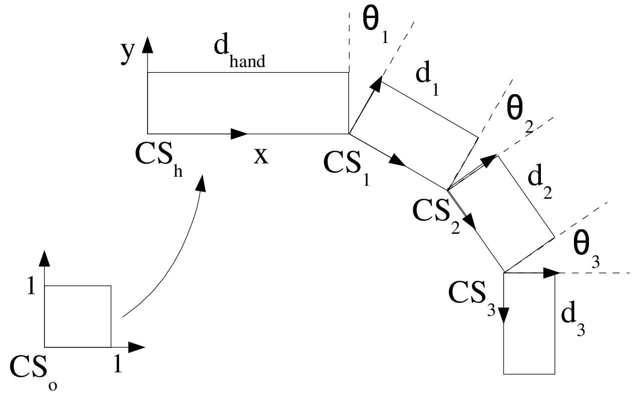
Transformation Hierarchies

Consider building the following model of a hand with one finger:



Note that each object part is constructed by transforming a single object part and is defined in local object part coordinates. Each object part is then positioned by a transformation that is performed relative to its parents coordinate frame.

This can be constructed using a transformation hierarchy. In the following object tree, circles represent transformations which position the object parts relative to their parents and squares represent geometries which define the object parts in their own local coordinate frames.

Transformations
M_i = initial transformation matrix (top of stack)
T_{hand} = translate to Cs_h origin.
T_{f1} = translate(d_{hand}, 0) rotate($-\theta_1$, z)
T_{f2} = translate(d_1, 0) rotate($-\theta_2$, z)
T_{f3} = translate(d_2, 0) rotate($-\theta_3$, z)

Local Object Geometries
draw hand = scale(d_{hand}, 1)
draw f_1 = scale(d_1, 1)
draw f_2 = scale(d_2, 1)
draw f_3 = scale(d_3, 1)

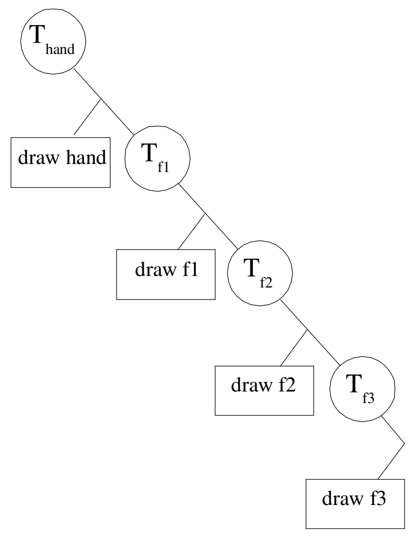


Fig. 3.1 Finger Hierarchy

Using these transformations, we can draw the object using the following pseudocode:
$M = M_i \ T_{hand}$
store M on stack and draw hand, restoring M after
$M = M_i \ T_{f1} = M_i \ T_{hand} \ T_{f1}$

store M on stack and draw f_1, restoring M after
$M = M_i \; T_{f2} = M_i \; T_{hand} \, T_{f1} \, T_{f2}$
store M on stack and draw f_2, restoring M after
$M = M_i \; T_{f3} = M_i \; T_{hand} \, T_{f1} \, T_{f2} \; T_{f3}$
store M on stack and draw f_3, restoring M after

Note: a stack can be implemented as a LIFO stack or by recursion. In our example, any time an object part needs to be drawn, the current matrix, M, is copied first. In general, for any branch in the tree hierarchy, a copy of the current matrix should be saved before the branch is made.

OpenGL maintains a matrix stack with the top of the stack being the current transformation matrix. Each OpenGL transformation function manipulates the current matrix and if a transformation needs to be reused, it can be copied and pushed on to the top of the stack using the command:

glPushMatrix(); // "remember where you are"

The top of the matrix stack can also be removed using the command:

glPopMatrix(); // "go back to where you were"

For our finger example we could write functions that define each object part in its own local coordinate system:

```
void drawHand() {
          glPushMatrix();
          glScalef(d_hand, d_hand/2, 1);
          drawSquare(1);
          glPopMatrix();
}

void drawF1() {
          glPushMatrix();
          glScalef(3*d_hand/4, d_hand/2, 1);
          drawSquare(1);
          glPopMatrix();
}

void drawF2() {
          glPushMatrix();
          glScalef(3*d_hand/4, 2*d_hand/5, 1);
          drawSquare(1);
          glPopMatrix();
}

void drawF3() {
          glPushMatrix();
          glScalef(d_hand/2, d_hand/3, 1);
          drawSquare(1);
          glPopMatrix();
}
```

We can then again utilize the OpenGL matrix stack to draw each part relative to its parent as described in our hierarchy tree:

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

glTranslatef(0,100,0);
drawHand();
glPushMatrix();//save matrix on stack in case we want to draw second finger
glTranslatef(d_hand, 0, 0);
glRotatef(angle, 0,0,1);
drawF1();
glTranslatef(3*d_hand/4, 0, 0);
glRotatef(angle, 0,0,1);
drawF2();
glTranslatef(3*d_hand/4, 0, 0);
glRotatef(angle, 0,0,1);
drawF3();
glPopMatrix();
glTranslatef(d_hand, 0, 0);
glRotatef(-10, 0,0,1);
drawF1();
glTranslatef(3*d_hand/4, 0, 0);
glRotatef(-10, 0,0,1);
drawF2();
glTranslatef(3*d_hand/4, 0, 0);
glRotatef(-10, 0,0,1);
drawF3();
```

A more general solution could implement a tree data structure with each node containing a shape descriptor and a transform from local object coordinates to that of its parent as shown in Fig. 3.2:

If, for example we wanted to render Obj 3b in its correct world coordinates position, we could specify it as follows:

$$\bar{P}_{world} = M_o M_{1b} M_{2b} M_{3b} \bar{P}_{Obj3b}$$

In order to render the entire object hierarchy, we can define a recursive function which renders a subtree at node j, with a transform, **M**, that maps the parent of node j to the world:

```
render(node j, matrix M) {
•  compute local object transform, $M_j$
•  compute transform $M_{world}$ , for this
   node: $M_{world} = M*M_j$
•  render object part j: $Q=M_{world}*P_{Obj\ j}$
•  for each child node i, recursively
   call render(node i, matrix $M_{world}$)
}
```

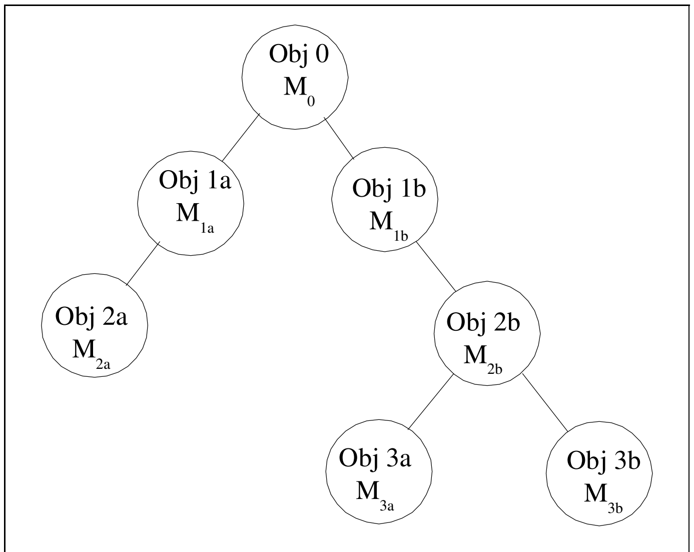


Fig. 3.2 Object Hierarchy