Modeling Objects by Polygonal Approximations

- Define volumetric objects in terms of surfaces patches that surround the volume
- Each surface patch is approximated set of polygons
- Each polygon is specified by a set of vertices
- To pass the object through the graphics pipeline, pass the vertices of all polygons through a number of transformations using homogeneous coordinates
- All transformation are linear in homogeneous coordinates, thus a implemented as matrix multiplications

Linear and Affine Transformations (Maps)

- A map $f\,()$ is linear if it preserves linear combinations, I.e. the image of a linear combination is a linear combination of the images with the same coefficients, that is , for any scalars a and b, and any vectors p and q,

  $$f(ap+ bq) = af(\mathrm{p})+ b\,f(p)$$
- Affine maps preserve affine combinations of points, I.e. the image of an affine combination is an affine combination of the images with the same coefficients, that is, for any scalars a and b, where $a+b=1,$ and any points P and Q,

  $$f(aP+ bQ) = a\,f(\mathrm{P})+ b\,f(\mathrm{Q}).$$

Linear and Affine Maps

- Recall that a line is an affine combination of two pints (thus an image of a line is a line under affine map).
- A polygon is a convex combination of its vertices, thus under an affine map, the image of the polygon is a convex combination of the transformed vertices.
- The vertices (in homogeneous coordinates) go through the graphics pipeline
- At the rasterization stage, the interior points are generated when needed and their color is obtained by bilinear interpolation
- Affine transformations compositions of some rotations, translations & scalings in some order
- Graphics API provide functions for translation, rotation, scale, anything else you get from these by composition

Bilinear Interpolation

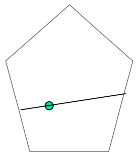
- Given the color at polygon vertices, assign color to the polygon points via bilinear interpolation:
  - An edge QR is convex combination of the two vertices Q and R, *0 £ a £ 1*,

    $$P(a) = (1-a)Q + aR$$
  - The color $C_{P(a)}$ at an edge point $P(a)$ is a linear interpolation of the color at the vertices C_Q, C_R

$$C_{P(a)} = (1-a)C_Q + aC_R$$

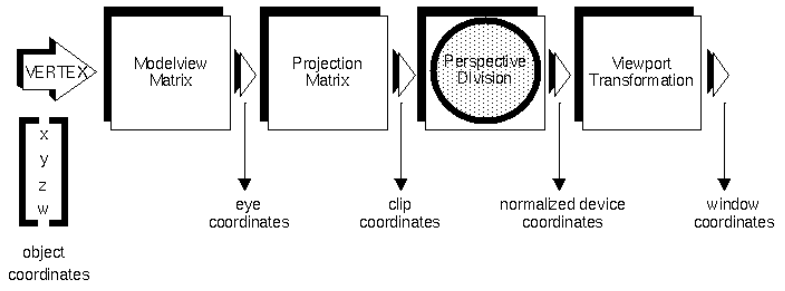
Bilinear Interpolation (cont)

- The color at an interior point is bilinear interpolation of the color at two edge points.



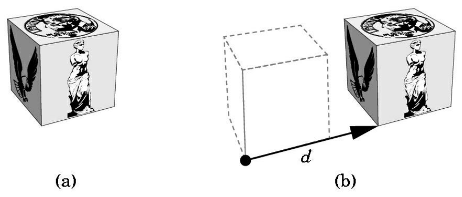
The polygon color is filled only when the polygon is displayed, during the the rasterization stage. The projection of the polygon is filled scan line by scan line. Each scan line intersects exactly 2 edges, thus color of an interior point is well-defined as bilinear interpolation of scan line intersections with the edges.

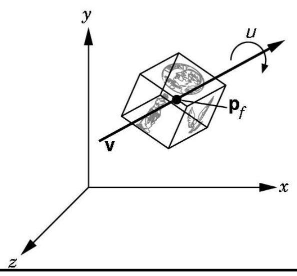
Modeling

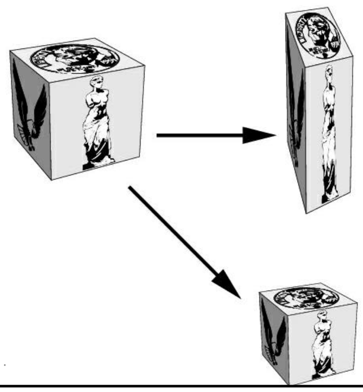


Affine Transformations

- Every affine transformation can be represented as a composition of translations, rotation, and scalings (in some order)



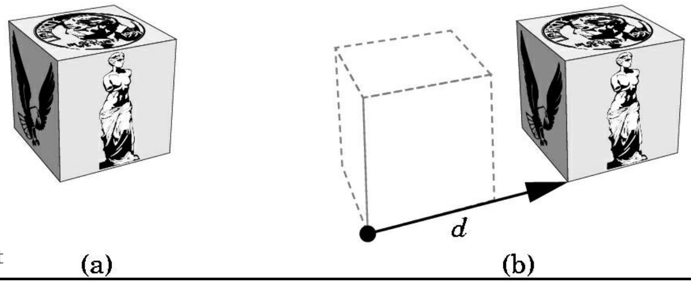




Translation

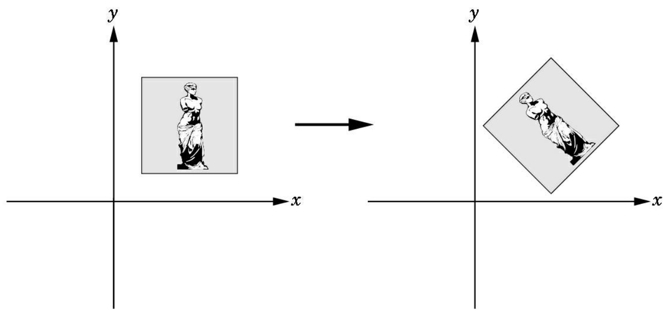
- Translation displaces points by a fixed distance in a given direction
- Only need to specify a displacement vector d
- Transformed points are given by

  $P¢ = P + d$



2D Rotations

Every 2D rotation has a fixed point



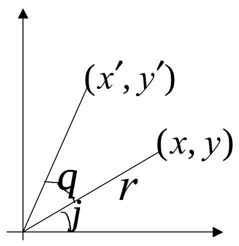
First: Matrix representation of 2D rotation around the origin

We want to find the representation of the transformation that rotates at angle q about the origin.

Since we talk about origin, we have fixed a frame.

Given a point with coordinates (x,y), what are Coordinates (x′,y′) of the transformed point?

2D Rotation on angle q around the origin



$$x = r\cos j$$
$$y = r\sin j$$
$$x' = r\cos(q + j)$$
$$y' = r\sin(q + j)$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} \cos q & -\sin q \\ \sin q & \cos q \end{bmatrix}}_{\text{matrix representing the rotation}} \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotations are represented by orthogonal matrices:

The rows (columns) are orthonormal.

3D Rotation around the z axis

- The origin is unchanged, called the fixed point of the transformation
- 2D rotation in the plane is equivalent to 3D rotation about the z axis: each point rotates in a plane perpendicular to z axis (I.e. z stays the same)
- Extend 2D rotation around the origin to 3D rotation around the z axis. Use the right-handed system. Positive rotation is counter clockwise when looking down the axis of rotation toward the origin
- Every rotation in 3D fixes and axis (I.e. is a rotation around a line)

3D rotation on angle q around the z axis

- The z axis is fixed by the rotation, the matrix representing the rotation is

$$\mathbf{R} = \begin{bmatrix} \cos q & -\sin q & 0 \\ \sin q & \cos q & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

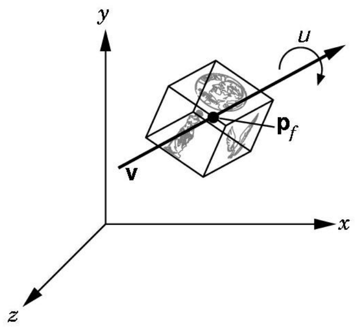
$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos q & -\sin q & 0 \\ \sin q & \cos q & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad P' = \mathbf{R}P$$

```
(      , 0.0, 0.0, 1.0)
```

Rotation in 3D around arbitrary axis

Must specify:
- rotation angle q
- rotation axis, specified by a point $P_{f,,}$ and a vector v

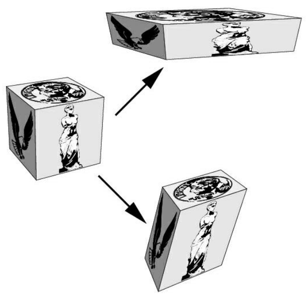
Note: openGL rotation is always around an axis through the origin



```
(     ,     ,     ,     )
```

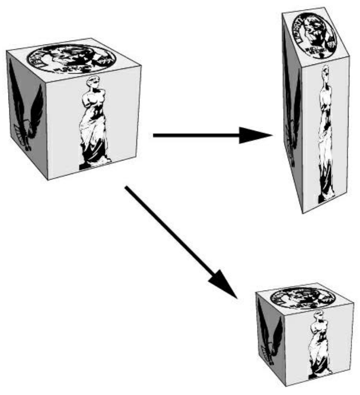
Rigid Body Transformation

•Rotation and translation are rigid-body transformations
•No combination of these transformations can alter the shape of an object



Non-rigid-body transformations

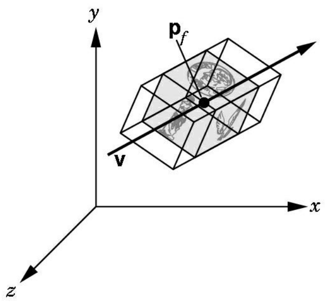
Scaling



Scaling

- Must specify:
  - fixed point P_f
  - direction to scale
  - scale factor α
- $\alpha > 1$ larger
  $0 \leq \alpha < 1$ smaller
  $-\alpha$ reflection
- Note: openGL scale more limited, allows simultaneous independent scale in each of the coordinate directions only.

The fixed point is the origin.



Scaling with fixed point the origin

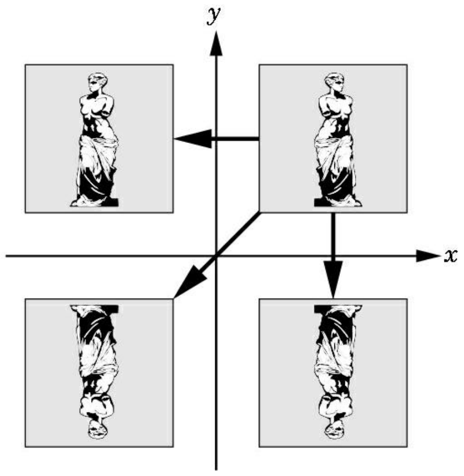
- Scaling has a fixed point
- Let the fixed point be the origin
- Independent scaling along the coordinate axes

$$x¢ = b_x x$$
$$y¢ = b_y y$$
$$z¢ = b_z z$$

- OpenGL:

```
glScalef(beta_x, beta_y, beta_z);
```

Reflections



Transformations in Homogeneous Coordinates

- Graphics systems work with the homogeneous-coordinate representation of points and vectors
- This is what OpenGL does too
- In homogeneous coordinates, an affine transformation becomes a linear transformations and as such is represented by 4x4 matrix, M.
- In homogeneous coordinates, the image of a point P, is the point MP, the image of a vector u , is the vector Mu.

Transformations in Homogeneous Coordinates

- In homogeneous coordinates, each affine transformation is represented by a 4 x 4 matrix (of a special form) and acts as matrix multiplication

$$\mathbf{v} = \mathbf{M}\mathbf{u}, \qquad \mathbf{M} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{d} \\ \mathbf{0} & 1 \end{bmatrix}, \quad \mathbf{A}_{3\times3}, \mathbf{d}_{3\times1}, \mathbf{0}_{1\times3},$$

$\mathbf{A}$, rotations and scalings

$\mathbf{d}$, translations

- In affine coordinates, not every affine transformation can be represented by a matrix, but it could be expressed in the form

$$\mathbf{v} = \mathbf{A}\mathbf{u} + \mathbf{d}$$

Translation

- Translation is an operation that displaces points by a fixed distance and direction given by a vector d
- Transformed points are given by

  $P¢ = P + d,$
- In homogeneous coordinates, this is

  $\mathbf{p}¢ = \mathbf{p} + \mathbf{d}$

  which can be represented as…

Translation by displacement **d**

Matrix form of the homogeneous coordinate equations:

$$\mathbf{p}' = \mathbf{T}\mathbf{p}, \quad \text{where}$$

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \qquad \mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}, \qquad \mathbf{d} = \begin{bmatrix} a_x \\ a_y \\ a_z \\ 0 \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & a_x \\ 0 & 1 & 0 & a_y \\ 0 & 0 & 1 & a_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{T}$ is called the translation matrix, the transformation is usually written as **T(d)** or

$$\mathbf{T}(a_x, a_y, a_z)$$

(, ,)

Translation

We can return to the original position by a displacement of $-d$, giving us the inverse:

$$\mathbf{T}^{-1}(a_x, a_y, a_z) = \mathbf{T}(-a_x, -a_y, -a_z) = \begin{bmatrix} 1 & 0 & 0 & -a_x \\ 0 & 1 & 0 & -a_y \\ 0 & 0 & 1 & -a_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translations commute, I.e. order does not matter

(, ,)

1 2 , (1+ 2) (1) (2)

Scaling with fixed point the origin

- Scaling has a fixed point
- Let the fixed point be the origin
- Independent scaling along the coordinate axes

$$x¢ = b_x x$$
$$y¢ = b_y y$$
$$z¢ = b_z z$$

Scaling with fixed point the origin

The homogeneous-coordinate equations in matrix form

$$\mathbf{p}' = \mathbf{S}\mathbf{p},$$

where

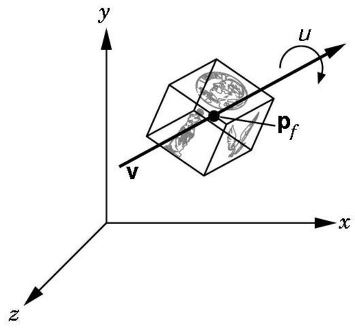
$$\mathbf{S} = \mathbf{S}(b_x, b_y, b_z) = \begin{bmatrix} b_x & 0 & 0 & 0 \\ 0 & b_y & 0 & 0 \\ 0 & 0 & b_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{S}^{-1}(b_x, b_y, b_z) = \mathbf{S}(\frac{1}{b_x}, \frac{1}{b_y}, \frac{1}{bz})$$

Two scale transformations with the same fixed point commute.

(, ,)

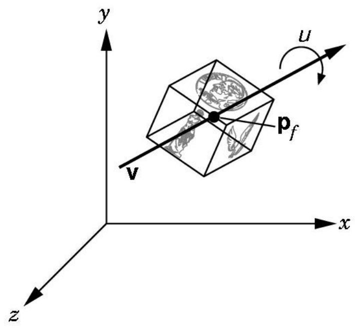
Rotation About an Arbitrary Axis **f** = (p_f,u)



Move the axis **f** so it goes through the origin, do this by moving the point, p_f, on the axis to the origin. Next apply a rotation around the new axis, u, at the origin. Finally move back the axis from origin to the original position, p_f.

$$\mathbf{T}(\mathbf{p}_f)\mathbf{R}_u(q)\mathbf{T}(-\mathbf{p}_f)$$

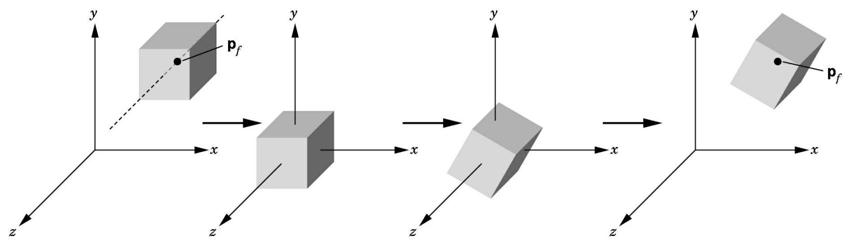
Rotation About an Arbitrary Axis **f** = (p_f,u)



$$\mathbf{T}(\mathbf{p}_f)\mathbf{R}_u(q)\mathbf{T}(-\mathbf{p}_f)$$

(, ,)
(, , ,)
(- , - , -)

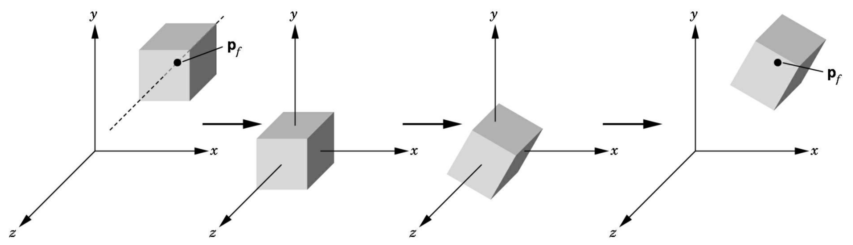
Example: Rotation around an axis **f** parallel to z axis



Move the axis **f** so it goes through the origin, achieve this by moving a point, p_f, on the axis to the origin. Next apply a rotation around the z axis. Finally move back the axis to the original position.

$$\mathbf{T}(\mathbf{p}_f)\mathbf{R}_z(q)\mathbf{T}(-\mathbf{p}_f)$$

Rotation around an axis parallel to z axis



$$\mathbf{T}(\mathbf{p}_f)\mathbf{R}_z(q)\mathbf{T}(-\mathbf{p}_f)$$

```
        (    ,    ,    )
    (        ,0,0,1)
        (-    ,-    ,-    )
```

Scaling with an arbitrary fixed point

- We know how to scale with a fixed point origin. How do we scale fixing an arbitrary point P?

  beta = [beta_x beta_y beta_z]' gives the scale factors in each of the coordiante directions.

  - **Translate so that P goes to the origin, T(-P)**
  - **Scale now with respect the origin**
  - **Translate back to P, T(P)**
  - **The composition represented by the matrix product**

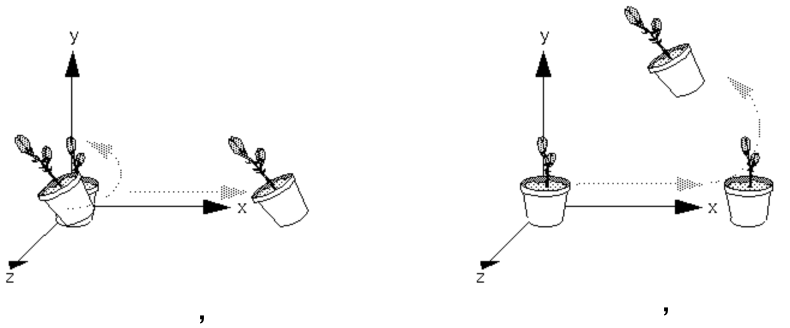
    T(P)S(beta)T(-P)

  - **OpenGL**

```
glTranslatef(p_x, p_y, p_z);
glScalef(beta_x, beta_y, beta_z);
glTranslatef(-p_x, -p_y, -p_z);
```

Composing Transformations

- Be careful when composing (concatenating) transformations: matrix multiplication is not commutative, and transformations composition is not commutative



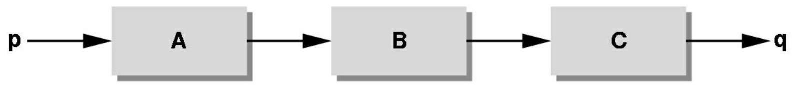
Concatenation of Transformations

- We can multiply together sequences of transformations – concatenating
- Works well with pipeline architecture
- e.g., three successive transformations on a point $\mathbf{p}$ creates a new point $\mathbf{q}$

  $\mathbf{q} = \mathbf{CBAp}$

  - **In code:**

    C
    B
    A
    Draw p



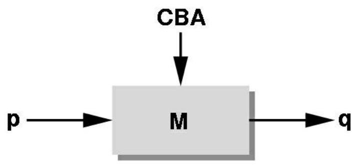
Concatenation of Transformations.

- If we have a lot of points to transform, then we can calculate

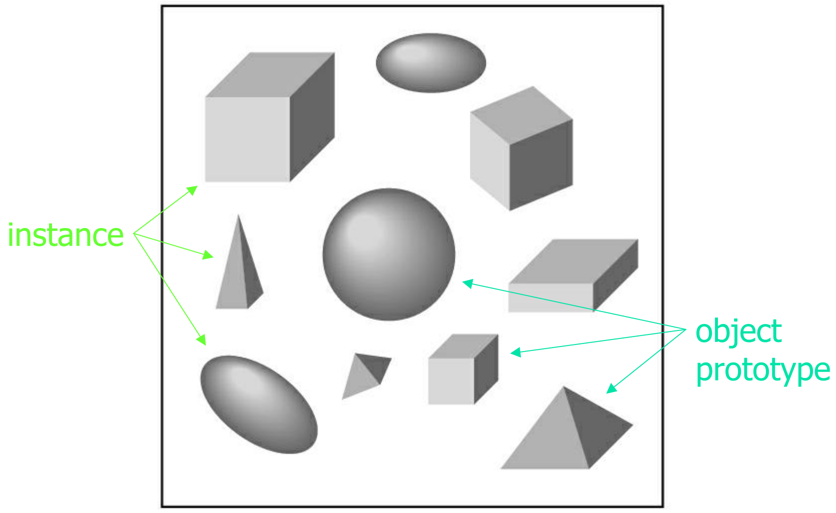
  $\mathbf{M} = \mathbf{CBA}$

  and then we use this matrix on each point

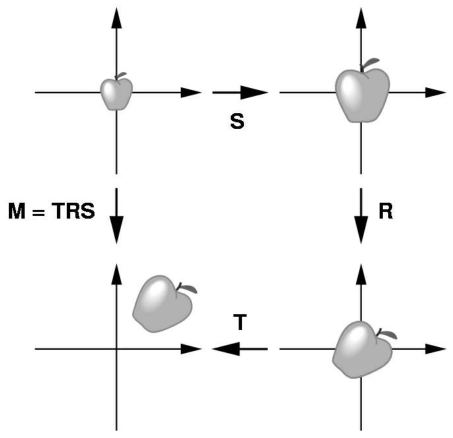
  $\mathbf{q} = \mathbf{Mp}$



Instance Transformation



Instance Transformation



//

( )

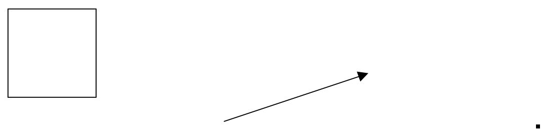
( )

( )

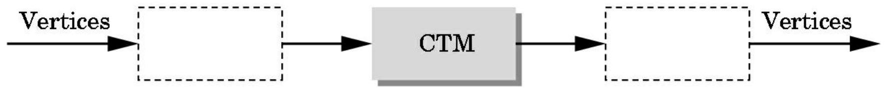
()

Instance Transformations

- Specify the affine transformation that will move the square so that its lower left corner will be at P, the vertical side will be parallel to u, and the size will be half the original size



Current Transformation Matrix



- Current Transformation Matrix (CTM) – defines the state of the graphics system. All drawings, (vertices) defined subsequently undergo that transformation.
- Changing the CTM, alters the state of the system.
- 4x4 matrix that can be altered by a set of functions provided by the graphics package
- Common to most systems. Part of the pipeline
- If $\mathbf{p}$ is a vertex, the pipeline produces $\mathbf{Cp}$

Current Transformation Matrix

Let **C** denote the CTM. It is set to the 4x4 identity matrix, initialliy
CTM=I, glLoadIdentity()
CTM=M (resets it), glLoadMatrixf(pM)
CTM=CTM*M (post multiplies CTM by M), glMultMatrixf(pM)

Application of the gl functions, post-multiplies CTM

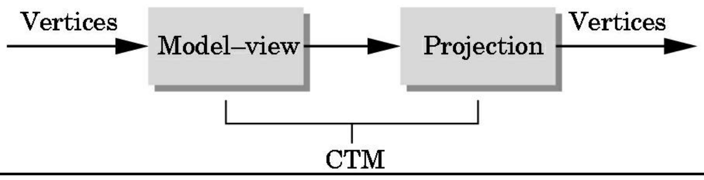
```
              ()     //1.
             (  )    //2.      *
             (  )    // .      *  *
       (          )
              ( )
     ()
```

The point will be transformed according to CTM at 3.

Current Transformation Matrix

In OpenGL the CTM is the product of model-view matrix (GL_MODELVIEW) and projection matrix (GL_PROJECTION).
The model-view matrix is product of modeling transformations (affine transformations on the objects) and viewing transformations (positioning and orienting camera).
The projection matrix is responsible for 3D to 2D.

The CTM is the product of these matrices!



Order of Transformations

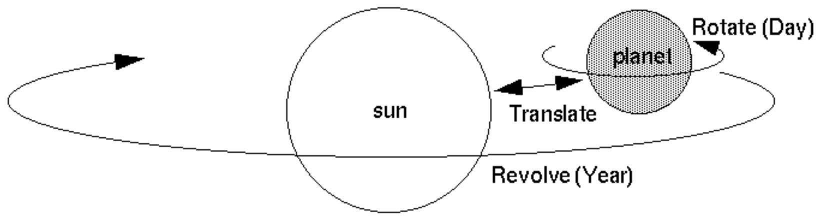
- Transformation specified most recently is the one applied first to the primitive

```
 // the transformed polygon
 glMatrixModel(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);
glBegin(GL_POLYGON);  // sample polygon
  ...
glEnd();
```

World and Local Coordinate Systems

- An object moving relative to another moving object has a complicated motion:
  - A waving hand on a moving arm on a moving body
  - A rotating moon orbiting a planet orbiting a star
- Directly expressing such motions with transformations is difficult
- More indirect approach works better

Example: planetary system



Example: solar system

```
#define RADMOON 0.3
#define RADSUN  1.0
#define RADORBIT 3.0

static int day, year;

void  myinit();
void  display();
void  idle();

void idle()
{
  day = (day+5)%360;
   year=(year+1)%360;
   glutPostRedisplay();
}
```

```
void init()
{
   glClearColor(0,0,0,0);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-2.25*RADORBIT,
            2.25*RADORBIT, …);
   glMatrixMode(GL_MODELVIEW);

   glEnable(GL_DEPTH_TEST);
}
```

Example: solar system

```
int main(int argc, char **argv)
{
   glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
  glutInitWindowSize(500,500);
   glutInitWindowPosition(0,0);
   glutCreateWindow("Solar System");
   glutDisplayFunc(display);
  glutIdleFunc(idle);
   myinit();
   glutMainLoop();
   return 0;
}
```

```
          ()

        (                                        )
             ()

          (  , 1, 0, 0)         //

         (1.0, 0.0, 0.0)         //
              (      , 20, 1 )   //
         (     ,  0, 0, 1)       //
            (        , 0, 0)     //
         (   , 0,0,1)            //
         (1.0, 1.0, 0.0)         //
              (        , 10,  )  //

            ()
                ()
```