8

Hierarchical structures

Hierarchical models of complex assemblies are generated by an aggregation of subassemblies, each one defined in a local coordinate system, and relocated by affine transformations of coordinates. This operation may be repeated hierarchically, with some subassemblies defined by aggregation of simpler parts, and so on, until one obtains a set of elementary components, which cannot be further decomposed.

Two main advantages can be found in a hierarchical modeling approach. Each elementary part and each assembly, at every hierarchical level, are defined independently from each other, using a *local* coordinate frame, suitably chosen to make its definition easier. Furthermore, only *one* copy of each component is stored in the memory, and may be instanced in different locations and orientations how many times it is needed.

In the present chapter a graph-theoretical model of hierarchical assemblies is discussed, also making some reference to its concrete implementation in standard graphics systems. The chapter includes several worked examples of hierarchical structures, including the 2D and 3D design of the furniture of a living room, an anthropomorphic body modeling, the preliminary sketching of a parametric umbrella, an algorithm for drawing tree diagrams, and operators to make complex arrangments of 2D symbols.

8.1 Hierarchical graphs

A hierarchical model, defined inductively as an assembly of component parts [LG85], is easily described by an *acyclic directed multigraph*, often called a *scene graph* or *hierarchical structure* in computer graphics. The main algorithm with hierarchical assemblies is the *traversal* algorithm, which transforms every component from *local coordinates* to global coordinates, called *world coordinates*.

Acyclic directed-multigraph

The standard definition of a *directed graph* G states that it is a pair (N, A), where N is a set of *nodes* and A is a set of directed *arcs*, given as ordered pairs of nodes. Such

Geometric Programming for Computer-Aided Design Alberto Paoluzzi
 ISBN 0-471-89942-9

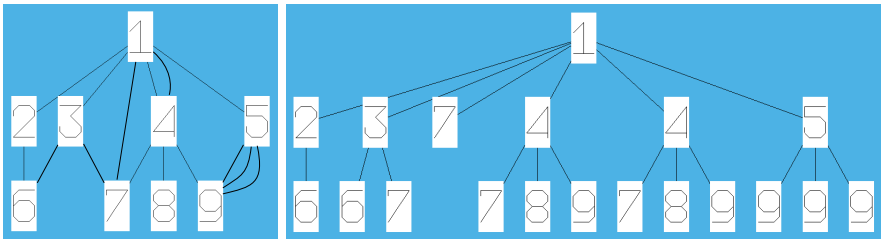

Figure 8.1 (a) Directed acyclic multigraph of an assembly (b) Tree of sub-assembly instances generated when traversing the multigraph

a definition is not sufficient when more than one arc is to be considered between the same pair of nodes.

In this case the notion of *multigraph* is introduced. A *directed multigraph* is a triplet $G := (N, A, f)$ where N and A are sets of nodes and arcs, respectively, and $f : A \to N^2$ is a mapping from arcs to node pairs. In other words, in a multigraph, the same pair of nodes can be connected by multiple arcs.

Directed graphs or multigraphs are said to be *acyclic* when they do not contain cycles, i.e. when no path starts and ends at the same vertex. *Trees* are common examples of acyclic graphs. Nodes in a tree can be associated with their integer *distance* from the root, defined by the number of edges on the unique path from the root to the node. A tree can be layered by *levels*, by putting in the same subset (level) all the nodes with equal distance from the root. A tree, where each non-leaf node is the root of a subtree, is the best model of the concept of *hierarchy*.

Acyclic graphs/multigraphs are also called *hierarchical graphs*, because they can be associated to a tree, generated at run-time by visiting the graph with some standard traversal algorithm, e.g. with a depth-first-search [AHU83]. The ordered sequence of nodes produced by the traversal is sometimes called a *linearized graph*. Each node in this sequence is suitably transformed from local coordinates to *world coordinates*, i.e. to the coordinates of the root, by the traversal algorithm.

8.1.1 Local coordinates and modeling transformation

A hierarchical multigraph is used to model a *scene database* in the sense described below. In particular, each *node* may be considered a *container* of geometrical objects, where:

1. The geometrical objects contained in a node a are defined using a system of coordinates which is *local* to a.
2. Each arc (a, b) is associated with an affine transformation of coordinates. In simplest cases the identity transformation is used.
3. The affine mapping of the arc (a, b) is used to transform the objects contained within the b node to the coordinate system of the a node.

The previous properties are extended inductively to the subgraphs rooted in each node. In particular:

1. the subgraphs rooted in the b_i sons of a, i.e., the geometrical data contained in such subgraphs, may be affinely mapped to the coordinates of a. The affine maps associated to (a, b_i) arcs are used at this purpose;
2. a subgraph may be instanced in a node (i.e., in its coordinate space) more than once. As shown in Figure 8.1 and discussed in Example 8.3.1, the number of instances of a subgraph in a node equates the number of different paths that connect the subgraph to the node.

Summary The main ideas concerning *scene graphs* can be summarized as follows. Nodes are *containers* of geometrical data stored in *local* coordinates. They are also used as root of subgraphs, whose data are transformed to the node coordinates by a traversal algorithm. Arcs (a, b) are associated with affine transformations, which map the data contained in b from their local coordinates to the coordinates of a. More than one arc may exist between the same node pair. This allows storage in memory only of *one copy* of each container. The composite transformations of coordinates applied to the linearized graph generated at traversal time are collectively known as the *modeling transformation*.

8.2 Hierarchical structures

Various kinds of hierarchical assemblies are used in standard graphical systems, such as GKS, PHIGS and VRML, as well as in graphics libraries like Open Inventor[1] and Java 3D. The model of hierarchical structures adopted by PLaSM is inspired, even in the name of the operator used for this purpose, by the one introduced by PHIGS.

GKS segments In GKS (Graphical Kernel System) [EKP87, ISO85] the storage of graphical *segments* was introduced, which defines a two-level hierarchical system. More specifically: graphical *primitives* like polylines, polygons and text strings can be grouped into named collections, called *segments*. Segments are stored in a "normalized" coordinate space, and cannot be nested. Geometric transformations, including composite translation, rotation and scaling, can be applied to segments. Also, segments can be made visible/invisible, *picked* interactively and highlighted.

PHIGS structure network In PHIGS (Programmer's Hierarchical Interactive Graphics System) [HHHW91, ANSI87] *structure networks* are used, which can be visualized as acyclic graphs, where *structures* give the nodes of the graph, and *references* between structures give the arcs between nodes. Hierarchical assemblies of any depth can be modeled by such acyclic graphs. Structures are stored in a *centralized structure store* (CSS) independent of workstations, were structures can be *posted*. Structures can be interactively edited, by inserting, replacing and deleting *structure elements*.

---

[1] Inventor, later called Open Inventor, was developed and marketed by SGI (Silicon Graphics Instruments) and by TGS (Template Graphics Software). An *OpenSource* version for both Linux and Windows platforms has recently been released by SGI.

Inventor's scene graphs In Open Inventor [WO94] scene *databases* are defined as collections of *scene graphs*. A scene graph is an ordered collection of *nodes*, which are basic building blocks holding shape descriptions, geometric transformations, light sources or cameras. In other words, each node represents a geometry, property, or grouping object. Hierarchical scenes are created by adding nodes as children of grouping nodes. This approach clearly results in building scene graphs as acyclic directed graphs. No properties or transformations are attached to graph arcs, which just represent the containment relation between nodes. Node kits are provided as C++ classes with a predefined behavior, which can be customized by the application programmer by subclassing.

VRML scene graphs in VRML (Virtual Reality Modeling Language) [ANM97, ISO97] the same idea of scene graphs as ordered collections of nodes is used. The reader should notice that VRML originates from the File Format of Open Inventor. Such VRML files, written either in ASCII or gzipped binary format, can be used to import scene graphs into a scene database or even as an alternative to creating scene graphs programmatically. For example, scene graphs can be imported in Java 3D [SRD00] using VRML files. Some non-trivial differences exist between the semantics of scene graphs with versions 1.0 and 2.0 of VRML.

Remarks The arcs of scene graphs are normally specified *implicitly* in real graphical systems. For example, an arc is actually specified when a node is contained or referred within another one. In particular, it is possible to specify a new container node together with either the matrix or the parameters of the transformation to be associated with the the arc that connects the new container to the current node.

8.2.1 Hierarchical structures in PLaSM

A *container* of geometrical objects is defined in PLaSM by applying the predefined operator STRUCT to the sequence of contained objects. The value returned from such application is of the *polyhedral complex* type. The coordinate system of the value returned from a STRUCT application is the one associated with the first object of the argument sequence. Also, the resulting geometrical value is often associated with a symbol used as the name of the container, as in

```
DEF obj = STRUCT:< obj1, obj2, ... , objn >;
```

The obj geometry can be pictorially described, using the previously discussed graph model of hierarchical structures, as shown in Figure 8.2a. Clearly, each component object may in turn be defined as a container of other objects, i.e. as the root of a subgraph, as shown in Figure 8.2b, according to the following definition:

```
DEF obj2 = STRUCT:< obj21, ... , obj2m >;
```

Exactly the same geometric result[2] would be generated by direct nesting of STRUCT sub-expressions:

---

[2] Actually, using an internally-generated symbol to name the second son of the scene graph root.

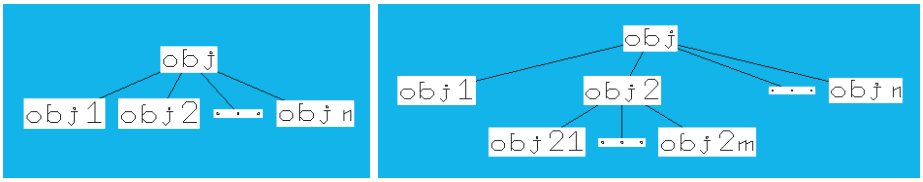

Figure 8.2 (a) Graph representation of a STRUCT assembly (b) Nested assembly

```
DEF obj = STRUCT:< obj1, STRUCT:< obj21, ... , obj2m >, ... , objn >
```

The sequence argument of the STRUCT operator may either contain or not affine transformations, together with polyhedral complexes. This fact results in generating an assembly either by using the same (global) coordinates for the various components or by using different (local) coordinate systems. The two cases are discussed in the two following subsections, respectively.

8.2.2 Assembly using global coordinates

Let us assume that the sequence argument of a STRUCT expression does not contain affine transformations. In other words we assume that the evaluations of the PLaSM expressions in the argument sequence return only polyhedral values.

In this case the output polyhedral complex is returned in the coordinate system of the first element of the input sequence, and no transformations of coordinates are applied to the assembly components, which are only aggregated within the same space, as shown by the following example.

Example 8.2.1 (STRUCT assembly (1))
The PLaSM expression given below returns the object displayed in Figure 8.3a. Notice that the local origin and coordinate axes of the three component shapes coincide. The @1 operator is used with the only purpose of generating the wire-frame drawing shown in the figure.

```
(@1 ~ STRUCT):< CUBOID:<2,2,2>, CUBOID:<1,1,1>, SIMPLEX:3 >
```

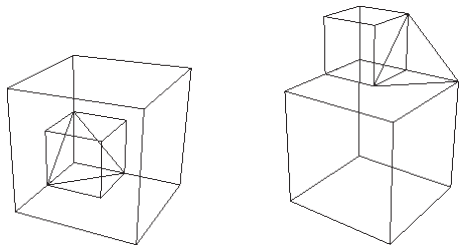

Figure 8.3 (a) Assembly without coordinate transformations (b) Assembly with coordinate transformations

8.2.3 Assembly with local coordinates

Let us conversely assume here that some affine transformations are contained within the sequence argument of a `STRUCT` expression. In this case, each transformation is applied to each polyhedral complex that follows it in the argument sequence.

From a user's viewpoint, the following equivalence holds, where $\texttt{pol}_1, \ldots, \texttt{pol}_n$ are polyhedral complexes and $\texttt{T}_1, \ldots, \texttt{T}_{n-1}$ are transformation tensors:

$$\texttt{STRUCT:<}\ \texttt{pol}_1, \texttt{T}_1, \texttt{pol}_2, \texttt{T}_2, \texttt{pol}_3, \ldots, \texttt{T}_{n-1}, \texttt{pol}_n\ \texttt{>} \equiv$$
$$\texttt{STRUCT:<}\ \texttt{pol}_1, \texttt{T}_1\texttt{:pol}_2, (\texttt{T}_1 \sim \texttt{T}_2)\texttt{:pol}_3, \ldots, (\texttt{T}_1 \sim \texttt{T}_2 \sim \cdots \sim \texttt{T}_{n-1})\texttt{:pol}_n\ \texttt{>}$$

Looking at the internal behavior of the geometric kernel of the language, the following transformation is applied to a `STRUCT` application at evaluation time:

$$\texttt{STRUCT:<}\ \texttt{pol}_1, \texttt{T}_1, \texttt{pol}_2, \texttt{T}_2, \texttt{pol}_3, \ldots, \texttt{T}_{n-1}, \texttt{pol}_n\ \texttt{>} \equiv \texttt{STRUCT:<}$$
$$\texttt{pol}_1,$$
$$(\texttt{T}_1 \sim \texttt{STRUCT})\texttt{:<}\ \texttt{pol}_2,$$
$$(\texttt{T}_2 \sim \texttt{STRUCT})\texttt{:<}\ \texttt{pol}_3,$$
$$\ldots$$
$$(\texttt{T}_{n-2} \sim \texttt{STRUCT})\texttt{:<}\ \texttt{pol}_{n-1}, \texttt{T}_{n-1}\texttt{:pol}_n\ \texttt{>}$$
$$\ldots\ \texttt{> > >}$$

Example 8.2.2 (STRUCT assembly (2))
Here we aggregate the same geometric components used in Example 8.2.1, but we also associate some transformations of coordinates, different from the identity, to the arcs of the graph representation of the resulting assembly.

In particular, let us describe their aggregation as done in the following.

```
(@1 ~ STRUCT):< CUBOID:<2,2,2>, T:3:2, CUBOID:<1,1,1>, T:2:1, SIMPLEX:3 >
```

Notice, looking at the geometric result shown in Figure 8.3b, that:

1. the output assembly is represented in the coordinate system of first cube;
2. the second cube is translated in z direction;
3. the unit tetrahedron is translated both in y and in z directions.

8.3 Traversal

The *traversal* of a hierarchical structure consists of a modified *Depth First Search* (DFS) of its acyclic multigraph,[3] where each arc — and not each node — is traversed only once. In particular, each node is traversed a number of times equal to the number of different paths that reach it from the root node.

The aim of the traversal algorithm is to "linearize" a structure network, by transforming all its substructures (i.e. all the subgraphs) from their *local coordinates* to the coordinates of the root node, assumed as *world coordinates*.

---

[3] Notice that the standard *dfs* graph traversal (see e.g. [AHU83]) visits all the nodes once, since it works by recursively visiting those sons of each node that it has not already visited.

For this purpose, a matrix denoted as the *current transformation matrix* (CTM) is maintained. Such a CTM is equal to the product of matrices associated with the arcs of the current path from the root to the current node. For the sake of efficiency, the traversal algorithm is implemented by using a stack of CTMs. When a new arc is traversed, the old CTM is pushed on the stack, and a new CTM is computed by (right) multiplication of the old one times the matrix of the arc. When unfolding from the recursive visit of the subgraph appended to the arc,[4] the CTM is substituted by the one popped from the stack. The TRAVERSAL algorithm is specified by some pseudo-language in Script 8.3.1.

Script 8.3.1 (Traversal of a multigraph)

```
algorithm TRAVERSAL ((N, A, f) : multigraph) {
    CTM := identity matrix;
    TraverseNode (root)
}

proc TRAVERSENODE (n : node) {
    foreach a ∈ A outgoing from n do TraverseArc (a);
    ProcessNode (n)
}

proc TRAVERSEARC (a = (n, m) : arc) {
    Stack.push (CTM);
    CTM := CTM * a.mat;
    TraverseNode (m);
    CTM := Stack.pop()
}

proc PROCESSNODE (n : node) {
    foreach object ∈ n do Process( CTM * object )
}
```

The CTM is normally used to (left) multiply the vertices of geometric objects stored in the traversed containers. But the reader should remember that equations of hyperplanes and normal vectors must be conversely (right) multiplied for the inverse of the applied transformation, according to the mapping of covectors discussed in Section 6.3.4. A double stack of matrices, where to push/pop both the CTM and its current inverse, may therefore speed up the traversal. As a result of the algorithm, a linearized model in world coordinates is produced, which may be used, e.g., for rendering purposes, as discussed in the next chapter.

Example 8.3.1 (Graph traversal)
In Figure 8.1a a directed multigraph over the set of nodes $\{1, 2, \ldots, 9\}$ is shown.

---

[4] Using a pictorial image, we could say: when the arc is traversed in the opposite direction.

According to a standard convention of hierarchical graph drawing [DBETT99], each edge should be considered as downwards oriented. In Figure 8.1b the tree generated when traversing the previous multigraph is given. As the reader may notice, it contains a higher number of node instances. In particular, the ordered set of nodes produced by the traversal algorithm discussed above is:

$$(6, 2, 6, 7, 3, 7, 7, 8, 9, 4, 7, 8, 9, 4, 9, 9, 9, 5, 1).$$

We would like to emphasize the fundamental property that *the number of instances of a node equates to the number of different paths that reach it from the root.* Notice, e.g., there are five different paths in the hierarchical multigraph of Figure 8.1a from root 1 to node 9, so that five instances of the node 9 are produced in the *linearized graph* generated at traversal time.

8.4 Implementations

We briefly summarize here some main aspects of management of structured assemblies in PHIGS and in VRML, mainly because the first standard introduced the conceptual model of hierarchical structures adopted by `PLaSM`, whereas VRML is used as main exporting format by our design language.

8.4.1 Structure network in PHIGS

A *structure* is defined as an ordered collection of structure elements, where a *structure element* is either:

1. an *output primitive*, say, a polyline, polymarker, text, fill area, fill area set, cell array or generalized drawing primitive (see Section 7.1.2);
2. an *affine transformation*, either through by setting a transformation matrix or by invoking some elementary transformation (say, a x-, y- or z-rotation, a scaling or a translation);
3. a *reference* to other structures. A reference behaves exactly like a procedure invocation in a programming language. When such a reference is encountered at traversal time, the current state of the graphics system is saved on a stack; the invoked procedure is executed, i.e. displayed; the saved state is then restored and finally the control is returned to the following structure element;
4. an *attribute* specification for either primitives or structures, i.e. a setting of some appearance characteristic of primitives or collections of primitives;
5. a *label*, i.e. a symbolic name associated with a numbered position within a structure, used to make the process easier and speed up the structure editing at run-time.

A collection of structures referring to each other is called *structure network*. This one has the topology of a directed acyclic graph. The processing of a structure network for display on a workstation by stepping through its structure elements is called the *structure network traversal*. While traversing the structure network, a set of information called the *traversal state list* is maintained, including the current values

of the *attributes* used to display the output primitives, and the current value of the *global modeling transformation*, that we called CTM, as well as the current value of the *local modeling transformation*, a sort of CTM internal to the current structure.

Remarks We note that:

- A structure is defined by invoking a `openStruct(`*`name`*`)` procedure and ended by invocation of a `closeStruct()` procedure. Structure definitions cannot be nested. Conversely, *references* to external structures are allowed within a structure, as invocations of the `executeStruct(`*`name`*`)` procedure.
- *Attribute specifications* modify the current values of the appropriate attributes. Attributes are values of properties (like, e.g. polyline color, text height or width, etc.) used to determine the appearance of output primitives encountered during the traversal. Current values of attributes are maintained in the traversal state list, and are saved on the stack when an `executeStruct()` element is encountered.
- *Labels* as structure elements are used to make the editing of structures at run-time easier. Since changes to a posted structure network are immediately displayed on a workstation, the editing of structures (i.e. the insertion, deletion or modification of structure elements) often results in some animation of the posted structure network.

Structure editing and animation As we already said, the animation of a posted structure network is obtained by editing the network, whose traversal is repeated several times at each time unit. Since changes of structure elements are immediately reflected on the workstation display, in order to animate a scene it will suffice to modify some element of posted structures, often by repeatedly changing some modeling transformation. In particular, a change to a modeling transformation may induce a change of the whole subgraph rooted on the edited node.

To obtain a continuous change of the scene appearance, i.e. a fluid animation, the frequency of traversal must exceed the latency of the perceived image on the observer′s retina. The traversal frequency will clearly depend on the complexity of the posted structure network and, in particular, on the number of output primitives contained there. Modern graphics hardware available as add-on for personal computers nowadays provides an amazing computational power, and is able to display scene graphs with more that one million of multiply-textured and shaded triangles at a traversal rate of 80 times for second, or to traverse 2 million triangles 40 times/second, and so on.

8.4.2 Hierarchical scene graph in VRML

The definition of hierarchies in VRML is obtained by using *grouping nodes*, and in particular by using `Transform` and `Group` nodes. The *Group* node has a similar use to the *Transform* node, but without including a transformation. Both nodes may contain any number of children nodes:

```
Group {
   bboxCenter 0 0 0 # SFVec3f
```

```
   bboxSize -1 -1 -1 # SFVec3f
   children [ ... ] # MFNode
}
```

The *bboxCenter* (bounding box center) and *bboxSize* (bounding box size) attributes are optionally used to speed up the operations, and in particular the hierarchical graph culling (see Section 10.3.2) of VRML viewers. The *children* field may contain either a single node or any number of ordered nodes.

Scene diagrams A useful tool for design and development of hierarchical assemblies is the drawing of scene diagrams, which help to visualize the structure of the scene. Nodes are shown as circles. In such diagrams dark circles are used for grouping nodes, i.e. for *Group* and *Transform* nodes, whereas light circles are used for the other types of nodes. According to a convention introduced by Open Inventor, the scene graph is drawn vertically, with children nodes aligned to the right of their father node, and connected to it by the same vertical line.

Example 8.4.1 (VRML scene graph)
In this example we introduce the VRML scene graph of the *Living room* example fully implemented in PLaSM in Section 8.5.1, and shown in Figure 8.5b.

The reader may easily figure out that a complete linear description, i.e. a complete tree of the scene may be too verbose and redundant, and also difficult to read. In order to make the description more compact and readable, it is customary to adopt a bottom-up approach based on separating the definition and the instantiation of scene components.

Let us therefore start our VRML coding by independently defining, within "local" coordinate systems, the elementary parts of our scene, i.e. the *Table* and *Chair* nodes. The *ArmChair* node is defined by invoke a transformed instance of the *Chair*. Let us remember that a *Shape* node usually contains an *appearance* field and a *geometry* field. Also remember the VRML rule that nodes are capitalized, whereas fields are not.

Script 8.4.1 (Table, Chair and ArmChair subgraphs)

```
DEF Table Shape {
   appearance Appearance  ...
   geometry Inline  url ...
} #Table

DEF Chair Shape {
   appearance Appearance  ...
   geometry inline  url ...
} #Chair

DEF ArmChair Transform {
   scale 1.2 1.2 1
   children USE Chair
} #ArmChair
```

The diagram of *Table* node defined in Script 8.4.1 is given in Figure 8.4a.

Analogously, to define the *Couch* node (Figure 8.4b), it is sufficient to invoking three instances of the `Chair` node, each one suitably rotated and translated. The VRML coding of the *Couch* as a *Group* node (and the subgraph rooted on it) is given in Script 8.4.2.

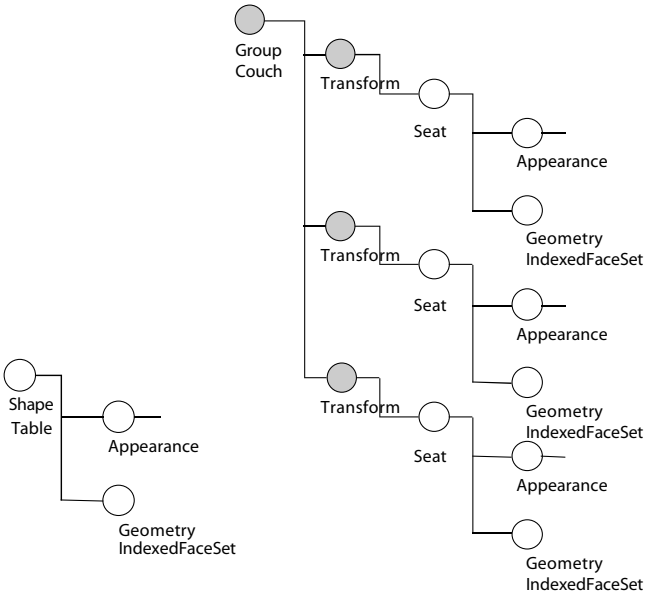

Figure 8.4 (a) Scene diagram for the *Table* node (b) Scene diagram for the *Couch* node

Script 8.4.2 (Couch subgraph)

```
DEF Couch Transform {
   scale 1.2 1.2 1
   children [
      USE Chair ,
      Transform {
         children USE Chair
         translation 6 0 0 },
      Transform {
         children USE Chair
         translation 12 0 0 }
   ] #children
} #Couch
```

The *Dinner* node, given in Script 8.4.3, is defined by an instance of *Table* node, translated to position its center in the origin, and by four instances of the *Chair* node, properly translated and rotated. A *Transform* node is used for this purpose where a translated chair is defined, denoted as *TranslChair*, which is further accessed three times, using different rotations, in the coordinate system of the *Dinner* node.

Script 8.4.3 (Dinner subgraph)

```
DEF Dinner Transform {
   translation -5 -5 0 },
   children [
      USE Table,
      Transform {
         rotation 0 0 1 1.57
         children
            DEF TranslChair
               Transform {
                  children USE Chair
                  translation -8 -2.5 0 } },
      Transform {
         rotation 0 0 1 3.14
         children USE TranslChair },
      Transform {
         rotation 0 0 1 4.71
         children USE TranslChair },
      Transform {
         rotation 0 0 1 6.28
         children USE TranslChair }
   ] #children
} #Dinner
```

Analogously, a *Conversation* object is defined as a *Group* node, by using the local coordinates of its first child *Table*, and containing also a rotated and translated instance of the *ArmChair* node and a translated instance of the *Couch* node.

Script 8.4.4 (Conversation subgraph)

```
DEF Conversation Group {
   children [
      USE Table ,
      Transform {
         translation 0 9 0
         rotation 0 0 1 -0.523
         center 3.6 3.6 0
         children USE ArmChair },
      Transform {
         translation 10 0 0
         children USE Couch }
   ] #children
} #Conversation
```

Finally, we give the root node of the scene graph, called *LivingRoom*, using the same local coordinate system of the *Conversation* node, and containing also a translated instance of the *Dinner* node.

Let us finally notice that the above VRML description of the scene was developed using a *bottom-up* approach, i.e. by starting from the more elementary components,

Script 8.4.5 (Living room subgraph)

```
DEF LivingRoom Group {
   children [
      USE Conversation ,
      Transform {
         children USE Dinner
         translation 30 30 0 }
   ] #children
} #LivingRoom
```

and by iteratively assembling them into more complex components. The above is the common development method when using scene-graph-based 3D development environments. Conversely, in the PLaSM description of the same scene discussed in the next section, we are able to use a *top-down* approach, starting from a high-level design of the scene, and making modifications until we produce the model of the scene with the desired level of detail. To work with a similar approach, a graphics application developer would need the support of some quite sophisticated 3D authoring software.

8.5 Examples

We discuss here some examples of quite complex assemblies in different application domains. The first one concerns the modeling of a living room; the second example discusses the modeling of a simplified human body; then a first implementation of the opening mechanism of a simplified umbrella is discussed. The umbrella example will be worked out with more detail in the chapters on curves and on surfaces, respectively. Finally, some strategies for generating complex assemblies by suitably aligning their component parts are introduced.

8.5.1 Living room modeling

In this section a top-down development of a 3D model of a (simplified) "living room" is presented. Our aim is to discuss a step-wise PLaSM refinement of quite a structured assembly. The example was already introduced in the previous section using VRML, with the further aim of comparing the "flavors" of the two languages when describing a quite complex scene graph.

In particular, a *LivingRoom* object is again defined as the aggregation of a *Dinner* and a *Conversation* objects. The first one contains a *Table* and four *Chair* objects; the second one contains another *Table*, an *ArmChair* and a *Couch*. The *Couch* is obtained by assembling three transformed instances of the *Chair* object.

In Script 8.5.1 a first draft of such definitions is given, by associating a transformation matrix to each object instance within an assembly. The reader may notice that the definitions in Script 8.5.1 are nothing more, nothing less, than a linguistic description of the information coded by the scene graph shown in Figure 8.5.

The above definitions actually specify a hierarchical relation *part_of* between the components of the LivingRoom assembly, by naming both each node of the graph and the matrices to be associated with the arcs. According to the discussion in Section 8.1.1, such matrices transform the geometry data contained in the second

Script 8.5.1 (Top-down design)

```
DEF LivingRoom = STRUCT: < T1:Dinner, T2:Conversation >
DEF Dinner = STRUCT: < T3:Table, T4:Chair, T5:Chair, T6:Chair, T7:Chair >
DEF Conversation = STRUCT: < T8:Table, T9:ArmChair, T10:Couch >
DEF ArmChair = STRUCT: < T11:Chair >
DEF Couch = STRUCT: < T12, Chair, T13:Chair, T14:Chair >
```

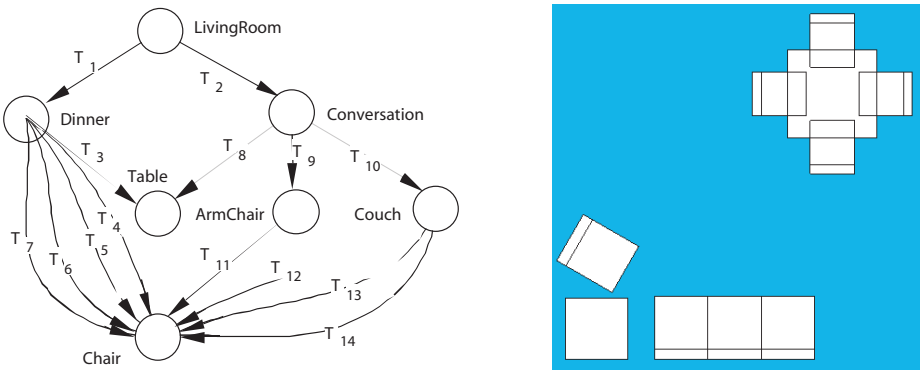

Figure 8.5 (a) Structure network (scene graph) of the `LivingRoom` design (b) 2D preliminary `LivingRoom`

node, into the coordinates of the first node of the arc.

Next we go on to detail both the geometric content of the scene components and the transformation matrices. We may assume in this phase that the scene is 2D.

Scene graph modeling

We start by setting, in Script 8.5.2, a first elementary specification of the basic parts of the scene; then we give a precise content, using local definitions, to the transformation matrices in Script 8.5.3.

Script 8.5.2 (First version - Basic parts)

```
DEF Chair = QUOTE:<1,5> * QUOTE:<5>;
DEF Table = CUBOID:<10,10>;
DEF ArmChair = S:<1,2>:<1.2,1.2>:Chair;
```

The higher-level assemblies in our scene are then fully detailed. The 1-skeleton of the resulting sketch of the scene, exported by the last expression of Script 8.5.3, is shown in Figure 8.5b.

A better implementation A much better implementation of `LivingRoom`, `Conversation` and `Dinner` assemblies is given in Script 8.5.4, by exploiting the very powerful semantics of structures introduced by PHIGS and inherited by `PLaSM` (see Section 8.2.3). Look in particular at the definition of the `Dinner` structure, given as a sequence that contains various instances of the `seat` geometric object and of the `Rxy`

Script 8.5.3 (First version - Assemblies)

```
DEF LivingRoom = STRUCT:< T1:Dinner, T2:Conversation >
WHERE
   T1 = T:<1,2>:<30,30>,
   T2 = T:1:0
END;

DEF Dinner = STRUCT:< T3:Table, T4:Chair, T5:Chair, T6:Chair, T7:Chair>
WHERE
   T3 = T:<1,2>:<-5,-5>,
   T4 = R:<1,2>:(1*PI/2) ∼ T:<1,2>:<-9,-2.5>,
   T5 = R:<1,2>:(2*PI/2) ∼ T:<1,2>:<-9,-2.5>,
   T6 = R:<1,2>:(3*PI/2) ∼ T:<1,2>:<-9,-2.5>,
   T7 = R:<1,2>:(4*PI/2) ∼ T:<1,2>:<-9,-2.5>
END;

DEF Conversation = STRUCT:< T8:Table, T9:ArmChair, T10:Couch >
WHERE
   T8 = S:<1,2>:<7/10,7/10>,
   T9 = T:2:9 ∼ (T:<1,2>:<3.6,3> ∼ R:<1,2>:(PI/-6) ∼ T:<1,2>:<-3.6,-3>),
   T10 = T:1:10
END;

DEF Couch = STRUCT:< T12, seat, T13:seat, T14:seat >
WHERE
   seat = (R:<1,2>:(PI/2) ∼ T:2:-5):Chair,
   T12 = S:<1,2>:<1.2,1.2>,
   T13 = T:1:5,
   T14 = T13 ∼ T13
END;

VRML:((STRUCT ∼ [ID,@1]):LivingRoom):'out.wrl';
```

transformation.

The associated multigraph would contain the `Dinner`, `Table` and `seat` nodes, an arc (with `ID` transformation) between `Dinner` and `Table` and four arcs between `Dinner` and `seat`. The transformation tensors associated with such four arcs would be `ID`, `Rxy`, `Rxy∼Rxy` and `Rxy∼Rxy∼Rxy`, respectively.

The simplification obtained in Script 8.5.4 is partially due to the use of local modeling transformations even within the single structure and also partially due to a better positioning of the local origin within the elementary parts of the assembly. In particular, the new definitions given in Script 8.5.5 set the origin of local systems to the midpoint of containment boxes.

3D scene modeling

A quite detailed 3D modeling of the elementary parts of the scene is given in Script 8.5.6. The `Chair` object is decomposed into the assembly of a frame, a seat and a back, colored with black, yellow and white predefined colors, respectively. The `Table` object is decomposed into the assembly of four rotated instances of a

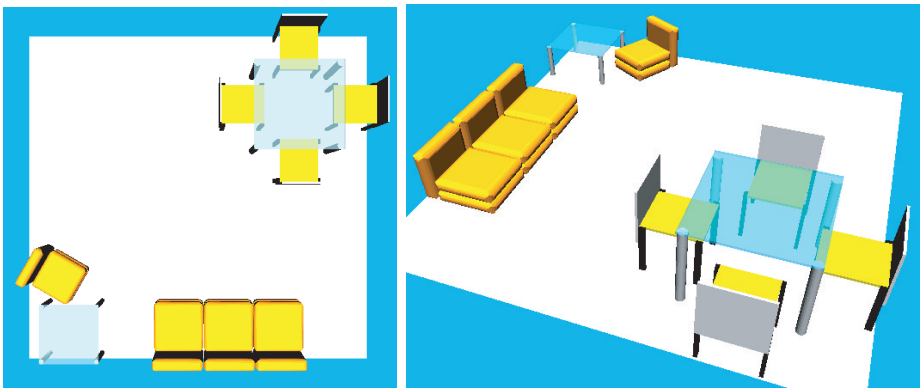

Figure 8.6 (a) Top view of the 3D version (b) View from the `dinner` angle

Script 8.5.4 (Better assembly)

```
DEF LivingRoom = STRUCT:< Conversation, T:<1,2>:<27,27>, Dinner >;

DEF Conversation = STRUCT:< table2, T9:ArmChair, T10:Couch >
WHERE
   table2 = S:<1,2>:<0.7,0.7>:Table,
   T9 = T:2:9 ~ R:<1,2>:(PI/-6),
   T10 = T:1:10
END;

DEF Dinner = STRUCT:< Table, seat, Rxy, seat, Rxy, seat, Rxy, seat >
WHERE
   Rxy = R:<1,2>:(PI/2),
   seat = T:1:-5:Chair
END;
```

translated cylindrical `leg` and one superimposed tiny plane of colored `glass`. The more interesting model is associated with the *ArmChair* symbol, where a local `basis` produces a properly dimensioned instance of a "parametrized puff", i.e. a cushion parametrized by the sizes of its containment box.

As it possible to see in Figure 8.6b, the `ArmChair` assembly is made by three translated and rotated instances of the `cushion` located at the cushion angles. The `Q` version of the `QUOTE` operator was defined in Script 1.5.5. Script 8.5.7 provides the `puff` function, used to generating the component cushions of both the `ArmChair` and the `Couch`.

Notice that such parametrized cushion is generated by `JOIN` of four rotated instances

Script 8.5.5 (Better parts)

```
DEF Couch = STRUCT:< seat, T:1:6, seat, T:1:6, seat >
DEF seat = R:<1,2>:(PI/2):ArmChair
DEF Chair = (T:<1,2>:<-3,-2.5>):(QUOTE:<1,5> * QUOTE:<5>);
DEF Table = (T:<1,2>:<-5,-5> ~ CUBOID):<10,10>;
DEF ArmChair = S:<1,2>:<1.2,1.2>:Chair;
```

Script 8.5.6 (Detail modeling — parts)

```
DEF Chair = STRUCT:< frame, seat, back >
WHERE
   frame = Q:<0.5,-4,0.5> * Q:<0.3,-4.4,0.3> * Q:4.5 COLOR BLACK,
   seat = Q:5 * Q:<-0.3,4.4> * Q:<-4.3,0.2> COLOR YELLOW,
   back = T:1:-0.2:(Q:0.2 * Q:5 * Q:<-3,5>) COLOR WHITE
END;

DEF Table =
   (T:<1,2>:<4.6,4.6> ~ STRUCT):< leg, Rz, leg, Rz, leg, Rz, leg >
   TOP (CUBOID:<10,10,0.2> MATERIAL glass)
WHERE
   leg = (T:<1,2>:<-4.6,-4.6> ~ Cylinder):<0.4,7,18> ,
   Rz = R:<1,2>:(PI/2),
   glasscolor = RGBCOLOR:<0.2,0.6,1>,
   glass = BASEMATERIAL:< glasscolor,glasscolor,0.2,BLACK,0.2,0.6 >
END;

DEF ArmChair = S:<1,2,3>:<5/8,5/8,5/8>:
   ((T:3:4 ~ R:<1,3>:(PI/2)):cushion RIGHT (cushion TOP cushion))
WHERE
   cushion = puff:<6,6,2> COLOR RGBCOLOR:<1,0.5,0>
END;

DEF Seat = ArmChair;
```

Script 8.5.7 (Detail modeling — puff)

```
DEF puff (a,b,c::IsReal) = (JOIN ~ STRUCT ~ ##:4):< theAngle, Rz >
WHERE
   Rz = R:<1,2>:(PI/2),
   cyl = (T:3:(c/-2) ~ cylinder):<c/2, c, 18>,
   corner = &:< cyl, R:<1,3>:(PI/2):cyl, R:<2,3>:(PI/2):cyl>,
   theAngle = T:<1,2>:<a/-2,b/-2>:corner
END;
```

of `theAngle`, a translated copy of `corner` generated by intersection of three rotated cylinders. The shape thus generated closely recalls at the corners the hand-made seams of real-world cushions, and gives more realistic results than using a small sphere — see Figure 8.6b.

Storage comparison It may be interesting to compare the storage used by the PLaSM description of the 3D `LivingRoom` discussed in this section, with the storage of the VRML file generated by it and displayed in Figure 8.6. On the Mac OS X file system, the first one needs 4KB, whereas the second one needs 112KB. The comparison between the number of characters contained in such files is even more unbalanced, going from $1,935$ to $103,621$ characters, with a ratio close to $\frac{1}{53}$.

8.5.2 Body model

A simplified model of the human body as a hierarchical assembly with rotational joints between parts is discussed in this section. A hierarchical structure is assumed as the basis of the modeling, where the `Body` symbol denotes the structure root. The body's main substructures, respectively called `top_limb`, `upper_limb` and `lower_limb`, are attached to it. The last two are clearly instanced twice. It is also assumed that all the degrees of freedom are rotational. In other words it is assumed that only rotations are allowed by body joints.

Anthropomorphic robot

The upper level object, i.e. the root of the assembly, is given as a `STRUCT` expression named `Body`. The interesting part of the definition is how the joint conditions are defined, using the `Tensor` function. The particular shapes of the body components will be detailed in Script 8.5.13. The simplest shape for each part is a parallelepiped. The resulting object, resembling an anthropomorphic robot, is shown in Figure 8.7.

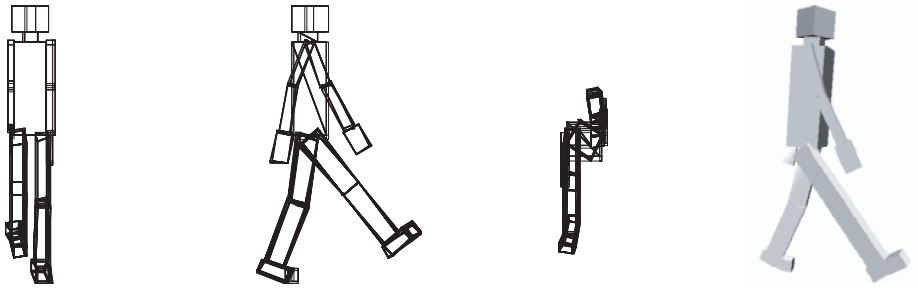

Figure 8.7 Front, side, top and dimetric projections of the anthropomorphic robot

The `Tensor` function is first applied to a sequence of integers, which specify the `axes` of rotations (either x, y or z, respectively corresponding to input values 1,2 or 3) allowed by a joint, and the corresponding `angles` in degrees. The output of the function is a transformation tensor, generated by composition of elementary rotations. Notice that `angles`, entered in degrees, are transformed to `convertedAngles` in radians. The `scalarVectProd` operator is given in Script 2.1.20.

Script 8.5.8 (Composite rotations)

```
DEF Tensor (axes::IsSeqOf:IsInt) (angles::IsSeqOf:IsReal) =
   (COMP ~ AA:APPLY ~ TRANS): < rotations, convertedAngles >
WHERE
   rotations = AS:SEL:axes:< R:<2,3>,R:<1,3>,R:<1,2> >,
   convertedAngles = PI/180 scalarVectProd angles
END;
```

In Script 8.5.9 the `Body` structure is given, i.e. the root of the structure network discussed in this section. The children structures `top_limb`, `upper_limb` and

`lower_limb` are invoked as functions, and applied to the actual values of the angles which determine their configuration. Notice that the rotations of each body's limb are about its local origin. Each joint which connects a limb to `torso` is then properly translated to its position in the body's coordinate system by a local $\texttt{joint}_i$ function.

Script 8.5.9 (Assembly root)

```
DEF Body = STRUCT:< torso,
   (joint1 ~ top_limb):<<0,0>,<0,0,-30>>,
   (joint2 ~ upper_limb):<<20,0,0>,<10,0>,<0>>,
   (joint3 ~ upper_limb):<<-20,0,0>,<10,0>,<0>>,
   (joint4 ~ lower_limb):<<40,0,-10>,<0>,<0>,<0>>,
   (joint5 ~ lower_limb):<<-10,0,10>,<-10>,<0>,<20>> >
WHERE
   torso = T:<1,2>:<-5,-5>:torso_shape,
   joint1 = T:3:30,
   joint2 = T:<1,3>:<6,30>,
   joint3 = T:<1,3>:<-6,30>,
   joint4 = T:1:4,
   joint5 = T:1:-4
END;
```

The `top_limb` structure, given in Script 8.5.10, contains a `neck` and a `head` object, respectively connected by two and three rotational degrees of freedom, which are passed via the `dof` (Degrees Of Freedom) formal parameter. Notice that the `neck` is allowed to rotate about its local x and y axes, whereas the `head` may rotate about x, y and z.

Script 8.5.10 (Neck and head)

```
DEF top_limb (dof::IsSeqOf:IsSeq) = STRUCT:<
   rot1, pos1:neck, joint, rot2, pos2:head >
WHERE
   pos1 = T:<1,2>:<-2.5,-2.5>,
   pos2 = T:<1,2>:<-4,-4>,
   rot1 = Tensor:<1,2>:(S1:dof),
   rot2 = Tensor:<1,2,3>:(S2:dof),
   joint = T:3:3
END;
```

The `upper_limb` parametric structure given in Script 8.5.11 connects into a kinematic chain the `upper_arm` to the `lower_arm` and the latter to the `hand` objects, and connects them with three, two and one rotational degrees of freedom, passed as subsequences of the `dof` sequence, respectively. Notice, e.g., that the `hand` object is positioned with respect to its local origin by the `pos3` tensor, then rotated about its origin by the `rot3` tensor. Finally, the local origin, i.e. the position of the rotational joint, is translated within the local system of its father object (in this case `lower_arm`) by the `joint2` tensor within the hierarchical assembly which models the kinematic

chain of the body arm. A similar pattern of transformations is used for the subsequent subassemblies and joints, i.e. for lower_arm and upper_arm.

Script 8.5.11 (Arm and hand)

```
DEF upper_limb (dof::IsSeqOf:IsSeq) = STRUCT:<
   rot1 ~ pos1, upper_arm,
   joint1 ~ rot2 ~ pos2, lower_arm,
   joint2 ~ rot3 ~ pos3, hand >
WHERE
   rot1 = Tensor:<1,2,3>:(S1:dof),
   rot2 = Tensor:<1,3>:(S2:dof),
   rot3 = Tensor:<2>:(S3:dof),
   pos1 = T:<1,2,3>:<-1,-1.5,-15>,
   pos2 = T:<1,2,3>:<-1,-1.5,-15>,
   pos3 = T:3:-8,
   joint1 = T:<1,2>:<1,1.5>,
   joint2 = T:<1,2>:<0.5,-1>
END;
```

Analogously, the lower_limb function given in Script 8.5.12 is a parametric structure, joining a big toe to the feet and this one to the lower_leg and then to the upper_leg, which is finally positioned into the local reference system of the lower_limb by the transformation tensors named pos1 and rot1.

Script 8.5.12 (Leg and feet)

```
DEF lower_limb (dof::IsSeqOf:IsSeq) = STRUCT:<
   rot1 ~ pos1, upper_leg,
   joint1 ~ rot2 ~ pos2, lower_leg,
   joint2 ~ rot3 ~ pos3, feet,
   joint3 ~ rot4, toe >
WHERE
   pos1 = T:<1,2,3>:<-2,-3,-20>,
   pos2 = T:<2,3>:<-3,-20>,
   pos3 = T:2:-1,
   rot1 = Tensor:<1,2,3>:(S1:dof),
   rot2 = Tensor:<1>:(S2:dof),
   rot3 = Tensor:<1>:(S3:dof),
   rot4 = Tensor:<1>:(S4:dof),
   joint1 = T:2:3,
   joint2 = T:3:-4,
   joint3 = T:2:9
END;
```

The sort of anthropomorphic robot depicted in Figure 8.7 is finally generated by assigning a parallelepiped shape to each Body part. The detailed definitions of all the component shapes are given in Script 8.5.13.

Script 8.5.13 (Body components)

```
DEF torso_shape = CUBOID:<10,10,30>;
DEF head = CUBOID:<8,8,8>;
DEF neck = CUBOID:<5,5,3>;
DEF upper_arm = CUBOID:<2,3,15>;
DEF lower_arm = CUBOID:<2,3,15>;
DEF hand = CUBOID:<1,5,8>;
DEF feet = CUBOID:<4,9,4>;
DEF toe = CUBOID:<4,3,4>;
DEF lower_leg = CUBOID:<4,6,20>;
DEF upper_leg = CUBOID:<4,6,20>
```

Note The important part of the structured assembly discussed above is the hierarchical relationship between its parts, and the chains of transformations between the local coordinate systems induced by the joint conditions. The shapes of the elementary parts may easily be changed without requiring any change of the upper-level structures, as it is shown by the two variations of the body model given in Figure 8.8.

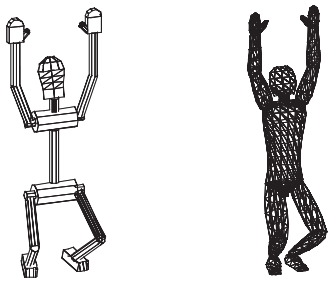

Figure 8.8 Two configurations of the `Body` model with different definitions for the elementary parts and same values for the joint angles

Actually, in order to be really invariant with respect to the dimensions and shapes of lower-level subassemblies and components, the $\texttt{pos}_i$ tensors should be moved on top of the children structures, and the $\texttt{joint}_i$ tensors should be written as parametric functions of the dimensions of the local root.

8.5.3 Umbrella modeling (1): structure

The goal of the geometric programming example given here is the generative modeling of a *parametric umbrella*, parametrized on the opening angle. In particular, we discuss the modeling from scratch of the kinematic mechanism, using wire-frame parts. In later chapters, this model will be step-wise refined by:

1. curving some rods as quadratic Bézier curves, depending on the opening angle of the umbrella (see Section 11.5.2);
2. modeling the umbrella canvas as Coon′s patches delimited by polynomial curves of degrees 2, 1 and 0 (see Section 12.6.1);
3. modeling all the umbrella rods as solid parts, to substitute their previous

definitions as wire frames (see Section 13.6.1).
4. animating the various umbrella versions (see Section 15.6.1).

This example aims to demonstrate that the PLaSM language, conversely than other development environments for graphics programming and virtual reality, allows for progressive refinement of working models and *top-down development*. The authors believe this fact should be the very distinctive feature of a *design language*.

Rod and axis modeling A 1D Rod of length len is defined along the z-axis in $\mathbb{R}^3$ by using the primitive constructor MKPOL. The core RodPair of the moving mechanism (shown in Figure 8.9a) is then described as a function depending on the height h and on the opening angle alpha. For this purpose two instances of the Rod primitive are properly combined with affine transformations. According to the semantics of standard ISO PHIGS structures, both Rod1 and Rod2 are defined in local coordinates and jointly transformed into world coordinates.

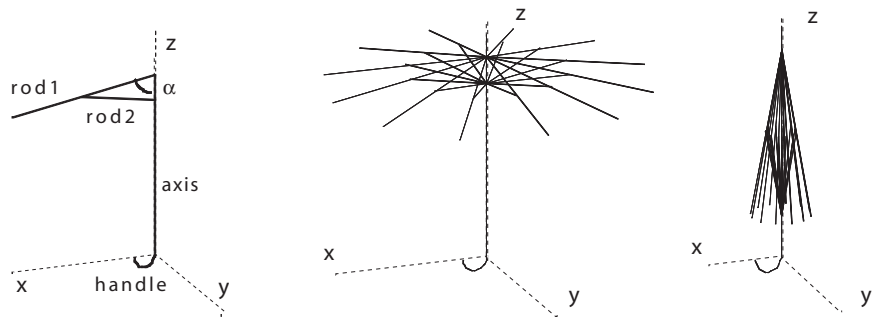

Figure 8.9 (a) Model generated by (STRUCT ∼ [Axis ∼ S1,Handle ∼ S1,RodPair]):<10,80> (b) Value of Umbrella:<10,80> (c) Value of Umbrella:<10,15>

Script 8.5.14 (Umbrella (1a))

```
DEF Rod (len::IsReal) = MKPOL:<<<0,0,0>,<0,0,len>>,<<1,2>>,<<1>>>;
DEF Axis (len::IsReal) = Rod:len;

DEF RodPair (h, alpha::Isreal) = STRUCT:<
   T:3:h, R:<3,1>:(-:alphaRad), Rod1,
   T:3:(-:AB), R:<3,1>:(2*alphaRad), Rod2 >
WHERE
   alphaRad = alpha*PI/180,
   Rod1 = S:3:-1:(Rod:(2*AB)),
   Rod2 = S:3:-1:(Rod:AB),
   AB = h*4/10
END;
```

Parametric umbrella The whole parametric `Umbrella` is then defined as a structure by catenating a sequence with one `Axis` and `Handle` and 12 pairs, each containing a rotation of $\pi/6$ around the z-axis and one instance of the `RodPair` model previously defined.

In particular, the `Handle` is defined as a properly positioned halfcircle linearly approximated with 12 segments. The whole `Umbrella` model at this stage is shown in Figure 8.9 for two different values of the opening angle.

Script 8.5.15 (Umbrella (1b))

```
DEF Umbrella (h, alpha::Isreal) = (STRUCT ~ CAT):
   <[Axis, Handle]:h, ##:12:< RodPair:<9/10*h,alpha>, R:<1,2>:(PI/6) > >;

DEF Handle (h::Isreal) = (T:1:Radius ~ S:<1,3>:<Radius,Radius>):
   (MAP:[COS ~ S1, K:0, SIN ~ S1]:dom )
WHERE
   dom = T:1:PI:(QUOTE:(#:12:(PI/12))),
   Radius = h/18
END;
```

8.5.4 Tree diagrams

In this section we develop and discuss a useful set of functions used to produce a graphical representation of a hierarchy, or, in other words, to perform some *tree drawing*. In particular, we assume here that every tree node is associated with a string. The given drawing approach can be slightly modified to produce a graphical representation of more general hierarchies, where a node may be associated to any kind of polyhedral complex of dimension 2 or 3. This approach could get useful to generate the hierarchical diagram of the *product model* in some industrial applications.

Drawing strategy

Very simple drawing rules are used, and no optimization of the drawing area is attempted. Two different styles for the drawing of the father-son relationship are alternatively implemented. The resulting diagrams of the same tree are shown in Figures 8.9a and 8.9b. The two simple style rules used in the following implementation may be summarized as follows:

1. the containment boxes of the subtrees rooted in a node (called brother subtrees in the following) are aligned with the top edges;
2. the containment boxes of brother subtrees are equally spaced.

Two more geometrical rules drive the whole drawing algorithm:

3. each node is defined in local coordinates, with the origin of the local system positioned in the centroid of the containment box of the node;
4. each subtree is defined in the local coordinates of its root node;

Implementation

In implementing the above drawing strategy, three main steps can be abstracted, concerning respectively (a) the drawing of a node; (b) the drawing of a subtree; and the (c) bottom up streaming aggregation of drawn subtrees, until one can draw or, better, generate a geometric model of the diagram of the whole input tree.

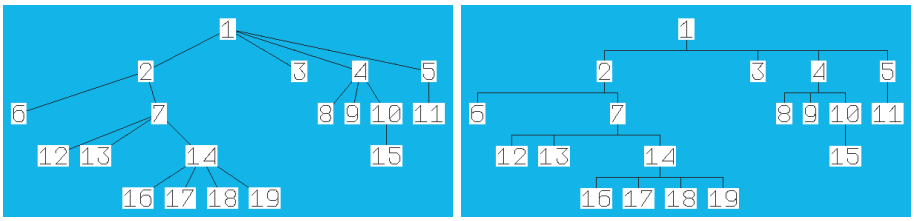

Figure 8.10 Two graphical representations of the same tree

Tree input The tree shown in Figure 8.10, using two different drawing styles for the arcs outgoing from each node, is coded in Script 8.5.16.

In particular, a tree is defined as a *sequence of levels*, where each level is a sequence of sequences of nodes. There are as many elements (subsequences) in a level as there are nodes in the previous tree level, so to define a *one-to-one mapping* between *nodes* and sequences of their *children*. Therefore, a leaf node is associated with an empty sequence in the next level, a node with two children is associated with a sequence with two nodes, and so on. Also, each node is represented by a `PLaSM` string. Clearly, there are as many (top-level) sequences in this description as there are levels in the tree.

Script 8.5.16 (Tree input)

```
DEF tree = <
   <<'1'>>,
   <<'2','3','4','5'>>,
   <<'6','7'>,<>,<'8','9','10'>,<'11'>>,
   <<>,<'12','13','14'>,<>,<>,<'15'>,<>>,
   <<>,<>,<'16','17','18','19'>,<>>
>;

VRML:(drawTree:tree):'out.wrl';
```

Notice that the diagrams of Figure 8.10 were produced by the last expression of Script 8.5.16.

Node drawing We assume here that each tree node is described by a label of type `string`, so that the `TEXT` operator defined in Script 7.2.13 can be applied to it. In Script 8.5.17 the `drawNode` operator is given for this purpose, which adds a properly scaled rectangle to the graphical text produced and centered about its midpoint by the `centerLabel` function.

Script 8.5.17 (Node drawing)

```
DEF Label = STRUCT ∼ [T:<1,2> ∼ AA:- ∼ MED:<1,2>, ID ] ∼ TEXT;
DEF drawNode = STRUCT ∼ [ ID, K:(S:<1,2>:<1.2,1.6>), BOX:<1,2> ] ∼ Label;
```

More general assumptions about the nature of the nodes might be postulated and easily implemented, with the aim of generating the diagram as a polyhedral complex of suitable dimension.

Subtree drawing The main design choices of the subtree drawing algorithm given in Script 8.5.18 concern the definition of each subtree in the local coordinate space of its own root node, with a local origin $(0, 0)$ coinciding with the center of the root′s containment box.

Notice that the discussed drawing strategy is *bottom-up*, from leaf nodes in the lowest hierarchical level, up to the root node in the top tree level. The main algorithm is codified by the `drawSubtree` function, which is repeatedly called by the `draw_level` function in Script 8.5.19.

The `drawSubtree` algorithm just aggregates the two structures returned by the `moved_sons` function and by the `drawEdges` function. In particular, the last one returns a set of polylines between the local origin and the `sonCenters`. The `moved_sons` operator returns the structure produced by equispacing the `subtree`(s) of its `son` arguments. The `translations` to be applied between adjacent `son` pairs are computed on the fly, by analyzing the `pairs` of already modeled subtrees.

Script 8.5.18 (Subtree drawing)

```
DEF pairs = CONS ∼ AA:CONS
   ∼ TRANS ∼AA:(AA:SEL∼FROMTO)∼[[ k:1,len-k:1],[ k:2,len ]];
DEF size_pairs = APPLY ∼ [pairs, ID] ∼ AA:(SIZE:1);
DEF translations = AA:(T:1 ∼ C:+:dx ∼ /∼[ID,K:2] ∼ +)
   ∼ AR ∼ [ID, K:0] ∼ size_pairs;
DEF last_transl (n::IsPol) = T:<1,2>
   ∼ [(SIZE:1 / K:-2) + K:(SIZE:1:n / 2), K: dy];

DEF drawSubtree = STRUCT ∼ [ moved_sons, drawEdges ];
DEF subtree = STRUCT ∼ CAT ∼ TRANS ∼ [ ID, translations ];
DEF moved_sons (sons::isSeqOf:isPol) =
   (STRUCT ∼ [ last_transl:(FIRST: sons), ID ] ∼ subtree):sons;

DEF drawEdges (sons::isSeqOf:isPol) =
   (STRUCT ∼ AA:Polyline ∼ DISTL): < <0,0>, sonCenters >
WHERE
   sonTransl = (AL∼[last_transl:(S1:sons)∼subtree, translations]): sons,
   sonCenters = (S1 ∼ UKPOL ∼ STRUCT ∼ CAT ∼ DISTR):<sonTransl,MK:<0,0>>
END;
```

Tree drawing The top-level operator, which is invoked to produce a diagram of a tree, is clearly named `drawTree`, and is given in Script 8.5.19. It is quite interesting,

because does not use either explicit iteration or recursion to make its work, but instead a pure FL style based on a sort of stream processing. The drawTree behavior can be summarized as follows.

1. First, the input tree is preprocessed by "drawing" every node, i.e. by transforming every string into a 2D polyhedral complex.
2. Then, the n levels in the input tree are paired bottom-up, and each level pair is repeatedly transformed into a single level, by reducing each corresponding (node, children sequence) into a single polyhedral complex, by using for this purpose the drawSubtree operator. This operation is repeated $n - 1$ times.
3. Finally, some house-cleaning is performed, in order to extract the 2D output complex from the generated data structure.

Script 8.5.19 (Tree drawing)

```
DEF drawTree (levels::IsSeq) =
   (S1 ~ S1 ~ S1 ~ COMP:(#:(n - 1):reducePair AR preProcess)):levels
WHERE
   n = LEN: levels,
   preProcess = REVERSE ~ AA: draw_leafs,
   reducePair = AL ~ [structMapping, TAIL~TAIL]
      ~ AL~[ AA: pairing ~ TRANS ~ [draw_level ~ S1, CAT ~ S2], TAIL ]
END;

DEF draw_leafs = AA:(IF:< isVoid, K:<>, AA:drawNode >);
DEF draw_level = AA:(IF:< isVoid, K:<>, drawSubtree >);
DEF pairing = IF:< isVoid ~ s1, s2, STRUCT ~ [s2,s1] >;
DEF structMapping = APPLY ~ [CONS ~ AA:(AS:SEL) ~ select ~ S2, S1];
DEF select = AA:(FROMTO ~ [ + ~ [- ~[+ , LAST],K:1], + ])
   ~ APPLY
   ~ [CONS ~ AA:(AS:SEL ~ INTSTO)~INTSTO ~ LEN, ID]
   ~ AA:LEN;
```

Drawing subtree forks The standard drawing style for a tree diagram puts a segment between non-leaf nodes and each of their children. More unusually, a single "fork" instead goes out from every non-leaf node, and enters each of the children of the node. The drawForks function given in Script 8.5.20 implements such a drawing style, and is intended to substitute the drawEdges call in the drawSubtree function of Script 8.5.18. An example of diagram with forks is given in Figure 8.10b. In this case a sort of "Manhattan" path is used when drawing edges between the starting point and the ending points. The mean function was given in Script 4.4.8.

8.5.5 Array of aligned graphics symbols

Our aim in this section is to discuss how to build complex arrangements of variously aligned graphics objects or symbols.

Script 8.5.20 (Subtree forks)

```
DEF drawForks (sons::isSeqOf:isPol) = (STRUCT ∼ AA:Polyline ∼ CAT):
   < LIST:middleExtremes, manhattan:<<0,0>, Meanpoint >,
      TRANS:< sonCenters, addedPoints >>
WHERE
   middleExtremes = DISTR:< [ S1∼first, S1∼last ]: sonCenters , dy / 2 >,
   Meanpoint = (AA: mean ∼ TRANS): middleExtremes,
   addedPoints = AA:S1: sonCenters DISTR dy / 2,
   manhattan = [[S1, [s1∼s1,s2∼s2]], [[s1∼s1,s2∼s2], S2]]
   sonTranslations =
      (AL ∼ [ last_transl:(FIRST: sons) ∼ subtree, translations ]): sons,
   sonCenters = (S1 ∼ UKPOL ∼ STRUCT ∼ TAIL ∼ CAT ∼ DISTL):
      < MK:<0,0>, sonTranslations >,
END;
```

For this purpose we first prepare a simplified 3D model of a house, given in Script 8.5.21, as a polyhedral complex assembling few 3D convex polygons. Such a `house` model is then projected in various ways, using methods discussed in Chapters 9 and 10, so generating several different 2D projections, shown in Figure 8.11. We do not discuss here how the projections are generated. An extensive discussion of standard projections may be found in Chapter 10. We are only interested here in the relative arrangements of the 2D projections.

Script 8.5.21 (House model)

```
DEF house = MKPOL:< verts,cells,pols >
WHERE
   verts = < <0,0,0>,<0,0,10>,<0,5,13>,<0,10,10>,<0,10,0>,
             <10,0,0>,<10,0,10>,<10,5,13>,<10,10,10>,<10,10,0>,
             <10,4,0>,<10,4,8>,<10,6,0>,<10,6,8>,
             <6,10,4>,<4,10,4>,<6,10,8>,<4,10,8> >,
   cells = < 15..18,<5,4,9,10>,11..14,6..10,<3,4,8,9>,
             <2,3,7,8>,<1,2,6,7>,1..5 >,
   pols  = < <1>,<2>,<3>,<4>,<5>,<6>,<7>,<8> >
END;
```

Each projection of the 3D `house` model generated (and assembled) by Script 8.5.22 produces a different 2D graphics object. In particular, a set of 15 different projections is organized in Script 8.5.22 as a 2D array of symbols with 5 rows and 3 columns. For this purpose the generating expressions (enclosed in round parentheses) are grouped into 5 subsets as `STRUCT` expressions, in turn assembled by an external `STRUCT` operator, together with translation tensors that relocate the graphics objects they refer to in proper locations of the 2D arrangement.

Notice that, according to the semantics of PHIGS structures, each `T:1:28` tensor in Script 8.5.22 applies *only* to graphics objects which follow it within the argument sequence the tensor belongs to.

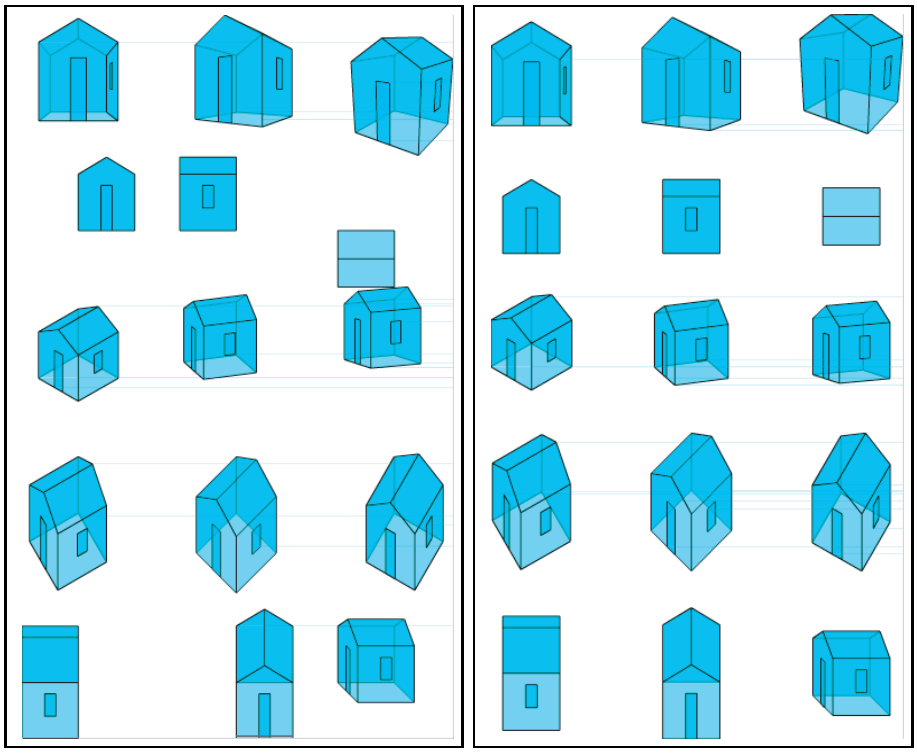

Figure 8.11 Flash exporting: (a) array of pictures, with alignment of origins of *local* systems (b) array of pictures *centered* around their (aligned) origins

Script 8.5.22 (array of artworks)

```
DEF MxMy = STRUCT ~ [T:<1,2> ~ AA:- ~ MED:<1,2>, ID];

DEF out = STRUCT:<
   STRUCT:<
      MxMy:(projection: perspective: onepoint: house), T:1:28,
      MxMy:(projection: perspective: twopoints: house), T:1:28,
      MxMy:(projection: perspective: threepoints: house) > ,
   T:2:-25, STRUCT:<
      MxMy:(projection: parallel: orthox: house), T:1:28,
      MxMy:(projection: parallel: orthoy: house), T:1:28,
      MxMy:(projection: parallel: orthoz: house) > ,
   T:2: -22, STRUCT:<
      MxMy:(projection: parallel: isometric: house), T:1:28,
      MxMy:(projection: parallel: dimetric: house), T:1:28,
      MxMy:(projection: parallel: trimetric: house) > ,
   T:2: -28, STRUCT:<
      MxMy:(projection: parallel: leftCavalier: house), T:1:28,
      MxMy:(projection: parallel: centralCavalier: house), T:1:28,
      MxMy:(projection: parallel: rightCavalier: house) > ,
   T:2: -30, STRUCT:<
      MxMy:(projection: parallel: xCavalier: house), T:1:28,
      MxMy:(projection: parallel: yCavalier: house), T:1:28,
      MxMy:(projection: parallel: cabinet: house) >
>;
```

Picture exporting The pictures shown in Figure 8.11 are produced by a web browser, e.g. *Microsoft Internet Explorer* or *Netscape Navigator*, equipped with a Flash plug-in, by loading the `out.swf` file exported by PLaSM at the evaluation of the expression given in Script 8.5.23. To successfully evaluate such expression, the PLaSM environment must have already loaded the library called `flash.psm`, where the definitions of the primitive functions FILLCOLOR, LINECOLOR and LINESIZE are contained. Notice that the argument sequence of FILLCOLOR and LINECOLOR contain 4 numbers in the interval $[0, 1]$, according to the $RGB\alpha$ (*Red*, *Green*, *Blue*, *Transparency*) color model used by Flash.

Script 8.5.23 (Flash exporting)

```
flash:( out
   FILLCOLOR RGBAcolor:< 0,1,1,0.5 >
   LINECOLOR RGBAcolor:< 0,0,0,1 >
   LINESIZE 1 )
:300:'out.swf';
```

Alignment operators

The picture shown in Figure 8.11b is generated by direct evaluation of the PLaSM code given above. The reader may find very instructive to re-evaluate the exporting

expression 8.5.23 after some editing and re-evaluation of Script 8.5.22. In particular, the reader should cancel all the instances of the string "`MxMy:`", i.e. all the applications of the function `MxMy` — that stands for *Middle x, Middle y* — to the symbol generating expressions. The specific aim of this operator, defined in Script 8.5.22 as

```
DEF MxMy = STRUCT ~ [T:<1,2> ~ AA:- ~ MED:<1,2>, ID]
```

is to center a 2D object around its local origin.

This operation is accomplished as follows by the `MxMy` operator:

1. compute, by using the `MED:<1,2>` operator, the middle point $\boldsymbol{p}$ of the containment box of the symbol;
2. then, generate the translation vector $\boldsymbol{o} - \boldsymbol{p}$, by just reversing the sign of all the $\boldsymbol{p}$ components;
3. finally, produce a `STRUCT` expression that contains the `T:<1,2>:`$-\boldsymbol{p}$ tensor as well as the symbol we want to relocate.

Several different *alignment operators* might easily be defined by substituting the `MED:<1,2>` operator with a different function, with the intent of suitably generating a different translation vector to be applied to the input symbol. For example, a graphics symbol can be usefully relocated by using one of the operators defined in Script 8.5.24.

Script 8.5.24 (Alignment operators)

```
DEF MxBy = STRUCT ~ [T:<1,2> ~ AA:- ~ [MED:1,MIN:2], ID]
DEF MxTy = STRUCT ~ [T:<1,2> ~ AA:- ~ [MED:1,MAX:2], ID]
DEF LxMy = STRUCT ~ [T:<1,2> ~ AA:- ~ [MIN:1,MED:2], ID]
DEF RxMy = STRUCT ~ [T:<1,2> ~ AA:- ~ [MAX:1,MED:2], ID]
```

Where:

1. `MxBy` stands for *Middle x, Bottom y*, and moves the center of the object baseline, clearly together with the whole object, to the origin of its local reference system.
2. `MxTy` stands for *Middle x, Top y*, and moves the center of the object topline to the origin of the local system.
3. `LxMy` stands for *Left x, Middle y*, and moves the center of left side of object′s containment box to the origin of the local frame, thus preparing the graphics symbol for a "left-hand" alignment.
4. `RxMy` stands for *Right x, Middle y*, and moves the center of right side of object′s box to the local origin, giving a "right-hand" alignment.

Other combinations of x and y alignments are clearly possible.