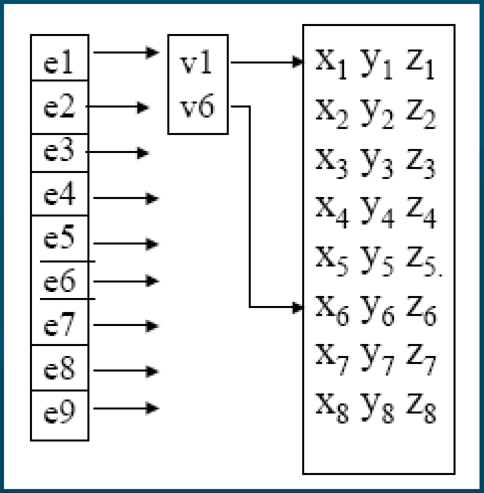


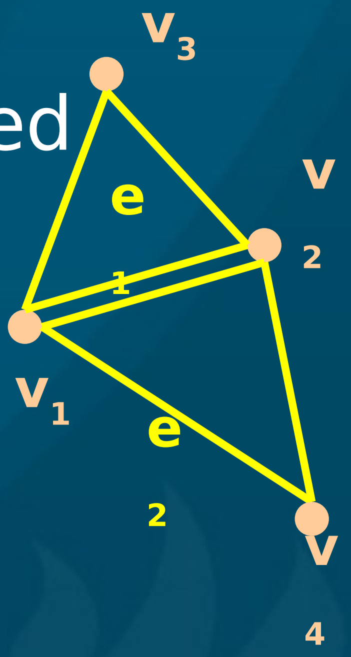
- Separate geometry from topology
  - Vertex coords are geometry
  - Connections between vertices (edges, polygons) are topology
- Vertex list:
  - $v_1 = \{x_1, y_1, z_1\}$
  - $v_2 = \{x_2, y_2, z_2\}$
- Polygon/face list:
  - $P_1 = \{v_1, v_2, v_3\}$
  - $P_2 = \{v_1, v_4, v_2\}$



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, 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