UNIT 4 :

GEOMETRIC OBJECTS AND TRANSFORMATION - 1

Syllabus

* Scalars, points, and vectors
* Three dimensional primitives
* Coordinate systems and Frames
* Modeling a colored cube
* Affine Transformations
* Rotation, Translation, and scaling

- 6 Hours.

* Minimum set of primitives from which we can obtain (build) more sophisticated objects are -

1. Scalars
2. points
3. Vectors

} Three basic elements.

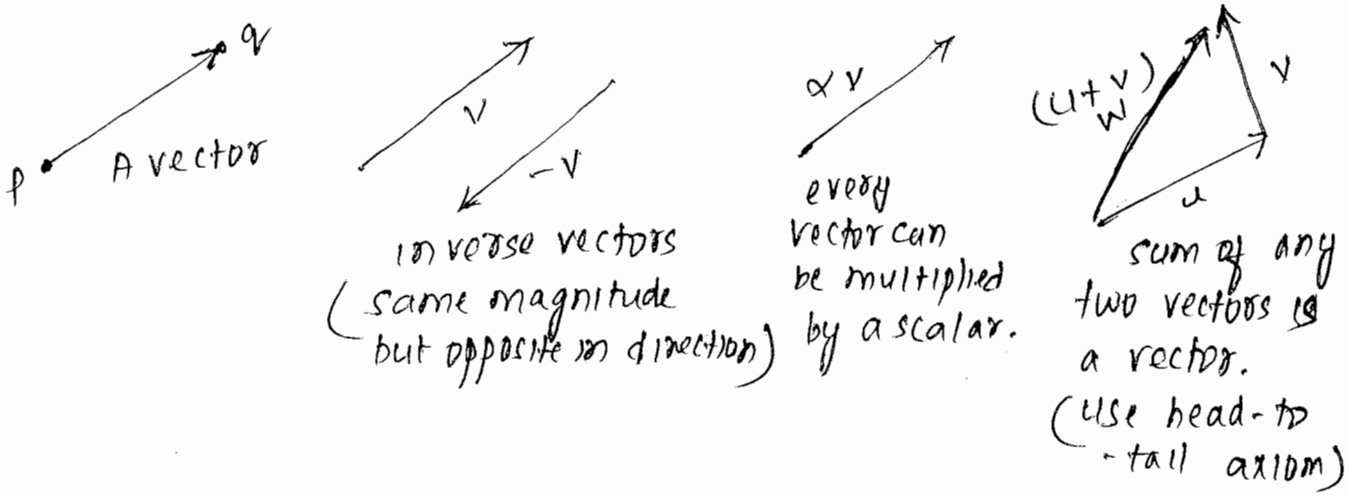
SCALARS, POINTS, AND VECTORS

Definitions

* A point is a fundamental geometric object.
A point is a location in space. Only property a point possess is location. It has neither shape nor a size.

* Scalars are quantities which have magnitude.
They obey operations of ~~geometry~~ ordinary arithmetic. Hence they obey addition, multiplication, subtraction, division, associativity and commutative rules.
Scalars alone have no geometric properties.

* A vector allows us to work with directions.
They have both magnitude and direction.
A directed line segment is a vector since it has both a magnitude (length) & direction (orientation)

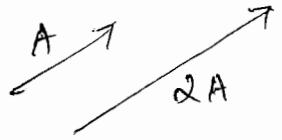


inverse vectors (same magnitude but opposite in direction)

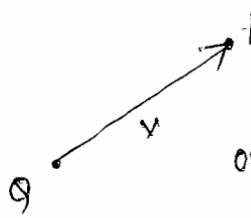
every vector can be multiplied by a scalar.

sum of any two vectors is a vector. (use head-to-tail axiom)

Some more ex:



identical vectors
(same mag & direct^n)



$V = P - Q$
or $P = V + Q$

→ point - point subtraction yields a vector.
It is equivalent to point - vector addition.

* f) Euclidean space is an extension of a vector space that adds a measure of size and distance and allows us to define such things as the length of a line segment.

* An Affine space is an extension of the vector space that includes additional type of object: the point

operations in Affine space -

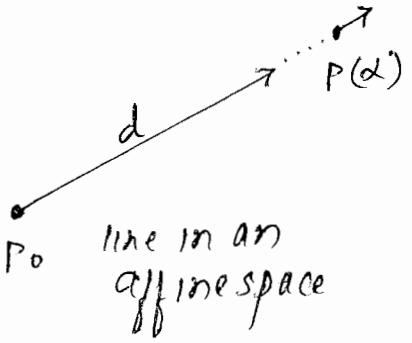
- Vector - Vector Addition
- Scalar - Vector Multiplication
- point - vector addition
- Scalar - Scalar operations.

note: For any point,

$$1 \cdot P = P$$
$$0 \cdot P = \mathcal{O} \text{ (zero vector)}$$

Lines

* Sum of point and a vector (or subtraction of two points) leads to the notion of a line in an affine space.



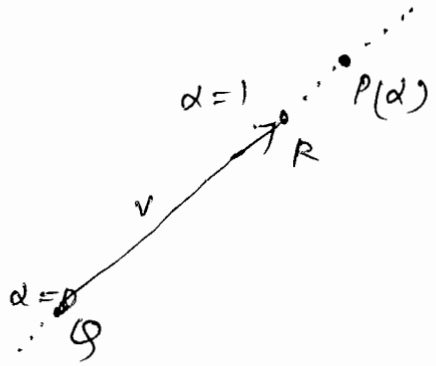
$P(\alpha) = P_0 + \alpha d$

It is the set of all points that passes through P_0 in the direction of a vector d.

Affine sum

$P = Q + \alpha v$ —(1)

describes all the points ~~from~~ on the line from Q in the direction of v. WKT -



$v = R - Q$

therefore (1) $\Rightarrow$

$P = Q + \alpha(R - Q)$

$= \alpha R + (1-\alpha) Q$.

or $\boxed{P = \alpha_1 R + \alpha_2 Q}$ where $\alpha_1 + \alpha_2 = 1$.

↳ It is called affine sum of points P & Q.

Gly Affine sum of points $P_1, P_2 \ldots P_n$ is -

$P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n$ where $\alpha_1 + \alpha_2 \ldots \alpha_n = 1$.

If $\alpha_i >= 0$ $i = 1, 2 \ldots n$ then it is called the convex Hull of the set of points.

Dot and cross product of vectors

Dot product

* Dot product of u & v is written as $u \cdot v$

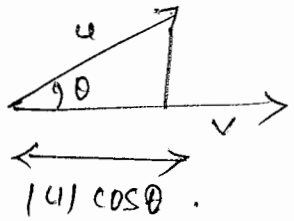
If $u \cdot v = 0$, u and v are said to be <u>orthogonal</u>

The square of magnitude of a vector is given by the dot product.

$$|u|^2 = u \cdot u$$

Cosine of the angle b/w two vectors is given by -

$$\cos\theta = \frac{u \cdot v}{|u||v|}$$



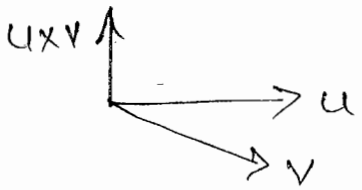
$|u|\cos\theta = \frac{u \cdot v}{|v|}$ is the length of the orthogonal projection of u onto v

cross product

If u & v are two vectors, then the third vector orthogonal to both u & v can be computed as

$n = u \times v$, it is called the cross product of u & v

$$|\sin\theta| = \frac{|u \times v|}{|u||v|}$$



planes

* A plane in an affine space can be defined as a direct extension of the parametric line.

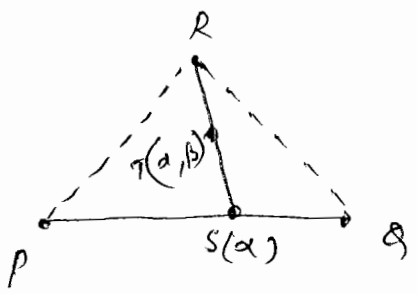
three points that are not on the same line determines a unique plane.

suppose P, Q, R are such points in an affine space.

line seg. that joins P and Q is the set of points of the form -

$$S(\alpha) = \alpha P + (1-\alpha) Q.$$

$$0 \le \alpha \le 1.$$



Suppose that we take an arbitrary point on this line seg and form a line seg from this point to R. using second parameter β, we can describe points along this line seg as

$$T(\beta) = \beta S + (1-\beta) R \qquad 0 \le \beta \le 1.$$

THREE DIMENSIONAL PRIMITIVES.

* three features characterize 3D objects that fit well with existing graphics h/w & s/w

1. Objects are described by their surfaces and can be thought of as being hallow.

⇒ Graphic package needs only 2D primitives to model 3D objects.

2. objects can be specified through a set of vertices in 3D.

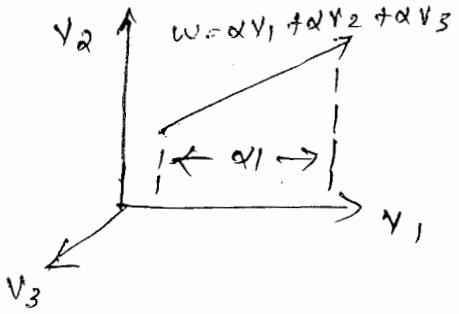
⇒ we can use pipeline architecture to process these vertices at high rates and generate images of object during rasterization

3. objects are either composed of or can be approximated by flat, convex polygons

COORDINATE SYSTEMS AND FRAMES

* In a 3D vector space, we can represent any vector w uniquely in terms of any 3 linearly independent vectors v_1, v_2, v_3 as -

$$w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$$



The scalars $\alpha_1, \alpha_2, \alpha_3$ are called components of w w.r.t the basis v_1, v_2, v_3

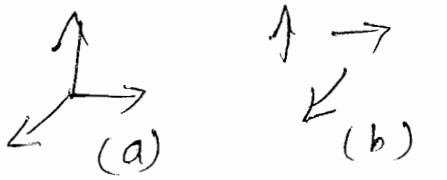
We can write the representation of w w.r.t to this basis as the column matrix -

$$a = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$

and for basis vectors -

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

note :



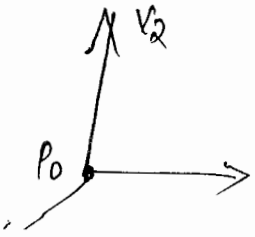
Both are correct since vectors have no fixed location

then,

$$w = a^T v \quad \text{ie} \quad w = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Frames

* A coordinate system is not sufficient to represent points.
* If we work in an affine space, we can add a single point, (the origin), to the basis vectors to form a frame.



A point p can be represented in terms of basis vectors as -

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3$$

$$P = P_0 + b^T \cdot v \quad , \quad b = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

Ie:

- A frame is determined by (P_0, v_1, v_2, v_3)
- within this frame,
  - Every vector can be written as -

$$w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$$

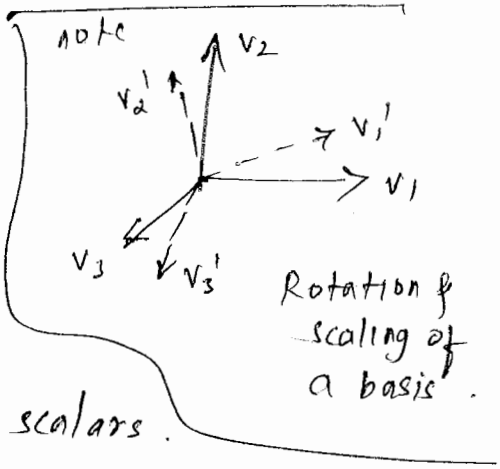
  - Every point can be written as

$$p = p_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3.$$

change of coordinate systems.

* Consider that $[v_1, v_2, v_3]$ and $[u_1, u_2, u_3]$ are two bases. Each basis vector in the second set can be represented in terms of the first basis vector and vice versa.
Hence there exists 9 scalar components $\{\gamma_{ij}\}$ as -

$$u_1 = \gamma_{11} v_1 + \gamma_{12} v_2 + \gamma_{13} v_3$$
$$u_2 = \gamma_{21} v_1 + \gamma_{22} v_2 + \gamma_{23} v_3$$
$$u_3 = \gamma_{31} v_1 + \gamma_{32} v_2 + \gamma_{33} v_3$$



Rotation & scaling of a basis.

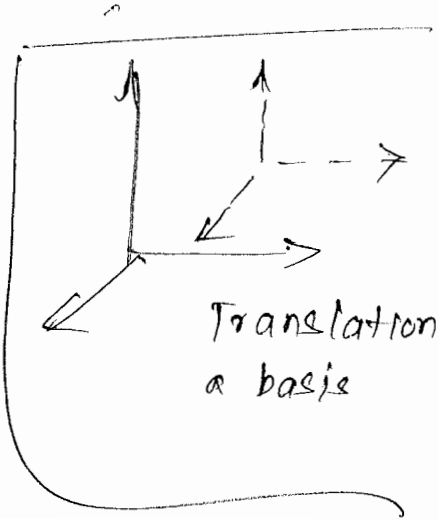
the 3×3 matrix

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{23} & \gamma_{33} \end{bmatrix}$$ defines these scalars.

Therefore,

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Or $u = Mv$.


Translation of a basis

* The matrix M contains the information to go from a representation of a vector in one basis to its representation in the second basis. The inverse of M gives the matrix representation of the change from $\{u_1, u_2, u_3\}$ to $\{v_1, v_2, v_3\}$

* Consider the vector w that has the representation $\{\alpha_1, \alpha_2, \alpha_3\}$ w.r.t $\{v_1, v_2, v_3\}$

ie $w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

$= a^T v$ where $a = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$ & $v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$

Assume that b is the representation of w wrt $\{u_1, u_2, u_3\}$

ie $w = \beta_1 u_1 + \beta_2 u_2 + \beta_3 u_3$

$= b^T u$.

Then, expressing second bases in terms of first basis -

$$w = b^T . u$$
$$= b^T . Mv$$
$$= a^T v$$

Thus, $a = M^T b$. $(\because a^T = b^T M)$

The matrix, $T = (M^T)^{-1}$ takes us from a to b, through the simple matrix eqn -

$$\boxed{b = Ta}$$

Example:

Suppose we have a ~~matrix~~ vector w whose representation is $a = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ along the basis vectors v_1, v_2, v_3

ie $w = v_1 + 2v_2 + 3v_3$.

suppose that we make a new basis from the vectors v_1, v_2, v_3

$$u_1 = v_1$$
$$u_2 = v_1 + v_2$$
$$u_3 = v_1 + v_2 + v_3$$

Then the matrix, M is

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

The above matrix converts the basis v_1, v_2, v_3 to u_1, u_2, u_3.

To do the opposite, the matrix is -

$$T = (M^T)^{-1}$$
$$= \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

In the new system, the representation of w is -

$$b = Ta$$
$$= \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 3 \end{bmatrix}$$

ie $\boxed{w = -u_1 - u_2 + 3u_3}$

Homogeneous Coordinates

* Homogeneous coordinates avoid potential confusion between a vector and point by using a 4-D representation for both points and vectors in 3-D.

1. In the frame specified by (v_1, v_2, v_3, P_0), any point P can be written uniquely as -

$$P = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + P_0$$

or $$P = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

note

$0 \cdot P = 0$

$1 \cdot P = P$.

Thus, P is represented by the column matrix -

$$P = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 1 \end{bmatrix}$$

2. In the same frame, any vector w can be written as.

$$w = \delta_1 v_1 + \delta_2 v_2 + \delta_3 v_3$$

or $$w = \begin{bmatrix} \delta_1 & \delta_2 & \delta_3 & 0 \end{bmatrix}^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

Thus, w is represented by the column matrix -

$$w = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ 0 \end{bmatrix}$$

change in frame

* If (v_1, v_2, v_3, P_0) and (u_1, u_2, u_3, Q_0) are two frames, then we can express the basis vectors and reference point of second frame in terms of the first as-

$$u_1 = \gamma_{11} v_1 + \gamma_{12} v_2 + \gamma_{13} v_3$$
$$u_2 = \gamma_{21} v_1 + \gamma_{22} v_2 + \gamma_{23} v_3$$
$$u_3 = \gamma_{31} v_1 + \gamma_{32} v_2 + \gamma_{33} v_3$$
$$Q_0 = \gamma_{41} v_1 + \gamma_{42} v_2 + \gamma_{43} v_3 + P_0 .$$

In matrix form -

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \quad \text{where} \quad M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

M is called the matrix representation of the change of frames.

* We can also use M to compute the changes in the representations directly.
Suppose a & b are homogeneous-coordinate representations either of two points or of two vectors in the two frames. Then -

$$b^T \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = b^T M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} = a^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

Hence $\boxed{a = M^T b}$.

Where $M^T = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Example -

Assume the two frames with basis vectors having the following relations -

$$u_1 = v_1$$
$$u_2 = v_1 + v_2$$
$$u_3 = v_1 + v_2 + v_3$$

The reference point does not change. So -

$$Q_0 = P_0.$$

The matrix in homogeneous form is -

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Suppose that in addition to changing the basis vectors, we also want to move the reference point to that point that has the representation $(1, 2, 3, 1)$ in the original system.

Then, $Q_0 = v_1 + 2v_2 + 3v_3 + P_0$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 1 \end{bmatrix}$$

$$T = (M^T)^{-1} = \begin{bmatrix} 1 & -1 & 0 & 1 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* Note that p is a point (1,2,3) in the original frame. Then the point p' in the new frame is

$$\begin{bmatrix} 1 & -1 & 0 & 1 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

↳ The origin in new system..

However, A vector (1,2,3) which is represented as

$$a = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 0 \end{bmatrix}$$ in original system is transformed to

$$b = \begin{bmatrix} -1 \\ -1 \\ 3 \\ 0 \end{bmatrix}$$ in the new system.

MODELLING A COLORED CUBE

A cube is a simple 3D object.

We start by assuming that the vertices of the cube are available through an array of vertices.

For ex -

```
GLfloat vertices[8][3] = {{-1.0, -1.0, -1.0}, {1.0, -1.0, -1.0}, {1.0, 1.0, -1.0}, {-1.0, 1.0, -1.0},
{-1.0, -1.0, 1.0}, {1.0, -1.0, 1.0}, {1.0, 1.0, 1.0}, {-1.0, 1.0, 1.0}};
```

We can use these list of points to specify the faces as -

```
glBegin (GL_POLYGON)
    glVertex3fv (vertices[0]);
    glVertex3fv (vertices[3]);
    glVertex3fv (vertices[2]);
    glVertex3fv (vertices[1]);
glEnd();
```

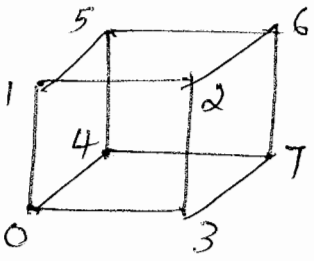
Similarly the other 5 faces can be defined.

The other five faces are -

(2,3,7,6) , (0,4,7,3) , (1,2,6,5)

(4,5,6,7) , (0,1,5,4)

* A vertex list data structure can be used to represent a cube as shown -



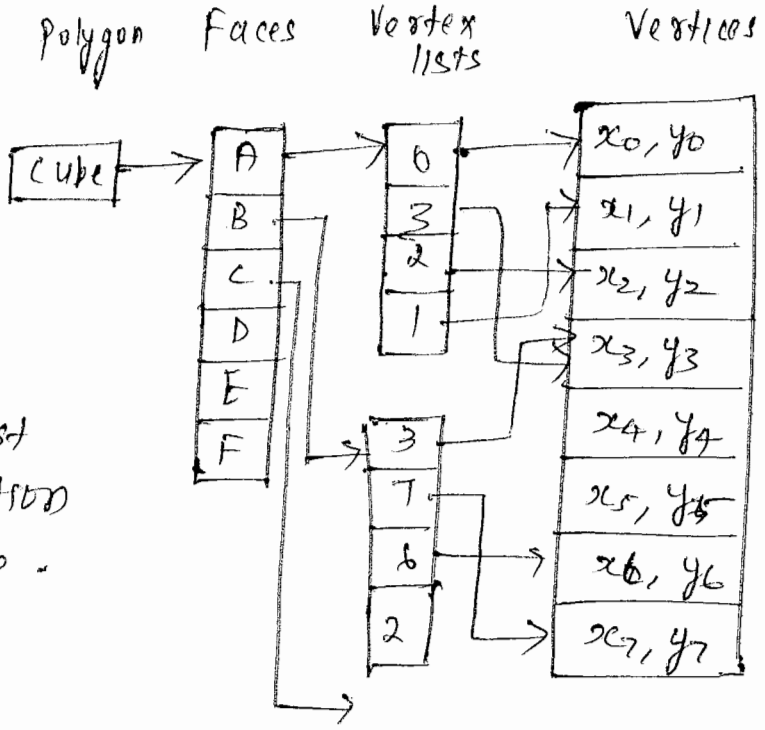


fig: Vertex list representation of a cube.

* A cube is composed of 6 faces. Each face is a quadrilateral which meets other quadrilateral at vertices.
Each vertex is shared by 3 faces. Each edge is shared by 2 faces.

* <u>vertex arrays</u> must first be enabled as -

```
glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);
```

The openGL must be specified as to where and in what format the vertex arrays are, using -

```
glVertexPointer (3, GL_FLOAT, 0, vertices);
glColorPointer (3, GL_FLOAT, 0, colors);
```

The adv of vertex arrays is that each geometric location appears only once. Instead of being repeated each time, the vertex needs a change only once.

* Bilinear Interpolation is used for coloring.
If c_0, c_1, c_2, & c_3 are colors assigned to the vertices in the application program. Then the first interpolation is used to interpolate colors along the edges b/w vertices 0 and 1, 2 and 3 creating RGB colors using the parametric equations -

$$c_{01}(\alpha) = (1-\alpha)c_0 + \alpha c_1$$

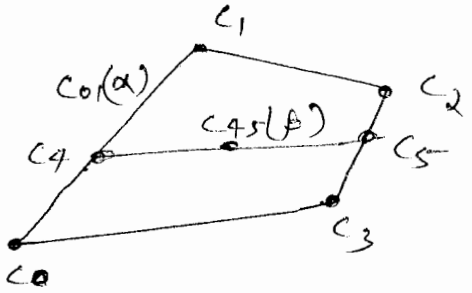


fig: Bilinear interpolation.

* As ~~col~~ α goes from 0 to 1, we generate different colors combination. This process continues until entire cube is colored.

* The code for generating a color cube is -

```
GLfloat vertices [8][3] = {  same as before
                          };
GLfloat colors [8][3] = {
                          same as vertices.
                        };
```

```
void quad (int a, int b, int c, int d)
{
    glBegin ( GL_QUADS ) ;
        glColor3fv ( colors[a] );
        glVertex3fv ( vertices[a]);
        glColor3fv ( colors[b] );
        glVertex3fv (vertices[b]);
        glColor3fv( colors[c] );
        glVertex3fv ( vertices[c]);
        glColor3fv (colors[d]);
        glVertex3fv(vertices[d]);
    glEnd();
}

void colorCube ()
{
    quad ( 0,3,2,1);
    quad ( 2,3,7,6);
    quad ( 0,4,7,3);
    quad ( 1,2,6,5);
    quad ( 4,5,6,7);
    quad ( 0,1,5,4);
}
```

AFFINE TRANSFORMATIONS

* A Transformation is a function that takes a point (or vector) and maps that point into another point (or vector)

we can picture such a function by looking at fig below or by writing down the functional form -

$q = T(p)$ for points

$v = R(u)$ for vectors

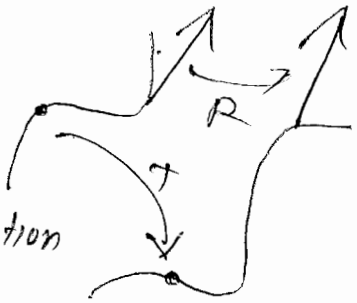


fig: transformation

* If we use Homogeneous coordinates then we can represent both vectors and points as 4D column matrix as -

$$q = f(p)$$
$$v = f(u)$$

* Using homogeneous coordinates, a linear transformation transforms the representation of a given point into another representation as (↳ or vector)

$$v = Cu$$

where

$$C = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

12 values of α can set arbitrarily & said that the transformation has 12 degrees of freedom.

* A point is represented in affine space as

$$p = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ 1 \end{bmatrix}$$

A vector is represented as -

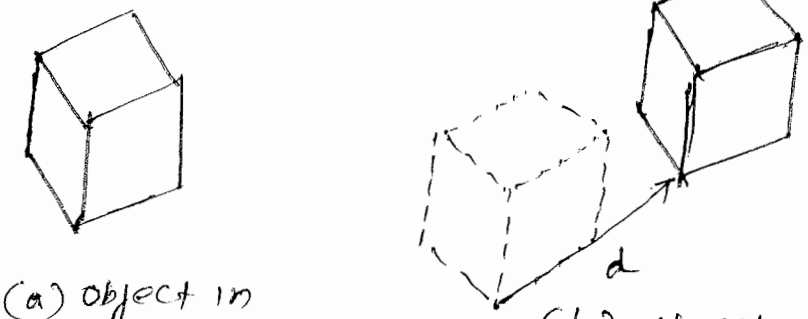
$$u = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 0 \end{bmatrix}$$

Hence a point has 12 degrees of freedom where as a vector has only 9.

TRANSLATION, ROTATION, AND SCALING.

Translation

* Translation is an operation that displaces points by a fixed distance in a given direction (refer fig)



(a) object in original position

(b) object translated.

To specify a translation, we need only to specify a displacement vector d, because the transformed points are given by -

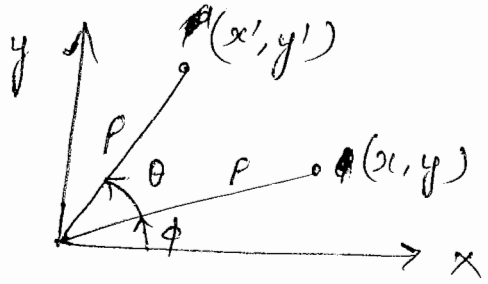
$$\boxed{p' = p + d}$$ for all points p on the object.

Translation has three degrees of freedom because we can specify 3 components of displacement vector arbitrarily.

Rotation

* Rotation operation accepts more than one parameters for its specification.

* fig show 2D rotation about origin



A 2D point at (x, y) is rotated about the origin by an angle θ to the position (x', y')

This can be represented as -

$$x = \rho \cos\phi$$
$$y = \rho \sin\phi$$
$$x' = \rho \cos(\theta + \phi)$$
$$y' = \rho \sin(\theta + \phi)$$

expanding, we get

$$x' = \rho\cos\phi\cos\theta - \rho\sin\phi\sin\theta$$
$$= x\cos\theta - y\sin\theta.$$
$$y' = \rho\cos\phi\sin\theta + \rho\sin\phi\cos\theta$$
$$= x\sin\theta + y\cos\theta.$$

where $x = \rho\cos\phi$

$y = \rho\sin\phi$.

In matrix form,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

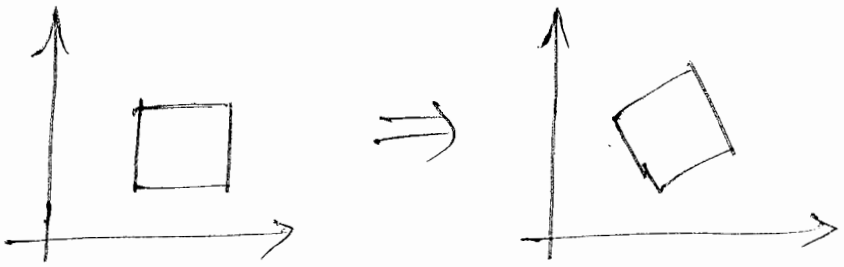


fig: Rotation about a fixed point.

note: Rotation & Translation are known as rigid body transformations since no combination of rotations & translations can alter the shape or volume of an object.

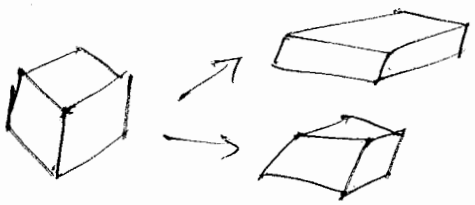


fig: non rigid body Transformation.

Scaling → <has 6 degrees of freedom>

* Scaling is an affine non rigid body transformation by which we can make an object bigger or smaller.
* Fig shows both uniform & non uniform scaling.

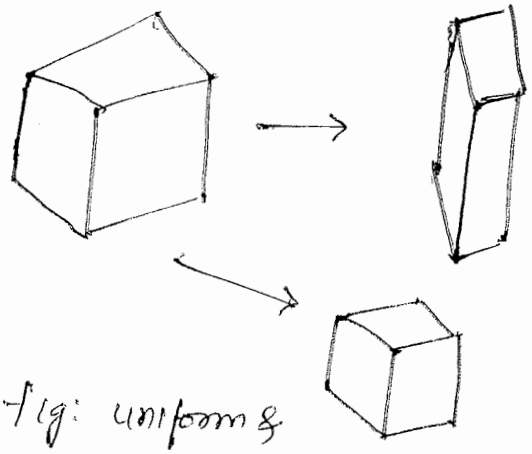


fig: uniform & non uniform scaling.

* Scaling have a fixed point
* To specify scaling, we should specify the fixed point, direction we wish to scale, & a scale factor (α)
  If $\alpha > 1$ ⇒ objects gets bigger
  If $\alpha < 1$ ⇒ Object gets smaller
  -ve value of α gives reflections (ref fig)

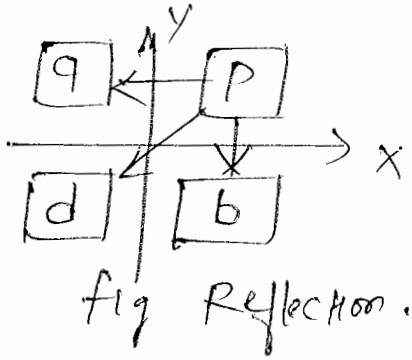
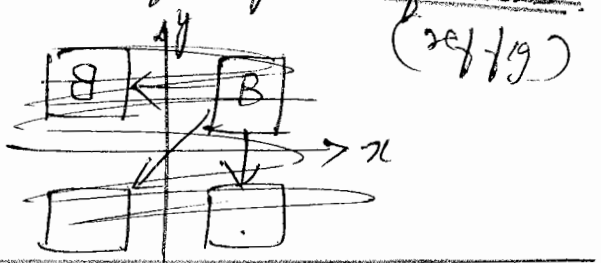


fig Reflection.



UNIT 5 :

GEOMETRIC OBJECTS AND TRANSFORMATIONS - 2

syllabus

* Transformations in homogeneous coordinates.
* Concatenation of Transformations.
* opengl Transformation matrices.
* Interfaces to three dimensional applications.
* Quaternions

TRANSFORMATION IN HOMOGENEOUS COORDINATES

* within a frame, each affine Transformation is represented by a 4 x 4 matrix of the form

$$A = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* Here we discuss 4 types of affine Transformation -
  - Translation
  - Rotation
  - Scaling
  - shear.

Translation

* Translation displaces points to new positions defined by displacement vector.
* If we move the point p to p' by displacing by a distance d, then $\boxed{p' = p + d}$
* Homogeneous coordinate forms of p, p', & d are

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad p' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \quad d = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ 0 \end{bmatrix}$$

* These equations can be written component by component as -

$$x' = x + \alpha_x$$
$$y' = y + \alpha_y$$
$$z' = z + \alpha_z$$

* However, we can also get this result using the matrix multiplication -

$$p' = Tp$$

where -

$$T = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

T is called translation matrix we sometimes write it as $T(\alpha_x, \alpha_y, \alpha_z)$

* we can obtain the <u>inverse of a translation matrix</u> as follows -

$$T^{-1}(\alpha_x, \alpha_y, \alpha_z) = T(-\alpha_x, -\alpha_y, -\alpha_z) = \begin{bmatrix} 1 & 0 & 0 & -\alpha_x \\ 0 & 1 & 0 & -\alpha_y \\ 0 & 0 & 1 & -\alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

* For both scaling and rotation, there is a <u>fixed point</u> that is unchanged by the transformation. (we let the fixed point to be origin here).

* A <u>scaling matrix</u> with a fixed point of the origin allows for independent scaling (increase or decrease size of primitive) along the coordinate axes.

* The three equations are :-

$$x' = \beta_x x$$
$$y' = \beta_y y$$
$$z' = \beta_z z$$

* These three eqns can be combined in homogeneous form as -

$$\boxed{p' = Sp}$$ where $S = S(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \end{bmatrix}$

* We can obtain the inverse of scaling matrix by applying the reciprocals of the scale factors -

$$S^{-1}(\beta_x, \beta_y, \beta_z) = S\left(\frac{1}{\beta_x}, \frac{1}{\beta_y}, \frac{1}{\beta_z}\right)$$

Rotation.

* Here we take the fixed point as origin.
* Rotation enables the programmer to rotate a given point w.r.t the three degrees of freedom.
* Suppose we rotate a point $p(x, y, z)$ about the z-axis to get the new point $p'(x', y', z')$. Then the equation for rotation about z axis by an angle θ is given by -

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$
$$z' = z$$

or, in matrix form -

$$p' = R_z p$$

where

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

111ly, eqn for rotation about x-axis is -

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

111ly, eqn for rotation about y axis is -

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

* A rotation by θ can always be <u>undone</u> by a subsequent rotation by $-\theta$. Hence -

$$R^{-1}(\theta) = R(-\theta)$$

note:
$$\cos(-\theta) = \cos\theta$$
$$\sin(-\theta) = -\sin\theta .$$

Shear

* Consider a cube centered at origin, aligned with the axes and viewed from +ve z axis. (refer fig)
If we pull the top of the object (cube) to the right and bottom to the left, we say that we <u>shear</u> the object in the x-direction.
<u>note:</u> neither y nor z values are changed by the shear.

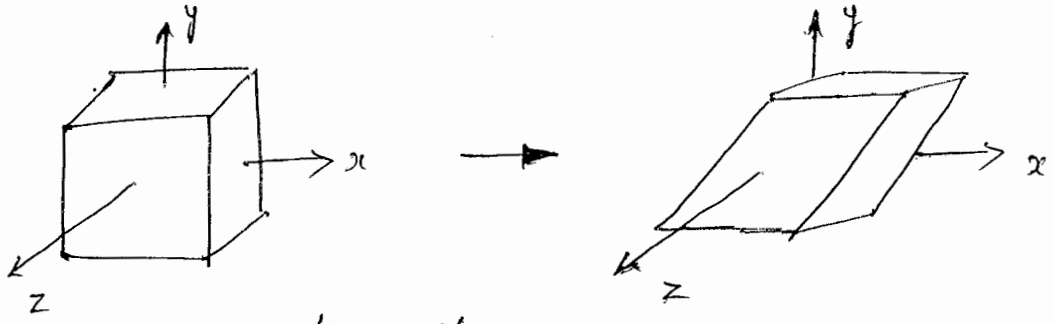


fig: shear.

* Using simple Trigonometry on below fig, we see that each shear is characterised by a single angle θ: The equations for this shear are

$$x' = x + y\cot\theta$$
$$y' = y$$
$$z' = z$$

, leading to shearing matrix

$$H_x(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

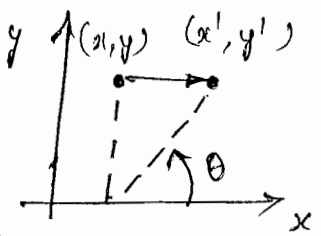


fig: composition of shear matrix.

* We can obtain inverse by shearing in opposite direction

CONCATENATION OF TRANSFORMATIONS

* It is nothing but multiplication of sequence of basic transformations in order to produce arbitrary transformation.

for ex: If we carry out three ~~consequetive~~ successive transformations on a point p, creating a new point q, then, $q = CBAp$ A, B, & C can be arbitrary 4 × 4 matrices.

order is important. Here we carry out A, followed by B, and followed by C.

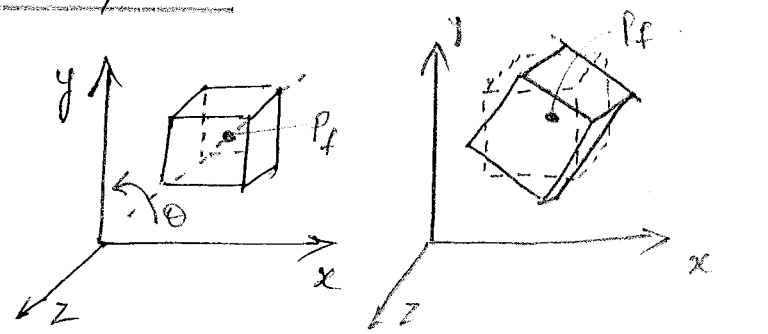
ie $q = (C(B(Ap)))$

$p \rightarrow [A] \rightarrow [B] \rightarrow [C] \rightarrow q$

* Here, we develop matrices for -
  - Rotation about a fixed point
  - General rotation.
  - Instance Transformation.
  - Rotation about an arbitrary axis

Rotation about a fixed point

* Consider a cube with its center at P_f and its sides aligned with axes.
* Assume that the cube is to be rotated about z axis (as shown) about its center P_f.



Rotation of cube about its center.

* It can be done as follows -

<u>step 1</u>: Move the cube to the origin by applying basic transformation of translate $T(-P_f)$.

<u>step 2</u>: Rotate w.r.t (or about) z axis by desired angle θ by applying $R_z(\theta)$.

<u>step 3</u>: Move the Object back such that its center is again at P_f by applying ~~reverse~~ translation as $T(+P_f)$

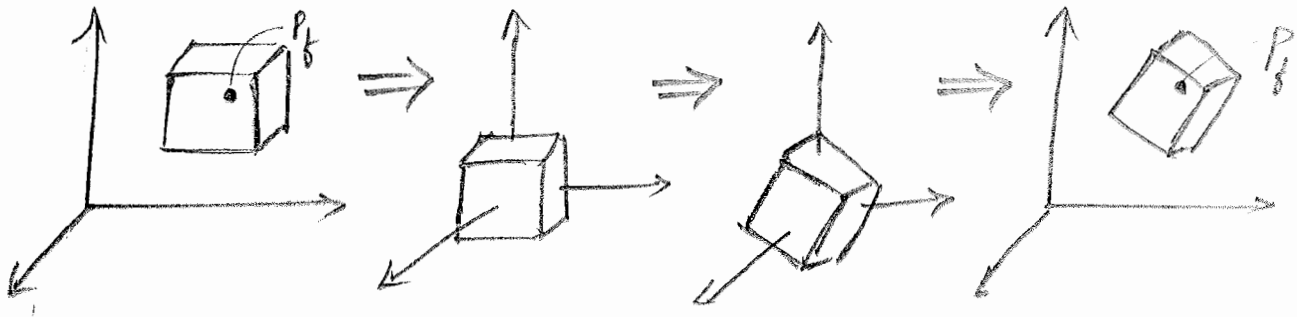
* Concatenating the above transformations, we get

$$\boxed{M = T(P_f)\, R_z(\theta)\, T(-P_f)}$$

If we multiply out the matrices, we find that

$$M = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & x_f - x_f\cos\theta + y_f\sin\theta \\ \sin\theta & \cos\theta & 0 & y_f - x_f\cos\theta - y_f\sin\theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* Fig below shows <u>sequence of transformations</u>.



<u>General Rotation</u>

* We now show that arbitrary rotation about the origin can be composed of three successive rotations about the three axis.

* we can obtain the desired matrix by first doing a rotation about x-axis and then about y axis and then about z-axis.

* However, the order of multiplication does not matter.

Therefore, the final rotation matrix M would be -

$$M = R_x(\alpha) * R_y(\beta) * R_z(\gamma)$$

By selecting appropriate values of α, β, and γ, we can achieve any desired orientation.

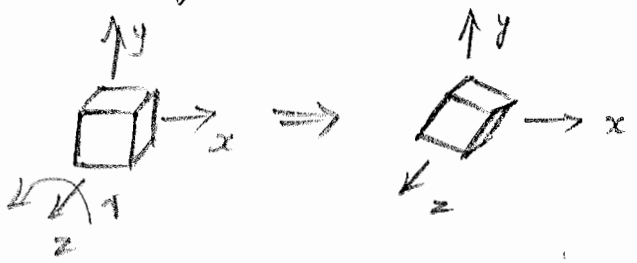


fig: rotation about z-axis.

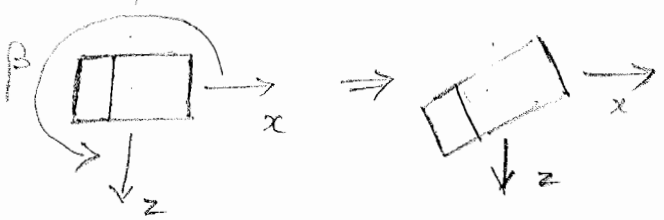


fig: Rotation about y-axis.

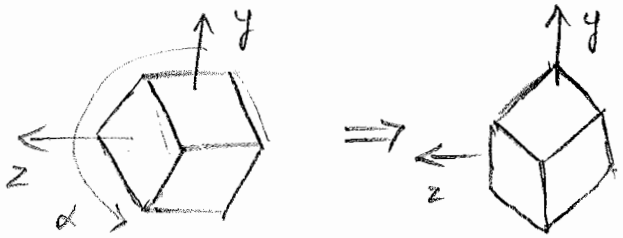


fig: Rotation about z-axis

Instance Transformation

* Consider a scene composed of many simple objects as shown.

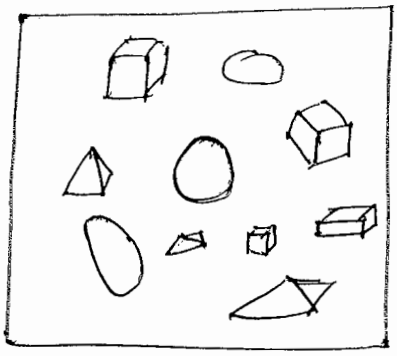


fig: scene of simple objects.

* There are two ways to create the above scene
1. Define each of these objects through its vertex, using glVertex()
2. An alternative is to define each of the objects types once at a convenient size, place, & orientation.
Each occurence of an object in the scene is an "instance" of that object's prototype, and we can obtain the desired size, orientation, and location by applying affine Transform called instance Transformation to the prototype.

* Instance transformation is applied in the order shown in the fig.
* In case of instance transformation, objects are originally defined in their own frames, First they are scaled to the desired size, Then they are oriented suitably using rotation matrix and then it is translated to the desired location.

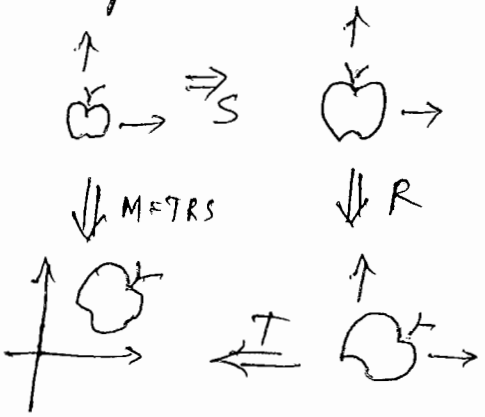


fig: Instance Transformatio

* Therefore, Instance Transformation matrix is of the form -

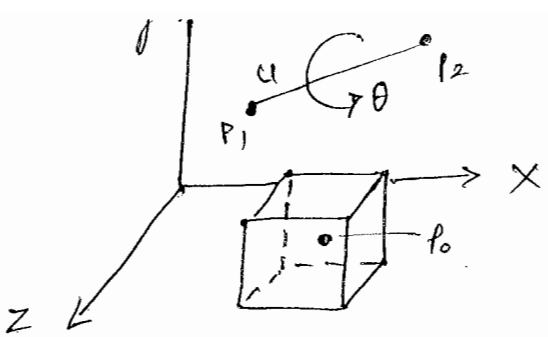
$$M = T * R * S$$

Rotation about an Arbitrary axis

* Consider a point P_0 as the center of the cube.
* The vector about which the cube is to be rotated can be specified by providing the points P_1 and P_2. The vector defined by P_1 and P_2 is u.

$$u = P_2 - P_1$$

~~* To rotate~~



* To rotate the cube about the vector u by an angle θ, the steps to be performed are as follows -

Step 1: Translate the fixed point, P_0 to the origin by performing $T(-P_0)$

Step 2: Rotate the cube about the x-axis by performing $R_x(+\theta_x)$

Step 3: Rotate the cube about the y-axis by performing $R_y(\theta_y)$

Step 4: Rotate the cube about the z-axis by angle θ (known) performing $R_z(\theta_z)$

~~Step 5: Perform inverse rotation about z axis ie $R_z(-\theta_z)$~~

Step 5: Perform inverse rotation about y axis ie $R_y(-\theta_y)$

Step 6: Perform inverse rotation about x axis ie $R_x(-\theta_x)$.

Step 7: perform inverse Translation of the fixed point P_0 using $T(P_0)$.

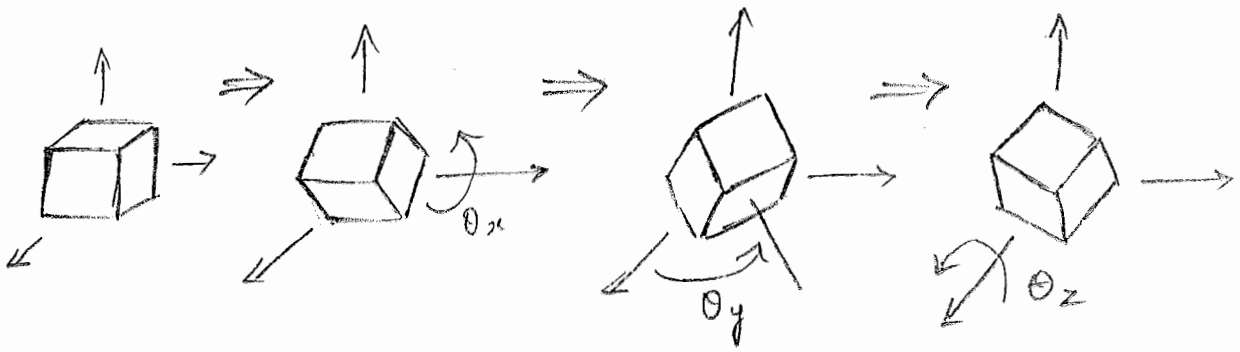
* Therefore, the concatenated matrix M is

$$M = T(P_0)\, R_x(-\theta_x)\, R_y(-\theta_y)\, R_z(\theta_z)\, R_y(\theta_y)\, R_x(\theta_x)\, T(-P_0)$$

* Our first and last transformation is the translation. ie $T(-P_0)$ and $T(P_0)$.
In between we perform rotation,

$$R = R_x(-\theta_x)\, R_y(-\theta_y)\, R_z(\theta)\, R_y(\theta_y)\, R_x(\theta_x).$$

This sequence of rotations is shown below.



Determining θ_x and θ_y.

* We replace 'u' with a unit length vector

$$V = \frac{u}{|u|} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \end{bmatrix}$$

we have, $\alpha_x^2 + \alpha_y^2 + \alpha_z^2 = 1$, since V is a unit length vect

* we draw a line segment from origin to $(\alpha_x, \alpha_y, \alpha_z)$
This line seg. has unit length & the orientation of V.
Next we draw perpendiculars from the point $(\alpha_x, \alpha_y, \alpha_z)$ to the co ordinate axes as shown.
The three direction angles - ϕ_x, ϕ_y, ϕ_z are the angles b/w the line seg (or v) and the axes.
The direction cosines are given by

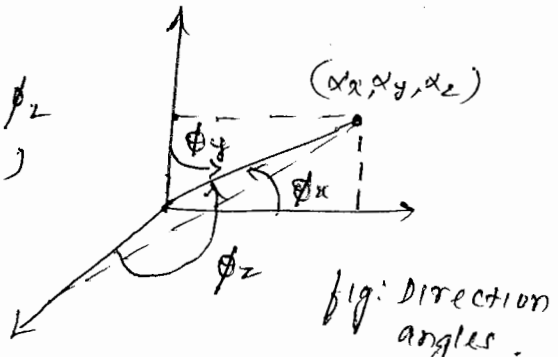


fig: Direction angles.

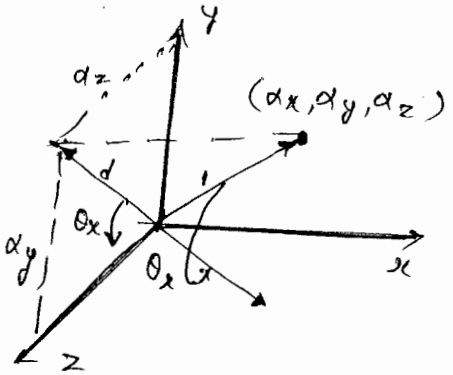
$$\cos\phi_x = \alpha_x$$
$$\cos\phi_y = \alpha_y$$
$$\cos\phi_z = \alpha_z$$

only two of the direction angles are independent because,

* We now calc θ_x & θ_y using these angles.

Computation of x rotation.

* Consider the fig shown,

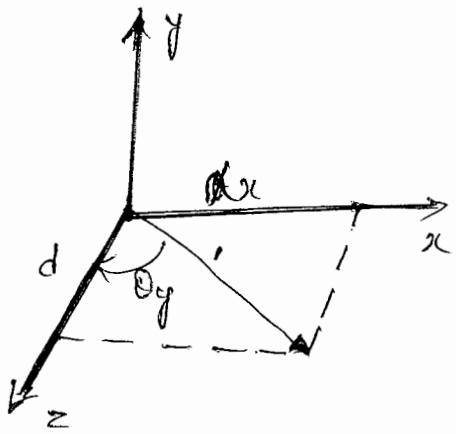


- If we ~~see~~ ~~a line~~ look at the projection of the line seg (before rotation) on the plane $x=0$, we will see a line seg of length d on this plane. The line that we see on the wall is the shadow of line seg from origin to $(\alpha_x, \alpha_y, \alpha_z)$
Note that length of shadow is less than length of line seg. We can say that the line seg has been foreshortened to $d = \sqrt{\alpha_y^2 + \alpha_z^2}$. We ~~get~~ never need to compute θ_x. rather we need to compute only -

$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z/d & -\alpha_y/d & 0 \\ 0 & \alpha_y/d & \alpha_z/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Computation of y rotation.

* It is similar to the above.



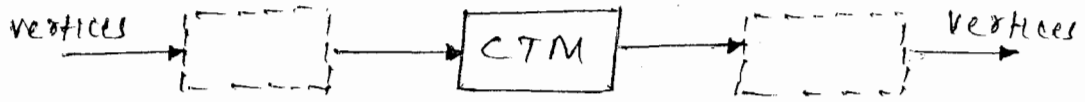
$$R_y(\theta_y) = \begin{bmatrix} d & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OPENGL TRANSFORMATION MATRICES

* Here we see the implementation of an homogeneous - coordinate transformation package and of that package's interface to the user.
* we use glMatrixMode to select the matrix to which the operations apply.
In opengl, the model-view matrix normally is an affine - transformation matrix and has only 12 degrees of freedom.

Current Transformation matrix. (CTM)

* It is the matrix that is applied to any vertex that is defined subsequent to its setting.
CTM is part of the pipeline. Thus, if p is a vertex specified in the application, then the pipeline produces Cp.



* CTM is an 4×4 matrix, initially set to identity matrix. It can be reinitialized as needed using '←'
eg C ← I
* we denote CTM by C. CTM can be altered by a set of functions provided by graphics package.
* The functions that alter C are of two forms
1. Those that load it with some matrix
2. Those that modify it by premultiplication or postmultiplication by a matrix.
opengl uses only post multiplication.
* The three transformations supported in most systems are
- Translation
- scaling with fixed point of the origin

* symbolically, we can write these operations in post multiplication form as -

$C \leftarrow CT$

$C \leftarrow CS$

$C \leftarrow CR$

and in load form as -

$C \leftarrow T$

$C \leftarrow S$

$C \leftarrow R$

* most systems allow us to load the CTM with an arbitrary matrix M -

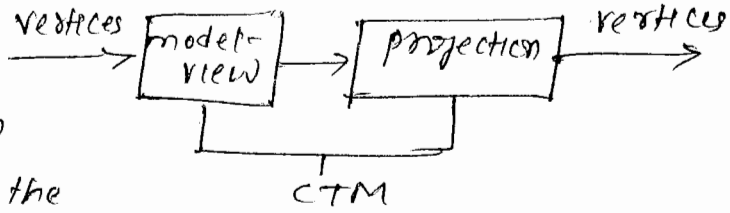
$C \leftarrow M$

or to postmultiply by an arbitrary matrix m -

$C \leftarrow CM$

Rotation, Translation and scaling

* In openGL, matrix that is applied to all primitives is the product of model-view matrix and projection matrix. ie CTM is a product of these two matrices. we can manipulate each individually by selecting the desired matrix by glMatrixMode.



→ We can load a matrix with the function

```
glLoadMatrixf(pointer_to_matrix);
```

→ or set a matrix to the identity matrix as -

```
glloadIdentity();
```

→ We can alter the selected matrix with

```
glMultMatrix(pointer_to_matrix);
```

→ Rotation, Translation, and scaling are provided thru' these functions

```
glRotatef (angle, vx, vy, vz);
glTranslatef (dx, dy, dy);
glScalef (sx, sy, sz);
```

Rotation about a fixed point in openGL.

* For a 45° rotation about the line through the origin and the point (1,2,3) with a fixed of (4,5,6), we have

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (4.0, 5.0, 6.0);
glRotatef (45.0, 1.0, 2.0, 3.0);
glTranslatef (-4.0, -5.0, -6.0);
```

Order of Transformations

* ~~The sequence of~~ Rule in openGL is this:
"Transformation specified last is the one applied first"

* The sequence of operations we specified above was

$C \leftarrow I$
$C \leftarrow CT(4.0, 5.0, 6.0)$
$C \leftarrow CR(45.0, 1.0, 2.0, 3.0)$
$C \leftarrow CT(-4.0, -5.0, -6.0)$

or

$C = T(4.0, 5.0, 6.0)\, R(45.0, 1.0, 2.0, 3.0)\, T(-4.0, -5.0, -6.0)$

Spinning of the cube // not needed, can be skipped.

* We define following three call back functions -

```
glutDisplayFunc (Display);
glutIdleFunc (spincube);
glutMouseFunc (mouse);
```

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    colorcube();
    glutSwapBuffers();
}

void mouse(int bt, int st, int x, int y)
{
    if(bt==GLUT_LEFT_BUTTON && st==GLUT_DOWN) axis = 0;
    --"-- _MIDDLE_ --"-- axis = 1;
    --"-- RIGHT --"-- axis = 2;
}

void spincube()
{
    theta[axis] += 2.0;
    if(theta[axis] > 360.0) theta[axis] -= 360;
    glutPostRedisplay();
}

void mykey(char key, int mousex, int mousey)
{
    if(key=='q' || key=='Q') exit();
}
```

Loading, Pushing, & popping Matrices

Sometimes if the programmer wants to perform a certain transformation and then return to the same state as before, then he can push the transformation matrix on to stack with glPushMatrix() before multiplication & recover it later with glPopMatrix().

eg

```
glPushMatrix ();
glTranslatef (...);
glRotatef (...);
glScalef (...);
    /* draw objects here */
glPopMatrix ();
```

note:

1. we can load a 4x4 homogeneous coordinate matrix as the current matrix as -

```
glLoadMatrixf (myarray);
```

2. we can also multiply on the right of current matrix by a user defined matrix as -

```
glMultMatrixf (myarray);
```

3. eg for forming myarray -

```
GLfloat m[4][4];
GLfloat myarray[16];
for (i=0; i<3; i++)
    for (j=0; j<3; j++)
        myarray[4*j+i] = m[i][j];
```

INTERFACES TO THREE-DIMENSIONAL APPLICATIONS

* Glut provides for smoother and more interesting interfaces to 3D applications by allowing the user to use keyboard along with mouse to provide interaction.
* Suppose that the user wants to use one mouse button for orienting an object, one for getting closer to or farther from the object, and one for translating the object to left or right.
* We can use the motion call back to achieve all these functions. The call back returns which button has been activated and where the mouse is located. This location of the mouse can be used to control the direction of rotation, translation, and to move in or move out.

A virtual Track Ball

* A virtual Track ball can be created using a mouse and the display device (eg: monitor)
* Adv of virtual device - it creates a frictionless track ball which once started to rotate will continue to rotate until stopped by the user. Thus it supports continuous rotations of objects but will still allow changes in the speed and orientation of the rotation.
* It can be achieved by mapping the position of the ~~mouse~~ trackball to that of a mouse.
* Consider the track ball as shown.
* We assume that the trackball has a radius of 1 unit. We can map a position on its surface to the plane $y=0$ by doing an orthogonal projection as shown below -

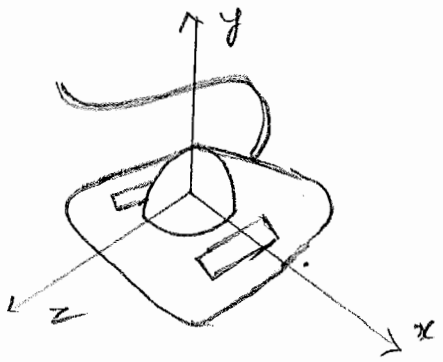
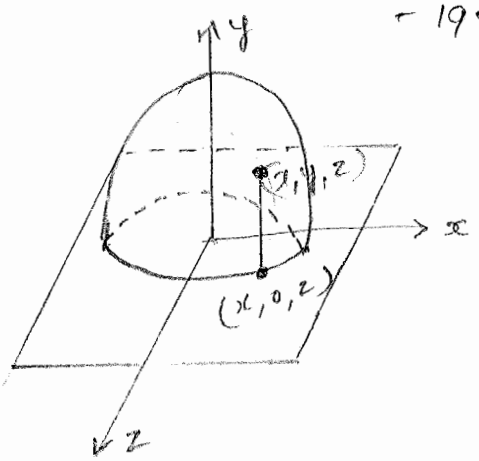
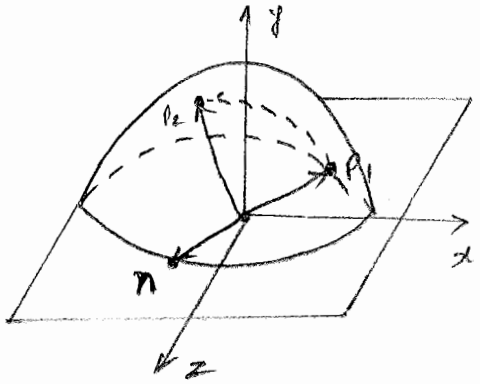


fig: Track ball frame



* The position (x, y, z) on the surface of the ball is mapped to $(x, 0, z)$ on the plane.
* The motion of the track ball that moves from one point p_1 to another point p_2 can be computed by computing the angle θ wrt p_1 and p_2



$$n = p_1 * p_2 .$$

$$\boxed{|\sin\theta| = |n|}$$

If we a tracking the mouse at high rate, then changes in position that we detect will be small. Hence rather than using inverse trigometric func. to find θ, we use the approximation

$$\sin\theta \cong \theta .$$

Incremental Rotation.

* GLUT also provides for smooth incremental rotations.
* Suppose that we are given two orientations of the camera and we want to move smoothly from one orientation to another, then the corresponding code will be -

```
for(i=0; i<imax; i++)
  {
    glRotatef(delta_theta, dx, dy, dz);
    draw_object();
  }
```

problem of this code - calculation of rotation matrix requires evaluation of sines and cosines of three angles.

~~* So, we~~

* So, to overcome this, we compute the rotation matrix once and reuse it through code such as the following -

```
GLfloat m[16];
glRotatef (dx, dy, dz, delta_theta);
glGetFloatv (GL_MODELVIEW_MATRIX, m);
for (i=0; i<max; i++)
  {
     glMultMatrixf (m);
     draw_object ();
  }
```

we could also use small angle approximations

$\sin\theta \simeq \theta$ } if θ is very small.
$\cos\theta \simeq 1$ }

An arbitrary axis rotation matrix with an angle ψ along z axis, ϕ along y axis, and θ along x axis can be achieved using -

$$R = R_z(\psi)\, R_y(\phi)\, R_x(\theta).$$

$$= \begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

for small angles of ψ, ϕ, and θ, we get

$$R = \begin{bmatrix} 1 & -\psi & \phi & 0 \\ \psi & 1 & -\theta & 0 \\ -\phi & \theta & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

QUATERNIONS

* Quaternions are an extension of complex numbers that provide an alternative method for describing and manipulating rotations.

* There are two methods of performing rotations
  - post multiplication of CTM with rotation matrix
  - performing rotations using quaternions.

Advantage of using second method is that it requires very little computations for rotations when compared to rotation matrices.

* specifying rotations in 3D space involves two things
  - specifying the direction of rotation which is a vector quantity.
  - specifying the amount of rotation which is a scalar quantity.

* Since a quaternion has both scalar and vector representation it can be used to specify rotation efficiently.

* A point P in space can be represented in the quaternion representation as

$$p = (0, P)$$

consider a quaternion in the polar form -

$$r = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2} * v\right)$$ v has unit length.

inverse quaternion is

$$r^{-1} = \left(\cos\frac{\theta}{2}, -\sin\frac{\theta}{2} * v\right)$$

* A new point p' after rotation can be obtained using the quaternion product

$$p' = qpq^{-1}$$

* This resultant quaternion has the form $(0, p')$ where

$$p' = \cos^2\frac{\theta}{2}p + \sin^2\frac{\theta}{2}(p \cdot v)v + 2\sin\frac{\theta}{2}\cdot\cos\frac{\theta}{2}(v * p) - 2\sin\frac{\theta}{2}(v \times p) \times v$$

* Above result represented by p' is the effect of rotation of point p by θ degrees about the vector v.

* If we ~~cannot~~ count the number of operations required to perform this task using matrix multiplication w.r.t the number of operations required using quaternions, it can be realised that using quaternions results in faster computations.