1 Robots! (I don't know why the scientists keep making them.)

We want to make a hierarchical model, like a robot with arms with hands with fingers.

Each part of the robot will be made using the Sphere class: elongated or squashed sphere, cylinder, or box. So we create a Part class that contains the appropriate sphere and transformations sphere2part and part2sphere. These take the place of model2object and object2model and are a product of a rotation, a scale to change the dimensions, and a translation matrix to move the origin (like we did with cube).

A part can move if it is connected to a *joint*. Rotation is the most common tranformation, but a sliding joint is possible in a robot. We will model all joints as ball-in-socket, like your shoulder. Your arm has three rotational degrees of freedom: up, forward, or rotate on its own axis. This is true of your hip, waist, neck, wrist, and ankle. It's not true of your elbow or knee. Even when it is true, there are limits. But to start, we will have a very flexible robot. So Part has part2joint and joint2part. These take the place of object2rotated and rotated2object and are rotation matrices.

A Part is not placed in the world. It is placed in the frame of its parent, the part of the robot to which it is attached. So for example a finger is placed on the hand, and different fingers have different locations on the hand. When the hand moves, the fingers move with it. The hand in turn, is placed on the arm. The arms, head, and legs are placed on the body. The body is placed in the world. So Shape will have joint2parent and parent2joint to represent where the part is in its parent's world. These take the place of rotated2world and world2rotated and are transformation matrices.

Each part has a matrix parent2world (and world2parent) that tells it where its *parent* is in the world. A point p on a part's sphere in the world at

- parent2world * joint2parent * part2joint * sphere2part * p.

(Actually, you'll have to use .times() because this is Javascript.) The body part has no parent, or in other words, its parent is the world. So body.parent2world is really "world2world" or the identity.

If we place a head on top of the body, what is head.parent2world? The head is not deformed by body.sphere2part. That's for shaping the body. However, it does move with the body. So

- head.parent2world = body.parent2world [the identity] * body.joint2parent * body.part2joint
- head.sphere2world = head.parent2world * head.joint2parent * head.part2joint
  * head.sphere2part
- head.parent2world (body2world) makes the head move with the body.
- head.joint2parent places the head on top of the body.

- head.part2joint nods.
- head.sphere2part scales the head and moves its origin to its neck connection.

Does this make sense? A point p on the head's sphere is moved by head.sphere2part, rotated by head.part2joint, and then put in the body's frame by head.joint2parent. To put it into the world's frame, we apply the body's rotation body.part2joint and the body's placement in the world body.joint2parent.

If something were attached to the head, like the jaw:

- jaw.parent2world = head.parent2world * head.joint2parent * head.part2joint
- jaw.sphere2world = jaw.parent2world * jaw.joint2parent * jaw.part2joint * jaw.sphere2part

Each Part has a list of children: the parts which are attached to it that have it as their parent. So the body has head, leftArm, rightArm, leftLeg, and rightLeg in its children list (assuming that we don't have upperBody and lowerBody–no waist.) rightArm has forearm. forearm has hand. hand has thumb, index, etc. index has middleFinger. middleFinger has fingerTip.

This is called a *hierarchical model*.

2 Setting every part's parent2world

A Part has a setParent2World method

- setParent2World (parent2world, world2parent)

which sets its local variables parent2world and world2parent from the parameters and sets

- sphere2world = parent2world * joint2parent * part2joint * sphere2part
- world2sphere = INVERSE OF THAT

and calls each element of children recursively with

- setParent2World(parent2world*joint2parent*part2joint, INVERSE OF THAT)

3 Rendering parts

A Part has a render method. It sets the uniform variables model2world and world2modelT (T means transpose) (why??) and calls sphere.render(). THEN it recursively calls render for each of the children.

4 Dragging parts

We also want to be able to rotate a part by dragging it. That should change its part2joint using the p,o,v,f,b,u technique from prog06. But first we need to figure out which part (if any) is clicked on, and at what point p.

We first need Sphere methods

- /*double*/ planeHit (/*int*/ face, /*PV*/ q, /*PV*/ u) // return s such that q + u s lies in place of face
- /*boolean*/ contains (/*int/ face, /*PV*/ p) // return true if p in plane also lies in face
- /*double*/ closestHit(/*PV*/ q, /*PV*/ u) // return minimum s for which q+u s hits a face or Infinity.

which figure out if the ray q + s u hits any face, and if so the closest hit. ("Infinity" is the Javascript infinity.)

Then we need the recursive Part method

- /*Part*/ closestHit (/*PV*/ front, /*PV*/ back)

This calls the Sphere closestHit on its own Sphere and closestHit on its children and returns the Part with the closest hit. It also sets pMin for that part if its sphere has a hit.

Finally, we need to implement

- drag (/*PV*/ front, /*PV*/ back)

which works to update part2joint (and joint2part) like dragging does now with object2rotated (and rotated2object). This time p is not a vertex of the model. Instead, it is the intersection of the ray with the sphere.