6. 3D Vectors, Geometry and Transformations

"It's magic, or geometry, or one of those things"
Terry Pratchett

6.1 From 2D to 3D

6.2 Vectors and Matrices in 3D

6.3 The Cross Product (Vector Product)

6.4 Straight Lines, Line Segments and Rays

6.5 The Geometry of Planes

6.6 More Common Graphics Problems

6.7 3D Transformations

6.8 Transformations in OpenGL

6.9 A Virtual Trackball

6.1 From 2D to 3D

- 3D points and vectors are 3-tuples
  - Coordinate space is defined by three orthogonal unit vectors.
- Convention: use upper-case letters for points, e.g. A, Q, bold lower case letters for vectors, e.g. **a**, **q**, and bold upper-case letters for matrices, e.g. **M**, **R**.
- Addition, scaling, subtraction, magnitude and normalisation all as for 2D, but with an extra coordinate.
- A convex combination of points defines a convex *polyhedron* rather than a polygon.
- *Products* of vectors are very important in 3D
  - *Dot Product (Scalar Product), Cross Product (Vector Product)*

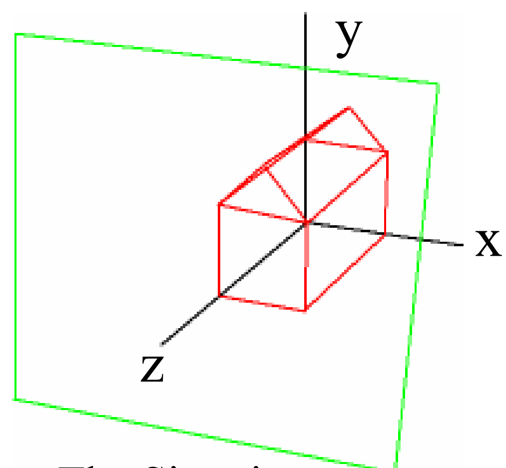
House3D

A simple OpenGL program displaying a 3D object

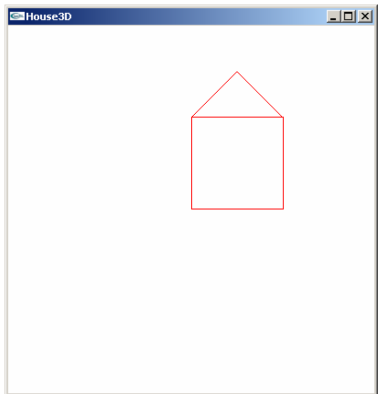
- What the program does
- The code
- Aspects of the code
  - Only consider aspects different from the 2D example:
    - Representing the 3D wireframe house
    - 3D Orthographic Projection
    - Resizing the display window
    - Drawing the picture
    - Exercises
    - Changing the View (GluLookAt)

What the program does

- Defines a simple house shape in *wireframe* form (i.e. made up of just straight lines representing the edges) in 3-space.
- Displays a picture of the house using a 3D *orthographic projection* along the z axis
  - Any point (x,y,z) projects to a point (x,y)
  - Much more on this later
- Note that the y-axis is UP



The Situation



The Program Output

```
// A basic OpenGL program displaying a 3D house shape
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>

const int windowWidth=400; const int windowHeight=400;

// define vertices and edges of the house
const int numVertices=10; const int numEdges=17;
const float vertices[numVertices][3] =  {{0,0,0},{1,0,0},{0,1,0},{1,1,0},{0,0,2},
                                          {1,0,2},{0,1,2},{1,1,2},{0.5f,1.5f,0},{0.5f,1.5f,2}};
const int edges[numEdges][2] = {{0,1},{1,3},{3,2},{2,0},{4,5},{5,7},{7,6},{6,4},{0,4},
                                {1,5},{3,7},{2,6},{2,8},{8,3},{6,9},{9,7},{8,9}};
void display(void){
   glMatrixMode(GL_MODELVIEW );              // Set the view matrix ...
   glLoadIdentity();                         // ... to identity.
   glClear(GL_COLOR_BUFFER_BIT);             // clear all pixels in frame buffer
   glColor3f (1.0, 0.0, 0.0);                // draw subsequent objects in red
   glBegin(GL_LINES);
   for(int i=0;i<numEdges;i++){
      glVertex3fv(vertices[edges[i][0]]);
      glVertex3fv(vertices[edges[i][1]]);
   }
   glEnd();
   glFlush ();
}
```

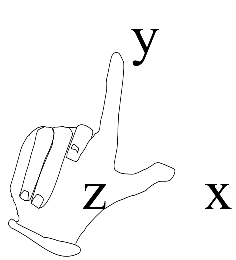
```
void init(void) {
   // select clearing color (for glClear)
   glClearColor (1.0, 1.0, 1.0, 0.0);          // RGB-value for white
   // initialize view (simple 3D orthographic projection)
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-2,2,-2,2,-3,3);
}

void reshape(int width, int height ) {       // Called at start, and whenever user resizes component
   int size = min(width, height);
   glViewport(0, 0, size, size);             // Largest possible square
}

// create a single buffered colour window
int main(int argc, char** argv){
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(windowWidth, windowHeight);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("House3D");
   init ();                                  // initialise view
   glutDisplayFunc(display);                 // Set function to draw scene
   glutReshapeFunc(reshape);                 // Set function called if window is resized
   glutMainLoop();
   return 0;
}
```

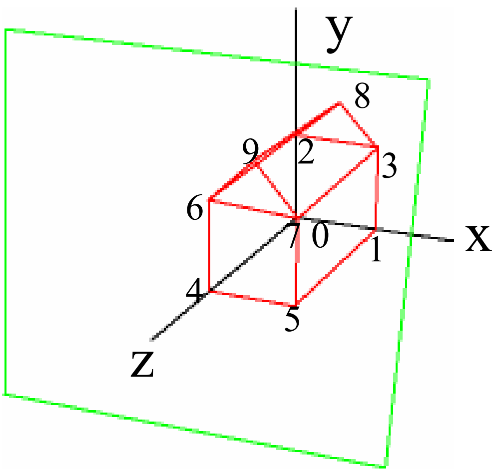
Representing the Wireframe House

- Have a vertex table and an edge table

```
const float vertices[numVertices][3] =  {{0,0,0},{1,0,0},{0,1,0},{1,1,0},{0,0,2},
                                          {1,0,2},{0,1,2},{1,1,2},{0.5f,1.5f,0},{0.5f,1.5f,2}};
const int edges[numEdges][2] = {{0,1},{1,3},{3,2},{2,0},{4,5},{5,7},{7,6},{6,4},{0,4},
                                 {1,5},{3,7},{2,6},{2,8},{8,3},{6,9},{9,7},{8,9}};
```

- Each vertex table array entry is itself an array of 3 floats, representing a point in R^3 [3D-space]
- Edge table values are *indices* into vertex table
  - e.g. edge {3,7} is the edge from V_3 (1,1,0) to V_7 (1,1,2)
- Coordinate system is *right handed*





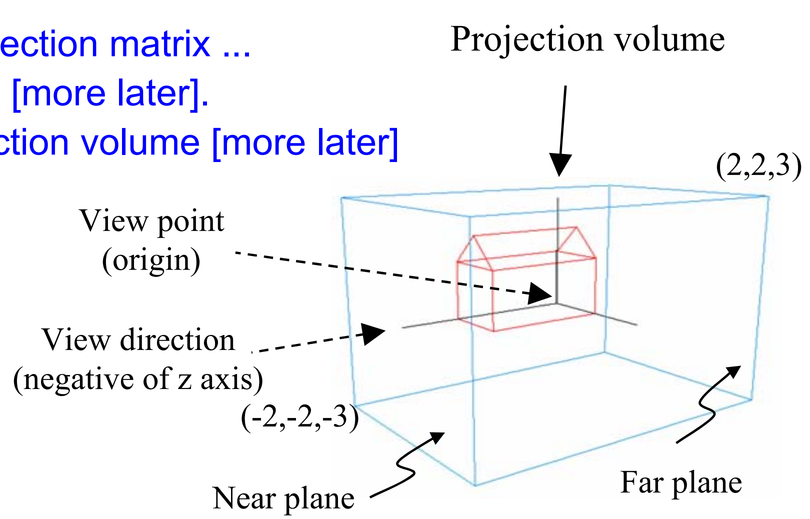
3D Orthographic Projection

- There are typically at least four phases to drawing ("*rendering*") a scene in *OpenGL*:
  (1) Define required projection
  (2) Define required view (allows you to to rotate, scale, translate, etc. the model)
  (3) Set up scene lighting
  (4) Output scene primitives (i.e. describe/define the scene)
- In this example we have a simple orthographic projection [i.e. $(x,y,z) \rightarrow (x,y)$], a trivial view and no lighting.

(1) Define required projection:

```
glMatrixMode(GL_PROJECTION);  // Initialise projection matrix ...
glLoadIdentity();             // ... to the identity matrix [more later].
glOrtho(-2,2,-2,2,-3,3);      // Set orthographic projection volume [more later]
```

glOrtho(left, right, bottom, top, near, far) defines the coordinates of the *projection volume*, with *near* and *far* being measured from the *view point* in the *view direction*, i.e. they are *depths* not z-values.

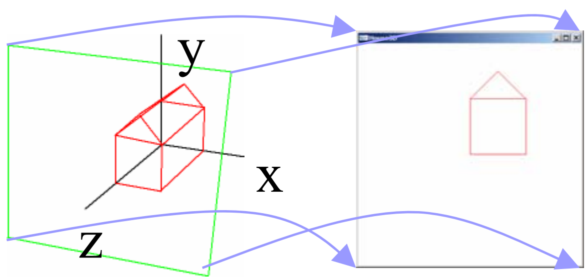


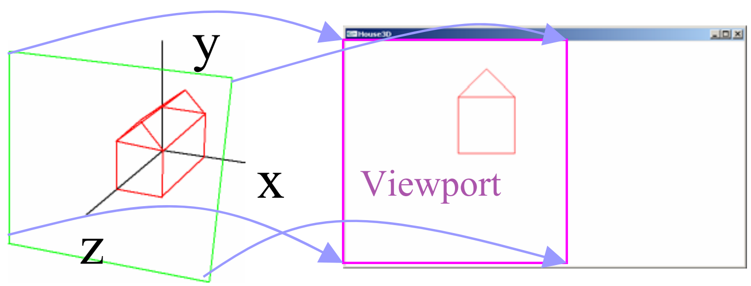
Resizing the Display Window

- The argument of *glutReshapeFunc* (the function *reshape*) is called at the start and whenever the display window gets resized
  - Specifies how the scene will be redrawn in the resized window
  - In the previous examples the viewport was the entire OpenGL window
  - In this example, we set the *viewport* to be the largest square possible.

```
int size = min(width, height);
glViewport(0, 0, size, size);
```

  - The projection matrix maps the scene onto the viewport
  - The rest (if any) of the window is unused
  - *glViewport* parameters are *x*, *y*, *width* and *height* in pixel coordinates, with (*x*, *y*) being the bottom left corner of the viewport.
    - In *OpenGL*, *y* coordinates always increase upwards, so (0,0) is the bottom left corner of the window, not the top left as is normal in screen coordinates.





Drawing the Picture

(2) Define required view of the model. The two lines

```
glMatrixMode(GL_MODELVIEW );
glLoadIdentity();
```

initialise the “model + view matrix” to the identity matrix, meaning “don’t transform the scene at all”. Don’t expect to understand this just yet!!

- View direction is along negative z axis.

(3) Set up scene lighting

- can not illuminate wireframes (since there is no surface normal)!

(4) Output scene primitives (as in the 2D example)

```
glClear(GL_COLOR_BUFFER_BIT);     // clear all pixels in frame buffer
glColor3f (1.0, 0.0, 0.0);        // draw scene in red
glBegin(GL_LINES);                // draw edges as line segments (3D vertices)
for(int i=0;i<numEdges;i++){
    glVertex3fv(vertices[edges[i][0]]);
    glVertex3fv(vertices[edges[i][1]]);
}
glEnd();
glFlush ();
```

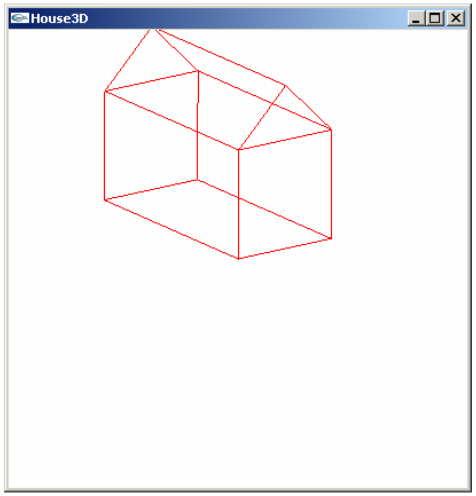
Exercises

- Change the program to use *glVertex3f* everywhere instead of *glVertex3fv*.
- How could you *centre* the picture in the output window? [Find a solution that involves only adjusting two of the numbers in the program]
- How could you increase the size of the picture in the output window?
- What is the effect of putting (a) the *near* plane, and (b) the *far* plane at z = 1?
- What happens if the *near* and *far* faces of the projection volume are rectangular rather than square?

Changing the View

- We can rotate the house to give a more useful view by changing step (2) to

```
glMatrixMode(GL_MODELVIEW );        // Set the view matrix ..
glLoadIdentity();                   // ... to identity.
glRotatef(-40,1,2,-0.3f);
                    // Rotate -40 degrees around an axis through the
                    // origin in the direction (1,2,-0.3).
```



- Looks vaguely OK, but how can we determine a suitable axis and angle, except by lots of experimentation?
- Answers: Either

  (a) We can’t. Yet. Need some maths! Or

  (b) Use a GLU function to do it for us.

GLULookAt

- Remember: GLU is a package of utility functions on top of GL.
- Replace start of *display()* with:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
gluLookAt(-1,2,5,  0.5f,0.75f,1,  0,1,0);
```

- Replace start of *init()* with:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-2,2,-2,2,0,7);
```

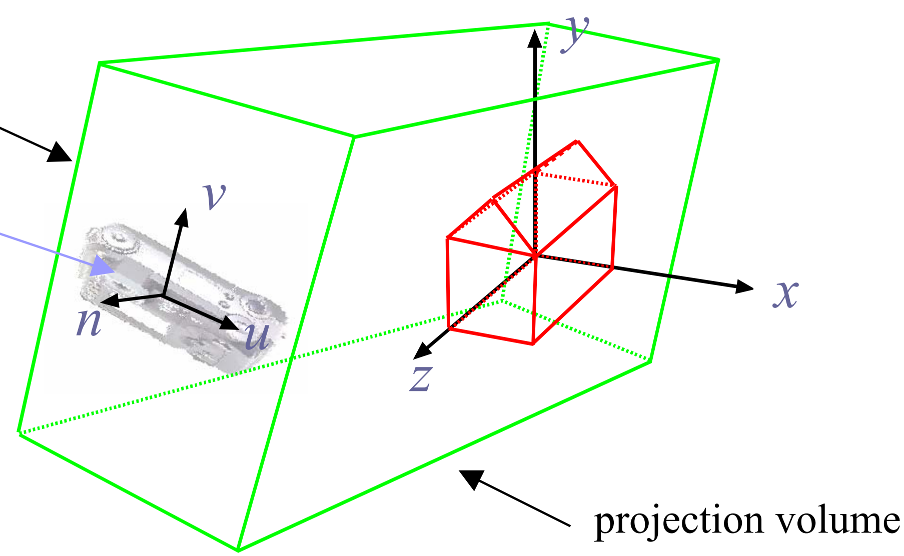
NB: projection volume coords are w.r.t. camera

The command `gluLookAt( cameraX, cameraY, cameraZ, lookatX, lookatY, lookatZ, UpX, UpY, UpZ)`

specifies where the virtual camera is, where it's pointing to, and how we're orienting (rotating) it.

GLULookAt (cont'd)

- What's happening?
  - Camera coordinate system (*u*, *v*, *n* axes) has origin at camera point, namely (-1, 2, 5).
    - vector from *camera* to *lookAt* defines the NEGATIVE *n* axis
    - House image is projected onto *viewplane*, which is defined to be perpendicular to *n* axis
    - Projection of *up* onto viewplane is *v* axis.
  - Projection volume is defined by *glOrtho*, in camera coordinates

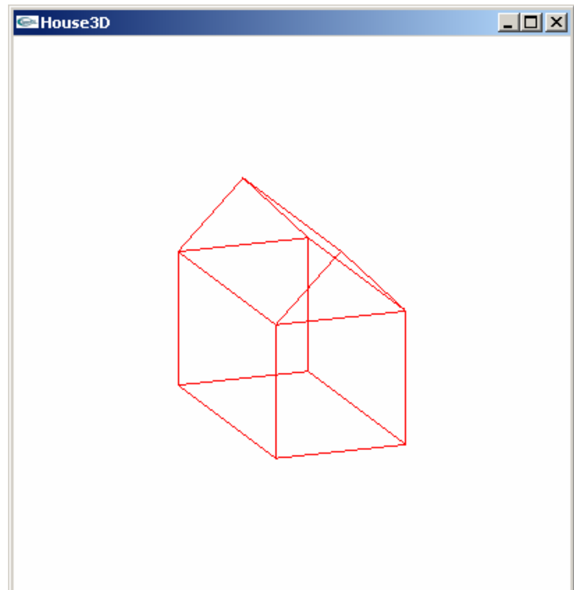


$$\text{Let } \mathbf{w} = \begin{pmatrix} UpX \\ UpY \\ UpZ \end{pmatrix}, \ \mathbf{e} = \begin{pmatrix} cameraX - lookAtX \\ cameraY - lookAtY \\ cameraZ - lookAtZ \end{pmatrix}$$

$$\text{then } \mathbf{n} = \frac{\mathbf{e}}{|\mathbf{e}|}, \ \mathbf{v} = \frac{\mathbf{w} - (\mathbf{w} \bullet \mathbf{n})\mathbf{n}}{|\mathbf{w} - (\mathbf{w} \bullet \mathbf{n})\mathbf{n}|}, \text{ and } \mathbf{u} = \mathbf{v} \times \mathbf{n}$$

GLULookAt (cont'd)

- Output of program is:



Each vertex has been projected in direction *n* onto the viewplane.

- But to understand *exactly* what's happening here we need to understand 3D vectors, transformations and other miscellaneous geometry.

6.2 Vectors and Matrices in 3D

- Vector and matrix operations as in 2D
- Determinant *det* **M** (also written | **M** |) of a 3x3 matrix **M**

$$\begin{vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{vmatrix} = m_{11}\begin{vmatrix} m_{22} & m_{23} \\ m_{32} & m_{33} \end{vmatrix} - m_{12}\begin{vmatrix} m_{21} & m_{23} \\ m_{31} & m_{33} \end{vmatrix} + m_{13}\begin{vmatrix} m_{21} & m_{22} \\ m_{31} & m_{32} \end{vmatrix}$$

- Inverse of a matrix:
  - The matrix $\mathbf{M}^{-1}$ is called the inverse of **M** if $\mathbf{M}^{-1}\mathbf{M}=\mathbf{M}\mathbf{M}^{-1}=\mathbf{I}$

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \Rightarrow \mathbf{M}^{-1} = \frac{1}{|\mathbf{M}|}\mathbf{A} \text{ where } \mathrm{a_{ij}} = (-1)^{i+j}\left|\mathbf{A}^{ji}\right| \quad , 1 \le i, j \le 3$$

and $\mathbf{A}^{kl}$ is the 2×2 matrix formed by deleting the kth row and lth column of $\mathbf{M}$.

The Dot Product

- Definition
  - The *dot product* (or *inner product,* or *scalar product*) of two 3D vectors $\mathbf{v} = (v_1,v_2,v_3)$ and $\mathbf{w} = (w_1,w_2,w_3)$ is: $\mathbf{v} \bullet \mathbf{w} = v_1w_1 + v_2w_2 + v_3w_3$.

- Properties (same as in 2D)
  - Symmetry: $\mathbf{a} \bullet \mathbf{b} = \mathbf{b} \bullet \mathbf{a}$
  - Linearity: $(\mathbf{a}+\mathbf{b}) \bullet \mathbf{c} = \mathbf{a} \bullet \mathbf{c} + \mathbf{b} \bullet \mathbf{c}$
  - Homogeneity: $(s\mathbf{a}) \bullet \mathbf{b} = s(\mathbf{a} \bullet \mathbf{b})$
  - $|\mathbf{b}|^2 = \mathbf{b} \bullet \mathbf{b}$
  - Example: Prove $|\mathbf{a} - \mathbf{b}|^2 = \mathbf{a} \bullet \mathbf{a} - 2\mathbf{a} \bullet \mathbf{b} + \mathbf{b} \bullet \mathbf{b}$

- Applications (same as in 2D, except where 2D "Perp" Vector is used)
  - Angle between two vectors
  - The sign of $\mathbf{a} \bullet \mathbf{b}$ and perpendicularity
  - Projecting vectors
  - Ray reflection

Coordinate Transformations

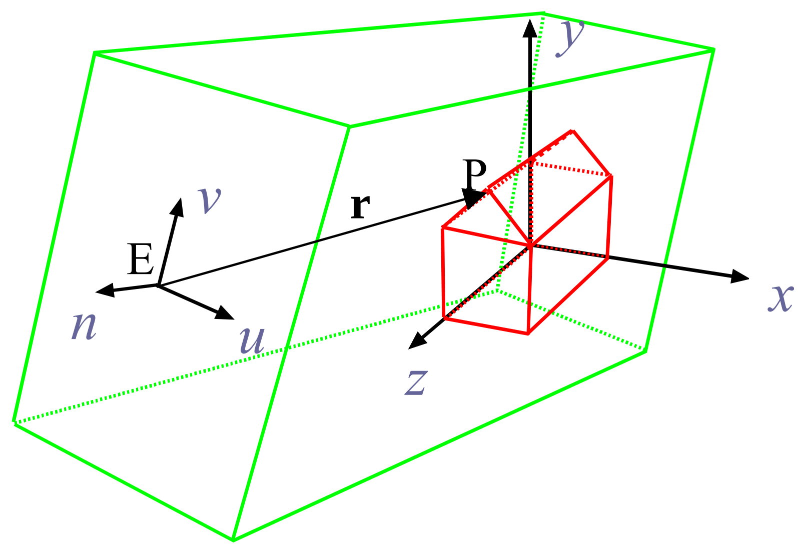
The component of a vector **v** in a direction represented by a unit vector **i** is the perpendicular projection of **v** onto the direction of **i**.

Given by (**v** • **i**) **i** (formula from Chapter 5, slide 12)

Let P be a point and E be the camera location given in x,y,z-coordinates

The components of the point P expressed in the (u,v,n) coordinate system are: (**r** • **u**), (**r** • **v**) and (**r** • **n**)

where **r** = P – E



6.3 The Cross Product

- Let **i**, **j**, and **k** be unit vectors along the *x*, *y* and *z* axes respectively.
- Define the *cross product* (or *vector product*) operator such that:
  - It's a linear operator, i.e.

    $\mathbf{a} \times (\mathbf{b}+\mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$
  - $|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin \theta$ where θ is the angle between **a** and **b** in range $[0,2\pi]$
  - It's homogeneous, i.e. $(s\mathbf{a}) \times \mathbf{b} = s\,(\mathbf{a} \times \mathbf{b})$
  - $\mathbf{i} \times \mathbf{j} = \mathbf{k}$, $\mathbf{j} \times \mathbf{k} = \mathbf{i}$, $\mathbf{k} \times \mathbf{i} = \mathbf{j}$
  - $\mathbf{j} \times \mathbf{i} = -\mathbf{k}$, $\mathbf{k} \times \mathbf{j} = -\mathbf{i}$, $\mathbf{i} \times \mathbf{k} = -\mathbf{j}$
  - $\mathbf{i} \times \mathbf{i} = \mathbf{j} \times \mathbf{j} = \mathbf{k} \times \mathbf{k} = \mathbf{0}$
- Can show from this (UDOO) that if

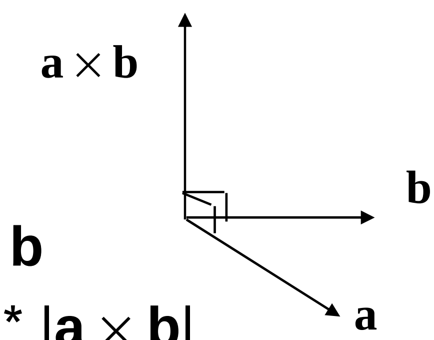
  $\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$ and $\mathbf{b} = b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k}$

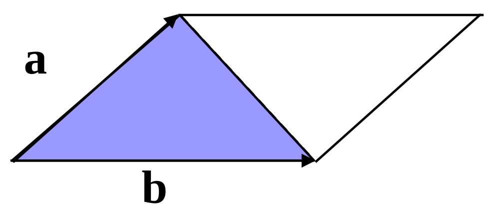
$$\text{then } \mathbf{a}\times\mathbf{b} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

Properties of Cross Product

Key properties (UDOO proofs):

- $\mathbf{a} \times \mathbf{b}$ is a vector perpendicular to both **a** and **b**
- Direction of $\mathbf{a} \times \mathbf{b}$ is given by "right-hand rule"
- $\mathbf{a} \times \mathbf{b} = -\,\mathbf{b} \times \mathbf{a}$
- $|\mathbf{a} \times \mathbf{b}|$ is area of parallelogram defined by **a** and **b**
- Hence area of triangle defined by **a** and **b** is 0.5 * $|\mathbf{a} \times \mathbf{b}|$





6.4 Straight Lines, Line Segments and Rays

- Use a *parametric* form for lines.
  - Straight line through two points P_1 and P_2 is

$$\begin{aligned} P(\alpha) &= (1-\alpha)P_1 + \alpha P_2 \\ &= P_1 + \alpha(P_2 - P_1) \\ &= P_1 + \alpha\mathbf{v} \end{aligned}$$

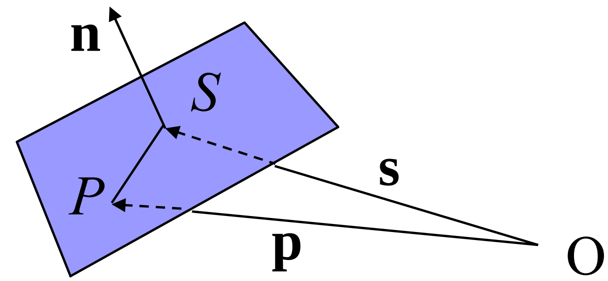
  - where $\mathbf{v} = P_2 - P_1$ is the displacement vector from P_1 to P_2.
- If α constrained to the range [0,1] we have a *line segment* – all points between P_1 and P_2.
- If α constrained to the range $[0,\infty]$ we have a *ray*.
- If α is any real number, we have a full line in *n*-space.

6.5 The Geometry of Planes

- The Point-Normal Form of a Plane Equation
- Distance of a Plane from the Origin
- Distance of a Point from a Plane
- Inside-Outside Half-Space Test
- Intersection Line-Plane

The Point-Normal Form of a Plane

- Can define a plane by giving one point on it, *S* say, and its unit normal **n**.



- Then for any point *P* on the plane, (*P*-*S*) is perpendicular to **n**.
  i.e. $\mathbf{n} \bullet (P - S) = 0$.
  ["Point-normal" form of plane equation]
- If **p** and **s** are the vectors to *P* and *S* then can write this as

$$\mathbf{n} \bullet (\mathbf{p} - \mathbf{s}) = 0$$

i.e. $\mathbf{n} \bullet \mathbf{p} = \mathbf{n} \bullet \mathbf{s}$

or $\mathbf{n} \bullet \mathbf{p} = d$ where $d = \mathbf{n} \bullet \mathbf{s}$

- If **n** is (a,b,c) and **p** is (x,y,z), then this is the familiar equation

$$ax + by + cz = \mathrm{d}.$$

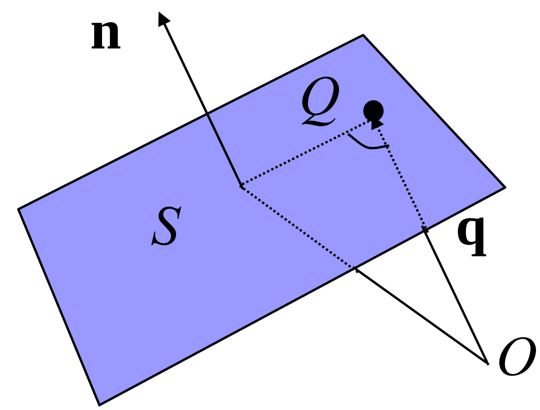
Distance of plane from origin

- Let Q be a point on the plane such that **q** is parallel to **n**. Then the length of **q** is the "shortest distance" to the plane from the origin.
- We have the plane equation $\mathbf{n} \bullet \mathbf{p} = d$

  for any point P on the plane. Hence $\mathbf{n} \bullet \mathbf{q} = d$.

  But since **n** is parallel to **q**, $\mathbf{n} \bullet \mathbf{q} = |\mathbf{q}|$.

  Thus $|\mathbf{q}| = d$.



- Hence, in the equations

  $\mathbf{n} \bullet \mathbf{p} = d$ and

  $ax + by + cz = d$

  d is the distance to the plane from the origin provided $\mathbf{n} = (a,b,c)$ is a unit vector.
- UDOO: How far is the plane $3x + y - 2z = 5$ from the origin?

Distance of Point from Plane

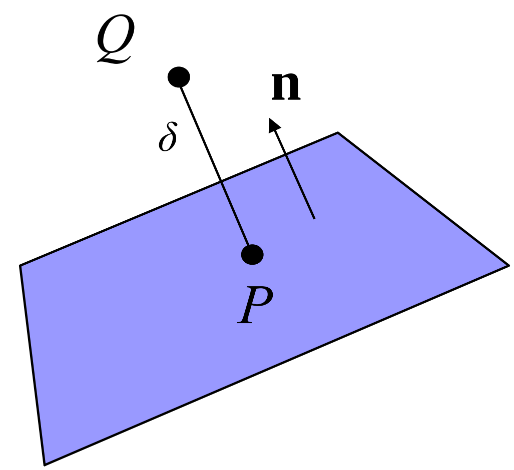
- How far is a point Q from the plane $\mathbf{n} \bullet \mathbf{p} = d$?
- Let P be the nearest point on the plane to Q, so that (Q-P) is parallel to **n**.
- Then the required answer δ is

  $$\delta = |Q-P| = |(\mathbf{q} - \mathbf{p})|$$

- Since **q** - **p** is parallel to **n**, can write as

  $$\delta = (\mathbf{q} - \mathbf{p}) \bullet \mathbf{n}$$
  $$= \mathbf{q} \bullet \mathbf{n} - \mathbf{p} \bullet \mathbf{n} = \mathbf{q} \bullet \mathbf{n} - d$$
  $$= aq_1 + bq_2 + cq_3 - d$$

  where $\mathbf{n} = (a,b,c)$ is the **unit normal** and $\mathbf{q} = (q_1 , q_2, q_3)$

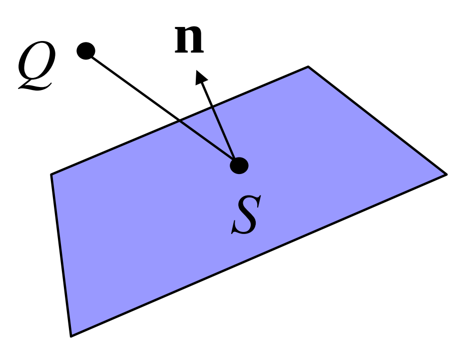


- δ is positive if Q is outside the plane, negative if Q is inside.
- WARNING: Always scale plane equation $ax+by+cz=d$ so that $(a^2+b^2+c^2)=1$.

Inside-Outside Half-Space Tests (cont'd)

- If plane defined in point-normal form, can by convention take **n** to be the outward normal and points "on that side" of the plane are said to be "outside", while points on the other side are "inside".
- Hence, if *S* is a point on the plane and *Q* is a point to be tested:
  - $(Q-S) \bullet \mathbf{n} > 0 \rightarrow Q$ is outside
  - $(Q-S) \bullet \mathbf{n} = 0 \rightarrow Q$ is on the plane
  - $(Q-S) \bullet \mathbf{n} < 0 \rightarrow Q$ is inside

  (see chapter 5, slide 11)



Inside-Outside Half-Space Tests

- NOTE:
  - Above plane equations assume 3D vectors.
  - If all vectors are 2D, the plane becomes a line, and the equations give the distance of a line from the origin, the distance of a point from a line, and categorise a 2D point as inside or outside a line.

Intersection Line-Plane

Given is a line

$\boldsymbol{p}(t) = \boldsymbol{p}_0 + t\boldsymbol{c}$ where **c** is the line's direction

and a plane $\mathbf{n} \bullet \mathbf{p} = d$

The line intersects the plane when $t = \dfrac{d - \mathbf{n} \bullet \mathbf{p}_0}{\mathbf{n} \bullet \mathbf{c}}$

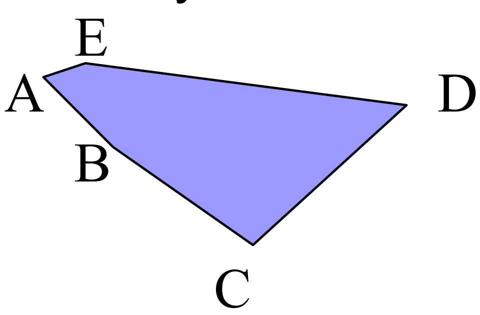
Q: What happens if $\mathbf{n} \bullet \mathbf{c} = 0$?

6.6. More Common Graphics Problems

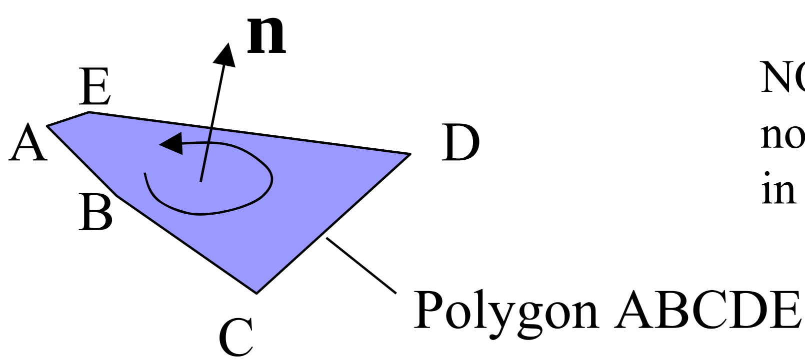
- The Area of a Triangle
  - use magnitude of a cross product
  - see "Properties of Cross Product"
- The Robust Normal to a Polygon

The Normal to a Polygon

- In principle, get normal **n** from the cross product of any two adjacent edge vectors, e.g. **n** = (D-C)×(B-C)
- But this is *non-robust* — gives erroneous or unrepresentative value when:
  - 3 vertices co-linear
  - 2 adjacent vertices very close together
    - Magnitude of cross product tends to zero and direction is sensitive to slight movement in either point
  - Polygon not coplanar
    - e.g. (B-A)×(E-A), above, not representative
- Warning: In computer graphics, exceptional conditions occur all the time!



A Robust Normal Algorithm



NOTE: The orientation of the resulting normal is such that the vertices are listed in counterclockwise order around it.

- Just add together all the cross products of adjacent edge vectors
  i.e. $(B–A) \times (E–A) + (C–B) \times (A–B) +$
  $(D–C) \times (B–C) + (E–D) \times (C–D) + (A-E) \times (D–E)$
- Normalise the result.
- Robust.
  - Short edges or nearly co-linear vertex triples give negligible cross product contribution
  - Long nearly-perpendicular edges give biggest contribution

6.7 3D Transformations

- Natural extension of 2D.
- We will use the homogeneous coordinate form for all transformations.
- To convert between ordinary 3D coordinates and homogenous 3D coordinates:
  - 3D ordinary $\rightarrow$ 3D homogeneous

    $(x, y, z)^T \rightarrow (x, y, z, 1)^T$
  - 3D homogeneous $\rightarrow$ 3D ordinary

    $(x, y, z, w)^T \rightarrow (x/w, y/w, z/w)^T$

Translation

The matrix for a translation by a vector **t** = (t_x, t_y, t_z) is:

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scaling

- The matrix for a scaling by factors of S_x, S_y, S_z in x, y and z respectively is:

$$\mathbf{S} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- A negative S_x gives a reflection about the x = 0 plane.
- Similarly for negative S_y or S_z.

Shearing

- The general shear matrix is:

$$\mathbf{H} = \begin{pmatrix} 1 & h_{yx} & h_{zx} & 0 \\ h_{xy} & 1 & h_{zy} & 0 \\ h_{xz} & h_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- 3D shearing in its most general form is very rare. Occasionally meet horizontal shearing e.g. a pile of paper pushed to one side so that the sides of the pile are still straight but not vertical.
- UDOO: What would the matrix for that look like?

Rotation

- Rotations are by far the most confusing of the transformations
- Most texts cover rotations around the three coordinate axes and try to build all other rotations from those
  - But some cases *very* difficult
- We will consider three different rotation situations:
  - Rotation around the three coordinate axes
  - Rotation to align an object with a new coordinate system
  - Rotation around an arbitrary axis

Rotation around Coordinate Axes

- Have three axes to rotate about, so three different matrices.
- Let C = cos θ and S = sin θ. Then the three matrices for positive (right handed) rotation are:
- Rotation about the x-axis:

$$\mathbf{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & C & -S & 0 \\ 0 & S & C & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Rotation about the y axis:

$$\mathbf{R}_y = \begin{pmatrix} C & 0 & S & 0 \\ 0 & 1 & 0 & 0 \\ -S & 0 & C & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{R}_z$: UDOO.

Note on 3 × 3 rotation matrices:

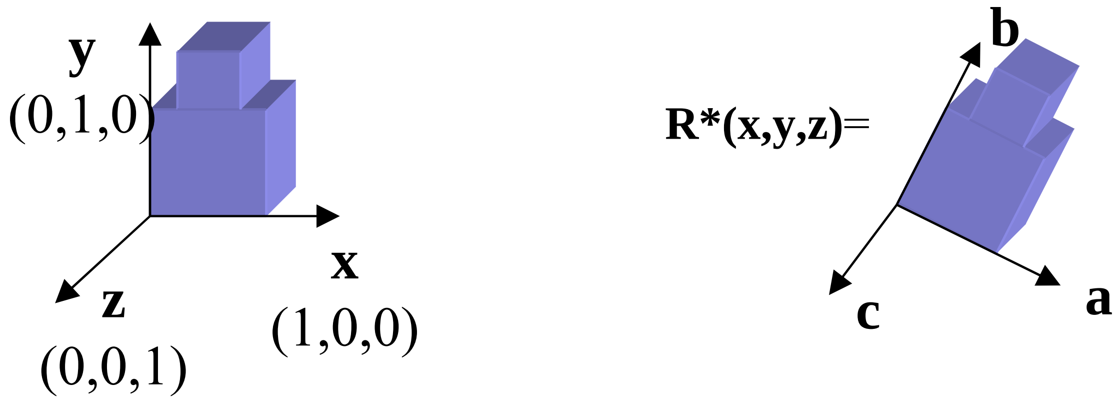
Row and column corresponding to axis of rotation are as for identity **I**

Other elements are C on diagonal, ± S off diagonal, so that **R**→ **I** if θ → 0.

Sign of S can be inferred from the fact that rotation around x,y,z by θ=90° transforms y→z , z → x, x → y, respectively.

Rotating to Align with New Coordinate Axes

- Often we have some object and want it at a new position and with a new orientation
  - Generally involves both rotation and translation
  - Translation trivial -- focus only on rotation here
- Problem: what is the rotation matrix **R** that rotates a coordinate system (**x**,**y**,**z**) to align with a new coordinate system (**a**,**b**,**c**) with the same origin, where **a**,**b**,**c** are unit vectors along the new axes.



Rotating to Align with New Coordinate Axes (cont'd)

- To get **R**: we have
  - **R** $(1\ \ 0\ \ 0)^T$ = **a**
  - **R** $(0\ \ 1\ \ 0)^T$ = **b**
  - **R** $(0\ \ 0\ \ 1)^T$ = **c**

$$\mathbf{R}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{pmatrix}$$

- Above 3 eqns equivalent to:

$$\therefore \mathbf{R} = \begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{pmatrix} \quad \text{or} \quad \mathbf{R}_{H.C.} = \begin{pmatrix} a_x & b_x & c_x & 0 \\ a_y & b_y & c_y & 0 \\ a_z & b_z & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- SO – IMPORTANT GENERAL RESULT: Columns of a 3 x 3 rotation matrix are unit vectors along the rotated coordinate axis directions
  - UDOO – derive $\mathbf{R}_x$, $\mathbf{R}_y$, $\mathbf{R}_z$ from this rule.

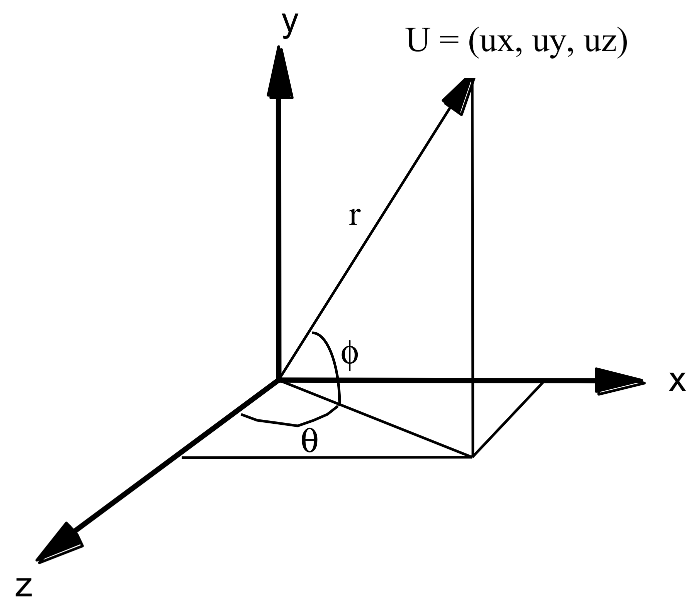
Rotation about an arbitrary axis

- Often, when building a 3D scene or object, need to rotate a component about some arbitrary axis through a reference point on it. [e.g. forearm of robot rotating around an axis through the elbow].
- Involves three steps:
  - (1) Translate reference point to origin
  - (2) Do the rotation
  - (3) Translate reference point back again
- Three approaches for step (2) [next 3 slides]:
  - Textbook method
    - Decompose rotation into primitive rotations about x,y,z axes
      - Nice exercise, but hard to get right in practice
  - Coordinate system alignment method
  - Generalised rotation matrix

- An aside: *Quaternions* provide an elegant way of manipulating (axis, angle) rotations directly.

Textbook Method

- Rotate the object so that the required axis of rotation **r** lies along the z axis [$\mathbf{R}_{alignZ}$]
- Do the rotation about z axis
- Undo original rotation [$\mathbf{R}_{alignZ}^{-1}$]
- How to get $\mathbf{R}_{alignZ}$?
  - Measure azimuth, θ , as a right handed rotation about the y axis, starting at the z axis.
  - Measure elevation, ø (or "latitude") as angle above plane y=0.
- $\mathbf{R}_{alignZ} = \mathbf{R}_x(\phi)\ \mathbf{R}_y(-\theta)$

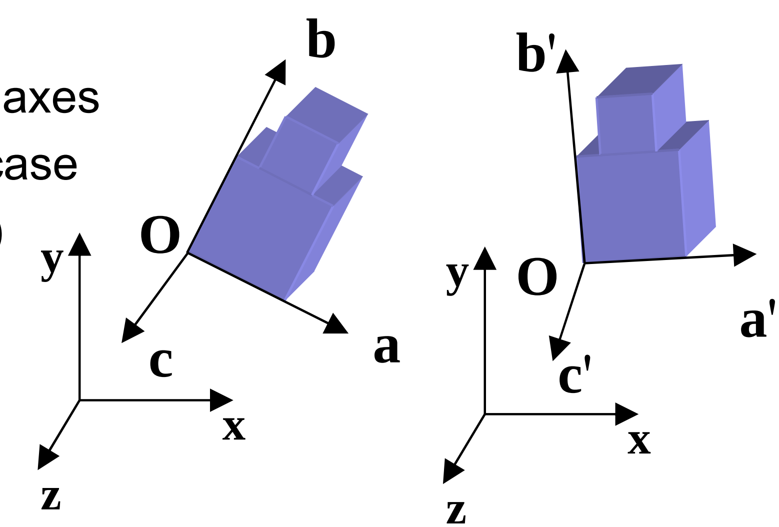


$$\phi = \tan^{-1} \frac{u_y}{\sqrt{u_x^2 + u_z^2}}$$

$$\theta = atan2(u_x, u_z) \text{ \{i.e. a 4-quadrant } \tan^{-1}\frac{u_x}{u_z}\}$$

Coordinate System Alignment Method

- Assume we have a coordinate system (**a**,**b**,**c**) attached to the object and want to rotate it to a new known orientation (**a**',**b**',**c**')
  - Slightly different problem from previous one
- Extension of "Rotate to align" problem
- Solution:
  - Translate object to origin
  - Rotate (**a**,**b**,**c**) to align with world coord axes
    - The *inverse* of the "rotate to align" case
  - Rotate coord axes to align with (**a**',**b**',**c**')
    - Same as "rotate to align" case
  - Translate back again
- Full matrix is: $\mathbf{T_O R_{a'b'c'} R^{-1}_{abc} T_{-O}}$



The Inverse of a Rotation Matrix
[needed on previous slide]

- Remember: columns of a rotation matrix are unit vectors along the rotated coordinate axis directions
  - So columns are orthogonal, i.e dot products = 0
- So:

$$\begin{pmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{pmatrix}\begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- i.e. $\mathbf{R}^T\mathbf{R} = \mathbf{I}$ and hence $\mathbf{R}^{-1} = \mathbf{R}^T$

So the inverse of a rotation matrix is its transpose

(Note: a matrix with this property is called *orthogonal*.)

Generalised Rotation Matrix

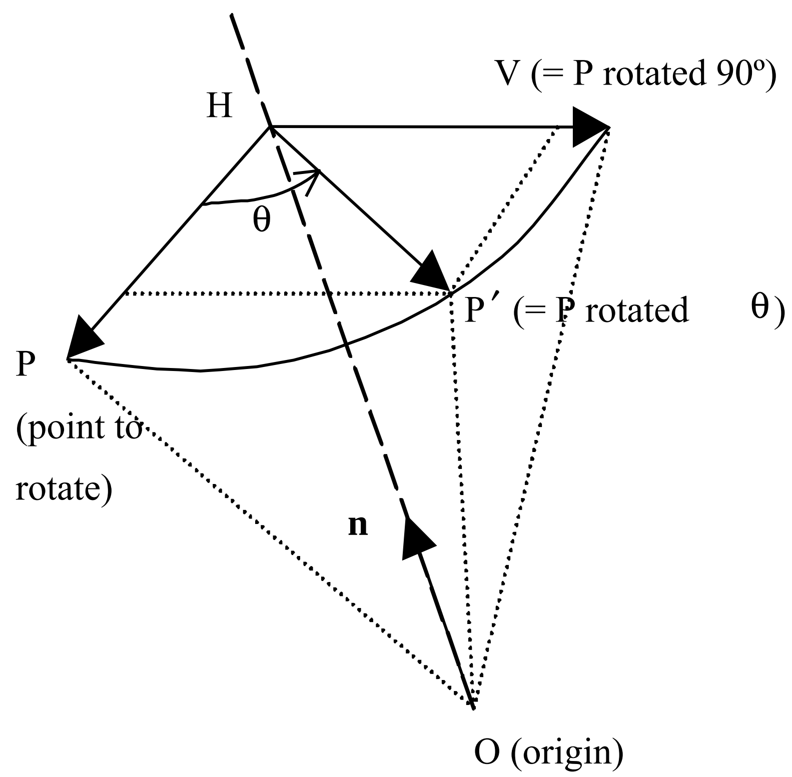
Not exam relevant!

- Matrix for an arbitrary rotation is:

$$R = \begin{pmatrix} tx^2 + c & txy - sz & txz + sy \\ txy + sz & ty^2 + c & tyz - sx \\ txz - sy & tyz + sx & tz^2 + c \end{pmatrix}$$

where the axis of rotation (normalised) is (x,y,z), *c* and *s* are resp. the cosine and sine of the angle of rotation, and *t*=(1-*c*).

- Proof outline [for enthusiasts only]
  - see Maillot, Graphics Gems I, P498 for details



6.8 Transformations in OpenGL

- *OpenGL* rendering has two 4 x 4 transformation matrices:
  - The *Projection* matrix, **P**
  - The *Model-View* matrix, **M**

- All vertices (i.e. points, polygon vertices, etc) are multiplied by **M** then **P** before the (x,y,z) → (x,y) projection is done

Transformations in OpenGL (cont'd)

- The **P** matrix handles perspective projections (see later) and scaling from world coordinates to screen coordinates.

- The **M** matrix handles both
  - ***modelling* operations**
    - i.e. the transformations that are part of the process of specifying a scene, e.g. positioning some generic chair to a certain point in the scene)
  - **the *viewing* transformation**
    - i.e. the rotation and translation required to allow us to view the scene from somewhere other than along the *z*-axis.

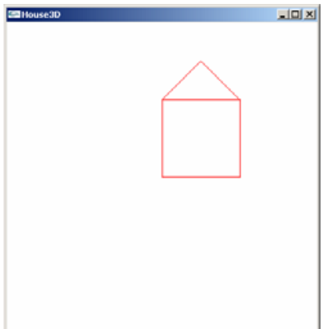
Transformations in OpenGL (cont'd)

- To set the value of one of the two matrices:
  - Select the one of interest, e.g.
    glMatrixMode(GL_MODELVIEW)
  - Set it to the identity, or load it with a specific matrix
    glLoadIdentity(), or
    glLoadMatrixf(const GLfloat **m*)
  - Multiply it *on the right* by one or more primitive matrices, e.g.
    glTranslatef(GLfloat dx, GLfloat dy, GLfloat dz)
    glScalef(GLfloat xFactor, GLfloat yFactor, GLfloat zFactor)
    glRotatef(GLfloat angleInDegrees, GLfloat axisX, GLfloat axisY, GLfloat axisZ)
    glMultMatrixf(const GLfloat **m*)   // general purpose matrix
    // Matrix is 16 floats, ***columnwise***, i.e. m_{00}, m_{10}, m_{20}, m_{30},m_{01}, m_{11}, ... m_{33}

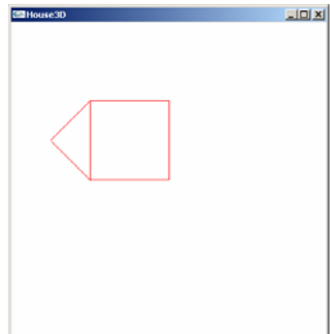
- Note: since matrices are multiplied on the right the *last* matrix multiplied in is the *first* to be applied to the vertices
  - Since   **(P Q R)v = P (Q (R v))**

Example MODEL_VIEW Transformations

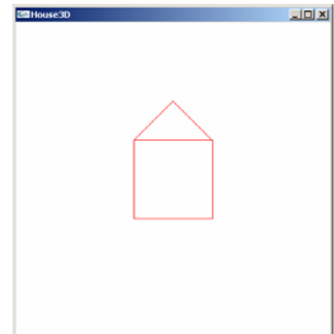
- In the *display* method of the `House3D` program …



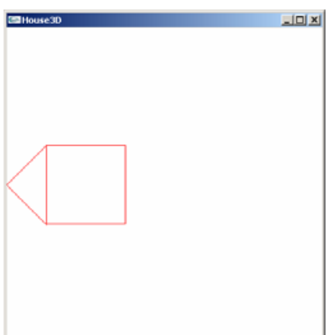
glLoadIdentity();



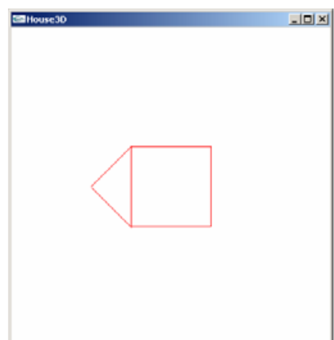
glLoadIdentity();
glRotatef(90,0,0,1);



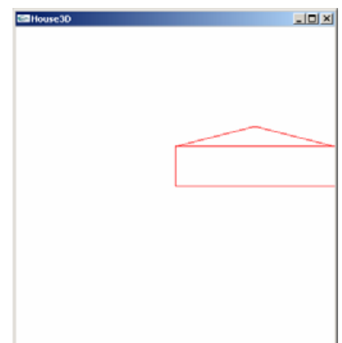
glLoadIdentity();
glTranslatef(-0.5f,-0.5f,-1.0f);



glLoadIdentity();
glTranslatef(-0.5f,-0.5f,-1.0f);
glRotatef(90,0,0,1);



glLoadIdentity();
glRotatef(90,0,0,1);
glTranslatef(-0.5f,-0.5f,-1.0f);



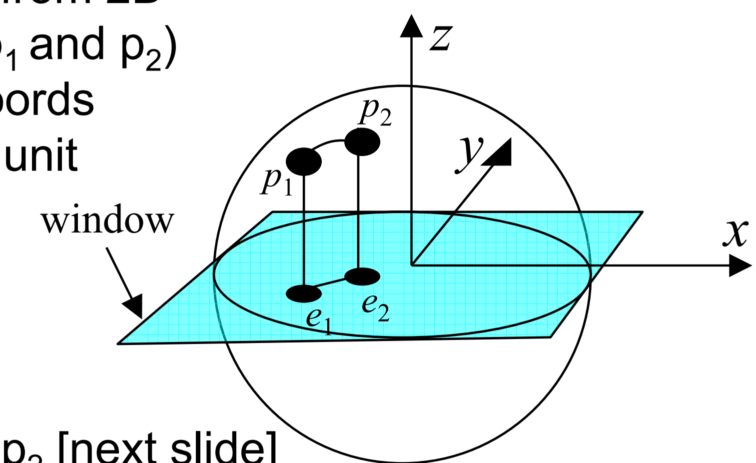
glLoadIdentity();
glScalef(2,0.5f,1);

6.9 A Virtual Trackball

(from Angel, section 4.10.2)

- A way of using the mouse to rotate the scene
- Imagine the scene is encased in a freely rotateable transparent sphere.
- Half of sphere is "sticking out" of screen window
- Clicking and dragging the mouse over the window is like rotating the sphere to a new position.
- To compute rotation:
  1. Map mouse drag end-points (e_1 and e_2) from 2D window coordinates to 3D coordinates (p_1 and p_2) on surface of virtual sphere. If window coords (x,y) are in range -1 to +1, and sphere is unit sphere, mapping is

$$(x, y) \to \left(x, y, \sqrt{1 - x^2 - y^2}\right)$$

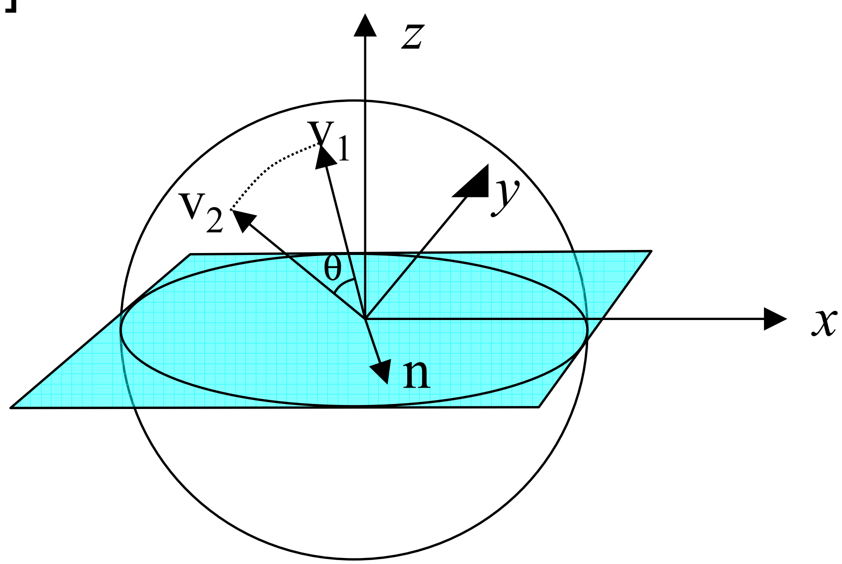


  2. Compute the rotation reqd to move p_1 to p_2 [next slide]

A Virtual Trackball (cont'd)

- Sphere rotation is about an axis $\mathbf{n} = \dfrac{\mathbf{v}_1 \times \mathbf{v}_2}{|\mathbf{v}_1 \times \mathbf{v}_2|}$
- where $\mathbf{v}_1$ and $\mathbf{v}_2$ are position vectors of p_1 and p_2 [NB: they have unit length. Why?]
- Rotation angle is

$$\theta = \cos^{-1} \mathbf{v}_1 \cdot \mathbf{v}_2$$



Trackball.h

Code not exam relevant!

```
class CTrackball
{
public:
    CTrackball();
    virtual ~CTrackball();
    void tbInit(GLuint button);
    void tbMatrix();
    void tbReshape(int width, int height);
    void tbMouse(int button, int state, int x, int y);
    void tbKeyboard(int key);
    void tbMotion(int x, int y);
private:
    GLuint   tb_lasttime;     GLfloat   tb_lastposition[3];
    GLfloat  tb_angle;        GLfloat   tb_axis[3];        // rotation axis and angle
    GLfloat  tb_transform[4][4];                   // current rotation matrix for GL_MODEL_VIEW
    GLuint   tb_width;        GLuint    tb_height;         // width and height of window
    GLint    tb_button;       GLboolean tb_tracking;
    void _tbPointToVector(int x, int y, int width, int height, float v[3]);
    void _tbStartMotion(int x, int y, int button, int time);
    void _tbStopMotion(int button, unsigned time);
};
```

Trackball.cpp

```
#include <math.h>
#include "Trackball.h"
#include <gl/glut.h>

CTrackball::CTrackball(){
    tb_angle = 0.0;
    tb_axis[0]=0.0;tb_axis[1]=0.0;tb_axis[2]=0.0;
    tb_tracking = GL_FALSE;
}

CTrackball::~CTrackball(){}

void CTrackball::_tbPointToVector(int x, int y, int width, int height, float v[3]){
    float d, a;

    // project x, y onto a hemi-sphere centered within width, height.
    v[0] = (float) ((2.0 * x - width) / width);
    v[1] = (float) ((height - 2.0 * y) / height);
    d = (float) (sqrt(v[0] * v[0] + v[1] * v[1]));
    v[2] = (float) (cos((3.14159265 / 2.0) * ((d < 1.0) ? d : 1.0)));
    a = (float) (1.0 / sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]));
    v[0] *= a; v[1] *= a; v[2] *= a;
}
```

Trackball.cpp (cont'd)

```
void CTrackball::_tbStartMotion(int x, int y, int button, int time)
{
    tb_tracking = GL_TRUE;
    tb_lasttime = time;
    _tbPointToVector(x, y, tb_width, tb_height, tb_lastposition);
}

void CTrackball::_tbStopMotion(int button, unsigned time)
{
    tb_tracking = GL_FALSE;
    tb_angle=0.0;
}

void CTrackball::tbInit(GLuint button)
{
    tb_button = button;
    tb_angle = 0.0;

    // put the identity in the trackball transform
    for(int i=0;i<4;i++){
       for(int j=0;j<4;j++) tb_transform[i][j]=0.0;
       tb_transform[i][i]=1.0;
    }
}
```

Trackball.cpp (cont'd)

```
void CTrackball::tbMatrix()
{
    glPushMatrix();
    glLoadIdentity();
    glRotatef(tb_angle, tb_axis[0], tb_axis[1], tb_axis[2]);
    glMultMatrixf((GLfloat *)tb_transform);
    glGetFloatv(GL_MODELVIEW_MATRIX, (GLfloat *)tb_transform);
    glPopMatrix();
    glMultMatrixf((GLfloat *)tb_transform);
}

void CTrackball::tbReshape(int width, int height)
{
    tb_width  = width;
    tb_height = height;
}

void CTrackball::tbMouse(int button, int state, int x, int y)
{
    if (state == GLUT_DOWN && button == tb_button)
        _tbStartMotion(x, y, button, glutGet(GLUT_ELAPSED_TIME));
    else if (state == GLUT_UP && button == tb_button)
        _tbStopMotion(button, glutGet(GLUT_ELAPSED_TIME));
}
```

Trackball.cpp (cont’d)

```
void CTrackball::tbKeyboard(int key)
{
    int i,j;
    for(i=0;i<4;i++)
      for(j=0;j<4;j++)
          tb_transform[i][j]=0.0;
    tb_transform[3][3]=1.0;
    switch (key)
    {
      case (int) 'z': tb_transform[0][0]=tb_transform[1][1]=tb_transform[2][2]=1.0; break;
      case (int) 'y': tb_transform[0][1]=tb_transform[1][2]=tb_transform[2][0]=1.0; break;
      case (int) 'x': tb_transform[0][2]=tb_transform[1][0]=tb_transform[2][1]=1.0; break;
      default:;
    }
    // remember to draw new position
    glutPostRedisplay();
}
```

Trackball.cpp (cont'd)

```
void CTrackball::tbMotion(int x, int y){
    GLfloat current_position[3], dx, dy, dz;
    if (tb_tracking == GL_FALSE) return;
    _tbPointToVector(x, y, tb_width, tb_height, current_position);

    // calculate the angle to rotate by (directly proportional to the
    // length of the mouse movement
    dx = current_position[0] - tb_lastposition[0];
    dy = current_position[1] - tb_lastposition[1];
    dz = current_position[2] - tb_lastposition[2];
    tb_angle = (float) (90.0 * sqrt(dx * dx + dy * dy + dz * dz));

    // calculate the axis of rotation (cross product)
    tb_axis[0] = tb_lastposition[1] * current_position[2] - tb_lastposition[2] * current_position[1];
    tb_axis[1] = tb_lastposition[2] * current_position[0] - tb_lastposition[0] * current_position[2];
    tb_axis[2] = tb_lastposition[0] * current_position[1] - tb_lastposition[1] * current_position[0];

    // reset for next time
    tb_lasttime = glutGet(GLUT_ELAPSED_TIME);
    tb_lastposition[0] = current_position[0];
    tb_lastposition[1] = current_position[1];
    tb_lastposition[2] = current_position[2];

    // remember to draw new position
    glutPostRedisplay();
}
```

House3DWithTrackball

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>
#include <iostream>
using namespace std;
#include "Trackball.h"

const int windowWidth=400;    const int windowHeight=400;

// define vertices and edges of the house
const int numVertices=10;
const int numEdges=18;
const float vertices[numVertices][3] =  {{0,0,0},{1,0,0},{0,1,0},{1,1,0},{0,0,2},
                                         {1,0,2},{0,1,2},{1,1,2},{0.5f,1.5f,0},{0.5f,1.5f,2}};
const int edges[numEdges][2] = {{0,1},{1,3},{3,2},{2,0},{4,5},{5,7},{7,6},{6,4},{0,4},
                                {1,5},{3,7},{2,6},{2,8},{8,3},{6,9},{9,7},{8,9}};

CTrackball trackball;                    // Add a trackball to our OpenGL program

void handleMouseMotion(int x, int y){ trackball.tbMotion(x, y); }
void handleMouseClick(int button, int state, int x, int y){ trackball.tbMouse(button, state, x, y);}
void handleKeyboardEvent(unsigned char key, int x, int y){ trackball.tbKeyboard(key);}
```

House3DWithTrackball (cont'd)

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);          // clear all pixels in frame buffer
    glColor3f (1.0, 0.0, 0.0);             // (red,green,blue) colour components

    glMatrixMode( GL_MODELVIEW );          // Set the view matrix ...
    glLoadIdentity();                      // ... to identity

    trackball.tbMatrix();

    // Rest is the same as in House3D
}

void init(void)
{
    // Rest of initialisation same as in House3D

    trackball.tbInit(GLUT_LEFT_BUTTON);
}

void reshape(int width, int height ) {
    // Rest of reshape is the same as in House3D

    trackball.tbReshape(width, height);
}
```

House3DWithTrackball (cont'd)

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(windowWidth, windowHeight);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("House3D");
    init ();                                        // initialise view

    glutMouseFunc(handleMouseClick);        // Set function to handle mouse clicks
    glutMotionFunc(handleMouseMotion);      // Set function to handle mouse motion
    glutKeyboardFunc(handleKeyboardEvent); // Set function to handle keyboard input

    glutDisplayFunc(display);               // Set function to draw scene
    glutReshapeFunc(reshape);               // Set function called if window gets resized
    glutMainLoop();
    return 0;
}
```

Notes on House3DWithTrackball

- The main class is almost identical to `House3D` except:
  - Add a global trackball variable
  - Pass callback functions to GLUT for mouse click events, mouse motion events and keyboard events. In more complex programs we have to decide which events apply to the trackball and which events are related to other parts of the program.
  - Initialise trackball and specify the associated mouse button.
  - Update trackball if the window is reshaped.
  - Add trackball rotation matrix to the MODEL_VIEW matrix stack.

- The *CTrackball* class contains functions for handling trackball events.
  - Mouse positions are transformed into rotations.
  - Trackball accumulates rotations
  - Use glutPostRedisplay() to redraw the window.