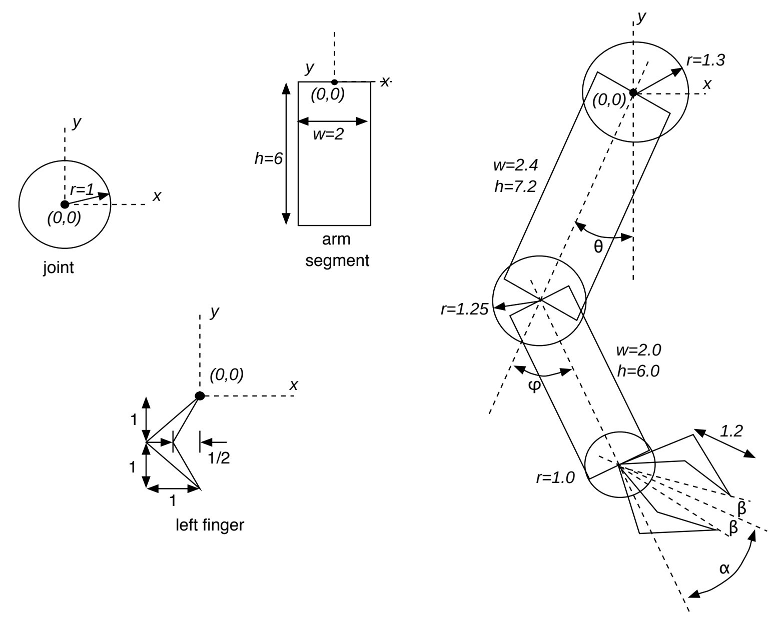
Hierarchical Modeling in OpenGL

CS 442/542

September 4, 2012

1 Hierarchical modeling and instancing

Many objects we wish to model form a natural *hierarchy* of connected components. "Movement" of one component causes another set of "attached" components to move in like fashion. This hierarchy can be represented by a *tree* (more generally a *graph*) where each child tree inherits some set of transformations (which define the movement) specified by the parent. For example, consider the robot arm below. A rotation of θ about the shoulder joint causes the entire arm to rotate, but a rotation of ϕ about the elbow joint only causes the forearm to move. The forearm inherits both rotations, whereas the biceps inherits only the shoulder rotation.

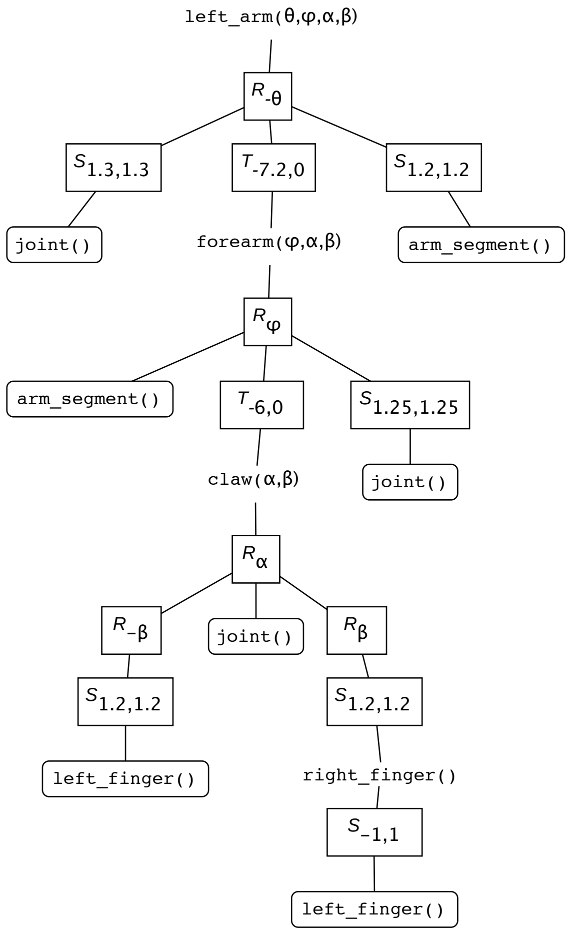


Here we construct our robot arm out of three primitive constructs: a *joint*, an *arm segment*, and a *left finger*. We have conveniently modeled each primitive in its own local coordinate system and have placed the origin at the appropriate "pivot point." We use our "joint template" to create the shoulder, elbow, and wrist joints. This process of creating multiple shapes from one master shape is called *instancing* and each resulting shape is called an *instance.*

2 Scene graph

The following tree illustrates how to create our robot's left arm which is parameterized by three angles:

- θ : CW rotation about shoulder;
- ϕ : CCW rotation about elbow;
- α : CCW rotation about wrist;
- β : half-angle of "claw opening."



Our transformations are composed of rotations (R_θ), scales ($S_{a,b}$), and translations ($T_{x,y}$). Note that the robot's right finger is made from mirroring the left finger about the y-axis (i.e., using a scale of -1 in x). We could use a similar trick to create the robot's right arm.

The above tree is an example of what is termed a *scene graph.* Other 3D programming libraries are based on a scene graph model (e.g., Java3D).

3 Implementation details

3.1 Matrices

Modern OpenGL no longer provides "built-in" matrices and leaves this up to the program. The examples below use the following "home-brewed" routines to manipulate matrices stored in the application memory space:

`matrixIdentity(GLfloat M[4*4])` Set `M` to 4×4 identity matrix.

`matrixScale(GLfloat M[4*4], GLfloat sx, GLfloat sy, GLfloat sz)`
Concatenate a scale S matrix onto M (*i.e.*, $M \leftarrow M \cdot S$).

`matrixTranslate(GLfloat M[4*4], GLfloat dx, GLfloat dy, GLfloat dz)`
Concatenate a translation matrix onto `M`.

`matrixRotate(GLfloat M[4*4], GLfloat theta, GLfloat x, GLfloat y, GLfloat z)`
Concatenate a CCW rotation matrix about the vector (x, y, z) onto `M` (`theta` is measured in degrees).

`matrixMultiply(GLfloat M[4*4], const GLfloat A[4*4], const GLfloat B[4*4]);`
$M \leftarrow A \cdot B$.

`matrixPush(GLfloat M[4*4]);` Push matrix `M` onto internal stack.

`matrixPop(GLfloat M[4*4]);` Pop matrix off internal stack and store in `M`.

We assume that the following client matrices are used

```
GLfloat ModelView[4*4];   // composite modeling and view matrix
GLfloat Projection[4*4];  // projection matrix
```

Before rendering any geometry, we construct the composite model-view-projection matrix and load it into a uniform variable for our vertex shader to use.

```
GLint ModelViewProjectionUniform;  // link address in shader program
...
GLfloat ModelViewProjection[4*4];
matrixMultiply(ModelViewProjection, Projection, ModelView);
glUniformMatrix4fv(ModelViewProjectionUniform, 1, GL_FALSE,
                   ModelViewProjection);
```

3.2 Articulated arm

Here is an OpenGL subroutine that renders the above tree. Note the use of `matrixPush` and `matrixPop` to save and restore the current model-view matrix. The `joint()` and `arm_segment()` routines implement two of our primitives. The `forearm()` routine renders the remaining portion of the tree.

```
void left_arm(GLfloat theta, GLfloat phi, GLfloat alpha, GLfloat beta) {
  matrixPush(ModelView);
  matrixRotate(ModelView, -theta, 0, 0, 1);

  matrixPush(ModelView);
  matrixScale(ModelView, 1.3, 1.3, 1);
  joint();
  matrixPop(ModelView);

  matrixPush(ModelView);
  matrixScale(ModelView, 1.2, 1.2, 1);
  arm_segment();
  matrixPop(ModelView);

  matrixTranslate(ModelView,0, -7.2, 0);
  forearm(phi, alpha, beta);

  matrixPop(ModelView);
}
```

3.3 Order is important – last shall be first

Note that the last transformation specified (via `matrixScale, matrixRotate, matrixTranslate`) has its effect first – this is critical to representing our hierarchy. Mathematically, we perform a *post concatenation* of the transformation T onto the current modeling matrix M yielding MT; Therefore, T has its effect before $M : (MT)x = M((T)x)$.

The general form for rendering a node in the hierarchy often looks like the following:

```
void drawComponent(void) {
  matrixPush(ModelView);   /* save parent's modeling matrix */
  apply global transformations we want children to inherit;
  for (each child component)
    draw child component;
  apply local transformations we do *not* want children to inherit;
  draw other primitives used to model this object;
  matrixPop(ModelView);    /* restore parent's modeling matrix */
}
```