Appendix A

The Peabody Object Representation

Peabody is a representation for complex articulated geometric objects. It represents *figures* composed of *segments* connected by *joints*, also under the influence of *constraints*.

Jack is an interactive system for constructing and manipulating peabody objects. It is crucial to understand peabody before using *Jack*, since so many of the features of *Jack* deal intrinsically with peabody objects and rely heavily on the syntax of the peabody language. You may design **peabody** environments without using *Jack* at all, but *Jack* was designed to make this task easier.

A.1 Introduction to Peabody

A peabody environment consists of a number of individual *figures*, each of which is a collection of *segments*. The segments are the basic building blocks of the environment. Each segment has a "geometry." It represents a single physical object or "part", which has shape and mass but no movable components. The geometry of each segment is represented by a *psurf*, which is generally a polyhedron or a polygonal mesh.

The term "figure" applies not only to articulated, jointed figures such as a human body: any single "object" is a figure. It need not have moving parts. A figure may have only a single segment, such as a coffee cup, or it may be composed of several segments connected by *joints,* such as a robot. The term "object" is used here very loosely; it has no special significance. We use it only to denote some part of the peabody environment.

The term *psurf* refers only to the representation for the geometry of a segment, which is the graph of nodes and edges typically drawn as the wireframe or shaded image of the segment. In the case of a figure with a single segment, it is sometimes convenient to refer to it as a "psurf", but that is not technically correct.

Joints connect segments at attachment points called *sites.* A site is a local coordinate frame specified relative to the coordinate frame of its segment. Each segment may have several sites. Joints connect sites on different segments within the same figure. Sites need not lie on the surface of a segment, but generally they will. A site is a coordinate frame which has an orientation as well as a location.

A.1.1 The Connectivity of Peabody Objects

The peabody environment can be visualized as a directed graph. The segments are the nodes of graph, and the joints form the edges. Figures are maximal subgraphs spanned by joints. It is important to remember that joints do not connect segments directly. Joints connect segments through sites.

A.1.1.1 The Hierarchy of Peabody Objects

Peabody avoids representing objects with a strict hierarchy by encouraging you to think of figures as collections of segments and joints. However, there *is* an underlying hierarchy, and as in the case of moving a figure or adjusting a joint, it is important to remember what the hierarchy is because it has an effect on which objects remain fixed and which move when the joints change angles.

In order for the location of an articulated figure to be well defined, we must designate some element of the figure as its origin. The location of a peabody figure is specified through a site designated as the *root*. The root site roughly corresponds to the "origin" of the figure, and it provides a handle by which the location of the figure may be specified. Viewing the figure as a tree, the root of the figure is the root of the tree. The root site of a figure may be changed interactively in *Jack* from time to time depending upon how you want to manipulate the figure. This is important because the root of the figure serves as the origin of rotation and translation when you manipulate a figure interactively.

It is not allowable to define figures with closed loops of joints in peabody, although the syntax of the peabody language does not prevent you from doing so.

A.1.1.2 A Metaphor for the Connectivity of Peabody Objects

To visualize the graph of a peabody environment, imagine a collection of simple objects, such as machine parts, floating around in zero-gravity space. Several of these objects are connected to each other with hinges. The objects are the segments and the hinges are the joints. The placement of the hinges on the objects is described by the placement of the sites on the segments. A collection of segments hinged together form a figure. There may be several figures floating around. Some figures may consist of lots of segments and hinges. Other figures may have only a single segment. No segment is part of a figure unless it is hinged to the rest of the figure's segments. You need not think of the segments and joints in the figure has having a strict hierarchy. The joints connect segments in completely arbitrary ways.

Each figure is nailed in place in space through its root site. The global placement of this field in space defines in turn the placement of all other segments in the figure. When you rotate a figure interactively, it rotates around its root site. When you translate a figure, it may move along the global coordinate axes of this frame. *Jack* allows you to interactively change the root, so if it becomes convenient to nail the figure down in a different way, it is possible to do so.

When you bend a hinge at one of the joints of a figure, the segments on one side of the joint will remain fixed, and the ones on the other side will move. Which side moves and which remains fixed depends on how the figure is rooted: the ones on the rooted site remain fixed.

We described the graph of the environment as *directed*, and the joints have a distinct direction. However, the directed-ness of a joint does not affect which segments stay fixed and which move as the joint angles change. The direction defines only the order in which the rotations of a joint are applied to produce the complete transformation between the sites which the joint connects. The details of joint definitions and degrees of freedom will be discussed in Section A.3.5.1.

A.1.2 The Geometry of Peabody Objects

It is very important to understand how the geometry is associated with a peabody figure. Each segment may have a psurf associated with it, but the shape of the psurf itself does not affect the topology of the figure. In *Jack*, it serves only as the *image* for the segment. The underlying topology of the figure, in terms of the "lengths" of the segments and the placement of the joints, is defined by the site locations relative to each segment, and it is completely independent of the psurf geometry. Sometimes sites will lie on the surface of the psurf, but this is by design rather than requirement.

There is no enforced relationship between the geometry and the length of a segment. In fact, segments don't really have a "length." If you define the length of a segment as the distance between the joints at either end, then this is only well defined when a segment has only two joints. But peabody allows segments to have any number of joints.

The geometry of a psurf is specified relative to the coordinate origin of the segment. This means that the (x, y, z) coordinates of the vertices of the polygons of the psurf are interpreted and drawn relative to the axes of the coordinate frame of the segment, *not* the world coordinate frame. Each psurf is designed in its own coordinate system.

Take a moment to consider what (x, y, z) cartesian coordinates really mean. These coordinates are only meaningful when interpreted in the context of a coordinate frame. Frequently, the coordinate frame is implicitly the world coordinate frame, but this not necessarily true. The (x, y, z) coordinates of a point define the location of a point relative to the coordinate frame by specifying displacements from the origin

of the frame along the x, y, and z axes of the frame, respectively: start at the origin of the frame, travel x units down the x axis, then travel y units along the y axis, then travel z units along the z axis.

As an example, consider the two cubes shown in Figures A.1 and A.2. Along with each picture is the syntax for the psurf which defines the geometry. The details of the syntax for the psurf files is described in Section A.4. These two objects have the same "shape" but their origins are different. In Figure A.1, the origin is at the one of the corners. In Figure A.2, the origin is in the middle of the cube. It is important to realize that this difference is in the *nodes* of the psurf, not in the definition of the location of the figure itself. The origin of the psurf is implicit: it is *with respect to this frame* that the polygons of the object are interpreted. Peabody gives you great freedom in how psurfs are designed, since the sites may be place anywhere on the segment. This gives you the ability to attach joints to segments in various places.

```
0.00    0.00    0.00
0.00    100.00  0.00
100.00  100.00  0.00
100.00  0.00    0.00
0.00    0.00    100.00
0.00    100.00  100.00
100.00  100.00  100.00
100.00  0.00    100.00
;;
1 2 3 4;
1 4 8 5;
3 7 8 4;
1 5 6 2;
5 8 7 6;
2 6 7 3;
;;
```

Figure A.1: `A Cube`

```
-50.00  -50.00  -50.00
-50.00  50.00   -50.00
50.00   50.00   -50.00
50.00   -50.00  -50.00
-50.00  -50.00  50.00
-50.00  50.00   50.00
50.00   50.00   50.00
50.00   -50.00  50.00
;;
1 2 3 4;
1 4 8 5;
3 7 8 4;
1 5 6 2;
5 8 7 6;
2 6 7 3;
;;
```

Figure A.2: `Another Cube`

This can sometimes be very confusing because it is the *sites* on the segment, and the joints which connect them, which define the location of the segment relative to other segments in the figure. Many times, it is convenient to define a site which lies at the coordinate origin of a segment. This is particularly true of an

elongated segment like an arm, which has a distinct proximal and distal end. In this case, the proximal end may lie at the coordinate origin of the segment, and the distal end may lie down the z axis, for example. In this case, it may seem that the geometry is specified relative to the proximal end, but remember that the geometry is relative to the origin of the segment. The proximal site just happens to lie at that origin.

Drawing a diagram may solve lots of confusion. It is usually a good idea to draw a diagram of the segment with an explicitly labeled origin. Then draw the sites, such as proximal and distal, *away* from the origin, even if in fact they are coincident. This will help to reinforce the fact that they are specified relative to the origin of the segment.

A.1.2.1 A Metaphor for the Construction of Peabody Figures

When designing the geometry of the segments of an articulated figure, it is best to proceed logically in a manner similar to what we might do if we were constructing a hinged mechanism out of wood, metal, or plastic. The most logical thing to do is to design each part one at a time. We begin by choosing an origin for the part and crafting its shape in terms of dimensions measured from that origin. After designing all of the individual moving parts, we proceed to drill holes in each part into which to bolt the hinges. Where do the holes go? We again calculate the location of the holes in terms of displacements from the origin of the segment. Next, we bolt the hinges into the holes and bend them to the correct angles (Let's assume that the hinges are stiff enough hold the figure in place). Finally, we choose a special point on the figure and place this point at the proper place so that the contraption is located in the desired position on the table, floor, or wall.

In the translation of this metaphor, the parts are the segments and the hinges are the joints. The shape of the part is defined in terms of a psurf, whose coordinates are interpreted relative to the coordinate origin of the segment. This coordinate origin is not relative to anything: other things are relative to it! The holes for the hinges correspond to the sites: they are measured from the coordinate origin of the segment. The special point on the contraption through which we fix it to the wall or floor corresponds to the figure's root site.

A.2 The Mechanics of the Peabody Language

The syntax of the peabody language somewhat resembles a programming language, except that it defines static elements, not actions.

A.2.1 Arithmetic Expressions

The peabody language employs a powerful arithmetic expression parser and symbol table, so that any part of the language which requires a numerical value accepts a general arithmetic expression. The syntax of the expressions is similar to an ordinary programming language. The operators and their precedence are shown in Figure A.3. The operators at the top have the greatest precedence.

Operator	Meaning
()	parentheses
-	unary minus
^	exponentiation
/	division
*	multiplication
-	subtraction
+	addition

Figure A.3: The Precedence of Peabody Arithmetic Operators

Variables need not be declared before use: their type is determined by context, although a warning message will be issued if a variable is used before being assigned a value.

A.2.2 Units

Many numerical values in the peabody language refer to physical measurements: angle, distance, mass. Peabody allows such values to be quantified by their type, so that values may be entered in any particular units. Whenever a value is specified, its units should also be specified so there is no confusion about which units are being used. This convention will save much confusion as files are written as used at later dates.

For angles, the legal types are radians and degrees, using the keywords `rad` and `deg`. The default is degrees. For distances, the legal types are millimeters (`mm`), centimeters (`cm`), meters (`m`), inches (`in`), feet (`ft`), and yards (`yd`). The default is centimeters.

A.2.3 Homogeneous Transformations

Peabody relies heavily on the specification of homogeneous transformations, and the peabody language has a rather simple mechanism for describing such transformations. Transformations may be expressed as a sequence of simpler, primitive transformations such as rotation and translation. The translation operator is `trans`, and it takes three arguments, giving the translation in x, y, and z. Rotation may be described with the `xyz` operator, which specifies rotation in terms of angles around the local x, y, and z axes, in that order. For example,

```
t = xyz(10deg,20deg,30deg);
```

specifies a rotation transformation which is formed by a rotation of 10° around the x axis, followed by a rotation of 20° around the *rotated* y axis, followed by a rotation of 30° around the *rotated* z axis. This operator can be used for simple rotations around a single coordinate axis by using zeros for two of the angles.

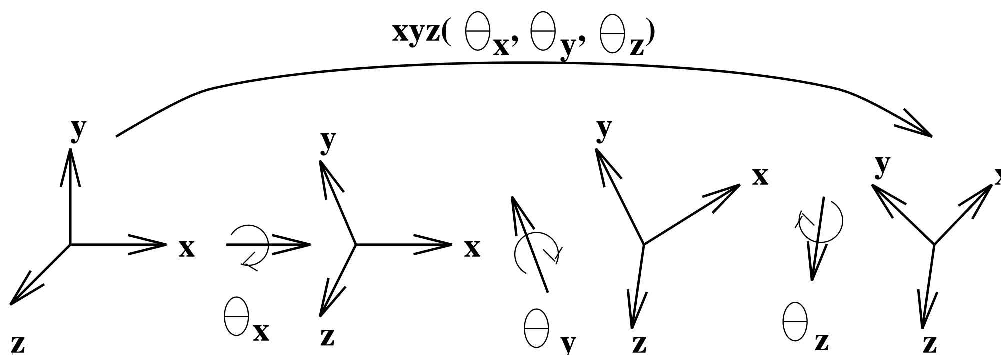

Figure A.4: The `xyz` Rotation Operator

A.2.3.1 Multiplying Homogeneous Transformations

The peabody language represents homogeneous transforms as 4×4 matrices in the form:

$$\begin{bmatrix} x_0 & x_1 & x_2 & 0 \\ y_0 & y_1 & y_2 & 0 \\ z_0 & z_1 & z_2 & 0 \\ p_0 & p_1 & p_2 & 1 \end{bmatrix}$$

The translation vector (p_0, p_1, p_2) is in the bottom row of the matrix.

When transformations are multiplied, the product may be interpreted in two ways. The most intuitive way corresponds to local transformations when applied *right* to *left*. For example, the transformation given by

```
t = xyz(90deg,0,0) * trans(0,100cm,0);
```

may be interpreted as first a translation of 100 cm along the y axis, followed by a rotation of 90° around the *translated* x axis. The ordering is critical, since transformations don't commute.

Alternatively, the product of transformations may be interpreted in global coordinates when read *left* to *right*. The above transformation may be interpreted as a rotation of 90° around the x axis, followed by a translation of 100 centimeters along the *original* y axis.

When *Jack* writes a transformation, it writes the rotation part followed by the translation part, expressed in terms of the `xyz` operator, no matter how the expression was originally specified. This product can be interpreted as a rotation followed by a translation with respect to the base coordinate frame, or alternatively a translation followed by a rotation around the *translated* axes.

The exponentiation operator may be applied to transforms, with the usual meaning. Raising a transform to the power of -1 yields the inverse of the transform.

A.2.3.2 Manipulating Transformations Interactively in Jack

You can experiment interactively with these transformations in *Jack* by moving a figure. As you move the figure, the peabody description of the figure's root appears at the bottom of the window. You can see the description of the transformation as it is being manipulated.

Another valuable exercise is to enter the transformations from the keyboard by hitting `^K` while moving the figure. You will be prompted in the message window to enter the transformation. Experiment with several transformations to get a feel for how this syntax works, particularly in terms of the ordering of the transformations. The details of how to enter values from the keyboard in *Jack* are described in Section 5.8.

A.2.4 Names and Identifiers

All constructs in the peabody language have names, and each name is local to the construct to which it belongs. This ambiguity may be resolved by prefixing the name of construct with the name of its parent, separated by a period. For example, each figure in the environment has a distinct name, and each segment within each figure has a name which is unique only within that figure. Therefore, two figures, say `fred` and `ethel`, may have segments named `head`. The identifier `head` does not uniquely specify *which* head, so the use of `head` must be prefixed with either `fred` or `ethel`, as in `fred.head` or `ethel.head`.

Likewise, joints are local to the figures to which they belong, and sites are local to the segment to which they belong. In the case of sites, a double prefix must be used to uniquely specify a site in the environment. This follows a general rule of thumb that when constructs are named, they must be properly qualified given the context. When naming segments within a figure, the figure name is usually understood and may be omitted.

One exception to the above rule is the pseudo-segment `world`, which is the base coordinate frame for the environment. The world is not a part of any figure, so a reference to one of its sites has only one prefix, as in `world.base`.

Names of constructs and variable identifiers must consist of upper and lower case alphabetic characters, digits, and underscores, and must not begin with a digit. It is important that names of constructs in peabody do not conflict with keywords. Since most keywords are frequently used terms, this rule must be carefully considered when constructing peabody figures. The peabody keywords are listed in Table A.1

It is important to remember the difference between identifiers and strings, especially in places in the peabody language where file names are required. All strings, such as file names, must be enclosed in double quotes. Otherwise, they would be interpreted as variable references.

A.3 Peabody Construct Declarations

The term "construct" refers to any of the basic elements of the peabody object representation: figures, segments, sites, joints, attributes, and lights. The peabody language consists primarily of assignment statements which define the properties of the peabody constructs. Each assignment must be qualified with the construct to which it applies. This is generally done by grouping the assignments together into *blocks* which set forth the construct name and type, similar to a data structure declaration in a programming language. The elements within the block are then automatically associated with that construct.

R	displacement	llimit	psurf	stiff
T	dullness	local	quat	stiffness
ambient	edge	localforce	radius	texture
archive	face	localmoment	refraction	texture
asurf	figure	location	rest	to
attribute	globalforce	mass	rgb	tolerance
attributes	globalmoment	maxtrans	root	trans
blur	glossiness	mintrans	scale	transexp
concentration	hsv	moment	segment	transparency
connect	include	node	shademode	type
constraint	inertia	nstrength	site	ulimit
damp	intensity	patch	specular	vrpd
density	joint	path	sphere	xyz
diffuse	light	pstrength	stexture	

Table A.1: Peabody Keywords

The peabody construct declarations create objects implicitly, not explicitly. Figures, segments, sites, and joints are created when they are defined for the first time. In this sense, the "definitions" of the constructs are not really *definitions* by *declarations.* It is entirely legal to have duplicate declarations for the same object. You should think of the process of reading a peabody file as a sequential stream of assignment statements which set the values of object parameters. The objects are created when they are first referenced. Duplicate assignment statements reset the parameter values.

A.3.1 Figure Declarations

A figure declaration begins with the keyword `figure`, followed by an identifier. A figure declaration may not occur inside any other block. Inside the figure block there may be:

- segment declarations
- site declarations
- joint declarations
- light declarations
- attribute declarations
- posture declarations

Site declarations must have names qualified with the segment name unless they are contained in a segment block. In addition, figures have the following assignment fields:

archive The archive specifies an archive file in which to look for psurf files for the segments in the figure. This field is optional, but if it is used it should occur before any segments which reference psurfs in the archive. The value is a character string giving the name of the archive file.

root The root specifies the root site for a figure. This is the site with respect to which the figure's `location` field describes its global placement. The value is the name of the site.

location The location gives the transform which defines the global placement of the figure's root. The placement of all other sites and segments in the figure is determined from this. The value is a homogeneous transformation.

A.3.2 Segment Declarations

A segment declaration begins with the keyword **segment**, followed by an identifier. A segment declaration may occur inside a figure block. If it lies outside a figure block, its name must be qualified with a figure name. Site declarations and attribute definitions may occur inside the segment block.

In addition, segments have the following assignment fields:

psurf The psurf gives the file which describes the geometry of the segment. The value is a character string which is the name of the psurf file.

attributes The set of attributes describes which surface attributes are to be associated with the psurf. The value is an array of attribute names, separated by commas. The number of attributes in this array must match the number of attribute indices used in the psurf file. If the psurf has a single attribute, then the value may be just the attribute name, and you may use the keyword `attribute`, without the 's'.

Peabody has a built-in set of rules for where it looks for psurf files. See Section 9.3 for a description of these rules. See Section A.4 for a description of the syntax of the psurf files.

A.3.3 Path Declarations

△

A *path* is a set of n sites on a segment, named $\{ pnt_0, pnt_1, \ldots pnt_n \}$, with one additional site called *point*. The peabody description of a path looks like:

```
segment paths {
    site base->location = trans(0cm,0cm,0cm);
    site point->location = trans(0cm,0cm,0cm);
    site pnt0->location = trans(0cm,0cm,0cm);
    site pnt1->location = xyz(0deg,0deg,-38deg) * trans(3cm,62cm,-11cm);
    site pnt2->location = xyz(-180deg,-75deg,141deg) * trans(45cm,62cm,-86cm);
    site pnt3->location = xyz(-180deg,-10deg,141deg) * trans(112cm,62cm,-1cm);
    site pnt4->location = trans(136cm,62cm,75cm);
    path = ("spline", 31, (site)point,
        (site)pnt0, 0,
        (site)pnt1, 0.25,
        (site)pnt2, 0.50,
        (site)pnt3, 0.75,
        (site)pnt4, 1);
}
```

The path field of the segment defines the interpolation method ("spline"), the number of generated samples (default 31), the point site, and the set of sites with their corresponding time values. The point site is used when animating a path, and the point site moves to a position on the path corresponding to a time fraction between 0 and 1 (i.e. the spline is evaluated at some time between 0 and 1, and the point site is moved to that location.

A.3.4 Site Declarations

A site declaration begins with the keyword `site`. A site declaration may occur inside a segment block. If it lies outside, its name must be fully qualified with the segment name. If it does not lie within a figure block, the name must also be qualified with the figure name.

A site has a single following assignment field, giving its location.

location The location is a transform describing the position and orientation of the site with respect to the coordinate frame of its segment. The value is a homogeneous transformation.

A.3.5 Joint Declarations

The declaration of a joint begins with the keyword `joint`. A joint connects two sites on two segments within the same figure, and represents a transformation between segments.

The principle statement in a joint declaration is the `connect` statement, which is not in the form of an assignment. It specifies the two sites which the joint connects, and it takes the form of "`connect site1 to site2`"

The joint has the following assignment fields:

type A functional expression describing the degrees of freedom of the joint. The value of the expression is described below.

displacement A vector of numbers specifying the current angles of the joint, having as many elements as the joint has degrees of freedom.

ulimit A vector of numbers describing the upper limit of the angle or distance of each degree of freedom, having as many elements as the joint has degrees of freedom.

llimit A vector of numbers describing the lower limit of the angle or distance of each degree of freedom, having as many elements as the joint has degrees of freedom.

By default, the transformation at a joint is an arbitrary homogeneous transformation, which specifies the relative position of the two sites which the joint connects. The `connect` statement specifies that the joint connects one site *to* another, and this defines the direction of the joint and the joint's displacement. The first site in the connect statement is sometimes called the "from" site; the second site is called the "to" site. The joint's displacement transform defines the global placement of the "to" site relative to the global placement of the "from" site[1]. The directionality of the joint is defined in terms of how the joint would normally behave in the figure, when the figure is rooted in its normal place. By "normal" behavior, we mean that when you interactively adjust a joint, the "from" side remains stationary and the "to" side moves. This directionality remains fixed even if the figure is re-rooted and the actual figure hierarchy is changed.

For example, in the human figure model, the shoulder is defined to connect the clavicle *to* the upper arm. This definition is convenient since we normally expect that when we change the transformation at the shoulder,the arm will move and the upper torso will remain fixed. This will not be the case, however, if the figure is rooted through his hand, possibly hanging from a rope or floating in zero gravity. In this case, we still define the transformation across the joint in the direction "from" the clavicle "to" the arm.

A.3.5.1 Degrees of Freedom

Joints may have specific degrees of freedom which restrict the transformation across the joint. The type of the joint is specified by the `type` field, whose value is an arbitrary expression composed of primitive rotation and translation operators. The rotation operator is `R`, and the translation operator is `T`. Each rotation and translation specifies an axis. The axis must be a coordinate axis of unit length: $(1,0,0)$, $(-1,0,0)$, $(0,1,0)$, $(0,-1,0)$, $(0,0,1)$, or $(0,0,-1)$.

The current angle associated with the axis comprises the *displacement* of the joint and is not part of the type. A simple joint may be defined as:

```
joint elbow {
    connect lower_arm.proximal to lower_arm.distal;
    type = R(1,0,0);
    displacement = (90deg);
    ulimit = (180deg);
    llimit = (0deg);
}
```

This joint rotates only around the x axis. The displacement field specifies that the transformation is a $90°$ rotation around the x axis.

[1]The global placement of the "from" site is determined in turn by its location relative to its segment's base coordinate frame. This in turn is determined by other site locations and joint displacements, up the figure hierarchy to the root of the figure, where the global placement is fixed.

Joints may have up to three degrees of freedom by multiplying primitive operators. In this case, the displacement field has the same number of arguments as the type expression has primitive elements. The complete transformation at the joint is the product of each primitive operator instantiated with the appropriate angle. As in the case of homogeneous transforms, the operators should be interpreted right to left as local transformations, i.e. with respect to the local, or current, transform. Alternatively, the displacement at the joint may interpreted from left to right as primitive transformations with respect to the parent coordinate frame, that is, the site on the "from" side of the joint.

For example, the transformation at the joint defined by:

```
joint luceille_ball_joint {
    connect torso.shoulder to arm.base;
    type = R(0,0,1) * R(0,1,0) * R(1,0,0);
    displacement = (90deg,45deg,30deg);
    ulimit = (180deg,90deg,60deg);
    llimit = (0deg,0deg);
}
```

may be interpreted as a rotation of 30° around the x axis of site `torso.shoulder`, followed by a rotation of 45° around the rotated x axis, followed again by a rotation of 90° around the *rotated* y axis. It may alternatively be interpreted as a rotation of 90° around the x axis of the site `torso.shoulder`, followed by a rotation of 45° around the y axis of the same frame, followed by a rotation of 30° around the x axis of the same frame as well.

A.3.6 Constraint Declarations

Constraints specify desired geometric relationships. The parameters of constraints are described in Chapter 10. The syntax of the constraint declaration in the peabody language has an assignment statement for each of the constraint's properties. There can be no blocks within a constraint block.

A constraint has the following assignment fields:

type The relationship type. The expression can be either a single string giving the position or orientation type if there is only one, or its is a 3-vector, with the first element a string describing the orientation relationship type, the second a string describing the position relationship type, and the third element a number giving the position/orientation weight.

end The end effector. The value may be a site or a node. It must be typecast to determine which type it is.

goal The goal. The value may be a site, node, face, or a matrix. A matrix type specifies a hold constraint. Otherwise, it must be typecast to determine which type it is.

startjoint The starting joint. If this field is absent, then the constraint is a rooting constraint.

weight The constraint weight.

A.3.7 Surface Attributes

Each polygon in the peabody environment has a surface attribute associated with it, and the surface attributes may be specified in the peabody file. Surface attributes are defined in a block structured manner similar to the other peabody constructs:

```
attribute brown {
    diffuse = (0.48,0.26,0.00);
}
```

Surface attributes have the following assignment fields

ambient This is a triplet of real numbers describing the ambient color of the surface in RGB coordinates.

diffuse This is a triplet of real numbers describing the diffuse color of the surface in RGB coordinates.

specular This is a triplet of real numbers describing the specular color of the surface in RGB coordinates.

glossiness This is an integer exponent describing the glossiness.

Alternatively, you may specify the ambient and diffuse parameters as scalar values and give the attribute a color with the `rgb` field. This makes the ambient and diffuse parameters different intensities of the same color. This is the most common way of describing attribute parameters.

The way in which the surface attribute parameters affect the shading of the surface is described in Section 12.

A.3.8 Light Source Declarations

Light sources are necessary for rendering and are an integral part of a computer graphics and animation environment. Light sources are represented in peabody as special types of segments. A light declaration begins with the keyword `light`. Usually, a light source will consist of a figure with a single segment, but it is possible to give light source properties to any segment on any figure. The origin of the light is the origin of the light segment, and the light is emitted equally in all directions. Lights are not directed. In the peabody grammar, lights are interchangeable with segments:

```
figure foo {
    light bar {
        site base->location = trans(0cm,0cm,0cm);
        color = (1,1,1);
    }
}
```

Lights have the following assignment assignment fields:

ambient This is a triplet of real numbers describing the ambient color of the light in RGB coordinates.

color This is a triplet of real numbers describing the color of the light's emission in RGB coordinates.

The way in which the light source parameters affect the shading of a surface is described in Section 12.

A.3.9 Peabody Block Structure

The block structure of the peabody language facilitates prefixing constructs with their "parent" construct by effectively prefixing everything within the block with the appropriate construct name. Thus, within a figure declaration, all segment names are taken relative to the figure, so there is no need to prefix them. Within a segment declaration, the site names are taken relative to the segment. The joint declaration references sites, which must always be qualified at least to the segment level.

The block structure exists primarily for convenience, since all constructs may occur in any arrangement provided they are fully qualified. For example, a segment block normally belongs inside a figure block, but it may occur outside, provided the segment name is prefixed with the proper figure name.

The curly braces serve to prefix each assignment statement in a construct with the proper construct name and type. They may be replaced by the "arrow" notation:

```
segment cube->psurf = "cube.pss";
```

This short form may be used for any assignment field in the construct.

The traditional block structure resembles a "definition," but actually each construct is defined whenever it is first used. There is no formal distinction between definitions and references. Any subsequent references to the construct refer to the original rather than creating a new one. This allows the same file to be read multiple times without side effect. The effect of reading a peabody file multiple times is to reset the values of the assignment fields rather than create new constructs. A common use of this is for separating joint displacements from figure definitions.

This also allows for very terse definitions. For example,

```
site fred.arm.base->location = trans(0,0,0);
```

is a valid figure definition.

As *Jack* reads a peabody file, it sets the given construct field regardless of its previous value. An exception to this rule applies to psurfs. If a psurf for the segment already exists and it was read from a file of the same name, then the psurf is not re-read. This allows figure definitions which include psurf specifications to be re-read without the overhead of re-reading the psurfs.

A.3.9.1 Figure Files

Sometimes it is convenient to create a *definition* for a certain type of figure, that is, a template which can be used to create different instances of the figure.

A single environment file cannot be used to create different instantiations of the same figure definition, since the names are associated directly with the figures, and figures must have unique names. Peabody allows you to define figures in special *figure files*. Different figures may be instantiated from a figure file with different names.

A figure file is simply a file containing the definition of a single figure, without the identifier following the `figure` keyword. For example, consider a file called `thing.fig`:

```
figure {
    segment x {
        psurf = "cube.pss";
        site base->location = trans(0,0,0);
        site p->location = trans(100,100,100);
    }
    segment y {
        psurf = "pyramid.pss";
        site base->location = trans(0,0,0);
    }
    joint elbow {
        connect x.p to y.base;
    }
}
```

Now, another file called, say `things.env`, could contain:

```
figure ["thing.fig"] a;
```

The syntax of the figure instantiation specifies the name of the figure file, `thing.fig`, in square brackets, followed by the name of the instantiated figure, `a`. Note that the name of the file must be enclosed in double quotes.

The file `things.env` may include an additional “thing”:

```
figure ["thing.fig"] a;
figure ["thing.fig"] b;
```

A.3.9.2 Figure Parameters

Figures may have parameters, much the same as function parameters in a programming language. This makes it possible to generalize a figure template to be instantiated with different sizes, types, etc. For example, the above figure may have been defined as:

```
figure (height,width) {
    segment x {
        psurf = "x.pss";
        site base->location = trans(0,0,0);
        site p->location = trans(width,height,0);
    }
    segment y {
        psurf = "y.pss";
```

```
        site base->location = trans(0,0,0);
    }
    joint elbow {
        connect x.p to y.base;
    }
}
```

and the instantiation the figure would look like:

```
figure ["thing.fig"] (50cm,100cm) a;
figure ["thing.fig"] (40cm,200cm) b;
```

A.3.10 Postures and Posture References

△

A posture block is declared within a figure block. It's purpose is to capture a posture (joint angles, figure location, root, and constraints and behaviors) for a particular figure. There are two forms of a posture declaration inside a figure block. The first one is called a *posture reference*, and uses the keyword **postureref**, and looks like:

```
figure human {
    postureref ["standing.post"] (10cm) standup;
    postureref ["sitting.post"] sitdown;
}
```

The above declaration adds two named postures to the figure *human*, and they are named *standup* and *sitdown*. The important point to know here is that a **postureref** declaration will **not** read the posture files (in this case, **.post**) when the postureref is read. It only adds the named posture (along with filename and arguments) to the list of postures attached to the figure. This is important, since the posture definition file (**.post** file) may contain references to peabody objects that do not yet exist in the environment. The posture file is read when a *Jack* command requests the posture to be instantiated. This also allows **postureref** declarations to be stored in figure definition files (**.fig** files).

The other form of a posture declaration is called a *posture instantiation* and looks just like above, except **postureref** is replaced with **posture**. It would look like:

```
figure human {
    posture ["standing.post"] (10cm) standup;
    posture ["sitting.post"] sitdown;
}
```

The difference here is that when the declaration is read, the posture file is read, and therefore the posture is instantiated (the figure is moved to the corresponding posture). In the above example, the figure would be moved to the *standup* posture, then the *sitdown* posture.

Posture definition files can be parameterized just like figure files and motion group files (see above and below). In the example above, the posture file **standing.post** takes one parameter, a length measurement. The file **standing.post** could look like:

```
posture (length) {
    root = lower_torso.distal;
    joint left_shoulder->displacement = ...
    ...
    location = trans(length, 0cm, 0cm);
}
```

A posture file must always start with **posture**, not **postureref**. Inside a posture block, you may have joint displacements, site locations, constraints, behaviors, root specifications, etc (any field or block which may appear in a figure block may appear in a posture block). Note that all segment/site/joint names are *relative*, they are not prefixed with the figure name. This allows posture files to be *shared* between similarly structured figures (i.e. the human).

A.3.11 Motion and Motion Group Declarations

△

The declaration of a motion begins with the keyword `motion`, followed by a name for the motion. The naming convention is the same as that for segments. A motion should not be declared inside any other peabody construct, except a `motiongroup` or at the top level by itself (in which case it belongs to the `default` motion group. A sample motion in peabody format looks like:

```
motion hand {
    figure = (figure)human5;
    type = "hand control";
    starttime = 0sec;
    duration = 2sec;
    off = 1;
    velocitycontrol = "ease in/ease out";
    data = ("right","site",(site)hand.paths.point,"waist");
}
```

Every motion type has a set of common fields (figure, type, starttime, duration, off, and velocitycontrol). Additionally, each motion type defines a field called data which is simply a vector of values. This vector is generated by the motion to save whatever parameters it needs. It is also parsed by the motion when being read from a peabody file. The only restriction on the values is that it must be a valid parse-able peabody value. The assignment fields for a motion must appear in the order that they are shown above. The assignment fields for a `motion` are:

figure This is a figure reference to the figure to which this motion is applied. Since this is a figure reference, you must precede a figure name with the `(figure)` casting operator. If this is a peabody variable, it must be a figure reference. (See the `motiongroup` below).

type The type field is a string describing the type of this motion. The type tells peabody how to interpret the data field below. Please see the table below for the values of type, and the corresponding format of the data field.

starttime This is the time, in seconds, when the motion will start. This may be ant valid peabody expression which resolves to a number.

duration An expression, evaluating to the duration, in seconds, of the motion.

off This field is optional, and if 1, means the motion is actually turned off (it won't execute).

velocitycontrol For motions which interpolate between a beginning and ending value, this is the method for controlling the velocity function of the interpolation. Valid values are: `"constant", "accelerate", "decelerate", "ease in/ease out"`.

data The data field is a vector of values representing the information needed by each motion type. It is different for each type. Below is a few examples from the simpler motion types:

joint $(n, jointname_1, (displacement_1) \ldots jointname_n, (displacement_n))$

n is the number of joints involved. For each joint involved, the $jointname_i$ is listed, followed by a vector giving the goal joint displacement.

figure $(xform)$

The $xform$ is the global transform for the goal of the figure motion.

camera $(xform, window_name, vrpd)$

The $xform$ is the global transform for the camera, the $window_name$ is the name of the peabody window, and $vrpd$ is the view reference point depth for the camera.

light $(segmentp, (s_R, s_G, s_B), (e_R, e_G, e_B))$

The *segmentp* is a segment pointer (to the light source), followed by the starting rgb color vector (s), and the ending color vector (e). If s is $(-1, -1, -1)$, the starting color is just the current color of the light source (when the motion starts).

item[path] $(segmentp, traversal)$

The *segmentp* is a segment pointer to the path, followed by *traversal*, which can be either `''forward''` or `''reverse''`.

item[figure path] $(segmentp, traversal, figurep)$

Same as above, except *figurep* points to the figure which moves along the path.

command $(preactionJCL, applyJCL, postactionJCL)$

These are three JCL command strings. The $preactionJCL$ is executed on the first frame of the motion, the $applyJCL$ is executed on each frame of the motion, and the $postactionJCL$ is executed on the final frame of the motion. If the JCL has double-quotes, they must be appropriately escaped (i.e. " becomes
").

Each motion can also be a member of a motion group. Motion groups can also be parameterized similar to the way figure files can be parameterized (See Section A.3.9.1). An example motion group file:

```
motiongroup (fig, start, durate) {
    motion arm4_arm1 {
        figure = fig;
        type = "joint";
        starttime = start + (0 * durate);
        duration = 0.67 * durate;
        velocitycontrol = "constant";
        data = (3,jointref(fig,''joint4''),(0.00deg,0.00deg)
        ,jointref(fig,''joint3''),(0.00deg,0.00deg)
        ,jointref(fig,''joint2''),(0.00deg,0.00deg)
        );
        }
    motion chain {
        figure = fig;
        type = "figure";
        starttime = start + (0.33 * durate);
        duration = 0.67 * durate;
        velocitycontrol = "constant";
        data = (trans(5.62cm,0.00cm,-202.27cm));
        }
}
```

If this motion group were in a file call `move.mgp`, then a valid peabody reference to create this motion would be:

```
motiongroup ["move.mgp"] ((figure)chain, 0sec, 1.50sec) move;
```

The motiongroup structure is useful for grouping related motions together, and the intention of it is also to be used for creating composite motion templates. But there is a major problem ... The problem is that peabody name references within the data field of motions need to be *relative* to the figure of the motion or relative to some parameter of the motiongroup.

A solution to this problem is to use the peabody built-in functions (siteref), `segmentref`, and `jointref`. These function takes two arguments, the first can either be a figure name (string) or a figure reference, and the second is a *relative* site/segment/joint name (i.e. of the form `segment.site`, `segment`, or `joint` respectively). These function will return a reference pointer to the actual object (site, segment or joint).

A.3.12 The Include Statement

The `include` statement nests a peabody file in another file, similar to the `#include` preprocessor control statement in the C programming language. Its format is simple:

```
include "file.env";
```

The effect is identical to inserting the entire contents of the file into the original file at the location of the include statement. Files may be nested to a level of 8.

A.4 The Syntax of Psurf Files

Psurfs may be described syntactically in text files. A psurf file is a textual representation for the nodes and faces of the psurf. By convention, these files have the suffix `.pss`. The file lists a set of nodes and faces.

```
{ a cube:  8 nodes, 6 faces }
0.00    0.00    0.00
0.00    100.00  0.00
100.00  100.00  0.00
100.00  0.00    0.00
0.00    0.00    100.00
0.00    100.00  100.00
100.00  100.00  100.00
100.00  0.00    100.00
;; { end of nodes }
1 2 3 4; { back }
1 4 8 5; { bottom }
3 7 8 4; { right side }
1 5 6 2; { left side }
5 8 7 6; { front }
2 6 7 3; { top }
;;
```

Figure A.5: `An Example psurf`

A psurf file begins with a list of nodes, which are specified as triplets of real numbers. There may be an optional comma between the triplets. The numbers may contain decimal points, but the decimals are not necessary. No leading 0 is required for fractions less than 1.0. The nodes are numbered implicitly starting at 1. The node table is terminated with two semicolons.

Following the nodes are the faces, which are lists of indices into the nodes. Each list specifies the vertices of the face, and is terminated with a semicolon. The faces are terminated by an empty vertex list, i.e. two adjacent semicolons. There is a predefined limit of 256 vertices in each face. The vertices should be in counter-clockwise order. This is discussed in greater detail in Section A.4.5.

Comments may appear anywhere in the file and are delimited by curly braces. The indices listed in the psurf file all start at 1 for historical reasons. A single psurf file may contain several individual “psurfs.” In other words, is is legal to concatenate several psurf files into one, which will make a single psurf out of several disconnected components.

Psurfs may be in any general scale, but it is best to make the coordinates correspond to centimeters. *Jack* assumes that all objects are scaled in centimeters, and it initializes the view so that objects in the same general scale as human bodies are displayed conveniently on the screen. You can change this scale if necessary, but it is still best to keep all object in the same basic range of sizes.

A.4.1 Surface Attributes

Between the index of the last vertex of each face and the semicolon which ends the face, there may be an attribute specification, which is the keyword `attribute` followed by an index into the psurf's attribute table, all delimited by square brackets. By default, the attribute index is 0, and its value carries over from one face to the next, so the attribute specification actually sets the "current" value, to be assigned to all subsequent faces until its value is changed again.

This is only part of the attribute information. The psurf itself does not define the material properties associated with each face, but it does specify which faces are made of the same material. When a psurf is associated with a segment, it will be instantiated with a set of surface attributes. The number of attributes will match the number of attribute indices specified in the psurf file, and the attributes will be associated with the faces according to each face's attribute index. The details of how attributes are associated with faces is described in Section A.3.7.

A.4.2 Face Smoothness

Along with the attribute specification, each face has a *smoothness* flag. This flag tells whether the face models a flat surface like a polyhedron, or whether it is being used to model a small piece of a curved surface. In computer graphics, it is sometimes convenient to model curved surfaces with a mesh of small polygons. The polygons themselves are an artifact of the model, not a geometric property.

When an object is drawn in wireframe, it's drawn with its edges. When it is shaded, its faces are filled in. If a face is flat, it has a constant surface normal across the entire face. If the face is smooth, the surface normal is computed at each of the vertices of the face in terms of the normals of the adjacent faces, and the normal at points in the interior of the face is interpolated from the values at the vertices. This is *phong shading*.

It is important *not* to use smooth shading with psurfs which are not polygon meshes. If a psurf consists of a few big faces and the angles between them are great, then the object will not look right if it is smooth shaded.

A.4.3 Node Coloring

It is also possible to assign rgb color values to individual nodes in a psurf file. The format is to place an rgb value after the node definition in the psurf file. A cube, with 7 white vertices and one red vertex, would look like the following:

```
    0.0     0.0     0.0 [ rgb( 1.0, 1.0, 1.0) ]
    0.0   100.0     0.0 [ rgb( 1.0, 1.0, 1.0) ]
  100.0   100.0     0.0 [ rgb( 1.0, 1.0, 1.0) ]
  100.0     0.0     0.0 [ rgb( 1.0, 1.0, 1.0) ]
    0.0     0.0   100.0 [ rgb( 1.0, 1.0, 1.0) ]
    0.0   100.0   100.0 [ rgb( 1.0, 1.0, 1.0) ]
  100.0   100.0   100.0 [ rgb( 1.0, 0.0, 0.0) ] {the red one}
  100.0     0.0   100.0 [ rgb( 1.0, 1.0, 1.0) ]
;;
1 2 3 4 [smooth] [attribute 0];
1 4 8 5 [smooth] [attribute 0];
3 7 8 4 [smooth] [attribute 0];
1 5 6 2 [smooth] [attribute 0];
5 8 7 6 [smooth] [attribute 0];
2 6 7 3 [smooth] [attribute 0];
;;
```

You can use the command `set node rgb` to edit the color of a node. Note that you should use `make segment smooth` to properly blend the node colors across a face, using Gouraud shading. Note that lighting will

not effect segments that have node colors. *Jack* uses the node colors to display the results of a radiosity rendering, therefore the lights are disabled in the environment for those segments which have node colors.

A.4.4 Big Psurfs

A psurf file may also have an optional specification `[big]` on the first line of the file. This specification will inhibit *Jack* from generating display lists for the edges of the psurf. This is useful if the psurf is imported from other CAD system that can't break up geometry and must send it as one long list of nodes and faces. *Jack* usually creates edge lists for displaying psurfs in wireframe. This spec is only necessary if the psurf has more then 1000-2000 nodes.

A.4.5 Face Orientation

Psurf faces have a specific orientation, which means that a face really has just one side. When viewing an object in wireframe, this is of no consequence since the object is displayed with its edges. However, when an object is shaded its intensity is defined in terms of the angle between the face and the light sources. The orientation of the face depends upon the order of the vertices. The surface normal is defined to point in the direction given by the right hand rule. This means that a counter-clockwise traversal by the right hand rule yields an outward pointing normal. In other words, place your right hand on the polygon and sweep your fingers around the vertices in the order they occur in the file. Your thumb then points in the direction of the surface normal.

It is imperative that the faces of psurfs be oriented consistently. When objects are shaded in *Jack*, only the side of the surface normal will be shaded correctly. The other side will be effectively shadowed and will be illuminated with ambient light.

Psurfs usually represent collections of polyhedra, so the nodes, edges, and faces form a planar graph[2]. This places some important restrictions on how the faces may be defined:

- Each edge (pair of nodes) should be contained in at most two faces.
- No pair of nodes (an edge) should be contained in two faces in the same order.

This means that if you traverse each face of a psurf, you should never travel along the same edge more than twice, and you should never travel along the same edge twice in the same direction. If you have, the psurf does not define a valid polyhedron, and if you try to shade it, its surface normals are not well-defined.

There are exceptions to these rules, particularly in the case of open objects, i.e. sets of faces which are connected but which do not bound a closed region in space.

A.4.6 Binary Psurfs and BPS

For faster reading and display, you may pre-process psurf files and convert them into "binary psurfs" using the program **bps**. **bps** reads a text psurf file and produces another file with the same base name but with the suffix `.bps`. This is a binary psurf file, and it represents the same geometric information about the psurf in a form which is easier for *Jack* to read and more efficient for it to display.

[2] A planar graph is a graph which is capable of be drawn in a plane without crossing edges. It does *not* mean that all nodes of the graph lie in a plane in space