General Transformation Commands

- **glMatrixMode()**
  - *Modelview*
  - *Projection*
  - *Texture*
  - *Which matrix will be modified*
    - *Subsequent transformation commands affect the specified matrix.*
- *void **glLoadIdentity**(void);*
  - *Sets the currently modifiable matrix to the 4 × 4 identity matrix.*
  - Usually done when you first switch matrix mode

Spring 2013

Object Coordinate System

- Used to place objects in scene
  - Draw at origin of WCS
  - Scale and Rotate
  - Translate to final position
- glMatrixMode(GL_MODELVIEW)
  - glScale[fd](x, y, z)
  - glRotate[fd](angle, x, y, z)
  - glTranslate[fd](x, y, z)
  - gluLookAt(eyex, eyey, eyez, x, y, z, upx, upy, upz)

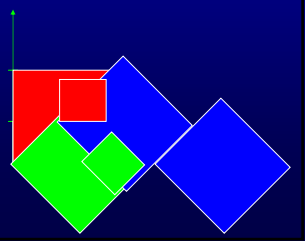
Spring 2013

Transformations

- In OpenGL, transformation are performed in the opposite order they are called

4 glTranslatef(1.0, 1.0, 0.0);
3 glRotatef(45.0, 0.0, 0.0, 1.0);
2 glScalef(2.0, 2.0, 0.0);
1 DrawSquare(0.0, 0.0, 1.0);

4 glScalef(2.0, 2.0, 0.0);
3 glRotatef(45.0, 0.0, 0.0, 1.0);
2 glTranslatef(1.0, 1.0, 0.0);
1 DrawSquare(0.0, 0.0, 1.0);

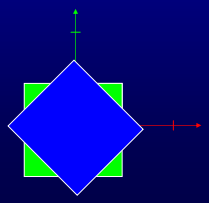


Spring 2013

Rotation and Scaling

- Rotation and Scaling is done about origin
  - You always get what you expect
  - Correct on all parts of model

4 glRotatef(45.0, 0.0, 0.0, 1.0);
3 glScalef(2.0, 2.0, 0.0);
2 glTranslatef(-0.5, -0.5, 0.0);
1 DrawSquare(0.0, 0.0, 1.0);



Spring 2013

Load and Mult Matrices

- *void **glLoadMatrix**{fd}(const TYPE *m);*
  - *Sets the sixteen values of the current matrix to those specified by m.*
- *void **glMultMatrix**{fd}(const TYPE *m);*
  - *Multiplies the matrix specified by the sixteen values pointed to by m by the current matrix and stores the result as the current matrix.*

Spring 2013

- OpenGL uses column instead of row vectors
- Let **C** be the current matrix and call **glMultMatrix*(M)**. After multiplication, the final matrix is always **CM**.
- Matrices are defined like this (use float m[16]);

$$M = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

Spring 2013

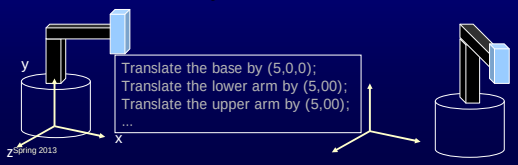
Stack Operations

- glPushMatrix
- glPopMatrix

Transformations

- Two ways to specify transformations
  - (1) Each part of the object is transformed independently relative to the origin

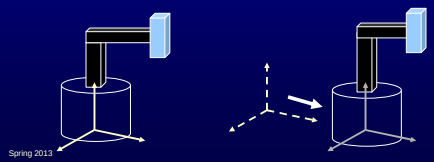
    Not the best way!



Relative Transformation

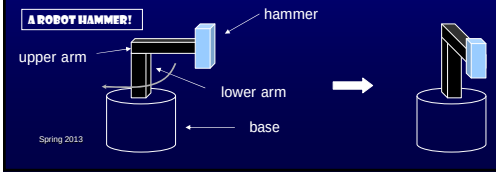
A better (and easier) way:

(2) Relative transformation: Specify the transformation for each object relative to its parent



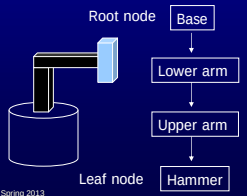
Object Dependency

- A graphical scene often consists of many small objects
- The attributes of an object (positions, orientations) can depend on others



Hierarchical Representation - Scene Graph

- We can describe the object dependency using a tree structure

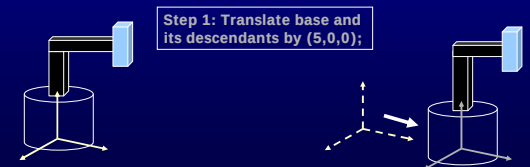


The position and orientation of an object can be affected by its parent, grand-parent, grand-grand-parent ... nodes

This hierarchical representation is referred to as Scene Graph

Relative Transformation

Relative transformation: Specify the transformation for each object relative to its parent



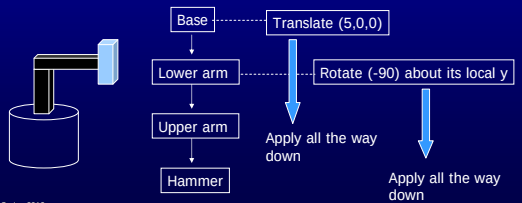
Relative Transformation (2)

Step 2: Rotate the lower arm and all its descendants relative to its local y axis by -90 degree

Spring 2013

Relative Transformation (3)

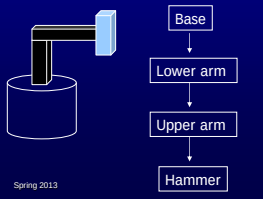
- Represent relative transformations using scene graph



Spring 2013

Do it in OpenGL

- Translate base and all its descendants by (5,0,0)
- Rotate the lower arm and its descendants by -90 degree about the locally

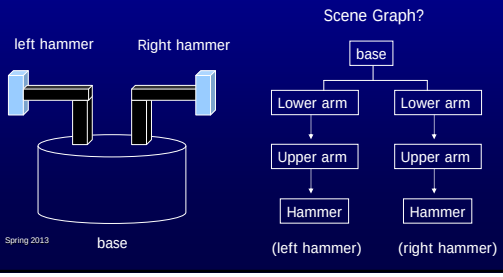



```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

... // setup your camera

glTranslatef(5,0,0);

Draw_base();

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

Draw_lower _arm();
Draw_upper_arm();
Draw_hammer();
```

Spring 2013

A more complicated example

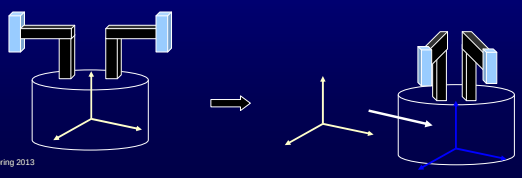
- How about this model?



Spring 2013

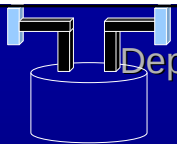
Do this …

- Base and everything – translate (5,0,0)
- Left hammer – rotate 75 degree about the local y
- Right hammer – rotate -75 degree about the local y

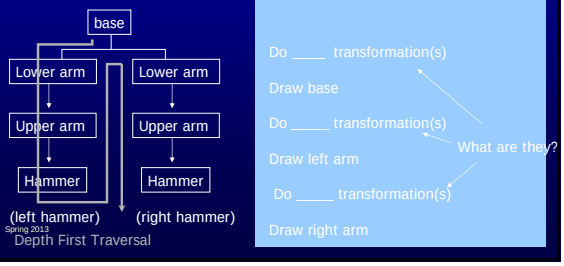


Spring 2013

Depth-first traversal



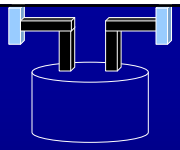
- Program this transformation by depth-first traversal

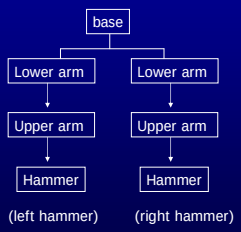


Spring 2013

Depth First Traversal

How about this?





Translate(5,0,0)

Draw base

Rotate(75, 0, 1, 0)

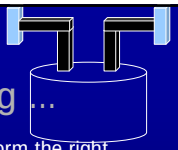
Draw left hammer

What's wrong?!

Rotate(-75, 0, 1, 0)

Draw right hammer

Something is wrong …



- What's wrong? – We want to transform the right hammer relative to the base, not to the left hammer

How about this?

Do **Translate(5,0,0)**

Draw base

Do **Rotate(75, 0, 1, 0)**

Draw left hammer

What's wrong?!

Do **Rotate(-75, 0, 1, 0)**

Draw right hammer

We should **undo the left hammer transformation** before we transform the right hammer

Need to undo this first

Undo the previous transformation(s)

- Need to save the modelview matrix right after we draw base

Initial modelView M

Translate(5,0,0) -> M = M x T

Draw base

Rotate(75, 0, 1, 0)

Draw left hammer

Rotate(-75, 0, 1, 0)

Draw right hammer

Undo the previous transformation means we want to restore the Modelview Matrix M to what it was here

i.e., save M right here

…

And then restore the saved Modelview Matrix

OpenGL Matrix Stack

- We can use OpenGL Matrix Stack to perform matrix save and restore

Initial modelView M

Do **Translate(5,0,0) -> M = M x T**

Draw base

Do **Rotate(75, 0, 1, 0)**

Draw left hammer

Do **Rotate(-75, 0, 1, 0)**

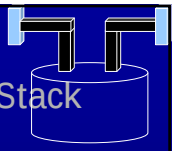
Draw right hammer

* Store the current modelview matrix
- Make a copy of the current matrix and **push** into OpenGL Matrix Stack
call glPushMatrix()

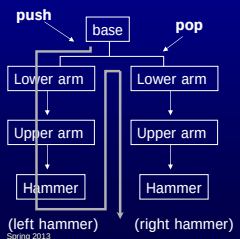
- continue to modify the current matrix

* Restore the saved Matrix
- Pop the top of the Matrix and copy it back to the current Modelview Matrix:
Call glPopMatrix()

Push and Pop Matrix Stack



- A simple OpenGL routine:




```
glTranslate(5,0,0)
Draw_base();
glPushMatrix();

glRotate(75, 0,1,0);
Draw_left_hammer();

glPopMatrix();
glRotate(-75, 0,1,0);
Draw_right_hammer();
```

Spring 2013

Depth First Traversal

Push and Pop Matrix Stack

- Nested push and pop operations

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
... // Transform using M1;
... // Transform using M2;
glPushMatrix();
... // Transform using M3
glPushMatrix();
.. // Transform using M4
glPopMatrix();
...// Transform using M5
...
glPopMatrix();
```

Modelview matrix (M)

M = I

M = M1

M = M1 x M2

M = M1 x M2 x M3

M = M1 x M2 x M3 x M4

M = M1 x M2 x M3

M = M1 x M2 x M3 x M5

M = M1 x M2

Stack

M1xM2

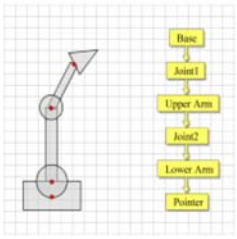
M1xM2xM3

M1 x M2

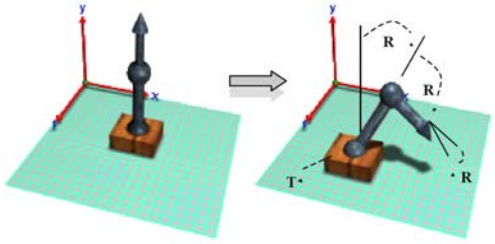
M1 x M2

Hierarchical Transformations

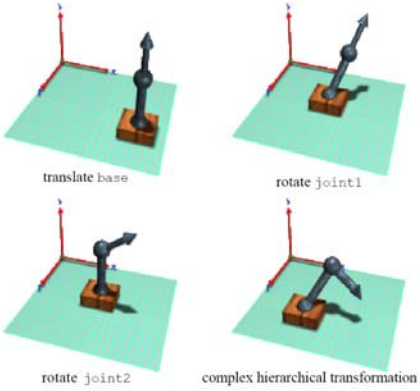
- For geometries with an implicit *hierarchy* we wish to associate local frames with sub-objects in the assembly.
- *Parent-child frames* are related via a transformation.
- Transformation linkage is described by a *tree*:
- Each node has its own *local co-ordinate system*.



Hierarchical Transformations

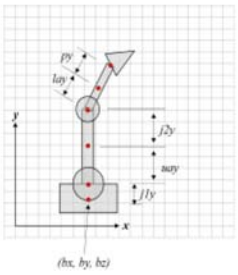


Hierarchical transformation allow independent control over sub-parts of an assembly



OpenGL® Implementation

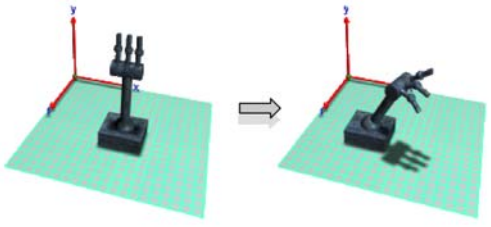
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(bx, by, bz);
  create_base();
glTranslatef(0, j1y, 0);
glRotatef(joint1_orientation);
  create_joint1();
glTranslatef(0, uay, 0);
  create_upperarm();
glTranslatef(0, j2y);
glRotatef(joint2_orientation);
  create_joint2();
glTranslatef(0, lay, 0);
  create_lowerarm();
glTranslatef(0, py, 0);
glRotatef(pointer_orientation);
  create_pointer();
```



Hierarchical Transformations

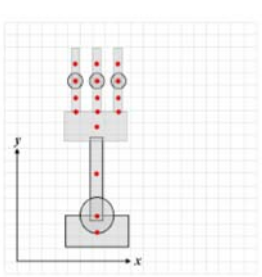
- The previous example had simple *one-to-one* parent-child linkages.
- In general there may be many *child frames* derived from a single parent frame.
- we need some mechanism to remember the parent frame and return to it when creating new children.
- OpenGL provide a matrix stack for just this purpose:
  - `glPushMatrix()` saves the *CTM*
  - `glPopMatrix()` returns to the last saved *CTM*

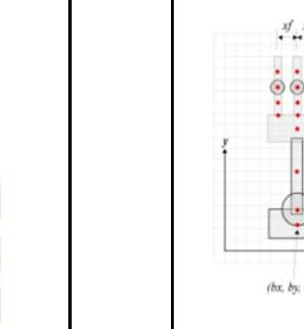
Hierarchical Transformations



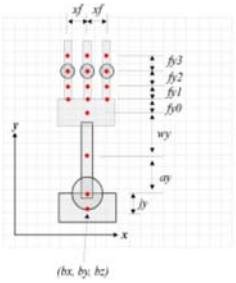
Each finger is a child of the parent (wrist)
⇒ independent control over the orientation of the fingers relative to the wrist

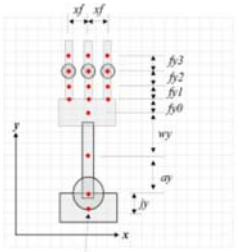
Hierarchical Transformations



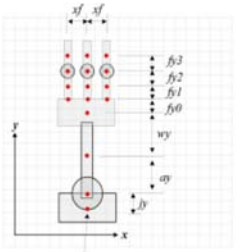



```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(bx, by, bz);
  create_base();
glTranslatef(0, jy, 0);
glRotatef(joint1_orientation);
  create_joint1();
glTranslatef(0, ay, 0);
  create_upperarm();
glTranslatef(0, wy);
glRotatef(wrist_orientation);
  create wrist();

  glTranslatef(-xf, fy0, 0);
  glRotatef(lowerfinger1_orientation);
  glTranslatef(0, fy1, 0);
    create_lowerfinger1();
  glTranslatef(0, fy2, 0);
  glRotatef(upperfinger1_orientation);
    create_fingerjoint1();
  glTranslatef(0, fy3, 0);
    create_upperfinger1();
```






```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(bx, by, bz);
  create_base();
glTranslatef(0, jy, 0);
glRotatef(joint1_orientation);
  create_joint1();
glTranslatef(0, ay, 0);
  create_upperarm();
glTranslatef(0, wy);
glRotatef(wrist_orientation);
  create wrist();

  glTranslatef(-xf, fy0, 0);
  glRotatef(lowerfinger1_orientation);
  glTranslatef(0, fy1, 0);
    create_lowerfinger1();
  glTranslatef(0, fy2, 0);
  glRotatef(upperfinger1_orientation);
    create_fingerjoint1();
  glTranslatef(0, fy3, 0);
    create_upperfinger1();

  glTranslatef (0, -fy3, 0) ;
  glRotatef (-upperfinger1_orientation) ;
  glTranslatef (0, -fy2, 0) ;
  glTranslatef (0, -fy1, 0) ;
  glRotatef (-lowerfinger1_orientation) ;
  glTranslatef (xf, -fy0, 0) ;

  // do finger 2
```



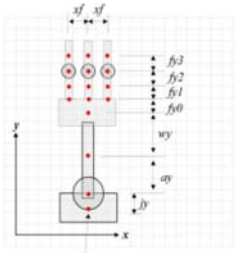

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(bx, by, bz);
  create_base();
glTranslatef(0, jy, 0);
glRotatef(joint1_orientation);
  create_joint1();
glTranslatef(0, ay, 0);
  create_upperarm();
glTranslatef(0, wy);
glRotatef(wrist_orientation);
  create wrist();
```

Save the matrix state

```
  glTranslatef(-xf, fy0, 0);
  glRotatef(lowerfinger1_orientation);
  glTranslatef(0, fy1, 0);
    create_lowerfinger1();
  glTranslatef(0, fy2, 0);
  glRotatef(upperfinger1_orientation);
    create_fingerjoint1();
  glTranslatef(0, fy3, 0);
    create_upperfinger1();

  glTranslatef (0, -fy3, 0) ;
  glRotatef (-upperfinger1_orientation) ;
  glTranslatef (0, -fy2, 0) ;
```

Restore the matrix state

```
  glTranslatef (xf, -fy0, 0) ;

  // do finger 2
```




```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(bx, by, bz);
  create_base();
glTranslatef(0, jy, 0);
glRotatef(joint1_orientation);
  create_joint1();
glTranslatef(0, ay, 0);
  create_upperarm();
glTranslatef(0, wy);
glRotatef(wrist_orientation);
  create_wrist();
glPushMatrix(); // save frame
  glTranslatef(-xf, fy0, 0);
  glRotatef(lowerfinger1_orientation);
  glTranslatef(0, fy1, 0);
    create_lowerfinger1();
  glTranslatef(0, fy2, 0);
  glRotatef(upperfinger1_orientation);
    create_fingerjoint1();
  glTranslatef(0, fy3, 0);
    create_upperfinger1();
glPopMatrix(); // restore frame
glPushMatrix();
  // do finger 2...
glPopMatrix();
glPushMatrix();
  // do finger 3...
glPopMatrix();
```

Finger1